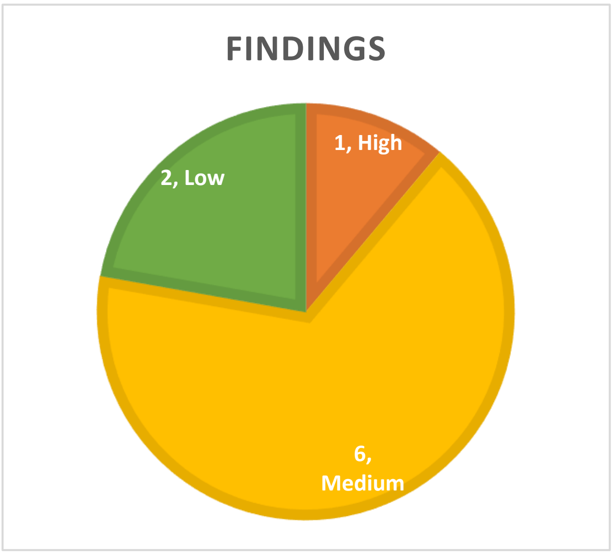
Trust Security

Smart Contract Audit

Mozaic Token

20/09/2023

Executive summary

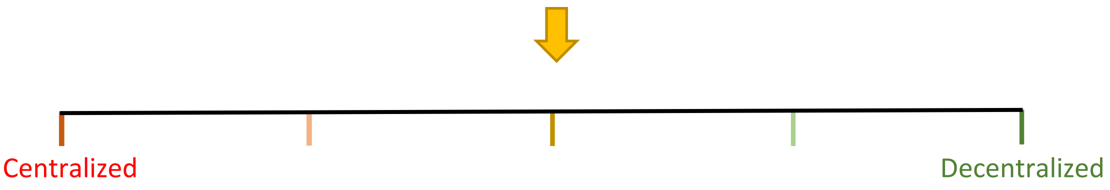


Category	Tokens
Audited file count	1
Lines of Code	341
Auditor	Trust
Time period	27/08-09/01

Findings

Severity	Total	Fixed	Acknowledged
High	1	1	-
Medium	6	3	3
Low	2	2	-

Centralization score



Signature

Digitally signed by Trust
Date: 2023-09-20 18:30:43

Document properties

Versioning

Version	Date	Description
0.1	20/09/2023	Client report

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

- MozToken.sol

Repository details

- **Repository URL:** https://github.com/Mozaic-fi/moz-staking
- **Commit hash:** 83e18e86e9487376388c68b51acee881bcf4701d
- **Mitigation review hash:** 05ea1c85182d77f019a7119482222ec8c82e2f70

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.

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.

Qualitative analysis

Metric	**Rating**	**Comments**
Code complexity	**Good**	Project kept code as simple as possible, reducing attack risks
Documentation	**Good**	Project is mostly well documented.
Best practices	**Good**	Project is aware of industry standards and mostly follows them.
Centralization risks	**Moderate**	The owner possesses some concerning privileges

Findings

High severity findings

TRST-H-1 Once minted, a Uniswap position can never be withdrawn

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

Description

In Mozaic token, the tax income is used to hold a Uniswap Mozaic/WETH LP position.

```
function addLiquidity(uint256 tokenAmount, uint256 wethAmount)
private {
    // Create new position if positionTokenId is not set
    if(positionTokenId == type(uint256).max) {
        positionTokenId = mintNewPosition(tokenAmount, wethAmount);
    } else {
        // Claim fee for the position
        collectAllFees(positionTokenId);
        // Add liquidity
        increaseLiquidityCurrentRange(positionTokenId, tokenAmount,
wethAmount);
    }
}
```

An issue arises as the tax contract does not support burning of the minted LP liquidity. This means governance is never able to cash out an ever-increasing amount of value.

Recommended mitigation

Incorporate a burning mechanism, only callable through governance.

Team response

Fixed.

Mitigation review

The code now has a *decreaseLiquidityCurrentRange()* function, callable by governance.

Medium severity findings

TRST-M-1 Funds allocated for liquidity will stay in the token contract

- **Category:** Logical flaws
- **Source:** MozToken.sol
- **Status:** Acknowledged

Description

In the *swapBack()* function, Moz tokens are split to treasury tokens and liquidity tokens. Treasury tokens are all converted to ETH and sent to the treasury, while the liquidity tokens are partially swapped to ETH and deposited as liquidity.

```
if (liquidityTokens > 0 && ethForLiquidity > 0) {
    addLiquidity(liquidityTokens, ethForLiquidity);
    emit SwapAndLiquify(
        amountToSwapForETH,
        ethForLiquidity,
        tokensForLiquidity
    );
}
```

The issue is that when depositing liquidity to Uniswap-v3, it's unlikely that all the liquidity tokens will be consumed. Anything that stays in the contract will not be marked as pending for future liquidity. This applies to both the ETH and Moz portions of the liquidity.

The effect is potentially a much lower distribution of funds sent to liquidity than intended.

Recommended mitigation

Store amount of liquidity owed in state variables and attempt to dispatch them on the next call to *swapBack()*.

Team response

Acknowledged. Funds not used for adding liquidity will be available in the MozToken contract.

TRST-M-2 The *swapTokensForEth()* function is vulnerable to a sandwich attack

- **Category:** MEV attacks
- **Source:** MozToken.sol
- **Status:** Acknowledged

Description

The *swapTokensForEth()* function will swap the input amount of Moz for Eth at spot price.

```
ISwapRouter.ExactInputSingleParams memory params = ISwapRouter
.ExactInputSingleParams({
tokenIn: address(this),
tokenOut: WETH,
fee: 3000,
recipient: address(this),
deadline: block.timestamp,
amountIn: tokenAmount,
amountOutMinimum: 0,
sqrtPriceLimitX96: 0
});
ISwapRouter(SwapRouter).exactInputSingle(params);
```

Note that the **amountOutMinimum** parameter provided to Uniswap is 0, allowing unlimited slippage.

An attacker can perform the following sandwich attack on a *swapBack()* call:

1. Perform a large Moz to WETH trade, dropping the value of Moz
2. Insert the victim TX, which would receive close to zero WETH for the trade
3. Perform a large WETH to Moz trade with the WETH from trade (1), pocketing a large amount of Moz which were traded at unfair pricing.

Note that this attack is easily performable on public mempool blockchains, but as Arbitrum has a centralized sequencer as of now, severity is reduced to medium. As the MozToken is planned to be deployed on other chains, and Arbitrum sequencer will eventually be decentralized, we highly recommend the issue to be addressed.

Recommended mitigation

Receive a slippage amount as an argument or compute it with an on-chain oracle.

Team response

Acknowledged.

TRST-M-3 Liquidity minting functions are vulnerable to sandwich attacks

- **Category:** MEV attacks
- **Source:** MozToken.sol
- **Status:** Acknowledged

Description

Liquidity is added to the MozToken's Uniswap position as shown below:

```
INonfungiblePositionManager.IncreaseLiquidityParams
memory params = INonfungiblePositionManager.IncreaseLiquidityParams({
tokenId: tokenId,
amount0Desired: amount0ToAdd,
amount1Desired: amount1ToAdd,
amount0Min: 0,
amount1Min: 0,
deadline: block.timestamp
});
nonfungiblePositionManager.increaseLiquidity(
    params
);
```

Note that when providing liquidity, it is important to provide slippage parameters for amount of tokens deposited. An attacker could manipulate the pool pricing to make the MozToken swap at any ratio they desire and profit using a similar sandwich attack to the one explained in M-2.

Recommended mitigation

Provide a tight spread in **amount0Min**,**amount1Min** which should be very close to the "amount to add" variables.

Team response

Acknowledged.

TRST-M-4 The swapping functionality is incorrectly paired to transfers, which creates DOS concerns

- **Category:** DOS attacks
- **Source:** MozToken.sol
- **Status:** Fixed

Description

In MozToken, whenever a transfer occurs and the Moz balance is above a threshold, the *swapBack()* function is called to handle swapping and adding liquidity for the protocol. However, the coupling of simple transfer requests and Uniswap position management introduces liveness issues. A caller may not provide enough gas for the *swapBack*() functionality, if at time of TX creation the *swapBack()* call was not scheduled, yet new tax tokens were collected to tip the scales. Also, there could be different reasons why the *swapBack()* call could revert, like slippage checks, and it's important not to disrupt the sensitive transferring functionality in case that occurs.

Recommended mitigation

Separate the *swapBack()* function and do not trigger it from transfer flows.

Team response

Fixed.

Mitigation review

Transfers now no longer trigger the *swapBack()* call.

TRST-M-5 The *swapBack*() function is not functional due to failing to unwrap WETH

- **Category:** ETH/WETH confusion
- **Source:** MozToken.sol
- **Status:** Fixed

Description

The *swapBack()* function calculates the amount of ETH gained from swapping Moz tokens as seen below:

```
uint256 initialETHBalance = address(this).balance;
swapTokensForEth(amountToSwapForETH);
uint256 ethBalance = address(this).balance - initialETHBalance;
uint256 ethForTreasury = (ethBalance * tokensForTreasury) /
(totalTokensToSwap - (tokensForLiquidity / 2));
uint256 ethForLiquidity = ethBalance - ethForTreasury;
```

Importantly, the *swapTokensForEth()* function swaps Moz into WETH and does not unwrap them. Therefore, the **ethBalance** calculated will be zero.

```
function swapTokensForEth(uint256 tokenAmount) private {
    _approve(address(this), SwapRouter, tokenAmount);
    ISwapRouter.ExactInputSingleParams memory params = ISwapRouter
    .ExactInputSingleParams({
    tokenIn: address(this),
    tokenOut: WETH,
    fee: 3000,
    recipient: address(this),
    deadline: block.timestamp,
    amountIn: tokenAmount,
    amountOutMinimum: 0,
    sqrtPriceLimitX96: 0
    });
    ISwapRouter(SwapRouter).exactInputSingle(params);
}
```

Therefore, all funds for the treasury and liquidity will remain in the MozToken contract. They can still be rescued using the *withdrawStuckToken()* function.

Recommended mitigation

Handle ETH/WETH conversion carefully.

Team response

Fixed.

Mitigation review

The code has been refactored to handle almost everything in WETH, and performs wrapping correctly.

TRST-M-6 ETH transfer to the treasury is not validated

- **Category:** Validation issues
- **Source:** MozToken.sol
- **Status:** Fixed

Description

In *swapBack()*, ETH is transferred to the treasury.

```
(success, ) = address(treasury).call{value: ethForTreasury}("");
```

The function never verifies the transfer has succeeded. If the call reverts, there would be excess ETH in the MozToken contract.

Recommended mitigation

Verify **success==True** after the call.

Team response

Fixed.

Mitigation review

The issue has been addressed.

Low severity findings

TRST-L-1 withdrawStuckMoz() uses deprecated transfer()

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

Description

In *withdrawStuckMoz()*, the remaining ETH balance is sent to the contract owner.

```
function withdrawStuckMoz() external onlyOwner {
    uint256 balance = IERC20(address(this)).balanceOf(address(this));
    IERC20(address(this)).transfer(msg.sender, balance);
    payable(msg.sender).transfer(address(this).balance);
}
```

It is frowned upon to use the *transfer()* call for Eth transfers, as it translated to a fixed 2300 gas stipend. As gas costs for opcodes change, the receiver's code may not be able to handle the transfer properly.

Recommended mitigation

Use the ".call" transfer variant to pass gas safely to the receiver.

Team response

Fixed.

Mitigation review

Suggested fix has been applied.

TRST-L-2 Popular ERC20 tokens could be stuck in the contract

- **Category:** ERC20 compatibility issues
- **Source:** MozToken.sol
- **Status:** Fixed

Description

The MozToken owner can rescue tokens accidentally sent to the contract.

```
function withdrawStuckToken(address _token, address _to) external
onlyOwner {
    require(_token != address(0), "_token address cannot be 0");
    uint256 _contractBalance =
IERC20(_token).balanceOf(address(this));
    IERC20(_token).transfer(_to, _contractBalance);
}
```

Note that it uses the standard ERC20 transfer() call. However, there are hundreds of tokens like BNB and USDT which would revert on this call, because they have different function signatures. This would make such tokens permanently stuck in the MozToken contract.

Recommended mitigation

It is recommended to use OpenZeppelin's *safeTransfer()* variant.

Team response

Fixed.

Mitigation review

Arbitrary tokens can no longer be rescued, so this issue is not relevant.

Centralization risks

TRST-CR-1 Owner has control of all funds in MozToken

The MozToken is implemented so that governance has ultimate control over the tax revenue. Owner can:

1. Choose the treasury address
2. Burn all supplied Moz/ETH liquidity
3. Exfiltrate any Moz or ETH residing in the contract

TRST-CR-2 Owner can control fees up to 10%

The owner can set up fees for Moz transfers up to 10% of the transferred amount. Note that the split between liquidity fee and treasury fee is not strictly enforced at the code level.