MCR369 - Buffet Audit Report

Audited By: zuhaibmohd And 33Audits

19th February 2024

Introduction

A time-boxed security review of the **MCR369 - Buffet** contract was done by **zuhaibmohd** and **33Audits**, focusing on the security aspects of the smart contracts.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where I try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any vulnerabilities. Subsequent security reviews, bug bounty programs, and on-chain monitoring are recommended.

About zuhaibmohd

zuhaibmohd is an independent smart contract security researcher. Check his previous work here or reach out on X @zuhaib44.

About 33Audits

33Audits is an independent smart contract security researcher. Check his previous work here or reach out on X @33Audits.

About MCR369 - Buffet

`MCR369 - Buffet` comprise a series of contracts enabling users to stake the MCR369 token. Users receive rewards in the form of reward tokens periodically, based on the amount of MCR369 tokens staked in the staking contracts.

Severity classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - The technical, economic, and reputation damage from a successful attack

Likelihood - The chance that a particular vulnerability gets discovered and exploited

Severity - The overall criticality of the risk

Informational - Findings in this category are recommended changes for improving the structure, usability, and overall effectiveness of the system.

Security Assessment Summary

review commit hash -
ebc1d5350b4898d843a234e1970897d393493898

fixes review commit hash -
b2025a8b28b2d21b39bc0f02dfd531e161fe8f70

Scope

The following smart contracts were in the scope of the audit:

- `contracts/MCR369.sol`
- `contracts/MCR369Collector.sol`
- `contracts/MCR369Staking.sol`

Findings Summary

ID	Title	Severity	Status
[H-01]	No new tokens can be added via `addRewardToken` because of incorrect implementation of `checkRewardToken`	High	Fixed

ID	Title	Severity	Status
[H-02]	Distribution of rewards token can be sandwiched	High	Fixed
[M-01]	Missing access control for `distributeFees`	Medium	Fixed
[M-02]	Solmate `safeTransfer` and `safeTransferFrom` does not check for codesize	Medium	Acknowledged
[M-03]	Fee on transfer tokens will lead to bad accounting of state variables	Medium	Acknowledged
[M-04]	Rebasing rewards get stuck in contract	Medium	Acknowledged
[L-01]	Missing function caller check for `MCR369Staking::distributeFees()`	Low	Acknowledged
[L-02]	Contracts do not implement 2-Step-Process for transferring ownership	Low	Fixed
[I-01]	Redundant code use all across smart contracts	Info	Acknowledged
[I-02]	Use named mappings for clarity	Info	Fixed

Detailed Findings

[H-01] - No new tokens can be added via addRewardToken because of incorrect implementation of checkRewardToken

Links:

https://github.com/MCR369/MCR369StakingContracts/blob/ebc1d5350b4898d843a234e1970897d393493898/contracts/MCR369Collector.sol#L160-L169

Description:

The `addRewardToken` function is used to include new reward tokens for distribution to MCR369 token stakers. This function utilizes a call to `checkRewardToken` to verify if the token has already been added. However, an issue arises after the addition of the first reward token. When the `onlyOwner` attempts to add a second reward token, the index starts with 1, even though it does not exist in the rewardToken array. Consequently, this leads to a reverting error due to an **out-of-bounds access**.

Impact:

The `onlyOwner` constraint allows operations on only one reward token at a time, undermining the entire purpose of the `rewardToken` array and preventing the addition of multiple reward tokens.

Proof Of Concept:

Execute the following test to validate the issue. Instructions for setting up the test suite can be found at https://gist.github.com/zzzuhaibmohd/e66a92be31326ca9c1e0c1fdf2ca9a3b.

```
    function test_addRewardToken_reverts() public {
        //Step 1: Add the second reward token (address(0x1111))
        vm.expectRevert();
        mcr369collector.addRewardToken(address(0x1111));

        //Step 2: Remove the existing token also fails
```

```
        //vm.expectRevert("panic: array out-of-bounds access
        (0x32)");
        vm.expectRevert();
        mcr369collector.removeRewardToken(address(0x1111));
    }
```

Recommendations:

Correct the logic in the `checkRewardToken` function and create a couple of unit tests to ensure its proper functionality.

[H-02] - Distribution of rewards token can be sandwiched

Links:

https://github.com/MCR369/MCR369StakingContracts/blob/ebc1d5350b4898d843a234e1970897d393493898/contracts/MCR369Collector.sol#L179

Description:

In an ideal staking contract, stakers should be rewarded based on the staked amount and the duration of their token staking. However, in the current implementation, there is a vulnerability where a user can front-run the `distributeFees` function either by calling it themselves or waiting for a permissioned user to do so. By staking a significant amount of MCR tokens, the malicious user can obtain the majority of rewards, and what's worse, this can be accomplished through a flash loan. Subsequently, the malicious user can unstake the MCR369 tokens, essentially stealing the rewards from honest stakers.

Impact:

Honest users are being stolen of their reward tokens, while malicious users exploit the system by staking tokens for a tiny period and illegitimately claiming the rewards.

Proof Of Concept:

Execute the following test to validate the issue. Instructions for setting up the test suite can be found at https://gist.github.com/zzzuhaibmohd/e66a92be31326ca9c1e0c1fdf2ca9a3b.

```
function test_frontRunning_distributeFees() public {

        //Alice stakes 50 ether - imagine for 5 days
        vm.startPrank(alice);
        mcr369Token.approve(address(mcr369staker), 5000 ether);
        mcr369staker.stake(50 ether);
        vm.stopPrank();

        //Bob stakes 50 ether - imagine for 10 days
        vm.startPrank(bob);
        mcr369Token.approve(address(mcr369staker), 5000 ether);
        mcr369staker.stake(100 ether);
        vm.stopPrank();

        //Eve is monitoring the public mempool, waiting for
        distributeFees to be called
        //Eve deposits large stake of mcr369Token buying it from
        the market or via flasloan
        vm.startPrank(eve);
        mcr369Token.approve(address(mcr369staker), 5000 ether);
        mcr369staker.stake(1000 ether);
        vm.stopPrank();

        //distribute reward is called, this tx was frontrun by
        eve
        mcr369collector.distributeFees();
        vm.prank(eve);
        mcr369staker.unstake(1000 ether);

        console2.log(mcr369staker.claimable(alice,
        address(rewardToken)));
        console2.log(mcr369staker.claimable(bob,
        address(rewardToken)));
        console2.log(mcr369staker.claimable(eve,
        address(rewardToken)));
    }
```

Recommendations:

You could include a block delay between the time a user stakes and the time they decide to unstake. For example if a user stakes you can store the current `block.timestamp` or `block.number` and then make sure that when the unstake at least a certain amount of blocks have passed (two blocks should suffice). This would stop a user from being able to take out a flashloan to carry out this attack. However, a whale could still carry out a similar attack though the chances are a bit smaller. Time locks also are

another solution as well as withdrawal queues which force users to stake for a minimum amount of time however all these solutions come at different costs.

Consider reviewing the implementation of Sushiswap's Masterchef contract which can be found at https://github.com/sushiswap/masterchef/tree/master.

To gain a better understanding of the issue, please refer to the following blog: https://www.rareskills.io/post/staking-algorithm.

[M-01] - Missing access control for `distributeFees`

Links:

https://github.com/MCR369/MCR369StakingContracts/blob/ebc1d5350b4898d843a234e1970897d393493898/contracts/MCR369Collector.sol#L172

Description:

The `distributeFees` function, being permissionless, allows users to monitor the reward tokens deposited in the collector contract and call the function independently. While this alone may not be exploitable, users can effectively track the instance when reward tokens are deposited, stake their tokens at that precise moment, distribute rewards to themselves, and subsequently unstake the tokens to benefit unfairly.

Impact:

Malicious users can exploit the system to steal the `rewardTokens` allocated for honest stakers.

Recommendations:

A good solution would be to implement `onlyOwner` control. If maintaining a permissionless approach is crucial, consider incorporating a timeout mechanism, such as 5 days or 10 days, after each call. This helps prevent users from spamming the function.

[M-02] - Solmate `safeTransfer` and `safeTransferFrom` does not check for codesize

Links:

https://github.com/MCR369/MCR369StakingContracts/blob/ebc1d5350b4898d843a234e1970897d393493898/contracts/MCR369Staking.sol#L8

Description:

The `safeTransfer` and `safeTransferFrom` in solmate's library don't check the existence of code at the token address. This is a known issue while using solmate's libraries.

Impact:

May lead to miscalculation and loss of funds.

Proof Of Concept:

This may lead to miscalculation of funds and may lead to loss of funds , because if `safeTransfer()` and `safeTransferfrom()` are called on a token address that doesn't resolve to a contract but instead an EOA, it will always return success, bypassing the return value check. Due to this protocol thinking that funds had been transferred successfully, and record would be accordingly calculated, but in reality funds were never transferred. For example in the `distributeRewards` function the `rewardAmount` gets added to the `_rewardDistributed` however in this casee the transfer would have failed leaving these values incorrect.

```
_rewardDistributed[rewardAsset] += rewardAmount;
_rewardIndex[rewardAsset] += (rewardAmount * _BASE) /
 totalMCR369Staked;
```

Recommendations:

Before calling the `safeTransfer` function ensure that the token address being passed is a contract and not an EOA using a codesize check. If it isn't a contract then revert.

[M-03] - Fee on transfer tokens will lead to bad accounting of state variables

Description:

Fee on transfer tokens will lead to improper accounting for `_rewardDistributed[rewardAsset]` and `_rewardIndex[rewardAsset]` leading to users getting an inflated rewardIndex causing them to miss out on potential rewards.

Proof Of Concept:

Imagine you want to distribute 100 FEE tokens to your users. You call `distributeReward` with an amount of 100 except every transaction of fee token takes a 10% fee only sending 90 to the contract however updating the `rewardIndex` with an amount of 100. This would be that the `rewardIndex` would start to deviate significantly from the actual amount of tokens being sent to the contract causing a loss to future users.

```
    function distributeReward(
        address rewardAsset,
        uint rewardAmount
    ) external nonReentrant {
        require(isRewardToken[rewardAsset],
        "INVALID_REWARD_ASSET");

        SafeTransferLib.safeTransferFrom(
            ERC20(rewardAsset),
            msg.sender,
            address(this),
            rewardAmount
        );
    //@audit - this doesn't work with fee on transfer tokens
    //@audit - this doesn't work with rebasing tokens
        _rewardDistributed[rewardAsset] += rewardAmount;
        _rewardIndex[rewardAsset] += (rewardAmount * _BASE) /
        totalMCR369Staked;

        emit RewardDistribution(rewardAsset, rewardAmount);
    }
```

Recommendations:

There isn't much concern for this issue unless you explicitly intend on using fee on transfer tokens though they aren't as common as they used to be. Ideally, just you know that fee on transfer tokens will not work with the contracts.

[M-04] - Rebasing rewards get stuck in contract

Description:

Rebasing tokens will lead to improper accounting for `_rewardDistributed[rewardAsset]` and `_rewardIndex[rewardAsset]` leading to users getting an inflated rewardIndex and potentially causing funds to get stuck in the contract.

Proof Of Concept:

Imagine you want to distribute 100 REBASE tokens to your users. Whenever a deposit is made a user passes a value as an argument to the `stake()` which the value then gets added to the `_claimable[staker][rewardAsset]` mapping. Let's imagine REBASE token then rebases by 50% burning 50% of the total supply. This would mean that the `_claimable[staker][rewardAsset]` would start to deviate significantly from the actual amount of tokens in the contract. The user would be able to withdraw their 50 REBASE tokens and then we would subtract `_claimable[staker][rewardAsset]` by this amount. The problem here is since the rebase happened that should be all we can withdraw from the contract but that isn't the case. The attacker could then call this function again and steal another user's 50 REBASE tokens that were in the contract.

```
  function unstake(uint value) external nonReentrant {
        _updateAllRewards(msg.sender);

        totalMCR369Staked -= value;
        stakeBalance[msg.sender] -= value;

        SafeTransferLib.safeTransfer(ERC20(mcr369), msg.sender,
        value);

        emit Unstake(msg.sender, value);
    }
```

The updateRewards function.

```
  function _updateRewards(address staker, address rewardAsset)
        internal {
        _claimable[staker][rewardAsset] += _calculateRewards(
            staker,
            rewardAsset
        );
        _rewardIndexOf[staker][rewardAsset] =
        _rewardIndex[rewardAsset];
    }
```

Recommendations:

There isn't much concern for this issue unless you explicitly intend on using rebasing (LDO, OHM) tokens though they aren't as common as they used to be. If you do however, deflating tokens would not work with the current implementation, and a function would need to be added to withdraw the rewards from inflating tokens.

[L-01] - Missing function caller check for `MCR369Staking::distributeFees()`

Links:

https://github.com/MCR369/MCR369StakingContracts/blob/ebc1d5350b4898d843a234e1970897d393493898/contracts/MCR369Staking.sol#L518-L521

Description:

The code currently lacks a security check in the `distributeReward` function, which is intended to be called exclusively from the collector contract. While not a security issue per se, the absence of a proper access control mechanism could potentially lead to unintended calls from unauthorized sources.

Impact:

Without the necessary verification, any random user could invoke the `distributeReward` function, leading to the undesired distribution of

`rewardTokens` to users. This could undermine the intended reward allocation process and compromise the integrity of the system.

Recommendations:

To enhance security and maintain control over the `distributeReward` function, it is advisable to add a require statement as follows:

```
require(msg.sender == address(mcr369collector), "Unauthorized
        caller");
```

By implementing this check, the function will only execute if the caller is the designated `MCR369Collector` contract, acting as a crucial authorization check. This modification helps prevent unauthorized parties from triggering the reward distribution process, reinforcing the overall security of the system.

[L-02] - Contracts does not implement 2-Step-Process for transferring ownership

Links:

https://github.com/MCR369/MCR369StakingContracts/blob/ebc1d5350b4898d843a234e1970897d393493898/contracts/MCR369Collector.sol#L6 https://github.com/MCR369/MCR369StakingContracts/blob/ebc1d5350b4898d843a234e1970897d393493898/contracts/MCR369Staking.sol#L371

Description:

The contracts `MCR369Collector.sol` and `MCR369Staking.sol` does not implement a 2-Step-Process for transferring ownership. So ownership of the contract can easily be lost when making a mistake when transferring ownership.

Since the privileged roles have critical function roles assigned to them. Assigning the ownership to a wrong user can be disastrous. So Consider using the Ownable2Step contract from OZ (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable2Step.sol) instead.

The way it works is there is a `transferOwnership` to transfer the ownership and `acceptOwnership` to accept the ownership. Refer the above Ownable2Step.sol for more details.

Impact:

Risk of losing ownership of the contracts due to incorrect assignment of the new owner.

Recommendations:

Implement 2-Step-Process for transferring ownership via OZ `Ownable2Step`.

[I-01] - Redundant code use all across smart contracts

Links:

https://github.com/MCR369/MCR369StakingContracts/tree/main/contracts

Description:

The code currently incorporates the solmate libraries for ERC20, Ownership, and Reentrancy checks; however, it seems that rather than importing the code directly through the libraries, the developer has copied and pasted the code into the contract.

To optimize the code and improve maintainability, it is advisable to utilize the `npm i solmate` command to install the solmate package and then import the required libraries directly. This approach not only enhances code organization but also reduces the contract size by avoiding unnecessary duplication.

By importing the solmate libraries directly from the npm package (https://www.npmjs.com/package/solmate?activeTab=dependents), you ensure that the contract benefits from updates and improvements made to the external libraries without the need for manual intervention. This promotes cleaner code practices and facilitates better collaboration within the development community.

Recommendations:

To implement this recommendation, execute `npm i solmate` in the project directory, and then modify the contract to import the required solmate

libraries directly. This not only streamlines the codebase but also aligns with best practices for code reuse and maintenance in Solidity development.

[I-02] - Use named mappings for clarity

Description:

There are a lot of different mappings that are tracking different values for users. Consider either consolidating these into a user struct or using named mappings for simplicity.

```
- mapping(address => uint) internal _rewardDistributed;
+ mapping(address asset => uint amount) internal
        _rewardDistributed;
```
