#HashEx
BLOCKCHAIN SECURITY

Moraswap Masterchef

smart contracts final audit report

October 2022

hashex.org

contact@hashex.org

Contents

1. Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and HashEx and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (HashEx) owe no duty of care towards you or any other person, nor does HashEx make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and HashEx hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, HashEx hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against HashEx, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed. HashEx owns all copyright rights to the text, images, photographs, and other content provided in the following document. When using or sharing partly or in full, third parties must provide a direct link to the original document mentioning the author (hashex.org).

2. Overview

HashEx was commissioned by the Moraswap team to perform an audit of their smart contract. The audit was conducted between 25/10/2022 and 28/10/2022.

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts
- Formally check the logic behind given smart contracts.

Information in this report should be used for understanding the risk exposure of smart contracts, and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.

The code is available at @moraswap/moraswap-core GitHub repository after the d71daae commit.

Update: the Moraswap team has responded to this report. The updated code is located in the same GitHub repository after the 7704690 commit.

2.1 Summary

Project name	Moraswap Masterchef
URL	https://moraswap.com
Platform	Neon EVM
Language	Solidity

2.2 Contracts

Name	Address
MasterChef	https://github.com/moraswap/moraswap-core/blob/d71daaeb456d3fdab7930f4fd9d887f3c12ed17e/contracts/MasterChef.sol

3. Found issues

High	1 (13%)
Medium	2 (25%)
Low	2 (25%)
Info	3 (37%)

C1. MasterChef

ID	Severity	Title	Status
C1-01	High	Emission rate not limited	Acknowledged
C1-02	Medium	Unfair distribution of awards without updateAllPoolsl()	Resolved
C1-03	Medium	Rewarder is not notified in emergency withdraw() about balance change	Resolved
C1-04	Low	Gas shortage in massUpdatePools()	Acknowledged
C1-05	Low	Gas optimization	Resolved
C1-06	Info	Reward token has a cap	Acknowledged
C1-07	Info	Tokens with commissions are not supported	Acknowledged
C1-08	Info	Redundant token burn functionality	Acknowledged

4. Contracts

C1. MasterChef

Overview

This contract is responsible for adding farming functionality to the project, i.e it allows you to invest in it a certain number of LP tokens in specially designated pools, for which you will be credited with project tokens (which are calculated according to the formula).

Issues

C1-01 Emission rate not limited

High Acknowledged

Using the `setEmissionRate()` function, the owner can change the `moraPerSecond` variable, which is used in calculating rewards in the system. With this feature, the owner can manipulate the number of rewards produced per second, obtain critical token share and withdraw liquidity from the reward token pair, bringing down its price.

```
function setEmissionRate(uint256 _moraPerSecond) external onlyOwner {
    updateAllPools();
    moraPerSecond = _moraPerSecond;

    emit SetEmissionRate(_moraPerSecond);
}
```

Recommendation

Remove the functionality or transfer ownership to `Timelock` contract with a minimum delay of at least 24 hours and `MultiSig` as admin. This won't stop the owner from possible rights abuse, but it will help users to be informed about upcoming changes.

Update

The Moraswap team said that will transfer ownership to a 24-hours TimeLock contract after deployment.

C1-02 Unfair distribution of awards without updateAllPoolsl()

Medium Resolved

The reward distribution for pools, where the `updatePool()` function is rarely called, can become too small (unfair) if new pools are added or updated without the `_withUpdate` flag.

Recommendation

Force a mass update without the flag.

C1-03 Rewarder is not notified in emergency withdraw() about balance change

Medium Resolved

The `deposit()` and `withdraw()` functions notify the `Rewarder` contract of a change in the number of LP tokens, but the `emergencyWithdraw()` function does not. Because of this, there may be errors in the calculation of rewards in the Rewarder contract.

```
function deposit(uint256 _pid, uint256 _amount) external nonReentrant {
        ...
        address rewarder = pool.rewarder;
        if (rewarder != address(0)) {
            IRewarder(rewarder).onReward(address(msg.sender), user.amount);
        }
        ...
    }

    function withdraw(uint256 _pid, uint256 _amount) external nonReentrant {
        ...
        address rewarder = pool.rewarder;
        if (rewarder != address(0)) {
            IRewarder(rewarder).onReward(address(msg.sender), user.amount);
        }
```

```
        ...
    }

    function emergencyWithdraw(uint256 _pid) external nonReentrant {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        uint256 amount = user.amount;
        user.amount = 0;
        pool.totalLp = pool.totalLp.sub(amount);
        user.rewardDebt = 0;
        pool.lpToken.safeTransfer(address(msg.sender), amount);

        // Rewarder is not notified!

        emit EmergencyWithdraw(msg.sender, _pid, amount);
    }
```

Recommendation

It is recommended to add a call to the `Rewarder` contract in `emergencyWithdraw()` to change the data about the stored LP tokens and enclose this call in a try-catch block. Regard, there are certain cases when the try-catch block does not save from reverting the transaction (for example, if there is a call not to the contract address or an infinite loop inside the block).

Update

Rewarder notification was added inside try-catch block with proper check preventing possible transaction reverting.

```
function emergencyWithdraw(uint256 _pid) external nonReentrant {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    uint256 amount = user.amount;
    user.amount = 0;
    address rewarder = pool.rewarder;
    pool.totalLp = pool.totalLp.sub(amount);
    user.rewardDebt = 0;
    pool.lpToken.safeTransfer(address(msg.sender), amount);
    emit EmergencyWithdraw(msg.sender, _pid, amount);
```

```
        uint256 size;
        assembly {
            size := extcodesize(rewarder)
        }
        if (rewarder != address(0) && size > 0) {
            try IRewarder(rewarder).onReward(address(msg.sender), 0) {} catch {
                emit FailedToNotifyRewarder(msg.sender, _pid);
            }
        }
    }
```

However, Rewarder contract itself is out of audit scope and we can't guarantee notification won't fail because of the `onReward()` method implementation and the attack can't take place.

C1-04 Gas shortage in massUpdatePools()

Low Acknowledged

The `massUpdatePools()` function cycles through the update of each pool from the `PoolInfo` array. If there are too many pools, then there may not be enough gas to process this function, which will cause the inability to set `moraPerSecond` and `burnPercent`.

Update

After resolvement of C1-03 `massUpdatePools()` became obligatory in `addPool()` and `setPool()` functions, what may lead to their DoS in case `massUpdatePools()` consumes too much gas. The owner should be careful while adding new pools ensuring functions containing mass pools update won't exceed the block gas limit.

C1-05 Gas optimization

Low Resolved

a. `name` should be constant or even removed as it is not addressed inside a contract;

b. `burnAddress` should be const;

c. Multiple `user.amount` reads from storage in `deposit()` and `withdraw()` methods.

C1-06 Reward token has a cap

Info Acknowledged

The reward token has max supply cap of `1000000000e18` tokens. When the total tokens supply reaches this amount the masterchef will stop generating rewards.

C1-07 Tokens with commissions are not supported

Info Acknowledged

Upon an attempt to enter commission tokens, `user.amount` is calculated incorrectly in the `deposit()` function, which leads to the fact that in the `withdraw()` function we will be able to withdraw more tokens than we actually have.

```
function deposit(uint256 _pid, uint256 _amount) external nonReentrant {
        ...
        uint256 amount = user.amount;
        ...
        if(_amount > 0) {
            pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
            user.amount = amount.add(_amount); // error!
            pool.totalLp = lpSupply.add(_amount);
        }
        ...
    }
```

C1-08 Redundant token burn functionality

Info Acknowledged

In the `updatePool()` function, the newly minted Mora tokens are immediately burned, which is a redundant action.

```
function updatePool(uint256 _pid) public {
        ...
        (uint256 burnMora, uint256 farmMora) = calculate(moraReward);

        mora.mint(address(this), burnMora.add(farmMora));
        if (burnMora > 0) {
            mora.transfer(burnAddress, burnMora);
        }
```

```
        ...
    }
```

5. Conclusion

1 high, 2 medium, 2 low severity issues were found during the audit. 2 medium, 1 low issues were resolved in the update.

The audit didn't reveal serious issues menacing staked users' funds. However, potential users should be informed about the reward token cap and possible owner's rights abuse with reward rate manipulation.

This audit includes recommendations on improving the code and preventing potential attacks.

Appendix A. Issues' severity classification

- **Critical.** Issues that may cause an unlimited loss of funds or entirely break the contract workflow. Malicious code (including malicious modification of libraries) is also treated as a critical severity issue. These issues must be fixed before deployments or fixed in already running projects as soon as possible.
- **High.** Issues that may lead to a limited loss of funds, break interaction with users, or other contracts under specific conditions. Also, issues in a smart contract, that allow a privileged account the ability to steal or block other users' funds.
- **Medium.** Issues that do not lead to a loss of funds directly, but break the contract logic. May lead to failures in contracts operation.
- **Low.** Issues that are of a non-optimal code character, for instance, gas optimization tips, unused variables, errors in messages.
- **Informational.** Issues that do not impact the contract operation. Usually, informational severity issues are related to code best practices, e.g. style guide.

Appendix B. List of examined issue types

- Business logic overview
- Functionality checks
- Following best practices
- Access control and authorization
- Reentrancy attacks
- Front-run attacks
- DoS with (unexpected) revert
- DoS with block gas limit
- Transaction-ordering dependence
- ERC/BEP and other standards violation
- Unchecked math
- Implicit visibility levels
- Excessive gas usage
- Timestamp dependence
- Forcibly sending ether to a contract
- Weak sources of randomness
- Shadowing state variables
- Usage of deprecated code

 contact@hashex.org

 @hashex_manager

 blog.hashex.org

 linkedin

 github

 twitter

#HashEx
BLOCKCHAIN SECURITY