#HashEx
Blockchain Security

CronaSwap Staking

smart contracts
final audit report

January 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 CronaSwap team to perform an audit of their smart contract. The audit was conducted between January 24 and January 28, 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 the @cronaswap/cronaswap-staking GitHub repository and was audited after the commit 69e6af0.

Update: the Cronaswap team has responded to this report. The updated code is located in the GitHub repository after commit 13f4bef.

The audited contracts are deployed to the Cronos Chain mainnet:

MasterChefTool: 0x69d850d2F4007902c8fF0f0d324a585dF916007B

MasterChefV2: 0x7B1982b896CF2034A0674Acf67DC7924444637E4

FeeDistributor: 0x7178E49EFBe21ef6F1F58f4e3bd9E79a9CeAC58B

RewardPool: 0x79956c0ccC9906Ee24B96CCF02234da1FB456dD8

VotingEscrow: 0x1E9d7DD649A1714f424f178036dbb79FA702b37d

TimeLock24H: 0xF4FD04bf83da4ac7C209A7A45eD8Ef17fd367EFB

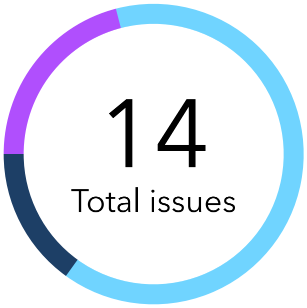
2.1 Summary

Project name	CronaSwap Staking
URL	https://cronaswap.org/
Platform	Cronos Network
Language	Solidity

2.2 Contracts

Name	Address
TimeLock24H	0xF4FD04bf83da4ac7C209A7A45eD8Ef17fd367EFB
VotingEscrow	0x1E9d7DD649A1714f424f178036dbb79FA702b37d
FeeDistributor	0x7178E49EFBe21ef6F1F58f4e3bd9E79a9CeAC58B
MasterChefV2	0x7B1982b896CF2034A0674Acf67DC7924444637E4
RewardPool	0x79956c0ccC9906Ee24B96CCF02234da1FB456dD8
MasterChefTool	0x69d850d2F4007902c8fF0f0d324a585dF916007B

3. Found issues



● Medium	3 (21%)
● Low	9 (64%)
● Info	2 (15%)

C1. TimeLock24H

ID	Severity	Title	Status
C1-01	● Medium	Low minimal delay	✓ Resolved
C1-02	● Low	Usage of external and public function types	✓ Resolved

C2. VotingEscrow

ID	Severity	Title	Status
C2-01	● Medium	Reward harvest recipient	✓ Resolved
C2-02	● Low	Using asserts instead of require statements	✓ Resolved
C2-03	● Low	Gas optimisation	✓ Resolved
C2-04	● Info	Emit event on try-catch failure to log errors	✓ Resolved
C2-05	● Info	Arrays are declared with a fixed size	✓ Acknowledged

C3. FeeDistributor

ID	Severity	Title	Status
C3-01	● Low	Using assert() instead of require()	✓ Resolved

C4. MasterChefV2

ID	Severity	Title	Status
C4-01	● Low	Harvest all rewards may run out of gas	✓ Acknowledged
C4-02	● Low	Function setCronaPerSecond may cause disruption in the reward calculations	✓ Acknowledged
C4-03	● Low	Usage of external and public function types	✓ Acknowledged
C4-04	● Low	Pool's working supply may not be correctly update in specific cases	✓ Resolved

C5. RewardPool

ID	Severity	Title	Status
C5-01	● Medium	Withrawals can be blocked if the owner transfers xCrona tokens	✓ Resolved
C5-02	● Low	Gas optimisations	✓ Resolved

4. Contracts

C1. TimeLock24H

Overview

A timelock contract aimed to provide a delay before a transaction can be executed. At first, the transaction is queued, after passing the delay the contract admin can execute it. The minimal delay for the audited contract is 6 hours.

Issues

C1-01 Low minimal delay ● Medium ✓ Resolved

The timelock contract can set the minimum delay of 6 hours. We recommend setting the `MINIMUM_DELAY` variable for at least 24 hours.

Update

The minimal timelock delay has been set to 24 hours in the update.

C1-02 Usage of external and public function types ● Low ✓ Resolved

The functions `setDelay()`, `acceptAdmin()`, `setPendingAdmin()`, `queueTransaction()`, `cancelTransaction()`, `executeTransaction()` can be declared as external to save gas.

C2. VotingEscrow

Overview

Allows users to stake their Crona tokens. Size and time of stake correlate with earning the amount of Crona tokens in the pools of the MasterChefV2 contract.

Staked users' Crona tokens are be redirected to the RewardPool contract, where they will be staked in the zero pool of the MasterChefV1 contract.

Issues

C2-01 Reward harvest recipient ● Medium ✓ Resolved

The VotingEscrow contract calls `masterchef.harvestAllRewards(address)` in several functions passing the address parameter to the function. The MasterChef checks if the address has deposited tokens and if so calls the `withdraw()` function to get the rewards. The problem is that the `withdraw()` function checks and sends rewards for the `msg.sender`, which in this case is the VotingEscrow contract, but to the passed address.

```
function harvestAllRewards(address _user) public {
    uint256 length = poolInfo.length;
    for (uint256 pid = 0; pid < length; ++pid) {
        if (userInfo[pid][_user].amount > 0) {
            withdraw(pid, 0);
        }
    }
}
```

C2-02 Using asserts instead of require statements ● Low ✓ Resolved

We recommend using `require()` statements instead of `assert()` for checking the input parameters to conform to the best practices in smart contracts development. This saves gas if a wrong parameter is passed to a function: the `assert()` statement uses all the gas provided in the transaction while the `require()` statement uses only gas spent before the failing statement has been reached.

C2-03 Gas optimisation ● Low ✓ Resolved

State variable `decimals` L37 can be declared as `immutable` to save gas.

C2-04 Emit event on try-catch failure to log errors ● Info ✓ Resolved

The function `emergencyWithdraw()` calls the `rewardPool` in a try-catch statement. In case of failure in the `rewardPool` call, the error will be silently dismissed.

```
function emergencyWithdraw() external {
    require(emergency, "Only can be called in an emergency");

    LockedBalance storage _locked = locked[msg.sender];
    uint256 value = uint256(_locked.amount);
    try rewardPool.withdrawFor(msg.sender, value) returns (bool) {
    } catch {
    }
    ...
}
```

Recommendation

Emit an Error event in a catch block:

```
    try rewardPool.withdrawFor(msg.sender, value) returns (bool) {
    } catch (bytes memory error) {
      emit Error(error);
    }
```

C2-05 Arrays are declared with a fixed size ● Info ✓ Acknowledged

The `Point` arrays are declared with a predefined size and cannot be extended. These arrays cover a large part of storage and thus make the collisions likely.

```
Point[100000000000000000000000000000] public pointHistory; // epoch -> point
mapping(address => Point[1000000000]) public userPointHistory; // user ->
```

```
Point[user_epoch]
```

Recommendation

We recommend using a dynamic-sized array for the `Point` arrays.

```
    Point[] public pointHistory; // epoch -> point
    mapping(address => Point[]) public userPointHistory; // user -> Point[user_epoch]
```

Team response

TBA

C3. FeeDistributor

Overview

A fork of the Curve's FeesDistributor contract rewritten in Solidity language.

Issues

C3-01 Using assert() instead of require() ● Low ✓ Resolved

For checking input parameters, we recommend using `require()` statements instead of `assert()` to conform to the best practices in smart contracts development. This saves gas if a wrong parameter is passed to a function: the `assert()` statement uses all the gas provided in the transaction while the `require()` statement uses only gas spent before the failing statement was reached.

C4. MasterChefV2

Overview

The contract allows to stake tokens in the pools, and get rewards. The main difference from the classic MasterChef contract is that reward amounts depend on the size of users' stake in the VotingEscrow contract.

Issues

C4-01 Harvest all rewards may run out of gas ● Low ✓ Acknowledged

The function harvestAllRewards iterates over an array of an unlimited size and may run out of block gas limit.

```
function harvestAllRewards(address _user) public {
    uint256 length = poolInfo.length;
    for (uint256 pid = 0; pid < length; ++pid) {
        if (userInfo[pid][_user].amount > 0) {
            withdraw(pid, 0);
        }
    }
}
```

The owner of the contract should be aware of the issue when adding pools and must check gas usage.

C4-02 Function setCronaPerSecond may cause disruption in the reward calculations ● Low ✓ Acknowledged

The owner of the contract may update Crona per second value with the `setCronaPerSecond()` function. If this value is not equal to the rate that the MasterChef receives Crona, the calculations for users' rewards will be disrupted.

C4-03 Usage of external and public function types ● Low ✓ Acknowledged

The functions setStartTime(), add(), set(), deposit(), harvestAllRewards(), emergencyWithdraw(), updateCronaPerSecond(), setCronaPerSecond() can be declared as external to save gas.

C4-04 Pool's working supply may not be correctly update in specific cases ● Low ✓ Resolved

If user.workingAmount is bigger than pool.workingSupply, the pool.workingSupply value will not be updated after withdrawal.

```
    function emergencyWithdraw(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        pool.lpToken.safeTransfer(address(msg.sender), user.amount);
        if (pool.workingSupply >= user.workingAmount) {
            pool.workingSupply = pool.workingSupply - user.workingAmount;
        }
        user.amount = 0;
        user.workingAmount = 0;
        user.rewardDebt = 0;

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

The possibility of such a case is minimal. Nevertheless, we recommend fixing the issue.

C5. RewardPool

Overview

The RewardPool contract is used to deposit staked users' Crona tokens in the zero pool of the MasterChefV1 contract. Also, it allows distributing rewards from the MasterChefV1 contract between users.

Issues

C5-01 Withrawals can be blocked if the owner transfers xCrona tokens ● Medium ✓ Resolved

When the RewardPool contract enters staking in the MasterChefV1 contract, it receives xCrona tokens. These tokens are needed to leave staking.

If the owner account gets compromised or acts maliciously it can withdraw xCrona tokens from the RewardPool contract utilizing the function `inCaseTokensGetStuck()`. This will block all withdrawals.

```
    function inCaseTokensGetStuck(address _token) external onlyOwner {
        require(_token != address(token), "Token cannot be same as deposit token");

        uint256 amount = IERC20(_token).balanceOf(address(this));
        IERC20(_token).safeTransfer(msg.sender, amount);
    }
```

Recommendation

Block withdrawals of the xCrona tokens in the `inCaseTokensGetStuck()` function.

C5-02 Gas optimisations ● Low ✓ Resolved

The state variable `operator` L36 can be declared as `immutable` to save gas.

Parameters `lastDepositedTime`, `cronaAtLastUserAction`, `lastUserActionTime` in the `UserInfo` structure are never read in the contract and could be removed.

```
    struct UserInfo {
        uint256 principals;
        uint256 shares; // number of shares for a user
        uint256 lastDepositedTime; // keeps track of deposited time for potential penalty
        uint256 cronaAtLastUserAction; // keeps track of crona deposited at the last user
action
        uint256 lastUserActionTime; // keeps track of the last user action time
    }
```

C6. MasterChefTool

Overview

The contract allows to manage the pools of the contract MasterChefV1 and synchronize the reward amount per second for the MasterChefV2 contract.

No issues were found.

5. Conclusion

3 medium severity issues were found. The code was provided without documentation and tests. We strongly recommend writing unit tests to have extensive coverage of the codebase minimize the possibility of bugs and ensure that everything works as expected.

The contracts are highly dependent on the owner's account. Users interacting with the contracts must trust the owner.

This audit includes recommendations on the code improving 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.
- **Info.** Issues that do not impact the contract operation. Usually, info 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

@hashexbot

blog.hashex.org

linkedin

github

twitter

HashEx
Blockchain Security