#HashEx
BLOCKCHAIN SECURITY

CERUS Reward Distribution

smart contracts
final audit report

April 2023

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). HashEx has exclusive rights to publish the results of this audit on company's web and social sites.

2. Overview

HashEx was commissioned by the CERUS team to perform an audit of their smart contract. The audit was conducted between 2023-04-09 and 2023-04-13.

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 https://github.com/CERUS-Nodes/CERUS-Contracts/CERUS-NFT-REWARD-DISTRIBUTION/CERUSNFTRewardDistribution.sol and was audited after the commit ed53946.

2.1 Summary

Project name	CERUS Reward Distribution
URL	https://cerusnodes.io
Platform	Metis
Language	Solidity

2.2 Contracts

Name	Address
CERUSNFTRewardDistribution	

3. Found issues

Critical	1 (8%)
Medium	3 (25%)
Low	2 (17%)
Info	6 (50%)

C7d. CERUSNFTRewardDistribution

ID	Severity	Title	Status
C7dI41	Critical	Reward distribution calculation errors	Open
C7dI40	Medium	Block gas limit problem	Open
C7dI44	Medium	Inconsistent distribution model	Open
C7dI46	Medium	Inconsistent rewards calculation for collection	Open
C7dI3d	Low	Gas optimizations	Open
C7dI43	Low	Stuck nextReward() getter	Open
C7dI47	Info	Compile warnings	Open
C7dI3e	Info	Typos	Open
C7dI3f	Info	Using of IERC721Enumerable by default	Open
C7dI48	Info	Default variable visibility	Open

C7dI42	Info	Possible reentrancy in METIS token	Open
C7dI45	Info	Inconsistent comment	Open

4. Contracts

C7d. CERUSNFTRewardDistribution

Overview

A staking contract to hold owner-selected NFT collections in exchange for external rewards in the form of either METIS or CERUS tokens, or both of them.

Issues

C7dI41 Reward distribution calculation errors

Critical Open

The `_distribute()` function is called with the manual distribution or during the withdrawals. Pending reward amounts in METIS and/or CERUS tokens are shared between all eligible users or tokens. However, actual calculations include dividing and multiplying by the same number, meaning the first user will take the full pending reward, and following users will receive the same reward to be transferred from the contract's reward balance.

```
function _distribute(address user) private {
    ...
    PendingReward storage reward = pendingRewards[i];
    address collection = reward.collection;
    uint256 numberOfUserTokens = users[user]
                    .tokens[collection]
                    .length;
    uint256 perTokenShareMetis = reward.amountMetis /
                    numberOfUserTokens;
    uint256 perTokenShareCerus = reward.amountCerus /
                    numberOfUserTokens;
    uint256 totalRewardMetis = perTokenShareMetis *
                    numberOfUserTokens;
    uint256 totalRewardCerus = perTokenShareCerus *
                    numberOfUserTokens;
    users[user].claimableMetis += totalRewardMetis;
    users[user].claimableCerus += totalRewardCerus;
```

```
        ...
    }
```

Recommendation

We recommend refactoring the reward distribution, adding documentation, and increasing test coverage.

C7dI40 Block gas limit problem

Medium Open

User length is practically unlimited, so the `addReward()` function may become inaccessible due to block gas limit exceedance. Reward distribution and claiming may suffer from this problem too.

```
    function addReward(
        address collection,
        uint256 amountMetis,
        uint256 amountCerus
    ) external onlyRole(REWARDER_ROLE) {
        ...
        for (uint256 i = 0; i < _userAddresses.length; i++) {
            address userAddress = _userAddresses[i];
            if (tokensOf(userAddress, collection).length > 0) {
                usersWithCollectionTokens = _addAddressToArray(
                    usersWithCollectionTokens,
                    userAddress
                );
            }
        }
        ...
    }
```

Recommendation

Consider code optimization to avoid massive array iterations. Take a look at OpenZeppelin's EnumerableSet.

C7dI44 Inconsistent distribution model

Medium Open

The `pendingRewardsUser()` function returns pending user reward, calculated by dividing by the number of eligible users, users.length. At the same time, actual distribution takes place in the `_distribute()` function, which includes dividing by the number of tokens. Such discrepancy must be justified by the documentation.

```
function pendingRewardsUser(address user)
    public
    view
    returns (uint256 unreleasedAmountMetis, uint256 unreleasedAmountCerus)
{
    for (uint256 i = 0; i < pendingRewards.length; i++) {
        if (hasReward) {
            unreleasedAmountMetis +=
                pendingRewards[i].amountMetis /
                pendingRewards[i].users.length;
            unreleasedAmountCerus +=
                pendingRewards[i].amountCerus /
                pendingRewards[i].users.length;
        }
    }
}

function _distribute(address user) private {
    ...
    PendingReward storage reward = pendingRewards[i];
    address collection = reward.collection;
    uint256 numberOfUserTokens = users[user]
                    .tokens[collection]
                    .length;
    uint256 perTokenShareMetis = reward.amountMetis /
                    numberOfUserTokens;
    uint256 perTokenShareCerus = reward.amountCerus /
```

```
                    numberOfUserTokens;
        uint256 totalRewardMetis = perTokenShareMetis *
                    numberOfUserTokens;
        uint256 totalRewardCerus = perTokenShareCerus *
                    numberOfUserTokens;
        users[user].claimableMetis += totalRewardMetis;
        users[user].claimableCerus += totalRewardCerus;
        ...
    }
```

Recommendation

Consider unifying the distribution model, adding documentation, and increasing test coverage.

C7dI46 Inconsistent rewards calculation for collection

Medium Open

The function `pendingRewardCollection()` uses a current number of tokens deposited into the contract. A calculated rewards value will change with every deposit of the collection's NFT token to the contract. This will lead to inconsistent function outputs.

```
    function pendingRewardCollection(address user, address collection)
        external
        view
        returns (uint256 amountMetis, uint256 amountCerus)
    {
        for (uint256 i = 0; i < pendingRewards.length; i++) {
            if (pendingRewards[i].collection == collection) {
                if (_addressIsInArray(pendingRewards[i].users, user)) {
                    uint256 numberOfTokens = IERC721Enumerable(collection)
                        .balanceOf(address(this));
                    uint256 perTokenShareMetis = pendingRewards[i].amountMetis /
                        numberOfTokens;
                    uint256 perTokenShareCerus = pendingRewards[i].amountCerus /
                        numberOfTokens;

                    amountMetis =
                        users[user].tokens[collection].length *
```

```
                        perTokenShareMetis;
                    amountCerus =
                        users[user].tokens[collection].length *
                        perTokenShareCerus;
                }
                break;
            }
        }
    }
```

Recommendation

Fix the number of users when a reward is distributed and calculate rewards based on this value.

C7dI3d Gas optimizations

Low Open

1. The variables `metis` and `treasury` should be declared as constants.

2. Multiple reads from storage of the `_userAddresses.length` variable in the `addReward()`, `distributePendingRewardToAllUsers()`, and `totalBalance()` functions.

3. Multiple reads from storage of the `_collections.length` variable in the `depositAll()` and `_withdrawAll()` functions.

4. Triple read from storage of the `pendingRewards.length` variable in the `pendingRewardsUser()` and `_removeRewardAtIndex()` functions.

5. Safety checks of `collection`/`tokenId` should be moved from the `_withdrawCollection()`/ `_withdraw()` to external functions, since `_withdrawAll()` always ensures the correctness of the collection address and `_withdrawCollection()` - of the token ID.

6. The UintSet from OpenZeppelin's EnumerableSet library should be used for `users[].tokens[]` in order to reduce gas costs of array iteration.

7. The AddressSet from OpenZeppelin's EnumerableSet library should be used for `_collections[]` and `pendingRewards.users[]` in order to reduce gas costs of array iteration.

8. `PendingReward.time` variable is stored but never used.

C7dI43 Stuck nextReward() getter

Low Open

The `nextReward()` view function returns the next pending reward parameters. It may become stuck since it returns the first element of the `pendingRewards[]` array, which is updated only when the previous `pendingRewards[0]` runs out of unclaimed users. If some users decide not to claim their rewards (or lose access to their accounts), `nextReward()` would become useless.

C7dI47 Compile warnings

Info Open

There are compilation warnings of unused arguments in the `onERC721Received()` function. Consider commenting out function parameters to suppress warnings.

```
function onERC721Received(
    address /*operator*/,
    address /*from*/,
    uint256 /*tokenId*/,
    bytes memory /*data*/
) public view returns (bytes4) {
    ...
}
```

C7dI3e Typos

Info Open

Typos reduce the code's readability. Typos in 'treausry', 'rewawrd', 'pedngin'.

C7dI3f Using of IERC721Enumerable by default

Info Open

The `tokenOfOwnerByIndex()` method is not a part of the ERC721 standard, so transactions may fail silently, complicating the debugging.

```
function depositCollection(address collection) public {
    ...
    uint256 tokenId = IERC721Enumerable(collection).tokenOfOwnerByIndex(
        msg.sender,
        0
    );
    ...
}
```

Recommendation

We recommend adding a security check to the `addCollection()` to ensure IERC721Enumerable support.

C7dI48 Default variable visibility

Info Open

The variable `isCerusSet` has a default `internal` visibility which means it can be accessed by other contracts within the same inheritance tree. We recommend setting explicit visibility of all variables to avoid unintended access or modification by other contracts and to increase the security of the smart contract.

C7dI42 Possible reentrancy in METIS token

Info Open

There's a possible reentrancy point in the `_claim()` function, which can be triggered if the METIS token would be updated to have a transfer hook.

```
function _claim(address user) private {
    ...
    uint256 claimableMetis = users[user].claimableMetis;
    uint256 claimableCerus = users[user].claimableCerus;
    ...
    users[user].claimableMetis = 0;
    IERC20(metis).transfer(user, claimableMetis);
    users[user].claimableCerus = 0;
    IERC20(cerus).transfer(user, claimableCerus);
    ...
```

```
    }
```

C7dI45 Inconsistent comment

Info Open

The `onERC721Received()` function contains an irrelevant comment on its behavior, stating that the contract verifies the incoming token ownership, which it does not.

```
    function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes memory data
    ) public view returns (bytes4) {
        // Verify that the token was transferred by the token owner
        /// @notice since we use multiple collections we check if it is registered instead
of
        // checking against a certain address
        require(_addressIsInArray(_collections, msg.sender));
        ...
        return IERC721Receiver.onERC721Received.selector;
    }
```

5. Conclusion

1 critical, 3 medium, 2 low severity issues were found during the audit. No issues were resolved in the update.

The reviewed contract is highly dependent on the owner’s account. Users using the project have to trust the owner and that the owner's account is properly secured.

This audit includes recommendations on code improvement and the prevention of 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. Issue status description

Resolved. The issue has been completely fixed.

Partially fixed. Parts of the issue have been fixed but the issue is not completely resolved.

Acknowledged. The team has been notified of the issue, no action has been taken.

Open. The issue remains unresolved.

Appendix C. 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