HashEx

BLOCKCHAIN SECURITY

CERUS

smart contracts
final audit report

January 2023

hashex.org

contact@hashex.org

Contents

1. Disclaimer

2. Overview

HashEx was commissioned by the CERUS team to perform an audit of their smart contract. The audit was conducted between 2023-01-03 and 2023-01-07.

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 in the https://github.com/CERUS-Nodes/CERUS-Contracts GitHub repository and was audited after the commit aa089c2.

Update. The CERUS team has responded to this report, the updated code is located in the same repository after the 63372d6 commit.

2.1 Summary

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

2.2 Contracts

Name	Address
CERUSToken	
CERUSVesting	

3. Found issues

5
Total issues

●	Critical	1 (20%)
●	Medium	1 (20%)
●	Low	1 (20%)
●	Info	2 (40%)

Cc3. CERUSToken

ID	Severity	Title	Status
Cc3Iaa	● Critical	Tokens can be burned from anyone	✓ Resolved
Cc3I7b	● Info	Tax distributor can be changed	✓ Resolved
Cc3Id6	● Info	Discrepancy in documentation and code behaviour	✓ Resolved

Ccf. CERUSVesting

ID	Severity	Title	Status
Ccfl6a	● Medium	Claim is not guaranteed	✓ Acknowledged
Ccfl68	● Low	Gas optimizations	✓ Partially fixed

4. Contracts

Cc3. CERUSToken

Overview

An ERC-20 standard implementation with taxable transfers. Tax is applied with several conditions: the recipient has to be a contract with an implemented `token1()` method, tax recipient and tax percent must have non-zero values, and the spender must be tax-applicable, i.e. not included in the whitelist. The tax percent is updatable with a default value of 5%; the updated value can't exceed 5%.

Issues

Cc3Iaa Tokens can be burned from anyone ● Critical ✓ Resolved

In the updated contract the `burn()` and `burnFrom()` functions were introduced. The latter can be used by an arbitrary caller to burn up to 20% of total supply from any account regardless the allowance.

```
function burnFrom(address _account, uint256 _amount) public override {
    require(totalBurned() + _amount <= MAX_BURNED, "burn: max burn!");
    _burn(_account, _amount);
}
```

Recommendation

Reduce current allowance for caller or remove the `burnFrom()` function.

Cc3I7b Tax distributor can be changed ● Info ✓ Resolved

A tax from sales is sent to the distributor contract. Users should be aware that the distributor contract address can be changed as well as the distribution itself.

Cc3Id6 Discrepancy in documentation and code behaviour ● Info ✓ Resolved

The documentation states:

Given that it [TaxDistributor] *is not set, when transfer is called, then it should revert with an error.*

The actual behavior is when the TaxDistributor address is not set, the tax won't be taken.

Recommendation

Update documentation to conform to the actual behavior.

Ccf. CERUSVesting

Overview

A sale contract with a whitelist and fixed price. Purchased CERUS tokens are vested within a fixed number of claimable periods of 4 weeks long each.

Issues

Ccfl6a Claim is not guaranteed ● Medium ✓ Acknowledged

The total amount of sale tokens is written during the contract initialization but the actual balance is not checked. Thus, the `claim()` function may constantly fail for some or even every user since the contracts will try to transfer nonexistent tokens.

```
function initialize(
    ...
    uint256 _totalCerus
) external {
    ...
    totalCerus = _totalCerus;
}

function claim() external nonReentrant {
    ...
    IERC20(cerusToken).transfer(msg.sender, _claimableAmount);
}
```

Recommendation

Ensure the `totalCerus` amount during the initialization either by checking the balance of the contract or by transferring them from the initializer.

Ccfl68 Gas optimizations

● Low — Partially fixed

1. The variables `treasury`, `investorInfo[].depositedAmount`, `totalVested`, and `cerusPrice` in the `deposit()` function are read from storage multiple times, which can be avoided by using local variables.
2. The variables `cerusToken`, `investorInfo[].claimedEpochs`, `investorInfo[].vestedAmount`, `totalEpochs`, and `claimStart` in the `claim()` function are read from storage multiple times, which can be avoided by using local variables.
3. The variable `isPublicSale` in the `withdrawUnsold()` function is read from storage multiple times, which can be avoided by using local variables.
4. The variable `claimStart` in the `currentEpoch()` function is read from storage multiple times, which can be avoided by using local variables.
5. In the case of a long whitelist, a Merkle tree pattern can be used to save gas.
6. The variables `initialized`, `totalEpochs`, and `epochLength` in the `currentEpoch()` function are read from storage multiple times, which can be avoided by using local variables. This issue was introduced in the code update.

5. Conclusion

1 critical, 1 medium, 1 low severity issues were found during the audit. 1 critical issue was 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.
- **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