#HashEx
BLOCKCHAIN SECURITY

Avata AVAXStaking

smart contracts final audit report

June 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 Avata team to perform an audit of their smart contract. The audit was conducted between 17/06/2022 and 20/06/2022.

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 GitHub repository @AVATA-Network/avata-contracts after the commit 6b7882c.

Update: the Avata team has responded to this report.

The updated code is located in the same repository after the commit 2e3d11a.

2.1 Summary

Project name	Avata AVAXStaking
URL	https://avata.network
Platform	Avalanche Network
Language	Solidity

2.2 Contracts

Name	Address
AVAXStaking	

3. Found issues

Critical	1 (10%)
High	2 (20%)
Low	3 (30%)
Info	4 (40%)

C1. AVAXStaking

ID	Severity	Title	Status
C1-01	Critical	Blocked rewards	Resolved
C1-02	High	Wrong calculations	Resolved
C1-03	High	Deposit fee isn't limited	Resolved
C1-04	Low	Gas optimizations	Partially fixed
C1-05	Low	Mass update is optional	Resolved
C1-06	Low	ACC_REWARD_PER_SHARE_PRECISION value	Resolved
C1-07	Info	Native currency transfers	Resolved
C1-08	Info	Possible reentrancy	Resolved
C1-09	Info	Inconsistent comment	Resolved
C1-10	Info	User's totalRewarded may be misleading	Resolved

4. Contracts

C1. AVAXStaking

Overview

Staking contract with fixed staking periods and WAVAX rewards.

Issues

C1-01 Blocked rewards

Critical Resolved

Every call to `updatePool(uint256 _pid)` for an arbitrary pid pool calculates the pool's part of reward and updates the `lastRewardBalance` variable. Since `lastRewardBalance` is shared across all the pools, part of the reward token balance would be locked forever. Assuming only 2 pools with the same allocation:

- let `lastRewardBalance = 0`, `wavax.balanceOf(AVAXStaking) = 100`
- call `updatePool(0)`: `poolInfo[0].accTokenPerShare += (100 * 1/2 * ACC_REWARD_PER_SHARE_PRECISION) / depositTokenSupply`, `lastRewardBalance = 100`
- call `updatePool(1)`: `poolInfo[0].accTokenPerShare += 0`

Moreover, if any user of the first pool collects their rewards, decreasing the contract's balance, then any `updatePool()` call would be reverted due to `currentRewardBalance < lastRewardBalance`.

```
    function updatePool(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];

        uint256 depositTokenSupply = pool.totalDeposits;
        uint256 currentRewardBalance = wavax.balanceOf(address(this));

        if (depositTokenSupply == 0 || currentRewardBalance == lastRewardBalance) {
            return;
        }
```

```
        uint256 _accruedReward = currentRewardBalance - lastRewardBalance;

        pool.accTokenPerShare = pool.accTokenPerShare + (((_accruedReward *
pool.allocPoint) / totalAllocPoint) * ACC_REWARD_PER_SHARE_PRECISION) /
depositTokenSupply;

        lastRewardBalance = currentRewardBalance;
    }
```

Recommendation

We strongly recommend redesigning the rewards calculation with additional testing.

C1-02 Wrong calculations

High Resolved

Typo in `_safeTransferReward()` calculations in L439 may cause a locked WAVAX rewards.

```
    function _safeTransferReward(address _to, uint256 _amount) internal {
        uint256 wavaxBalance = wavax.balanceOf(address(this));

        if (_amount > wavaxBalance) {
            lastRewardBalance = lastRewardBalance - wavaxBalance;
            paidOut += wavaxBalance;

            wavax.withdraw(wavaxBalance);
            payable(_to).transfer(wavaxBalance);
        } else {
            lastRewardBalance = lastRewardBalance - wavaxBalance;
            paidOut += _amount;

            wavax.withdraw(_amount);
            payable(_to).transfer(_amount);
        }
    }
```

Recommendation

Fix the typo and increase testing coverage:

```
else {
    lastRewardBalance = lastRewardBalance - _amount;
    (...)
}
```

C1-03 Deposit fee isn't limited

High Resolved

The owner is able to add pools with any arbitrary `depositFeePercent` parameter. Setting it over the `DEPOSIT_FEE_PRECISION` value would cause a completely malfunctioning pool. The `setDepositFee()` function allows the owner to update the pool deposit fee to any value up to 100%. Users must pay attention and check the deposit fee prior to the deposit act.

Recommendation

Limit the maximum fee value to significantly less than 100% and/or transfer the ownership to a Timelock-like contract.

C1-04 Gas optimizations

Low Partially fixed

a. Multiple reads from storage: `user.stakesCount` in L133-158, `stake.id` in L162, `stake.amount` in L182, L190, L217-228, `pool.accTokenPerShare` in L223-228, `ACC_REWARD_PER_SHARE_PRECISION` in L182-190, L223-228, `lastRewardBalance` in L363-364, L412-416.

b. Excessive calculations in L190 (always zero), L228.

c. `ACC_REWARD_PER_SHARE_PRECISION` should be declared constant.

d. The `initialize()`, `add()`, `deposit()`, `withdraw()`, `collect()`, `setAllocation()`, `userStakesCount()`, `getPool()`, `getUserStakes()`, `getUserStake()`, `getUserStakeIds()`, `deposited()`, `totalDeposited()`, `pending()`, `totalPending()` functions can be declared as

external to save gas.

C1-05 Mass update is optional

Low Resolved

An optional boolean flag for `massUpdatePools()` in `add()` and `setAllocation()` functions allows the owner to redistribute unclaimed rewards, e.g. disabling the pool's allocation. Freshly added pools receive historical rewards after the `lastRewardBalance` timestamp. We recommend mandatory enabling mass update with every change in allocation scheme, taking in mind possible gas limit problems in case of an extremely large `poolInfo[]` array.

C1-06 ACC_REWARD_PER_SHARE_PRECISION value

Low Resolved

`ACC_REWARD_PER_SHARE_PRECISION` float multiplier has the immutable value of `1e36`, making it plausible to overflow in rewardDebt calculations, see L151, L182, L368. The problem is prominent if `pool.depositToken.decimals()` is significantly less than `wavax.decimals()`.

```
    function updatePool(uint256 _pid) public {
        (...)
        pool.accTokenPerShare += _accruedReward * pool.allocPoint / totalAllocPoint *
ACC_REWARD_PER_SHARE_PRECISION / depositTokenSupply;
    }
```

We recommend decreasing the multiplier value in order to support wider range of staking tokens.

C1-07 Native currency transfers

Info Resolved

Using the `transfer()` method is discouraged in favor of `call()` with gas customization and/or reentrancy protection.

C1-08 Possible reentrancy

Info Resolved

Reentrancy is possible in the `withdraw()` function if the `pool.depositToken` token contains transfer hooks. The owner must avoid adding pools with such tokens.

C1-09 Inconsistent comment

Info Resolved

The comment says that depositFeePercent is limited both from below and above, but the owner is able to set any value, see Deposit fee isn't a limited issue. A minimum value check isn't implemented either.

```
uint256 depositFeePercent; // Percent of deposit fee, must be >= depositFeePrecision / 100
and less than depositFeePrecision
```

C1-10 User's totalRewarded may be misleading

Info Resolved

In the `collect()` function, there's an update of the `user.totalRewarded` variable, but the actual transferred amount during the `_safeTransferReward()` may differ from the pendingAmount:

```
    function collect(uint256 _pid, uint256 _stakeId) public whenNotPaused {
        (...)
        _safeTransferReward(msg.sender, pendingAmount);
        user.totalRewarded = user.totalRewarded + pendingAmount;
        (...)
    }

    function _safeTransferReward(address _to, uint256 _amount) internal {
        uint256 wavaxBalance = wavax.balanceOf(address(this));
        if (_amount > wavaxBalance) {
            (...)
            payable(_to).transfer(wavaxBalance);
        } else {
            (...)
            payable(_to).transfer(_amount);
        }
```

```
    }
```

5. Conclusion

1 critical, 2 high, and 3 medium severity issues were found, all of them were fixed with the code update.

This audit includes recommendations on code improvement and the prevention of potential attacks. We strongly suggest adding unit and functional tests with at least 80% coverage.

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