Avata Farming

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 27/05/2022 and 30/05/2022.

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 901169a. Update: the Avata team has responded to this report. The updated code is located in the GitHub repository after the commit 9e9ac73. The contracts are deployed on Avalanche C-Chain network at 0x05991ADb5024C30Fe4D72Da3544D03Ad1Cb33c80.

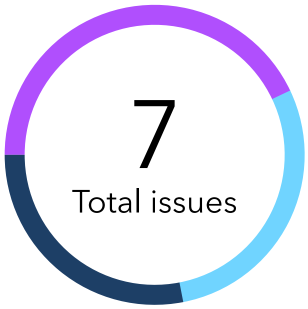
2.1 Summary

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

2.2 Contracts

Name	Address
Farming	0x05991ADb5024C30Fe4D72Da3544D03Ad1Cb33c80
FarmingFactory	0x05991ADb5024C30Fe4D72Da3544D03Ad1Cb33c80
OwnableTimelock	0x05991ADb5024C30Fe4D72Da3544D03Ad1Cb33c80

3. Found issues



Medium	3 (43%)
Low	2 (29%)
Info	2 (28%)

Cc2. Farming

ID	Severity	Title	Status
Cc2l96	Medium	Bonus token rewards may be lost	Acknowledged
Cc2l97	Low	Gas optimizations	Partially fixed
Cc2l98	Info	Possible reentrancy	Resolved

Cc3. FarmingFactory

ID	Severity	Title	Status
Cc3l99	Medium	Rewards distribution	Acknowledged
Cc3l9a	Low	Gas optimizations	Resolved

Cc8. OwnableTimelock

ID	Severity	Title	Status
Cc8Iac	Medium	Delay is zero	Resolved
Cc8Iab	Info	Owner accepts ownership changes	Acknowledged

4. Contracts

Cc2. Farming

Overview

A single pool Farm is to be deployed and administered by the FarmingFactory contract. Can have a single reward type of Avata token or bonus tokens specified on the contract creation, or both of them. The amount of rewards is controlled by the factory.

Issues

Cc2I96 Bonus token rewards may be lost — Medium — Acknowledged

The `_safeTransfer()` function may cause users to lose their rewards in bonus tokens if the Farming contract balance does not have enough funds.

```
function _safeTransfer(
    IERC20Upgradeable token_,
    address to_,
    uint256 amount_
) internal {
    uint256 balance = token_.balanceOf(address(this));

    if (amount_ > balance) {
        token_.safeTransfer(to_, balance);
    } else {
        token_.safeTransfer(to_, amount_);
    }
}
```

Recommendation

The rewards that can not be paid at the moment could be stored in the mapping `unpaidRewards[_userAddress]`, which should be accounted in the `pending()` calculations.

Cc2I97 Gas optimizations

Low · Partially fixed

a. `ATPS_PRECISION` variable should be declared constant.

b. Multiple reads from storage of `user.amount` in `deposit()` and `withdraw()` functions.

c. Multiple reads from storage of `lastRewardTimestamp` in `updatePool()` function.

d. `deposit()`, `withdraw()`, and `emergencyWithdraw()` functions could be declared external.

Cc2I98 Possible reentrancy

Info · Resolved

A reentrancy attack is possible in `emergencyWithdraw()` and in `deposit()`/`withdraw()` functions. The owner must avoid adding `lpToken` and `bonusToken` tokens with hooks in transfers.

Cc3. FarmingFactory

Overview

Factory contract allows the owner to create Farming contracts and update their reward allocation scheme. Avata token rewards are minted via the DistributionV2 external contract.

Issues

Cc3I99 Rewards distribution

Medium · Acknowledged

Rewards distribution may be updated (or disabled) for rarely updated Farmings by changing the allocation scheme (via `deployFarming()` or `updateFarmingAllocationPoint()`).

```
Farming:
   function updatePool() public {
      (...)
      reward = farmingFactory.countRewardAmount(
         lastRewardTimestamp,
         block.timestamp,
         address(this));
      (...)
   }
FarmingFactory:
   function countRewardAmount(...) external view returns (uint256) {
      (...)
      farmingReward = reward.mul(farmingAllocationPoint[farmingAddress])
         .div(totalFarmingAllocationPoint);
      (...)
   }

   function updateFarmingAllocationPoint(...) external onlyOwner {
      (...)
      totalFarmingAllocationPoint = totalFarmingAllocationPoint.sub(farmingAllocationPoint[
farmingAddress_]).add(allocationPoint_);
      farmingAllocationPoint[farmingAddress_] = allocationPoint_;
      (...)
   }

   function deployFarming(...) external onlyOwner returns (address) {
      (...)
      farmingAllocationPoint[clone] = allocationPoint_;
      totalFarmingAllocationPoint = totalFarmingAllocationPoint.add(allocationPoint_);
      (...)
    }
```

Also, the distribution may give out more rewards than necessary if the allocation scheme changes. For example, there are two pools: one is allowed to receive 100% of the rewards, and the second is 0%. By calling the `countRewardAmount()` function, the first pool takes its 100% of the rewards, after that, you can change the allocation scheme to the reverse, the second pool is allowed to receive 100% of the rewards. Thus, it will be possible to pick up 200% of the rewards, instead of 100%, and the emission scheme of the DistributionV2 contract may be disturbed.

Recommendation

Factory ownership should be transferred to a Timelock with >24h of minimum delay.

Cc3l9a Gas optimizations

Low | Resolved

a. The `avatToken` variable should be declared immutable.

b. Direct boolean comparison could be eliminated in the `deployFarming()` function.

Cc8. OwnableTimelock

Overview

The contract will change the address of the owner after a delay time. This contract was added to the scope with the code update.

Issues

Cc8Iac Delay is zero

Medium | Resolved

The `transferOwnershipDelay` state variable, which is responsible for the delay time in seconds, is zero. This will lead to the fact that the owners can be changed instantly.

Cc8Iab Owner accepts ownership changes

Info | Acknowledged

The `confirmTransferOwnership()` function is executed under the `onlyOwner()` modifier, but it will be more correct from the logic side if only the one to whom ownership rights are transferred can accept changes.

Recommendation

Set default value before mainnet deployment or add a setter function.

5. Conclusion

3 medium, 2 low, 2 info severity issues were found.1 medium, 1 low, and 1 informational severity issues have been resolved in the update. 1 low severity issue has been partially resolved.

The reviewed contracts are 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 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