HashEx
BLOCKCHAIN SECURITY

Avata

smart contracts final audit report

March 2022

hashex.org

contact@hashex.org

Contents

1. Disclaimer 3

2. Overview 4

3. Found issues 6

4. Contracts 8

5. Conclusion 13

Appendix A. Issues severity classification 14

Appendix B. List of examined issue types 15

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 06/03/2022 and 10/03/2022.

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 @i-link-pro-team/avata-network GitHub repository and was audited after the commit 79c3d0c.

Update: the Avata team has responded to this report. The updated code is located in the GitHub repository after commit bf23c07.

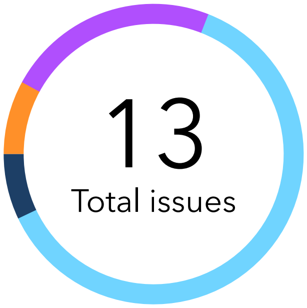
2.1 Summary

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

2.2 Contracts

Name	Address
SaleAvat	
Distribution	

3. Found issues



●	High	1 (8%)
●	Medium	3 (23%)
●	Low	8 (62%)
●	Info	1 (7%)

C1. SaleAvat

ID	Severity	Title	Status
C1-01	● High	Owner can withdraw all tokens	✓ Resolved
C1-02	● Medium	Possible discrepancy in periods config	✓ Resolved
C1-03	● Medium	Period update may be incorrect	✓ Resolved
C1-04	● Low	Lack of events/ events are not indexed	✓ Resolved
C1-05	● Low	Gas optimization	✓ Partially fixed
C1-06	● Low	Floating pragma	✓ Resolved
C1-07	● Info	Not accurate documentation	✓ Resolved

C2. Distribution

ID	Severity	Title	Status
C2-01	● Medium	Exaggerated owner's rights	✓ Acknowledged
C2-02	● Low	Gas optimization	✓ Resolved
C2-03	● Low	Unnecessary require	✓ Resolved
C2-04	● Low	Lack of input address check in the constructor	✓ Resolved
C2-05	● Low	No reason message in require statement	✓ Resolved
C2-06	● Low	Floating pragma	✓ Resolved

4. Contracts

C1. SaleAvat

Overview

The contract allows the owner to create a sale of an ERC-20 token for different ERC-20 tokens. Every user's tokens have its own rate - the price for the sold token.

Issues

C1-01 Owner can withdraw all tokens ● High ✓ Resolved

The contract owner has the ability to withdraw all `_withdrawToken` and `_depositTokens`leaving nothing to users.

Users deposit `_depositTokens` to contract and wait until for the `distributionDate` to be unlocked to claim `_withdrawToken`. At the same time the contract owner can call `collectToken()` and `leftover()` functions to withdraw all tokens from the contract.

Recommendation

The admin and the owner must be `Timelock` contracts with a minimum delay of at least 24 hours. This won't stop the admin and the owner from possible right abuses but it will help users to be informed about upcoming changes.

C1-02 Possible discrepancy in periods config ● Medium ✓ Resolved

The `_distributionDates` should be also deleted after L80, otherwise the timestamps in `_distributionDates` and `_distributionPeriods` will be different.

C1-03 Period update may be incorrect ● Medium ✓ Resolved

If `createDistributionPeriod()` is called in order to update current period, but `startSale` is set to zero in the previous call, `_distributionPeriods` won't be updated correctly. New periods will be pushed to existing ones.

Recommendation

Require `startSale_` to not equal zero, while calling the `createDistributionPeriod()` function.

C1-04 Lack of events/ events are not indexed ● Low ✓ Resolved

a. No events are emitted in `updateTokenRates()`, `leftover()`, `collectToken()`, `claimToken()`, `buyToken()` functions.

b. Events don't have indexed fields. Indexation allows searching events by the topic after they are emitted.

C1-05 Gas optimization ● Low ✓ Partially fixed

a. The state variable `signatory` can be declared as `immutable` to save gas.

b. The library ECDSA.sol (L7) is never used. Removing the library might decrease contract size and its deployment costs.

c. The comparison of the parameters `startSale_` and `endSale_` of the functions `createDistributionPeriod()` should be executed before assignment at L83-84.

d. The conversion of `msg.sender` to type `address` at L128, 132, 133, 135, 141 are redundant.

e. There is no need to cast the address `ownerAddress_` to payable type at L231, L242, because no native token is sent to this address.

f. Since the arguments of the function `updateTokenRates()` are read-only, they can be declared as calldata instead of memory to save gas.

g. Since the mapping nonces is updated only once, the L172 is redundant. Consequently the function checkSig() can be deleted and the function permitBySig() can be declared as public.

h. The _distributionDates array completely duplicates the values from _distributionPeriods . Thus, you can refuse to use the _distributionDates array.

i. The mapping _depositTokens is used to store same values as keys.

C1-06 Floating pragma ● Low ✓ Resolved

A general recommendation is that pragma should be fixed to the version that you are intending to deploy your contracts with. This helps to avoid deploying using an outdated compiler version and shields from possible bugs in future solidity releases.

C1-07 Not accurate documentation ● Info ✓ Resolved

a. The parameter startSale_ (L73) of the function createDistributionPeriod() has incrorrect description in documentation.

b. The parameter precision_ (L75) of the function createDistributionPeriod() is never used and can be removed from documentation.

C2. Distribution

Overview

The contract allows the owner to create distributions of tokens for specific users. Distribution is carried out by minting tokens at a given point in time.

Issues

C2-01 Exaggerated owner's rights ● Medium ✓ Acknowledged

a. The owner can deactivate users' distributions with the `removeDistributions()` function, which disables the token claiming function for deactivated users.

b. The owner can mint distributed tokens using the `mintTokens()` function. Since the distributed token has cap, the owner must control that after the mint pending distributions won't exceed the cap. Otherwise, users can loose their funds as `claimToken()` function will be reverted on `_token.mint()` stage.

c. The owner can change vesting time periods while they are in progress. E.g. it can shift them on a long period of time.

Recommendation

The admin and the owner must be `Timelock` contracts with a minimum delay of at least 24 hours. This won't stop the admin or the owner from possible right abuses but it will help users to be informed about upcoming changes.

Update

The Avata team added timelock period for `mintToken()` function. Now the function is available when`timeLockPeriod` seconds pass after the contract deployment. It should be noted that this approach doesn't save from the stated problem and of the owner's account is compromised, the attacker still can mint tokens that were meant to be distributed or extend vesting periods.

C2-02 Gas optimization ● Low ✓ Resolved

a. The functions `setDistribution()`, `removeDistributions()`, `claimTokens()`, `getDistribution()`, `mintTokens()` can be declared as `external` to save gas.

b. Since the argument `distributionAddresses` of the functions `setDistribution()`, `removeDistributions()` is read-only, it can be declared as `calldata` instead of `memory` to save gas.

c. There is no need to converse the address `to` to payable type at L262, because no native token is sent to this address.

C2-03 Unnecessary require ● Low ✓ Resolved

Since all periods of distributions are ordered and previous periods already checked at L176, the require statement at L204 is unnecessary.

C2-04 Lack of input address check in the constructor ● Low ✓ Resolved

There is no non-zero check for `admin_` argument in the constructor.

C2-05 No reason message in require statement ● Low ✓ Resolved

There is no reason message in the `require` statement at L54.

C2-06 Floating pragma ● Low ✓ Resolved

A general recommendation is that pragma should be fixed to the version that you are intending to deploy your contracts with. This helps to avoid deploying using an outdated compiler version and shields from possible bugs in future solidity releases.

5. Conclusion

1 high and 3 medium severity issues were found during the audit. 1 high and 2 medium issues were resolved in the update.

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. We recommend putting the contract behind a Timelock and multisig wallet.

Also, we strongly recommend adding unit and integration tests for SaleAvat and Distribution contracts.

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

@hashex_manager

blog.hashex.org

linkedin

github

twitter

HashEx
BLOCKCHAIN SECURITY