#HashEx
BLOCKCHAIN SECURITY

Avata Sale

smart contracts final audit report

May 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 contracts. The audit was conducted between 19/04/2022 and 23/04/2022.

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 was provided in a zip. The MD5 sums of the files are:Sale.sol 85f8cfc23364c619a47b8d4115ad490d,SalesFactory.sol 1c903b4d3775210dd4e77b5dc119364a,SaleToken.sol f6975c867963c8d1d1745933c6748059.

Update: the Avata team has responded to this report. The updated code is located in the GitHub repository after the commit 23a8bfe.

The audited contracts: Sale.sol, SalesFactory.sol, SaleToken.sol.

2.1 Summary

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

2.2 Contracts

Name	Address
Sale	https://github.com/AVATA-Network/avata-contracts/blob/23a8bfeb82b1887348383ddd1d4e047ed9ca2c84/contracts/sale/Sale.sol
SalesFactory	https://github.com/AVATA-Network/avata-contracts/blob/23a8bfeb82b1887348383ddd1d4e047ed9ca2c84/contracts/sale/SalesFactory.sol
SaleToken	https://github.com/AVATA-Network/avata-contracts/blob/23a8bfeb82b1887348383ddd1d4e047ed9ca2c84/contracts/sale/SaleToken.sol
Multiple contracts	

3. Found issues

High	3 (21%)
Medium	1 (7%)
Low	6 (43%)
Info	4 (29%)

C1. Sale

ID	Severity	Title	Status
C1-01	High	Reducing the number of distribution periods	Resolved
C1-02	High	Exaggerated owner's rights	Resolved
C1-03	High	Double-Spend attack	Resolved
C1-04	Medium	Calling a function in the wrong place	Resolved
C1-05	Low	Multiplication after division	Resolved
C1-06	Low	Function input values are not checked	Acknowledged
C1-07	Low	Gas optimization	Resolved
C1-08	Info	Getting a chainId	Resolved
C1-09	Info	Incorrect documentation	Resolved

C2. SalesFactory

ID	Severity	Title	Status
C2-01	Low	Lacks a zero-check on constructor	Resolved
C2-02	Low	Gas optimization	Partially fixed

C3. SaleToken

ID	Severity	Title	Status
C3-01	Low	Gas optimization	Resolved
C3-02	Info	Lack of increase and decrease allowance functions	Acknowledged

C4. Multiple contracts

ID	Severity	Title	Status
C4-01	Info	Floating Pragma	Resolved

4. Contracts

C1. Sale

Overview

The contract conducts the sale of the ERC20 token. Users can buy a token using AVAT. After the sale, users can pick up the purchased tokens. The contract is deployed using SalesFactory. The purchase of tokens occurs by using a signed message at the address of the signature.

Issues

C1-01 Reducing the number of distribution periods

High Resolved

If the number of distribution periods is reduced via `updateDistributionPeriods()` function after some users had bought tokens with previous distribution periods length, they don't get the rewards in full.

```
function claimToken(uint256 periodId_) external onlyInitialized {
    (...)
    require(_distributionAddresses[participant][periodId_].isClaimed == false &&
_distributionAddresses[participant][periodId_].amount != 0, "claimToken: Participant does
not have funds for withdrawal");
    require(block.timestamp >= _distributionPeriods[periodId_].timestamp,
"Sale::claimToken: Claim date is not arrived");
    (...)
}
```

Recommendation

In the `updateDistributionPeriods()` function, you should to check the number of periods in the `distributionPeriods_` variable. if it is less than the current `distributionPeriods`, revet the transaction.

C1-02 Exaggerated owner's rights

High Resolved

a. The owner is able to infinitely delay the time of vesting periods by `updateDistributionPeriods()` functions;

```
function updateDistributionPeriods(DistributionPeriod[] calldata distributionPeriods_)
external onlyOwner onlyInitialized {
    (...)
    delete _distributionPeriods;
    for (uint256 i = 0; i < periodsLength; i++) {
        amountPercent = amountPercent.add(distributionPeriods_[i].percent);
        _distributionPeriods.push(distributionPeriods_[i]);
    }
    (...)
}
```

b. The owner can withdraw all reward tokens with `collectRaisedTokens()`.

```
function collectRaisedTokens(
    address depositAddress,
    address to_,
    uint256 amount_
    ) external onlyOwner {
    IERC20Decimals(depositAddress).safeTransfer(to_, amount_);
    emit AvatTokensCollected(to_, amount_);
}
```

Recommendation

a. We advise storing user's periods unlock timestamps for the moment he entered the vesting, so possible future terms changes won't affect him and the initial arrangements won't be violated.

b. Add a requirement `depositAddress` is not equal to `sellingTokenAddress`.

C1-03 Double-Spend attack

High Resolved

The `_permitLotteryAllocation()` function is vulnerable to Double-Spend attacks. A malefactor can unlimitedly substitute an already used signature with a different `to` parameter to get extra allocation points.

```
function takeLotteryAllocation(
        address to,
        address participant,
        uint256 tokenAmount,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external onlyInitialized {
    require(_permitLotteryAllocation(participant, tokenAmount, deadline, v, r, s) == true,
"Sale::takeLotteryAllocation: Invalid signature");
    (...)
    _mint(to, tokenAmount);
}
```

```
function _permitLotteryAllocation(
        address participant,
        uint256 tokenAmount,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) private view returns (bool) {
    (...)
    bytes32 domainSeparator = keccak256(abi.encode(DOMAIN_TYPEHASH,
keccak256(bytes(saleName)), _getChainId(), address(this)));
    bytes32 structHash = keccak256(abi.encode(PERMIT_TYPEHASH, participant, tokenAmount,
deadline));
    bytes32 digest = keccak256(abi.encodePacked("\x19\x01", domainSeparator, structHash));
    address _signatory = ECDSA.recover(digest, v, r, s);
    (...)
}
```

Recommendation

Remove the `participant` input parameter in the `takeLotteryAllocation()` function, and use `msg.sender` instead of `participant` in 325L.

C1-04 Calling a function in the wrong place

Medium Resolved

In the `buyTokenOnFCFS()` function, the `_createDistributionAddress()` function has to be called after the if-statement in 388L, otherwise the `participants` variable won't be incremented.

C1-05 Multiplication after division

Low Resolved

In 450L, 458L, and 528L there is multiplication after division, which increases the possibility of an error in integer math.

C1-06 Function input values are not checked

Low Acknowledged

The `start_` and `end_` input values are not checked in the `updateTranferPossibilityPeriod()` function. Let `start_` be less than `end_`, then many functions in the `SaleToken` contract will be blocked.

Developer response

The Avata team explained that they understand possible consequences of invalid input and argue that this check won't save from intentional transfer stops.

C1-07 Gas optimization

Low Resolved

a. `_distributionAddresses[participant][periodId_].amount` is read from storage multiple times in the `claimToken()` function;

b. The following lines use redundant comparison with true: 120L, 325L, 347L, 379L, 536L;

c. The `./interfaces/ISalesFactory.sol` and `hardhat/console.sol` imports are not used in the

contract;

d. It's not necessary to put `msg.sender` to `address()` method L354;

e. `sellingTokenAddress`, `stakingAddress`, `farmLensV2Address` can be marked as immutable.

C1-08 Getting a chainId

Info Resolved

Starting with version 0.8 of the Solidity language, chainId can be obtained not only with `assembly`, but also with a global variable `block.chainid`.

C1-09 Incorrect documentation

Info Resolved

`FCFS` period is used in documentation 258L-260L instead of `allocation` period.

C2. SalesFactory

Overview

The contract deploys sales contracts and stores their addresses. Anyone can call the lookup method to get the addresses of deployed contracts.

Issues

C2-01 Lacks a zero-check on constructor

Low Resolved

There are no zero address checks for input addresses in the contract constructor. Thus, an error is possible during the contract deployment.

C2-02 Gas optimization

Low Partially fixed

a. `_signatory` can be marked immutable;

b. To minimize gas consumption during contracts deployment, consider using Openzeppelin Clones library;

c. The `ReentrancyGuard` inheritance is not used.

Update

The Avata team decided to leave it at issue B. The rest of the issues have been resolved.

C3. SaleToken

Overview

The contract is an ERC20 token and defines the initial functionality of the sale.

Issues

C3-01 Gas optimization

Low Resolved

The `ERC20Pausable.sol` and `ERC20Capped.sol` imports are not used in the contract.

C3-02 Lack of increase and decrease allowance functions

Info Acknowledged

The contract lacks increase and decrease allowance functions. These functions are among the most important when managing a token. To see more follow the link.

C4. Multiple contracts

Overview

Issues in this section are related to various contracts.

Issues

C4-01 Floating Pragma

Info Resolved

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

The issue is typical for `SaleToken`, `Sale`, `SalesFactory` contracts.

Recommendation

Lock the pragma version and also consider known bugs (link) for the compiler version that is chosen.

5. Conclusion

3 high, 1 medium, 6 low severity issues were found.

3 high, 1 medium, and 5 low severity issues have been resolved in the update. 1 low severity issues have been resolved partially.

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.
- **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