#HashEx
BLOCKCHAIN SECURITY

Node.sys

smart contracts final audit report

October 2023

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 Node.sys team to perform an audit of their smart contract. The audit was conducted between 27/09/2023 and 02/10/2023.

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 @sokol/nys Gitlab repository after the 053011fc commit.

Update: the Node.sys team has responded to this report. The updated code is located in the same repository after the 3278815 commit.

2.1 Summary

Project name	Node.sys
URL	https://nodesys.io
Platform	Binance Smart Chain
Language	Solidity
Centralization level	High
Centralization risk	Medium

2.2 Contracts

Name	Address
Nodesys	
Consensus	
Vesting	

3. Project centralization risks

The Nodesys token can be minted by a consensus decision with a threshold of confirmations from consensus' owners.

Any `confirmationsRequired` number of owners can exclude other owners from consensus mechanism.

4. Found issues

High	2 (9%)
Medium	4 (17%)
Low	11 (48%)
Info	6 (26%)

C39. Nodesys

ID	Severity	Title	Status
C39Ia5	High	DoS attack from malicious vested account	Resolved
C39I9e	Medium	Blacklisting by the owner	Resolved
C39Ia4	Medium	Possible overriding user record in vesting mapping	Resolved
C39I94	Low	Owner can perform an arbitrary external call	Resolved
C39Ie3	Low	Iteration over unlocked vestings on each transfer	Acknowledged
C39I91	Low	Gas optimizations	Resolved
C39I93	Info	Typos	Resolved
C39I92	Info	Default visibility	Resolved

C3a. Consensus

ID	Severity	Title	Status
C3aI95	High	No proposal deadline	Resolved
C3aI96	Medium	No safety guard for minimum confirmations	Resolved
C3aI98	Low	Function signature collision	Acknowledged
C3aI97	Low	Gas optimizations	Acknowledged
C3aI99	Info	Lack of require message	Resolved
C3aI9b	Info	Typos	Acknowledged
C3aI9a	Info	Default visibility	Acknowledged

C3b. Vesting

ID	Severity	Title	Status
C3bIb0	Medium	Logic of assignRandomAddresses	Resolved
C3bI9c	Low	Gas optimizations	Partially fixed
C3bIa0	Low	Implicit visibility of a variable	Resolved
C3bI9d	Low	Deterministic random	Acknowledged
C3bIa2	Low	Lack of events	Resolved
C3bI9f	Low	Unnecessary import of console.log in Production Code	Resolved

C3bIa1	Low	Address parameters not indexed in events	Resolved
C3bIa3	Info	Non-conventional naming for external functions	Resolved

5. Contracts

C39. Nodesys

Overview

An ERC-20 standard token with minting function governed by list of owners (see Consensus contract) and vesting functionality (see Vesting contract).

Issues

C39Ia5 DoS attack from malicious vested account

High Resolved

If user's tokens are vested and the user is added to exceptions, he is able to transfer tokens to other users. According to the internal documentation, these tokens should be also locked on the recipient account until the vesting is unlocked.

```
    function _transfer(address from, address to, uint256 amount) internal override{
        //Is the user a member of Vesting?
        if(_vesting[from] != address(0)){
            Vesting vesting = Vesting(_vesting[from]);
            //We receive data on whether the user has been unlocked or whether the vesting
itself has been unlocked. We get data whether the user is an exception
            (bool unlock, bool exception) = vesting.checkUnlock(from);
            if(exception){
                if(!unlock){
                    //if it is not yet unlocked, and the user is in the exception list, he
is allowed to make a transfer, but the recipient ends up in the vesting
                    vesting.addAddress(to);
                }
            }else{
                require(unlock,"Your not unlocked");
            }
        }
        super._transfer(from, to, amount);
    }
```

The recipient address is added to the vesting, but the Nodesys contract won't check it if the recipient sends his tokens: Nodesys contract checks its `_vesting` mapping which is not updated.

Recommendation

Modify the contract logic to ensure that vested tokens transferred to a recipient's account are locked. However, be cautious: if all tokens are locked, it could introduce a potential vulnerability. Specifically, an account with an exception might exploit this by sending minimal token amounts to another account, effectively locking it and potentially facilitating a DoS attack.

C39I9e Blacklisting by the owner

Medium Resolved

Any of the contract's owners can block transfers of arbitrary addresses by assigning them to a new vesting contract.

```
function addAddressToVesting(address[] memory _users) external onlyOwner{
    Vesting vesting = new Vesting(address(this));
    for(uint i; i < _users.length; i++){
        ...
        _vesting[user] = address(vesting);
    }
    ...
}

function _transfer(address from, address to, uint256 amount) internal override{
    if(_vesting[from] != address(0)){
        (bool unlock, bool exception) = vesting.checkUnlock(from);
        if(exception){
            if(!unlock){
                vesting.addAddress(to);
            }
        }else{
            require(unlock,"Your not unlocked");
        }
    super._transfer(from, to, amount);
```

```
    }
```

Recommendation

Restrict `addAddressToVesting()` to `onlyConsensus`.

C39Ia4 Possible overriding user record in vesting mapping

Medium Resolved

The function `addAddressToVesting()` creates a vesting contract and sets its address to the `_vesting` mapping. In case the owner of the contract passes the same user again amongst the `_users` parameter, the `_vesting` record for that user will be overridden.

```
    function addAddressToVesting(address[] memory _users) external onlyOwner{
        Vesting vesting = new Vesting(address(this));

        for(uint i; i < _users.length; i++){
            address user = _users[i];
            vesting.addAddress(user);

            _vesting[user] = address(vesting);
            emit checkUnlock(user,_vesting[user]);
        }
        vestings.push(address(vesting));

    }
```

Recommendation

Check if a user has already an assigned vesting before updating it.

C39I94 Owner can perform an arbitrary external call

Low Resolved

The `addException()` function allows the contract owner to execute arbitrary code. The owner of the contract can pass arbitrary address parameter `_vestingAddr` and call function with the same signature as `_addException(address)`.

```
    function addException(address _vestingAddr, address _user) external onlyOwner{
        require(_vestingAddr != address(0),"Vesting cannot be a zero address");
        require(_user != address(0),"User cannot be a zero address");

        Vesting vesting = Vesting(_vestingAddr);
        vesting._addException(_user);
    }
```

Recommendation

Add requiment for the `_vestingAddr` to be one of already created by the Nodesys vesting contracts.

C39le3 Iteration over unlocked vestings on each transfer

Low Acknowledged

The function `_checkTransfer()` is invoked for every token transfer to assess vesting conditions. Currently, it iterates over all vestings associated with a user's account, regardless of whether they are already unlocked. In scenarios where an account has a substantial number of vestings, this can lead to significant gas consumption, as each iteration consumes gas and the majority of the iterations may be unnecessary if the vestings are already unlocked.

```
    function _checkTransfer(address _from, address _to, uint256 _amount) internal {
        uint length = vestingUsers[_from].length();

        if(length > 0){
            uint256 totalSumLock;
            //Checking all available vestings
            for(uint i = 0; i < length; i++){
                address vesting = vestingUsers[_from].at(i);
            ...
    }
```

C39I91 Gas optimizations

Low Resolved

1. Multiple reads from storage in the `unlockAllUsers()` function: `vestings.length` variable.

C39I93 Typos

Info Resolved

Typos reduce code readability. Typos in 'your', 'adress', 'adresss', 'unloked'.

C39I92 Default visibility

Info Resolved

No explicit visibility is defined for the `vestings` variable.

C3a. Consensus

Overview

Governance contract to authorize access to certain functions with threshold confirmations.

Issues

C3aI95 No proposal deadline

High Resolved

Proposals have the timestamp field, which is not used during the execution. A malicious proposal may be pended silently for an arbitrary period of time.

Recommendation

Consensus members list may be updated to contracts only, which have safety check implemented.

C3al96 No safety guard for minimum confirmations

Medium Resolved

Minimum value of `confirmationsRequired` should be added as safety guard in the `assignRequiredConf()` function. Otherwise, a malicious executed proposal can grant full access to a single address.

Recommendation

Consensus members list may be updated to contracts only, which have this safety check implemented.

C3al98 Function signature collision

Low Acknowledged

The incoming proposal is stored in form of `ExecProposal.func::string`, `ExecProposal.data::bytes`. A malicious proposal may be constructed to exploit `bytes4(keccak256(bytes(execProposal.func)))` hash collision.

Recommendation

Consensus members must calculate function signature locally before approving proposal.

C3al97 Gas optimizations

Low Acknowledged

1. Multiple reads from storage in the `confirm()` and `cancelConfirmation()` functions: `eps[_txId].confirmations` variable.

2. Multiple reads from storage in the `discardExecProposal()` function: `owners.length` variable.

3. Multiple reads from storage in the `delOwner()` function: `owners.length` variable.

4. Ineffective removal of array item in the `delOwner()` function, only a single storage write should be performed.

5. Code with no effect in the `delOwner()` functions: `delete owners[owners.length-1]` before `owners.pop()` should be removed.

C3aI99 Lack of require message

Info Resolved

The constructor has a `require` statement without a message.

```
constructor(address[] memory _owners){
    require(_owners.length >= confirmationsRequired);
    ...
}
```

C3aI9b Typos

Info Acknowledged

Typos reduce code readability. Typos in 'execut', 'allready'.

C3aI9a Default visibility

Info Acknowledged

No explicit visibility is defined for the `confirmationsRequired`, `queue`, `isOwner` variables.

C3b. Vesting

Overview

A locker contract meant to be deployed and governed by the Nodesys contract. Doesn't hold or transfer tokens. Vested funds can be unlocked once for all users or individually.

Issues

C3bIb0 Logic of assignRandomAddresses

Medium Resolved

The `assignRandomAddresses()` may result in math underflow in `currentIndex--` operation. The input percentage parameter lacks the NatSpec description.

The example: in the first call the lower 10% of users have been unlocked, the second call random is unlucky to reduce `currentIndex` to lower 10% and check of

`addressAssigned[candidate]` fails, leading to skip of last 10% of candidates and causing `currentIndex` to be lowered to 0 and below.

Recommendation

Fix the function's logic according to internal documentation.

C3bI9c Gas optimizations

Low Partially fixed

1. Multiple reads from storage in the `addAddress()` function: `lastIndex` variable.

2. Unnecessary comparison in while loop: `numAddressesToSetTrue` is always not less than `currentIndex`.

3. Unnecessary read from storage in the `_assignRandomAddresses()` function: `addressUnlock[candidate]` is always true in the `UnlockAddress()` event.

4. Multiple reads from storage in the `assignRandomAddresses()` function: `indexToAddress.length` variable.

C3bIa0 Implicit visibility of a variable

Low Resolved

The variable `unlock` has been declared without an explicit visibility specifier (i.e., `public`, `internal`, or `private`). In Solidity, if no visibility is specified for state variables, they are treated as `internal` by default. Relying on implicit default settings can lead to misunderstandings and unintended consequences. It is always a best practice to specify visibility explicitly to ensure clarity and avoid potential vulnerabilities or misinterpretations.

Using explicit visibility for state variables enhances code readability, reduces potential ambiguities, and ensures that smart contract developers and auditors can readily understand the intended access patterns for each variable.

C3bI9d Deterministic random

Low Acknowledged

The `_assignRandomAddresses()` function is designed to be called by the Nodesys owner to unlock vested funds for a specified percentage of randomly chosen users. The randomization relies on the on-chain data, and therefore, it cannot be considered as a truly secure random.

```
    function _assignRandomAddresses(uint8 percentage) external Nodesys{
        ...

        uint256 seed = getRandom();
        uint256 currentIndex = lastIndex;

        while (numAddressesToSetTrue > 0 && currentIndex > 0) {
            address candidate = indexToAddress[currentIndex];

            if (!addressAssigned[candidate] && seed % currentIndex <
numAddressesToSetTrue) {
                ...
            }
            currentIndex--;
            seed = uint256(keccak256(abi.encodePacked(seed))); // Update the seed
        }
    }

    function getRandom() internal view returns(uint256) {
       uint256 random = uint256(keccak256(abi.encodePacked(block.prevrandao,block.timestamp
,block.gaslimit,block.basefee)));
        return random;
    }
```

Recommendation

Use VRF oracle for random seed to achieve trustless unlocking.

C3bIa2 Lack of events

Low Resolved

Several key functions (`_addException()`, `unlockAll()`, `_assignRandomAddresses()`) do not emit any events. Emitting events in smart contract functions, especially those that change the state,

is essential for tracking and verifying contract operations from off-chain services. Without these events, it becomes challenging for DApps, external systems, or end-users to get insights or notifications of the state changes caused by these functions.

Recommendation

Define appropriate events for each of these functions. Update each function to emit these events at relevant places.

C3bI9f Unnecessary import of console.log in Production Code

Low Resolved

The contract imports "hardhat/console.log" for debugging. While `console.log` is highly beneficial during development and debugging phases, it's not suitable for production code. Deploying this code on the main network would lead to unnecessary gas costs.

```
// Import this file to use console.log
import "hardhat/console.sol";
...
```

Recommendation

Remove the unnecessary import of the `console.log` contract from the contract.

C3bIa1 Address parameters not indexed in events

Low Resolved

the `address` parameters in the events `UnlockAddress` and `UserVesting` are not indexed. Events in Ethereum smart contracts allow parameters to be indexed, which greatly enhances the efficiency of querying these events. In particular, addresses, which are commonly used as lookup keys, should generally be indexed to optimize for frontend or off-chain services that filter or search for events based on specific addresses.

```
event UnlockAddress(
    address sender,
```

```
    bool unlock
);
event UserVesting(
    uint256 totalLocked,
    address user,
    uint blockNumber
);
```

Recommendation

Update the event definitions to index the address parameters:

```
event UnlockAddress(
    address indexed sender,
    bool unlock
);
event UserVesting(
    uint256 totalLocked,
    address indexed user,
    uint blockNumber
);
```

C3bla3 Non-conventional naming for external functions

Info Resolved

Two external functions, `_addException()` and `_assignRandomAddresses()`, employ a naming convention typically reserved for internal or private functions. Conventionally, functions that start with an underscore (`_`) are perceived as having `internal` or `private` visibility. When used for `external` (or `public`) functions, it can lead to confusion regarding the function's intended visibility and usage.

6. Conclusion

2 high, 4 medium, 11 low severity issues were found during the audit. 2 high, 4 medium, 6 low issues were resolved in the update. The reviewed contracts are highly dependent on the owner's account. See the centralization risks chapter.

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