HashEx
BLOCKCHAIN SECURITY

Metarix

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 **Metarix** team to perform an audit of their smart contract. The audit was conducted between 22/04/2022 and 25/04/2022.

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 is available at the @Metarix-Network/Smart-Contracts GitHub repository and was audited after the commit 9bd0ffd.

Update: the Metarix team has responded to this report. The updated code is located in the GitHub repository after the commit 78ae852.

The audited contract is deployed to the Binance Smart Chain mainnet:

LockableToken: 0x55382eEEF32EB87AA27D13d7637954C344154151

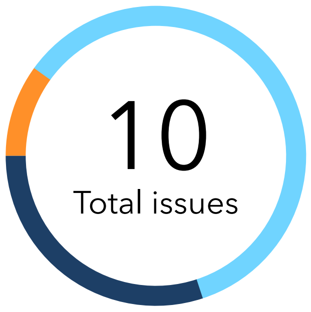
2.1 Summary

Project name	Metarix
URL	https://metarix.network/
Platform	Binance Smart Chain
Language	Solidity

2.2 Contracts

Name	Address
LockableToken	`0x55382eEEF32EB87AA27D13d7637954C344154151`

3. Found issues



●	High	1 (10%)
●	Low	6 (60%)
●	Info	3 (30%)

C1. LockableToken

ID	Severity	Title	Status
C1-01	● High	Broken unlock() due to gas limit	✓ Acknowledged
C1-02	● Low	Lack of emitting event	✓ Acknowledged
C1-03	● Low	Using assert() instead of require()	✓ Resolved
C1-04	● Low	Variable default visibility	✓ Resolved
C1-05	● Low	Gas optimization	✓ Partially fixed
C1-06	● Low	Reason message on failure	✓ Partially fixed
C1-07	● Low	No checks for lock() parameter	✓ Acknowledged
C1-08	● Info	Typos	✓ Resolved
C1-09	● Info	Incorrect reference in documentation	✓ Resolved
C1-10	● Info	Outdated libraries and compiler version	✓ Partially fixed

4. Contracts

C1. LockableToken

Overview

An ERC20-like token with the possibility of locking tokens for a specific period of time.

Issues

C1-01 Broken unlock() due to gas limit ● High ✓ Acknowledged

The function `unlock()` uses a for-loop to run overall user's locks. This may result in an inability to return funds belonging to a user in the following example.

```
function transferWithLock(address _to, bytes32 _reason, uint256 _amount, uint256 _time)
    public
    returns (bool)
{
    uint256 validUntil = now.add(_time); //solhint-disable-line

    require(tokensLocked(_to, _reason) == 0, ALREADY_LOCKED);
    require(_amount != 0, AMOUNT_ZERO);

    if (locked[_to][_reason].amount == 0)
        lockReason[_to].push(_reason);

    transfer(address(this), _amount);

    locked[_to][_reason] = lockToken(_amount, validUntil, false);

    emit Locked(_to, _reason, _amount, validUntil);
    return true;
}


function unlock(address _of)
```

```
    public
    returns (uint256 unlockableTokens)
{
    uint256 lockedTokens;

    for (uint256 i = 0; i < lockReason[_of].length; i++) {
        lockedTokens = tokensUnlockable(_of, lockReason[_of][i]);
        if (lockedTokens > 0) {
            unlockableTokens = unlockableTokens.add(lockedTokens);
            locked[_of][lockReason[_of][i]].claimed = true;
            emit Unlocked(_of, lockReason[_of][i], lockedTokens);
        }
    }

    if (unlockableTokens > 0)
        this.transfer(_of, unlockableTokens);
}
```

User Alice locks a large amount of tokens. Later the attacker creates locks for Alice for only 1 token every few blocks using the `transferWithLock()` function. When the total number of Alice's locks exceeds approximately 1500, she stops being able to unlock even her initial tokens. This may happen because the gas spent on the performing loop in the function exceeds the block gas limit.

Recommendation

It is necessary to limit the number of iterations in the loop. This can be implemented using batches.

Update

In the updated code, the users may still face with mentioned problem if they have more than 100 locks. Each next lock after the 100th will not be available for unlocking due to the limitation of the loop steps of the `maxLength` variable.

```
    function unlock(address _of)
        external
        override
        returns (uint256 unlockableTokens)
    {
        uint256 lockedTokens;

        uint256 maxLength = lockReason[_of].length;
        if(maxLength>100){
            maxLength = 100;
        }
        for (uint256 i = 0; i < maxLength; i++) {
            lockedTokens = tokensUnlockable(_of, lockReason[_of][i]);
            if (lockedTokens > 0) {
                unlockableTokens = unlockableTokens.add(lockedTokens);
                locked[_of][lockReason[_of][i]].claimed = true;
                emit Unlocked(_of, lockReason[_of][i], lockedTokens);
            }
        }

        if (unlockableTokens > 0)
            this.transfer(_of, unlockableTokens);
    }
```

C1-02 Lack of emitting event ● Low ✓ Acknowledged

The `constructor()` of the Ownable contract does not emit the `OwnershipTransferred` event.

C1-03 Using assert() instead of require() ● Low ✓ Resolved

For checking input parameters, we recommend using `require()` statements instead of `assert()` (L25, L31) to conform to the best practices in smart contracts development. This saves gas if a wrong parameter is passed to a function: the `assert()` statement uses all the gas provided in the transaction while the `require()` statement uses only gas spent before a failing statement is reached.

C1-04 Variable default visibility ● Low ✓ Resolved

The variables `balances` (L104), `allowed` (L143) have default visibility. Labeling the visibility explicitly makes it easier to catch incorrect assumptions about who can access the variable.

C1-05 Gas optimization

● Low Partially fixed

a. The functions `transferOwnership()`, `allowance()`, `transferFrom()`, `approve()`, `increaseApproval()`, `decreaseApproval()`, `lock()`, `tokensLockedAtTime()` , `totalBalanceOf()`, `extendLock()`, `increaseLockAmount()`, `unlock()`, `getUnlockableTokens()`, `transferWithLock()` can be declared as external to save gas.

b. The functions of the `Ownable` contract are not used anywhere. It is recommended to remove this inheritance to save gas when deploying the contract.

C1-06 Reason message on failure

● Low Partially fixed

The reason message is not defined in require statements at L60, L69, L112, L113, L152-L155, L173.

This significantly complicates the debugging of the transactions.

Update

Some of the reason messages does not correspond to require statements:

a. L110 - `require(_to != address(0), "Address should not a owner address!")` ,

b. L151 - `require(_to != address(0), "Address should not a owner address!");`,

c. L29 - `require(c >= a, "a should not greater than c")` since the `c` is not the function parameter, the message should be about 'overflow'.

C1-07 No checks for lock() parameter

● Low Acknowledged

The input parameter `_time` of the `lock()` function is not checked. This can cause a user to create a lock with too long a duration (for example, if they mistakenly specify the time in milliseconds).

Update

In the updated code users can lock tokens for period more than 1 year, due to incorrect `nextYear` variable.

```
function lock(bytes32 _reason, uint256 _amount, uint256 _time)
        external
        override
        returns (bool)
    {
        uint256 nextYear = block.timestamp + 31536000;
        require(_time <= nextYear, "Time should not greater than next 1 year!");

        uint256 validUntil = block.timestamp.add(_time); //solhint-disable-line
        ...
        locked[msg.sender][_reason] = lockToken(_amount, validUntil, false);
        ...
    }
```

In this case user can lock their tokens for approximately 52 years + 1 year (block.timestamp ~ 52 years).

We recommend to fix `nextYear` variable in the following way:

```
uint256 nextYear = 31536000;
```

Also note that this issue also affects the `transferWithLock()` function.

C1-08 Typos ● Info ✓ Resolved

There are typos in the code documentation:

a. L390 'adress' should be 'address';

b. L390 'transfered' should be 'transferred';

c. L392 'transfered' should be 'transferred'.

C1-09 Incorrect reference in documentation ● Info ✓ Resolved

There is a reference to an openZeppelin package in the contract documentation at L347. But no package is imported into the contract.

C1-10 Outdated libraries and compiler version ● Info ✓ Partially fixed

We recommend using recent versions of the contract libraries (eg SafeMath, ERC20, etc.) and the Solidity compiler. This can help to remedate disclosed bugs and issues that affect the current library or compiler version.

Update

The updated code still uses outdated contract libraries (eg SafeMath, ERC20, etc.). We recommend importing the latest versions from the OpenZeppelin package.

5. Conclusion

1 high, 6 low, and 3 informational severity issue were found.

2 low and 2 informational severity issues have been resolved in the update. 2 low and 1 informational severity issues have been partially resolved.

We strongly suggest adding unit and functional tests for the contract.

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

@hashexbot

blog.hashex.org

linkedin

github

twitter

HashEx
BLOCKCHAIN SECURITY