Security Audit ERC-20 Smart Contract

Homihelp

2020-08-13

Introduction

Homihelp is a customer support services platform that sells its services through cryptocurrency, aims to make cryptocurrency more accessible and implement decentralized features in software one by one so that users can easily understand the decentralized apps feature wise by using in daily business tools.

Homihelp is the communication bridge that fills the gap between your customers and your business. Homihelp provides omnichannel support tools for businesses to provide top class support for their customers.

As requested by Homihelp and as part of the vulnerability review and management process, Red4Sec has been asked to perform a security code audit and a cryptographic assessment in order to evaluate the security of the Homihelp Smart Contract source code.

Scope

The scope of this evaluation includes:

- Description: Homihelp Smart Contract Security Audit.
- Smart Contract: https://cn.etherscan.com/token/0xCa208BfD69ae6D2667f1FCbE681BAe12767c0078

Conclusions

The general conclusion resulting from the conducted audit, is that the *Homihelp's Smart Contract* is secure and does not present any known vulnerabilities.

The overall impression about code quality and organization is very positive, although Red4Sec has found some potential improvements, these do not pose any risk by themselves. We have classified such issues as informative only, but they will help Homihelp to continue to improve the security and quality of its developments.

Recommendations

Outdated Compiler Version

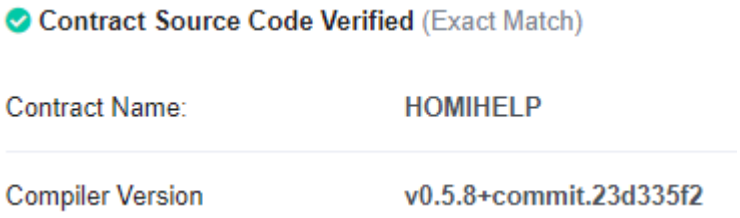
Solc frequently launches new versions of the compiler. Using an outdated version of the compiler can be problematic, especially if there are errors that have been made public or known vulnerabilities that affect such version.

We have detected that the audited contract uses the following version of Solidity *pragma ^0.5.8*:

```
1  /**
2   *Submitted for verification at Etherscan.io on 2020-04-13
3   */
4
5  pragma solidity ^0.5.8;
6
7  library SafeMath {
```

Nevertheless, when the deploy was made (13th of April 2020), the last available version was 0.6.5, therefore, the pragma used should have been that one.

Finally, the contract was compiled with the version **v0.5.8+commit.23d335f2**, as we can observe in the following image.



0.6.5 (2020-04-06)

Important Bugfixes:

- Code Generator: Restrict the length of dynamic memory arrays to 64 bits during creation at runtime fixing a possible overflow.

Language Features:

- Allow local storage variables to be declared without initialization, as long as they are assigned before they are accessed.
- State variables can be marked `immutable` which causes them to be read-only, but assignable in the constructor. The value will be stored directly in the code.

Compiler Features:

- Commandline Interface: Enable output of storage layout with `--storage-layout`.
- Metadata: Added support for IPFS hashes of large files that need to be split in multiple chunks.

Bugfixes:

It is always of good policy to use the most restrictive and up to date version of the pragma.

References

- https://github.com/ethereum/solidity/releases/tag/v0.6.5
- https://github.com/ethereum/solidity/blob/develop/Changelog.md

Provide License for Third-Party Code

The ***SafeMath.sol*** contract from OpenZeppelin, which is used for arithmetic operations, has been detected as included in the Homihelp project.
However, these contracts have been included in the repository by copying it rather than by package manager.

This is not recommended by OpenZeppelin, although is not necessarily incorrect, it can be considered as a vector of attack. We could obtain this update automatically by using the original sources and if the project resolves any vulnerability or bug in the code, which results in avoiding known vulnerabilities.

Additionally, these OpenZeppelin contracts are under the MIT license, which requires its license to be included within the code. For this reason, we highly recommend including a reference or copyright in the Homihelp project.

References

- https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol
- https://github.com/homihelp/smart-contract/blob/master/homihelp.sol

Recommendations

- Include third-party code through package managers.
- Include in the Homihelp project a reference to OpenZeppelin code since it's under the MIT license and it's required by such.

GAS Usage Optimization

Software optimization is the process of modifying a software system to make an aspect of it work more efficiently or use less resources. This premise must be applied to smart contracts as well, so that they execute faster or in order to save GAS.

On Ethereum blockchain, GAS is an execution fee which is used to compensate miners for the computational resources required to power smart contracts. If the network usage is increasing, so will the value of GAS optimization.

These are some of the requirements that must be met to reduce GAS consumption:

- Short-circuiting.
- Remove redundant or dead code.
- Delete unnecessary libraries.
- Explicit function visibility.
- Use of proper data types.
- Use hardcoded CONSTANT instead of state variables.
- Avoid expensive operations in a loop.
- Pay special attention to mathematical operations and comparisons.

Remove Unnecessary Steps

Following, an example that results unalarming, because it only affects the cost of the deploy:

The *constructor* of the *BaseToken* class sets the *owner's* balance to zero, this step is necessary since its value it's later established in the *HOMIHELP* class.

```
constructor() public
{
    balances[msg.sender] = totalSupply;
}
```

Reference

- https://github.com/homihelp/smart-contract/blob/df99a9983b1d5ec85fa0c4aca5f024edd31de7ea/homihelp.sol#L205

Unused Variables

Just as the previous case, the declaration of the two following variables, which are not used, result in a higher cost of the deploy. Is of good practice to eliminate such variables.

```
string constant internal ERROR_DATE_TIME_NOT_VALID         = 'Reason: Datetime must grater or equals than zero.';
string constant internal ERROR_DUPLICATE_ADDRESS = 'Reason: msg.sender and receivers can not be the same.';
```

References

- https://github.com/homihelp/smart-contract/blob/df99a9983b1d5ec85fa0c4aca5f024edd31de7ea/homihelp.sol#L176
- https://github.com/homihelp/smart-contract/blob/df99a9983b1d5ec85fa0c4aca5f024edd31de7ea/homihelp.sol#L376

Logic Optimizations

Unlike the previous cases, this optimization affects all variables and not only during the deployment. Therefore, by optimizing this function, the cost of GAS in each transaction will be lower, saving the users costs in GAS.

In the following ***balance*** variable (represented by point #1) the variable should be declared in the ***else*** of the condition. This prevents reading the memory of such balance, since it is not used in case that the locked balance is less than or equal to 0.

Next, represented by point #2, you can observe a condition that can be erased without affecting the logic of the contract.

```
// true: _who can transfer token
// false: _who can't transfer token
function isLocked(address _who, uint256 _value) view public returns(bool)
{
    uint256 lockedBalance = lockedBalanceOf(_who);
    uint256 balance = balanceOf(_who);   // 1

    if(lockedBalance <= 0)
    {
        return false;
    }
    else
    {
        return !(balance > lockedBalance && balance.sub(lockedBalance) >= _value);   // 2
    }
}
```

Reference

- https://github.com/homihelp/smart-contract/blob/df99a9983b1d5ec85fa0c4aca5f024edd31de7ea/homihelp.sol#L256

Use of Same Variable for Multiple Purposes

This issue is not critical, nor it poses a risk. The issue was added to this report to improve the good practices of the developers of the future.

In the 143 line, we can observe that when a new owner is set, it is verified that the *msg.sender* is equal to the previously proposed owner (*proposedOwner*) the problem is that *proposedOwner* is also used as a flag with *address(0)* as an inactive/false value.

In order to make a more resilient code, checking that *msg.sender* is never equal to this inactive flag, i.e. *address(0)*, is of good practice. Because, if it's able to send a transaction with an empty sender, a takeover of the Smart Contract could happen.

Nowadays, this is practically impossible because it would need the private key of that address, however, it is worth mentioning that Ethereum is a living project and that it is continuously undergoing modifications, a clear example is the *EIP86* designed to abstract the verification of signatures in which the *NULL_SENDER* is set to: *2**160 – 1.*

Therefore, it would be pretentious to state that in a near future, special transactions will not be from *address(0)* as special feature. Which would mean having to redeploy the contract, so it doesn't become vulnerable.

```
function claimOwnership() public
{
    require(msg.sender == proposedOwner, ERROR_NOT_PROPOSED_OWNER);

    emit OwnershipTransferred(_owner, proposedOwner);

    _owner = proposedOwner;
    proposedOwner = address(0);
}
```

References

- https://github.com/homihelp/smart-contract/blob/df99a9983b1d5ec85fa0c4aca5f024edd31de7ea/homihelp.sol#L141
- https://github.com/ethereum/EIPs/blob/master/EIPS/eip-86.md

Recommendations

- Check that *msg.sender* is different from 0.
- Or, create a new flag variable for the *proposedOwner*.

Denial of Service by Locks

The logic executed to check if a balance is locked might trigger a denial of service (DoS).

A denial of service (DoS) attack is an attack on a computer system, functionality or network that causes a service or resource to be inaccessible to legitimate users.

Loops without limits are considered a bad practice in the development of Smart Contracts, because they can cause a denial of service or overly expensive executions such is the case affecting Homihelp.

In the following image, we can observe that when the locked balance of an account is calculated, all possible locks of the account are traversed, this allows a denial of service when the account has enough locks to exceed the maximum allowed GAS per block, which currently is of 12 million approximately.

```
function lockedBalanceOf(address _who) view public returns (uint256)
{
    require(_who != address(0), ERROR_ADDRESS_NOT_VALID);

    uint256 lockedBalance = 0;
    if(lockup[_who].length > 0)
    {
        Lock[] storage locks = lockup[_who];

        uint256 length = locks.length;
        for (uint i = 0; i < length; i++)
        {
            if (now < locks[i].expiresAt)
            {
                lockedBalance = lockedBalance.add(locks[i].amount);
            }
        }
    }

    return lockedBalance;
}
```

The criticality of this vulnerability is drastically reduced since the administrator is the only one allowed to transfer balances with locks and to eliminate them. This makes such practice unadvisable, since it gives the administrator the possibility to directly make a denial of service in a specific account, even if it's at a high cost.

It shall be noted that eliminating the expired *Locks*, is convenient, since they produce more GAS consumption in the transfers of the users which have locks, even if they have expired.

Decentralization Recommendation

In order to promote decentralization, it would be advisable to improve the start-stop logic of the contract.
Currently, this logic works as a switch and the only one capable of altering it is the administrator.

While we understand that business logic is necessary for the project, it could be implemented in a way that the stop function detains the contract transactions during a determined number of blocks, this would force the administrator to have to make another transaction if he wants to extend that time.

Therefore, this would prevent all users of the token from being affected if the *start* method is not called again after a temporary stop of the contract.