Smart Contract Security Audit

SmartKey

2021-02-12

Content

Introduction

SmartKey is the missing part of the puzzle that connects the world of decentralized finance (DeFi) and blockchain with the world of physical assets. They are the first working platform that allows you to combine physical values and assets (Blockchain Of Things) with DeFi projects operating on the Ethereum and Waves blockchain.

As requested by **SmartKey** and as part of the vulnerability review and management process, **Red4Sec** has been asked to perform a security code audit to **evaluate the security of** its smart contract.

Disclaimer

This document only represents the results of the code audit conducted by Red4Sec Cybersecurity and should not be used in any way to make investment decisions or as investment advice on a project.

Likewise, the report should not be considered neither "endorsement" nor "disapproval" of the guarantee of the correct business model of the analyzed project.

Scope

The detailed audit below has been based on the contract deployed on Ethereum MainNet address `0x06A01a4d579479Dd5D884EBf61A31727A3d8D442`.

SmartKey Token Smart Contract

- Token.sol
  - SHA256: `AA791156A89126A56C06AD0EB538C8EB99FAA510819A012B47E010FA82888F1A`

Results Overview

To this date, 12th of February 2020, the general conclusion resulting from the conducted audit is that **SmartKeys's smart contract is secure and does not present any known vulnerabilities** that could compromise the security of the users.

Nevertheless, Red4Sec has found some minor potential improvements, these do not pose any risk and we have classified such issues as informative only, but they will help SmartKey to continue to improve the security and quality of its developments.

The current implementation of SafeKey, guarantees the transparency and decentralization of the token by not containing methods that allow blocking balances or altering the total supply. SafeKey only contains the necessary methods for the correct operation of the ERC 20 standard.

Recommendations

ERC20 Approval Race Condition

SmartKey token does not have any sort of protection against the well-known attack "*Multiple Withdrawal Attack*"[1] on the *approve/transferFrom* methods of the ERC20 standard.

Although this attack poses a limited risk in specific situations, it is worth mentioning to consider it for possible future operations.

There are a few solutions to mitigate this front running such as, to first reduce the spender's allowance to *0* and set the desired value afterwards; another solution could the one that Open Zeppelin offers, where the non-standard *decreaseAllowance* and *increaseAllowance* functions have been added to mitigate these well-known issues, involving setting allowances.

[1] https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM

Outdated Compiler Version

Solc frequently launches new versions of the compiler and 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 this version.

We have detected that the audited contract sets the following version of Solidity pragma *^0.5.0.*

```
/**
 *Submitted for verification at Etherscan.io on 2020-08-28
*/

pragma solidity ^0.5.0;

contract Token {
    // Track how many tokens are owned by each address.
    mapping (address => uint256) public balanceOf;
```

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

Use of Require statement without reason message

Throughout the audit, it was verified that the reason message is not specified in some require methods; this will give the user more information and consequently, make it more user friendly.

An example of this issue can be found in: *SmartKey.sol:26* even though the same modus operandi can be found through all the smart contract.

```
function transfer(address to, uint256 value) public returns (bool success) {
    require(balanceOf[msg.sender] >= value);

    balanceOf[msg.sender] -= value;  // deduct from sender's balance
    balanceOf[to] += value;          // add to recipient's balance
    emit Transfer(msg.sender, to, value);
    return true;
}
```

This functionality is compatible since *0.4.22* release and the contract were compiled with the *0.5.16* version; this will result in compatibility with this feature.

GAS 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 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 hard-coded CONSTANT instead of state variables.
- Avoid expensive operations in a loop.
- Pay special attention to mathematical operations and comparisons.

Executions Cost

The use of constants is recommended as long as the variables are never to be modified. In this case the variables "*name*", "*symbol*" and "*decimals*" of the SmartKey's contract should be declared as constants since they would not be necessary to access the storage and to read the content of these variables, therefore, the execution cost is much lower.

```
// Modify this section
string public name = "SmartKey";
string public symbol = "Skey";
uint8 public decimals = 8;
uint256 public totalSupply = 1000000000 * (uint256(10) ** decimals);
```


RED4SEC
Invest in Security, invest in your future