DOOR Token
Smart Contract ERC20 Audit

Performed by Kishan Patel
Certified Auditor

Prepared on July 15, 2021

Table of Contents

Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

Attacks made to the contract

In order to check for the security of the contract, we tested several attacks in order to make sure that the contract is secure and follows best practices.

Over and under flows

An overflow happens when the limit of the type variable uint256, 2 ** 256, is exceeded. What happens is that the value resets to zero instead of incrementing more. On the other hand, an underflow happens when you try to subtract 0 minus a number bigger than 0. For example, if you subtract 0 - 1 the result will be = 2 ** 256 instead of -1. This is quite dangerous. This contract does check for overflows and underflows by using OpenZeppelin's SafeMath to mitigate this attack, but all the functions have strong validations, which prevented this attack.

Short address attack

If the token contract has enough tokens and the buy function doesn't check the length of the address of the sender, the Tron's virtual machine will just add zeros to the transaction until the address is complete. Although this contract is not vulnerable to this attack, there are some points where users can mess themselves due to this (Please see below). It is highly recommended to call functions after checking the validity of the address.

Visibility & Delegate call

It is also known as, The Parity Hack, which occurs while misuse of Delegate call.
No such issues found in this smart contract and visibility are also properly addressed. There are some places where there is no visibility defined. Smart Contract will assume "Public" visibility if there is no visibility defined. It is good practice to explicitly define the visibility, but again, the contract is not prone to any vulnerability due to this in this case.

Reentrancy / TheDAO hack

Reentrancy occurs in this case: any interaction from a contract (A) with another contract (B) and any transfer of Tron hands over control to that contract (B).
This makes it possible for B to call back into A before this interaction is completed.
Use of "require" function in this smart contract mitigated this vulnerability.

Forcing Tron to a contract

While implementing "selfdestruct" in smart contract, it sends all the tron to the target address. Now, if the target address is a contract address, then the fallback function of target contract does not get called. And thus Hacker can bypass the "Required" conditions. Here, the Smart Contract's balance has never been used as guard, which mitigated this vulnerability.

Good things in smart contract

SafeMath library:

You are using the SafeMath library. This protects you from underflow and overflow attacks.

```
84 library SafeMath {
85     function sub(uint256 a, uint256 b) internal pure returns (uint256) {
86       assert(b <= a);
87       return a - b;
```

Good required condition in functions:

Here you are checking that numTokens value should be smaller or equal to balance of msg.sender(caller of function).

```
53
54     function transfer(address receiver, uint256 numTokens) public override returns
55         require(numTokens <= balances[msg.sender]);
56         balances[msg.sender] = balances[msg.sender].sub(numTokens);
57         balances[receiver] = balances[receiver].add(numTokens);
```

Here you are checking that numTokens value should be smaller or equal to balance of owner account, and allowance value to caller of function from account address should be bigger than or equal to numTokens.

```
72     function transferFrom(address owner, address buyer, uint256 numTokens) public
73         require(numTokens <= balances[owner]);
74         require(numTokens <= allowed[owner][msg.sender]);
75
```

Critical vulnerabilities found in the contract

No Critical vulnerabilities found

Medium vulnerabilities found in the contract

No Medium vulnerabilities found

Low severity vulnerabilities found

Short address attack

This is not a critical issue in solidity, because this s increased In the new solidity version. But it is good practice to check for the short address.

- After updating the version of solidity it's not mandatory.
- In some functions you are not checking the value of the address parameter.

Function: - transfer ('receiver')

```
53
54      function transfer(address receiver, uint256 numTokens) public override returns
55          require(numTokens <= balances[msg.sender]);
56          balances[msg.sender] = balances[msg.sender].sub(numTokens);
57          balances[receiver] = balances[receiver].add(numTokens);
```

It's necessary to check the address value of "receiver". Because here you are passing whatever variable comes in the "receiver" address from outside.

Function: - transferFrom ('owner', 'buyer')

```
72      function transferFrom(address owner, address buyer, uint256 numTokens) public
73          require(numTokens <= balances[owner]);
74          require(numTokens <= allowed[owner][msg.sender]);
75
76          balances[owner] = balances[owner].sub(numTokens);
```

It's necessary to check the addresses values of "owner", "buyer". Because here you are passing whatever variable comes in "owner", "buyer" addresses from outside.

Compiler version is not fixed

- In this file you have put "pragma solidity ^0.6.0;" which is not a good way to define the compiler version.
- Solidity source files indicate the versions of the compiler they can be compiled with. Pragma solidity 0.6.0; // bad: compiles 0.6.0 and above pragma solidity 0.6.0; //good: compiles 0.6.0 only
- If you put(>=) symbol then you are able to get compiler version 0.6.0 and above. But if you don't use(^/>=) symbol then you are able to use only the 0.6.0 version. And if there are some changes in the compiler and you use the old version then some issues may come at deploy time.

Check user balance in approve

- I have found that in approve function user can give more value than their balance.
- It is necessary to check that the user can give less or equal value to their amount.
- There is no validation about user balance. So it is good to check that a user did not set approval wrongly.

Function: - approve

```
62 ▾    function approve(address delegate, uint256 numTokens) public override returns (
63          allowed[msg.sender][delegate] = numTokens;
64          emit Approval(msg.sender, delegate, numTokens);
65          return true;
66      }
```

Here you can check that balance of delegare address should be bigger or equal to value.

Summary of the Audit

Overall the code is well and performs well. The smart contract has no backdoor in the code

Please try to check the address and value of the token externally before sending it to the solidity code. Our final recommendation would be to pay more attention to the visibility of the functions , hardcoded address and mapping since it's quite important to define who's supposed to execute the functions and to follow best practices regarding the use of assert, require etc. (which you are doing).

Good Point: code performance is good. Address validation and value validation is done properly.

Suggestions: Please add address validations at some place and also try to use the static version of solidity, check user balance in the approve function.