SMART CONTRACT AUDIT REPORT

For

Kudoken (MATIC)

Prepared By: Kishan Patel

Prepared For: Kudoken Ecosystem

Prepared on: 18/06/2021

Table of Content

• 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.

• Overview of the audit

The project has 1 file. It contains approx 228 lines of Solidity code. All the functions and state variables are well commented using the natspec documentation, but that does not create any vulnerability.

• 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 amount of tokens and the buy function doesn't check the length of the address of the sender, the Eth's virtual machine will just add zeros to the transaction until the address is complete.

Although this contract **is not vulnerable** to this attack, but there are some point where users can mess themselves due to this (Please see below). It is highly recommended to call functions after checking 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 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 Eth 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 Ethereum to a contract

While implementing "selfdestruct" in smart contract, it sends all the eth 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

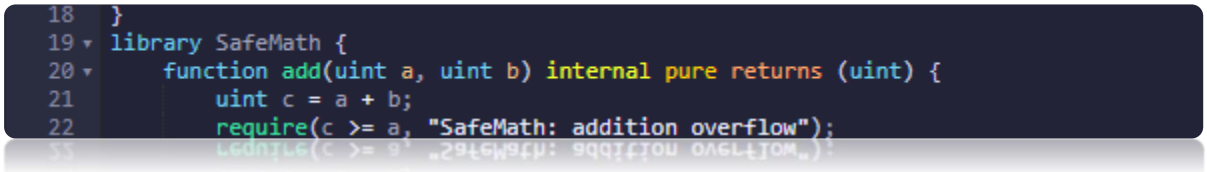
- ### Compiler version is static:-

  => In this file you have put "pragma solidity 0.8.0;" which is a good way to define compiler version.

  => Solidity source files indicate the versions of the compiler they can be compiled with. Pragma solidity 0.8.0; // bad: compiles 0.8.0 and above pragma solidity 0.8.0; //good: compiles 0.8.0 only

  => If you put(^) symbol then you are able to get compiler version 0.8.0 and above. But if you don't use(^) symbol then you are able to use only 0.8.0 version. And if there are some changes come in the compiler and you use the old version then some issues may come at deploy time.

- ### SafeMath library:-
  - You are using SafeMath library it is a good thing. This protects you from underflow and overflow attacks.



- **Good required condition in functions:-**
  - Here you are checking that sender and recipient addresses value are proper and valid addresses.

```
function _transfer(address sender, address recipient, uint amount) internal{
    require(sender != address(0), "ERC20: transfer from the zero address");
    require(recipient != address(0), "ERC20: transfer to the zero address");

```

  - Here you are checking that owner and spender addressed value are proper and valid addresses, and owner balance should be equal or bigger to amount value.

```
function _approve(address owner, address spender, uint amount) internal {
    require(owner != address(0), "ERC20: approve from the zero address");
    require(spender != address(0), "ERC20: approve to the zero address");
require(_balances[owner] >= amount, "ERC20: sender has not enough balance");

```

  - Here you are checking that value is not zero and allowance method of token contract is successfully called.

```
function safeApprove(IERC20 token, address spender, uint value) internal {
    require((value == 0) || (token.allowance(address(this), spender) == 0),
        "SafeERC20: approve from non-zero to non-zero allowance"
    );
    callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
```

  - Here you are checking that checking that token address is contract address and function of token contract is successfully called.

```
function callOptionalReturn(IERC20 token, bytes memory data) private {
    require(address(token).isContract(), "SafeERC20: call to non-contract");

    // solhint-disable-next-line avoid-low-level-calls
    (bool success, bytes memory returndata) = address(token).call(data);
    require(success, "SafeERC20: low-level call failed");

```

- **Critical vulnerabilities found in the contract**

=> No critical vulnerabilities found

- **Medium vulnerabilities found in the contract**

=> No Medium vulnerabilities found

- # Low severity vulnerabilities found

=> No Low vulnerabilities found

- # Summary of the Audit

Overall the code is well and performs well. **There is no**

backdoor to still fund from this contract.

Please try to check the address and value of token externally before sending 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 executed the functions and to follow best practices regarding the use of assert, require etc. (which you are doing ;)). .

- **Good Point:** Latest version of solidity is used. Code performance is good. Address and value validation is done properly.