SMART CONTRACT AUDIT REPORT

For

Egyptian mau

Prepared By: Kishan Patel **Prepared For**: Ahmad Khalid

Prepared on: 09/11/2021

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 50 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 ethereum’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 ethereum 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.

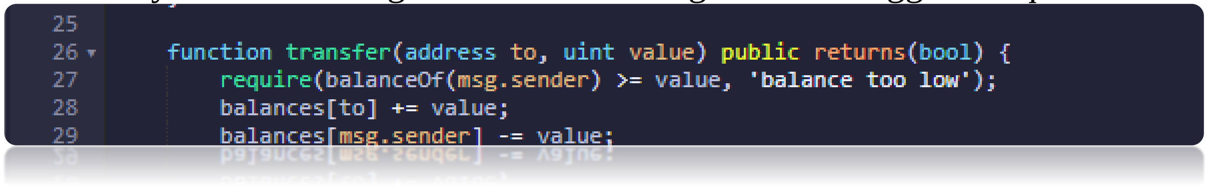
- **Forcing Ethereum to a contract**

While implementing “selfdestruct” in smart contract, it sends all the ethereum 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

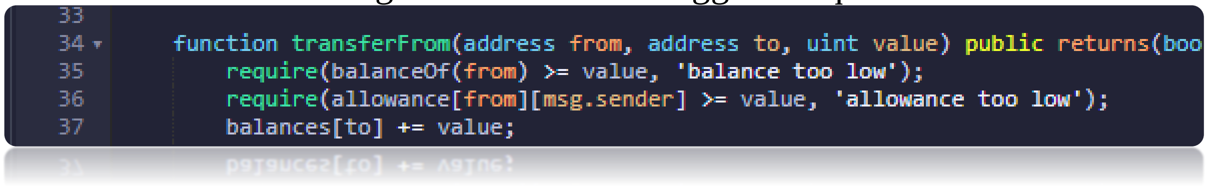
• Good required condition in functions:-

- Here you are checking that balance of msg.sender is bigger or equal to value.



```
    function transfer(address to, uint value) public returns(bool) {
        require(balanceOf(msg.sender) >= value, 'balance too low');
        balances[to] += value;
        balances[msg.sender] -= value;
```

- Here you are checking that balance of from should be bigger or equal to value, and allowance to msg.sender should be bigger or equal to value.



```
    function transferFrom(address from, address to, uint value) public returns(boo
        require(balanceOf(from) >= value, 'balance too low');
        require(allowance[from][msg.sender] >= value, 'allowance too low');
        balances[to] += value;
```

• Critical vulnerabilities found in the contract

=> No Critial vulnerabilities found

• Medium vulnerabilities found in the contract

=> No Medium vulnerabilities found

- ## Low severity vulnerabilities found

  - ### 7.1: Compiler version is not fixed:-

    => In this file you have put “pragma solidity ^0.8.2;” which is not a good way to define compiler version.

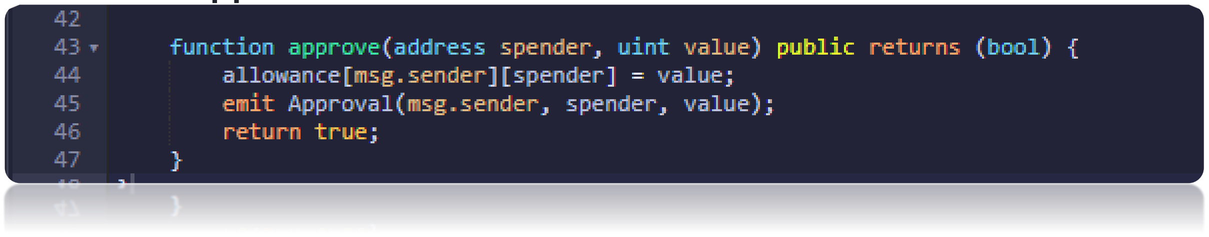
    => Solidity source files indicate the versions of the compiler they can be compiled with. Pragma solidity ^0.8.2; // bad: compiles 0.8.2 and above pragma solidity 0.8.2; //good: compiles 0.8.2 only

    => If you put(>=) symbol then you are able to get compiler version 0.8.2 and above. But if you don’t use(^/>=) symbol then you are able to use only 0.8.2 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.

  - ### 7.2: Approve given more allowance:-

    => I have found that in approve function user can give more allowance to a user beyond their balance.
    => It is necessary to check that user can give allowance less or equal to their amount.
    => There is no validation about user balance. So it is good to check that a user not set approval wrongly.

Function: - approve



```
    function approve(address spender, uint value) public returns (bool) {
        allowance[msg.sender][spender] = value;
        emit Approval(msg.sender, spender, value);
        return true;
    }
```

  - Here you can check that balance of spender should be bigger or equal to value.

• Summary of the Audit

Overall, the code is written with all validation and all security

is implemented. Code is performs well and there is no way to

steal 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:** Code performance and quality is good. Address validation and value validation is done properly.
- **Suggestions:** Please use the static version of solidity, and try to check spender balance in approve method.