SMART CONTRACT AUDIT REPORT

For

Liquid ICP

Prepared By: Kishan Patel

Prepared For: Liquid ICP

Prepared on: 06/11/2021

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 359 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.

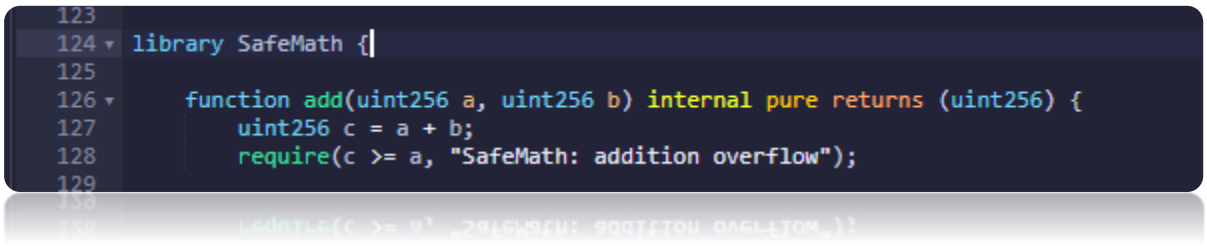
Use of "require" function in this smart contract mitigated this vulnerability.

- ## Forcing Ethereum to a contract

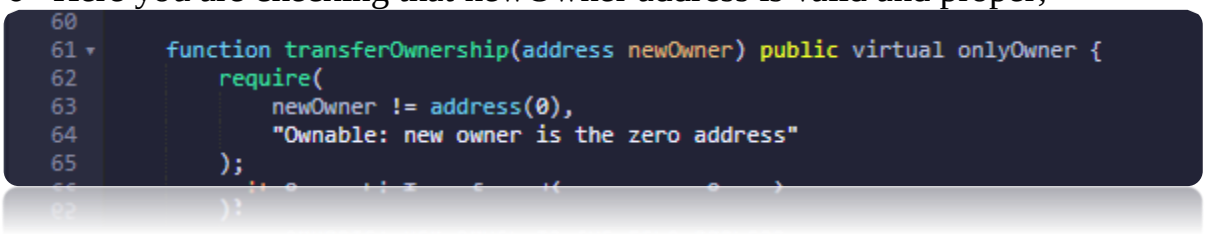
While implementing "selfdestruct" in smart contract, it sends all the ethereum to the target address. Now, if the target address is a contract address, then the fallback function of target contract does not get called. And thus Hacker can bypass the "Required" conditions. Here, the Smart Contract's balance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

• Good things in smart contract

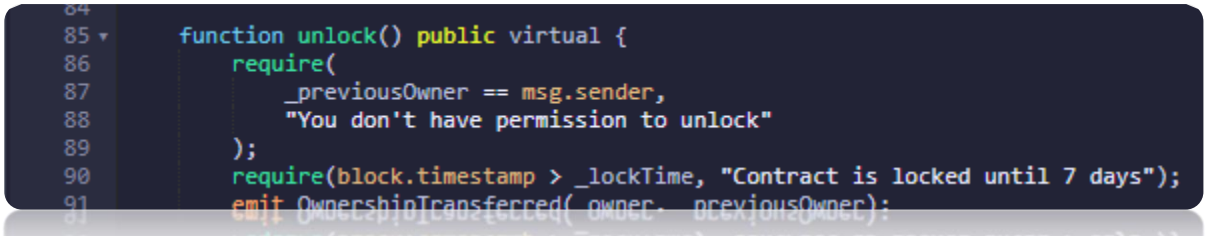
- ### SafeMath library:-
  - You are using SafeMath library it is a good thing. This protects you from underflow and overflow attacks.



- ### Good required condition in functions:-
  - Here you are checking that newOwner address is valid and proper,

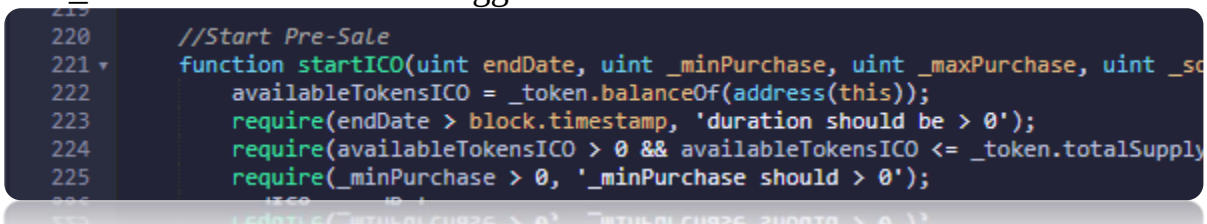


- Here you are checking that previous owner is not msg.sender and _lockTime is smaller than current time.



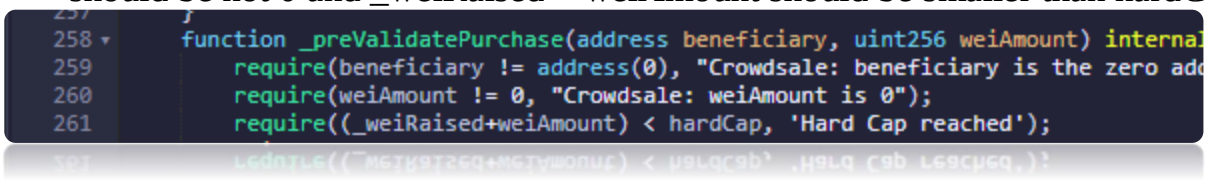
```
function unlock() public virtual {
    require(
        _previousOwner == msg.sender,
        "You don't have permission to unlock"
    );
    require(block.timestamp > _lockTime, "Contract is locked until 7 days");
    emit OwnershipTransferred( owner,  previousOwner);
```

- Here you are checking that endDate is bigger than current time, availableTokensICO should be bigger than 0 and smaller than totalSupply, and _minPurchase should be bigger than 0.



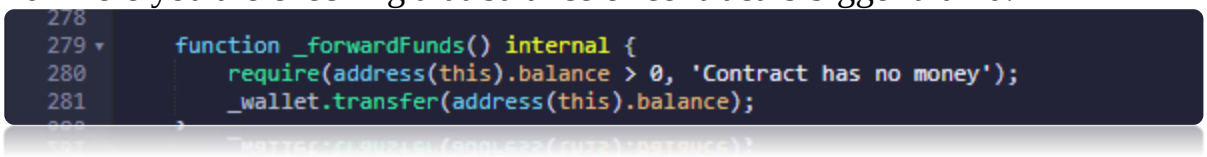
```
//Start Pre-Sale
function startICO(uint endDate, uint _minPurchase, uint _maxPurchase, uint _so
    availableTokensICO = _token.balanceOf(address(this));
    require(endDate > block.timestamp, 'duration should be > 0');
    require(availableTokensICO > 0 && availableTokensICO <= _token.totalSupply
    require(_minPurchase > 0, '_minPurchase should > 0');
```

- Here you are checking that beneficiary address is valid and proper, weiAmount should be not 0 and _weiRaised + weiAmount should be smaller than hardCap.



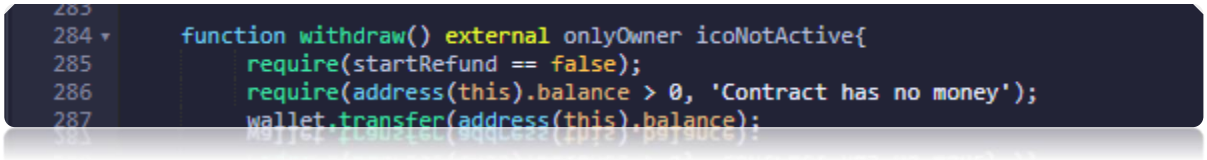
```
function _preValidatePurchase(address beneficiary, uint256 weiAmount) internal
    require(beneficiary != address(0), "Crowdsale: beneficiary is the zero add
    require(weiAmount != 0, "Crowdsale: weiAmount is 0");
    require((_weiRaised+weiAmount) < hardCap, 'Hard Cap reached');
```

- Here you are checking that balance of contract is bigger than 0.



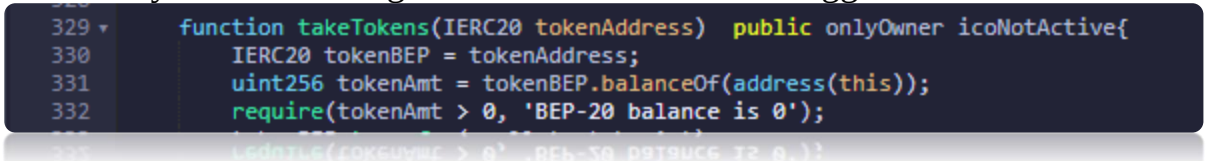
```
function _forwardFunds() internal {
    require(address(this).balance > 0, 'Contract has no money');
    _wallet.transfer(address(this).balance);
```

- Here you are checking that startRefund is false and balance of contract is bigger than 0.



```
function withdraw() external onlyOwner icoNotActive{
     require(startRefund == false);
     require(address(this).balance > 0, 'Contract has no money');
     wallet.transfer(address(this).balance);
```

- Here you are checking that tokenAmt should be bigger than 0.



```
function takeTokens(IERC20 tokenAddress)  public onlyOwner icoNotActive{
    IERC20 tokenBEP = tokenAddress;
    uint256 tokenAmt = tokenBEP.balanceOf(address(this));
    require(tokenAmt > 0, 'BEP-20 balance is 0');
```

- Here you are checking that startRefund should be true.

```
336     function refundMe() public icoNotActive{
337         require(startRefund == true, 'no refund available');
338         uint amount = _contributions[msg.sender];
339         if (address(this).balance >= amount) {
```

• 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 ^0.8.4;” 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 ^0.8.4; // bad: compiles 0.8.4 and above pragma solidity 0.8.4; //good: compiles 0.8.4 only

=> If you put(>=) symbol then you are able to get compiler version 0.8.4 and above. But if you don’t use(^/>=) symbol then you are able to use only 0.8.4 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: Suggestions to add validations:-

=> You have implemented required validation in contract.

=> There are some place where you can improve validation and security of your code.

=> These are all just suggestion it is not bug.

Function: - setRate, setAvailableTokens, setHardCap, setSoftCap, setMaxPurchase, setMinPurchase

```
293
294     function setRate(uint256 newRate) external onlyOwner icoNotActive{
295         _rate = newRate;
296     }
297
298     function setAvailableTokens(uint256 amount) public onlyOwner icoNotActive{
299         availableTokensICO = amount;
300     }
301
```

```
312
313     function setHardCap(uint256 value) external onlyOwner{
314         hardCap = value;
315     }
316
317     function setSoftCap(uint256 value) external onlyOwner{
318         softCap = value;
319     }
320
321     function setMaxPurchase(uint256 value) external onlyOwner{
322         maxPurchase = value;
323     }
324
325     function setMinPurchase(uint256 value) external onlyOwner{
326         minPurchase = value;
327     }
```

- Here in all functions you need to check that parameter value should be bigger than 0.

7.3: Uncheck return response of transfer method:-

=> I have found that you are transferring fund to address using a transfer method.

=> It is always good to check the return value or response from a function call.

=> Here are some functions where you forgot to check a response.

=> I suggest if there is a possibility then please check the response.

- **Function: - _deliverTokens, _forwardFunds, withdraw, takeTokens, refundMe**

```
265     function _deliverTokens(address beneficiary, uint256 tokenAmount) internal {
266         _token.transfer(beneficiary, tokenAmount);
267     }
```

```
279     function _forwardFunds() internal {
280         require(address(this).balance > 0, 'Contract has no money');
281         _wallet.transfer(address(this).balance);
282     }
```

```
284     function withdraw() external onlyOwner icoNotActive{
285         require(startRefund == false);
286         require(address(this).balance > 0, 'Contract has no money');
287         _wallet.transfer(address(this).balance);
288     }
```

```
330         IERC20 tokenBEP = tokenAddress;
331         uint256 tokenAmt = tokenBEP.balanceOf(address(this));
332         require(tokenAmt > 0, 'BEP-20 balance is 0');
333         tokenBEP.transfer(_wallet, tokenAmt);
334     }
```

```
341             if (amount > 0) {
342                 address payable recipient = payable(msg.sender);
343                 recipient.transfer(amount);
344                 emit Refund(msg.sender, amount);
345             }
```

- Here you are calling transferFrom method 5 times. It is good to check that the transfer is successfully done or not.

• 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, try to include suggested validation in code and check return response of transfer and transfer method call.