SMART CONTRACT AUDIT REPORT

For

DePayLiquidityStaking

(Order#FO81AC06C7346)

Prepared By: Kishan Patel **Prepared For**: depayapp

Prepared on: 12/12/2020

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 205 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 Ethereums'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.

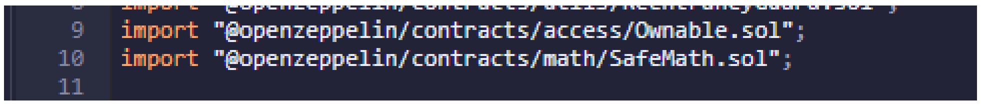
Use of “require” function in this smart contract mitigated this vulnerability.

- ### Forcing Ethereum to a contract

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

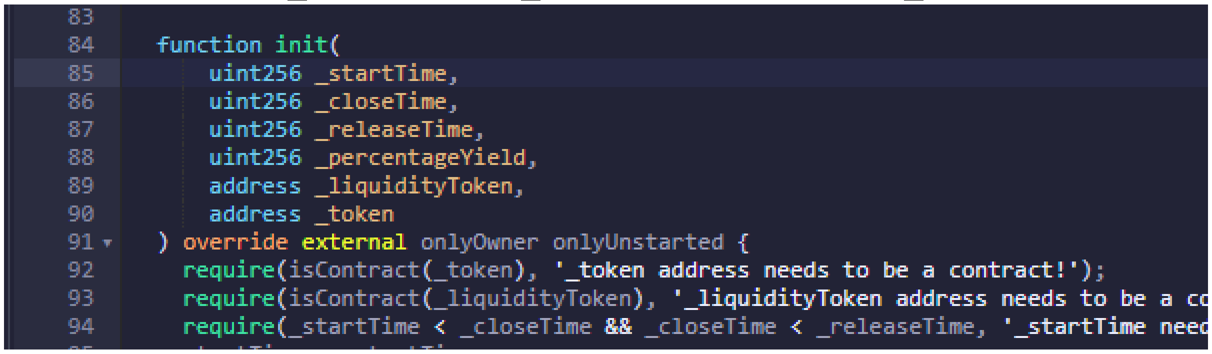
- ## Good things in smart contract

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



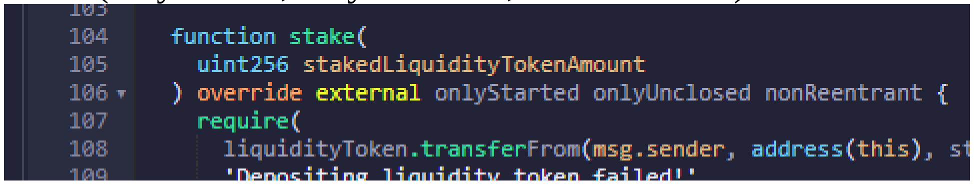
```
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/math/SafeMath.sol";
```

- ### Good required condition in functions:-
  - Here you are checking that _token and _liquidityToken addresses are contract addresses and _startTime and _releaseTime is less than _closeTime.



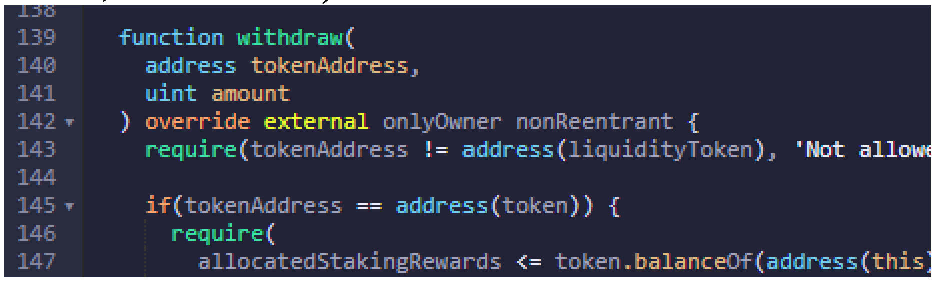
```

  function init(
      uint256 _startTime,
      uint256 _closeTime,
      uint256 _releaseTime,
      uint256 _percentageYield,
      address _liquidityToken,
      address _token
  ) override external onlyOwner onlyUnstarted {
    require(isContract(_token), '_token address needs to be a contract!');
    require(isContract(_liquidityToken), '_liquidityToken address needs to be a co
    require(_startTime < _closeTime && _closeTime < _releaseTime, '_startTime need
```

  - Here you are checking that transferFrom method is called of liquidityToken, reward is calculations is strict to reserved token, allocatedStakingRewards is equal or less than rewardsAmount, and stake function has good allocated modifiers(onlyStarted, onlyUnclosed,nonReeentrant).



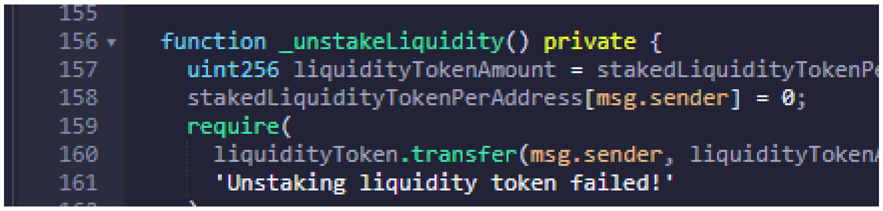
```
  function stake(
    uint256 stakedLiquidityTokenAmount
  ) override external onlyStarted onlyUnclosed nonReentrant {
    require(
      liquidityToken.transferFrom(msg.sender, address(this), st
      'Depositing liquidity token failed!'
```

- Here you are checking that tokenAddress is not liquidityToken, if tokenAddress is token address then allocatedStakingRewwards will be equal or bigger than balance of this contract in token, and good has good modifiers allocated(onlyOwner, nonReentrant).



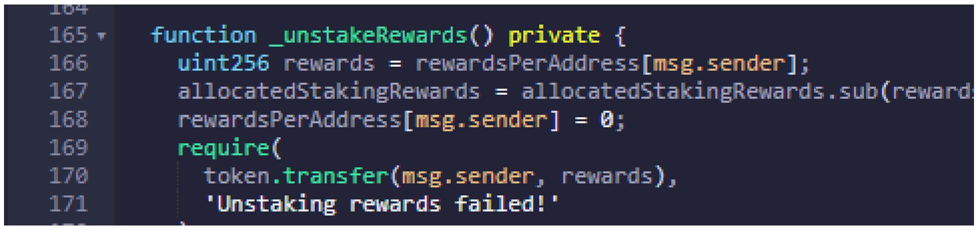
```
  function withdraw(
    address tokenAddress,
    uint amount
  ) override external onlyOwner nonReentrant {
    require(tokenAddress != address(liquidityToken), 'Not allowe

    if(tokenAddress == address(token)) {
      require(
        allocatedStakingRewards <= token.balanceOf(address(this)
```

- Here you are checking that liquidityTokenAmount is tnrasfer to liquidityToken or not if it is not transfer then failed the execution.



```
  function _unstakeLiquidity() private {
    uint256 liquidityTokenAmount = stakedLiquidityTokenPe
    stakedLiquidityTokenPerAddress[msg.sender] = 0;
    require(
      liquidityToken.transfer(msg.sender, liquidityTokenA
      'Unstaking liquidity token failed!'
```

- Here you are checking that token is successfully transfer to the reward address if not then fail the execution.



```
  function _unstakeRewards() private {
    uint256 rewards = rewardsPerAddress[msg.sender];
    allocatedStakingRewards = allocatedStakingRewards.sub(reward
    rewardsPerAddress[msg.sender] = 0;
    require(
      token.transfer(msg.sender, rewards),
      'Unstaking rewards failed!'
```

• Critical vulnerabilities found in the contract

=> No Critial vulnerabilities found

• Medium vulnerabilities found in the contract

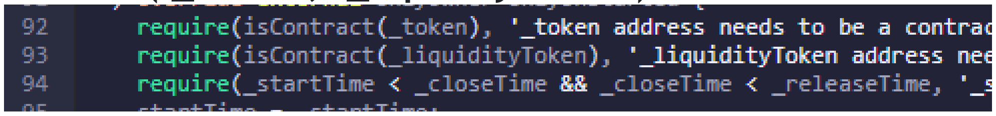
=> No Medium vulnerabilities found

• Low severity vulnerabilities found

○ 7.1: Short address attack:-

=> This is not big issue in solidity, because now a days is 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 address parameter.

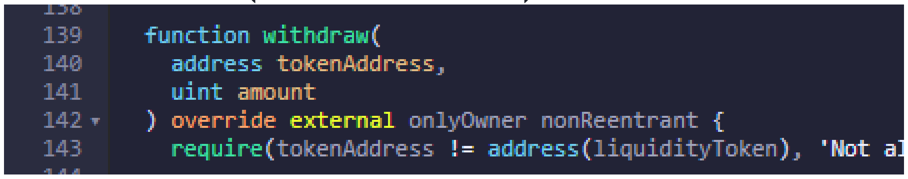
Function:- init (‘_token’, ‘_liquidityToken’)



```
    require(isContract(_token), '_token address needs to be a contrac
    require(isContract(_liquidityToken), '_liquidityToken address nee
    require(_startTime < _closeTime && _closeTime < _releaseTime, '_s
```

- It's necessary to check the addresses value of "_token","_liquidityToken". Because here you are passing whatever variable comes in "_token","_liquidityToken" addresses from outside.

Function: - withdraw (‘tokenAddress’)



```
  function withdraw(
    address tokenAddress,
    uint amount
  ) override external onlyOwner nonReentrant {
    require(tokenAddress != address(liquidityToken), 'Not al
```

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

○ 7.2: Compiler version is not fixed:-

=> In this file you have put “pragma solidity ^0.7.5;” 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 >=0.7.5; // bad: compiles 0.7.5 and above pragma solidity 0.7.5; //good: compiles 0.7.5 only

=> If you put(>=) symbol then you are able to get compiler version 0.7.5 and above. But if you don’t use(^/>=) symbol then you are able to use only 0.7.5 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.

7.3: Unchecked return value or response:-

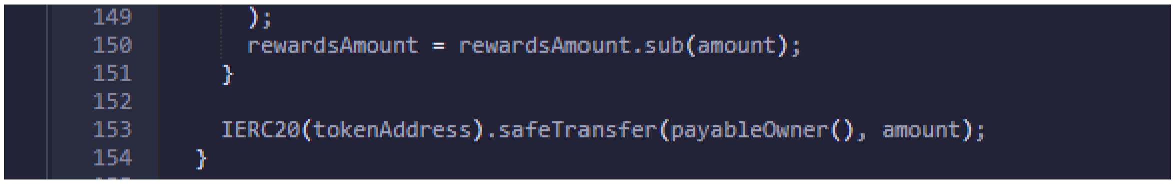
=> I have found that you are transferring fund to address using a transfer method.

=> 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: - withdraw



```
            );
            rewardsAmount = rewardsAmount.sub(amount);
        }

        IERC20(tokenAddress).safeTransfer(payableOwner(), amount);
    }
```

- Here you are calling safeTransfer method 1 time. It is good to check that the transfer is successfully done or not.

• Summary of the Audit

Overall the code is well and performs well.

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 ;)).

- **Note:** Please focus check addresses, and return response (at 1 place).
- I have seen that a developer is using block’s timestamp and now method so, I like to tell you that write smart contracts with the notion that block values are not precise, and the use of them can lead to unexpected effects.