SMART CONTRACT AUDIT REPORT

For
Recurring Income Token

Prepared By: Kishan Patel

Prepared For: REIT Finance Labs

Prepared on: 21/10/2021

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 6 files. It contains approx 1300 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.

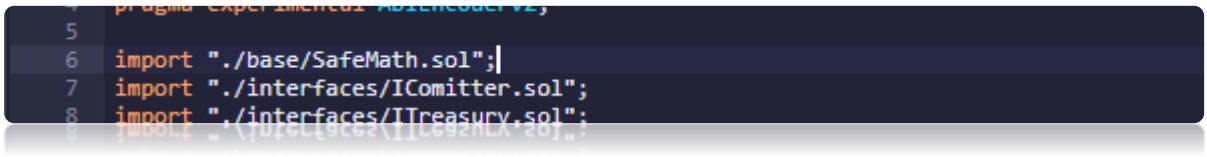
Use of "require" function in this smart contract mitigated this vulnerability.

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



- **Compiler version is fixed:-**
  - **Filename:- All Files**

=> In this file you have put "pragma solidity 0.6.12;" 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 ^0.6.12; // bad: compiles 0.6.12 and above pragma solidity 0.6.12; //good: compiles 0.6.12 only

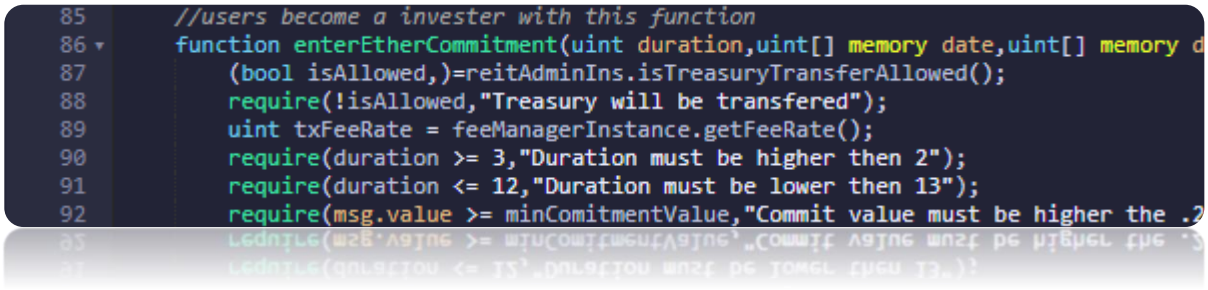
=> If you put(>=) symbol then you are able to get compiler version 0.6.12 and above. But if you don't use(^/>=) symbol then you are able to use only 0.6.12 version. And if there are some changes come in the compiler and you use the old version then some issues may come at deploy time.

=> Try to use latest version of solidity.

- ## Good required condition in functions:-

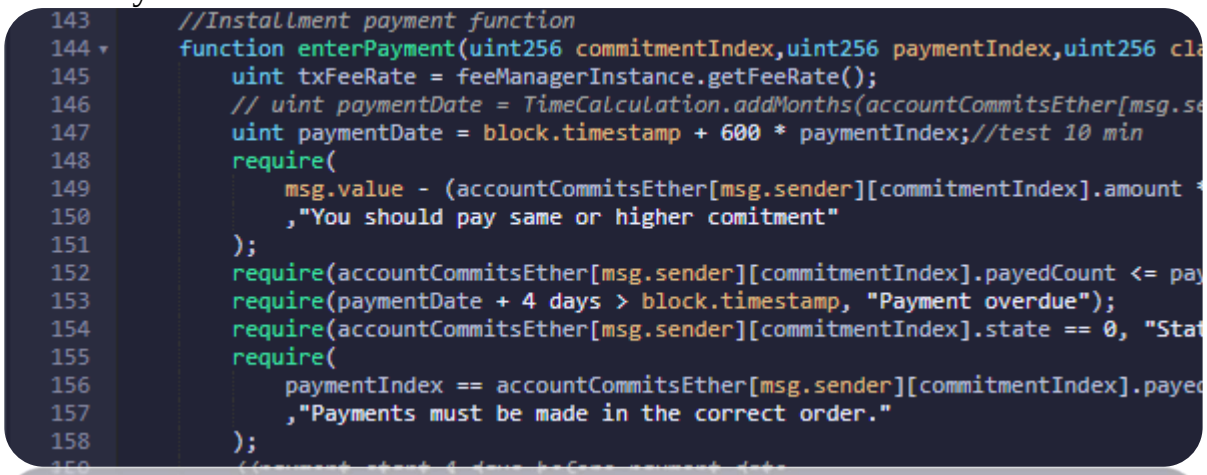
❖ Filename:- Comitter.sol

- Here you are checking that treasury transfer is active or not, duration should be between 3 to 12 and msg.value should be bigger or equal to minComitementValue.



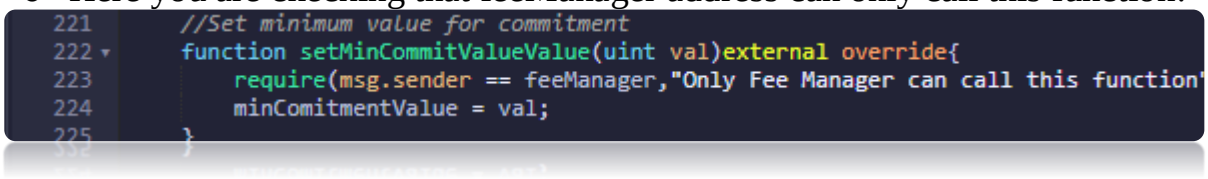
```
85      //users become a invester with this function
86      function enterEtherCommitment(uint duration,uint[] memory date,uint[] memory d
87          (bool isAllowed,)=reitAdminIns.isTreasuryTransferAllowed();
88          require(!isAllowed,"Treasury will be transfered");
89          uint txFeeRate = feeManagerInstance.getFeeRate();
90          require(duration >= 3,"Duration must be higher then 2");
91          require(duration <= 12,"Duration must be lower then 13");
92          require(msg.value >= minComitmentValue,"Commit value must be higher the .2
```

- Here you are checking that correct commitment should be paid, commitment should not be already paid, Payment is not overdue, commitment state should not be failed, payment is in correct order or not, payment should accepted before 4 days.



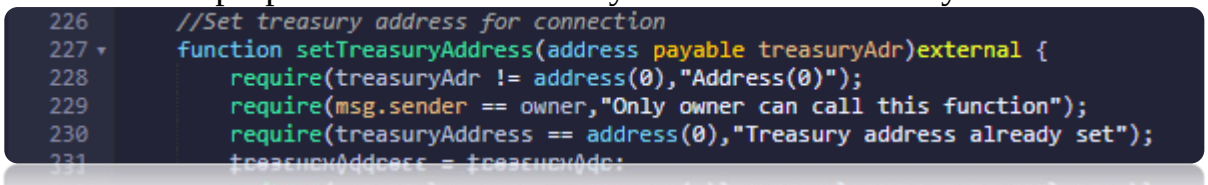
```
143     //Installment payment function
144     function enterPayment(uint256 commitmentIndex,uint256 paymentIndex,uint256 cla
145         uint txFeeRate = feeManagerInstance.getFeeRate();
146         // uint paymentDate = TimeCalculation.addMonths(accountCommitsEther[msg.se
147         uint paymentDate = block.timestamp + 600 * paymentIndex;//test 10 min
148         require(
149             msg.value - (accountCommitsEther[msg.sender][commitmentIndex].amount *
150             ,"You should pay same or higher comitment"
151         );
152         require(accountCommitsEther[msg.sender][commitmentIndex].payedCount <= pay
153         require(paymentDate + 4 days > block.timestamp, "Payment overdue");
154         require(accountCommitsEther[msg.sender][commitmentIndex].state == 0, "Stat
155         require(
156             paymentIndex == accountCommitsEther[msg.sender][commitmentIndex].payed
157             ,"Payments must be made in the correct order."
158         );
```

- Here you are checking that feeManager address can only call this function.



```
221     //Set minimum value for commitment
222     function setMinCommitValueValue(uint val)external override{
223         require(msg.sender == feeManager,"Only Fee Manager can call this function"
224         minComitmentValue = val;
225     }
```

- Here you are checking that only owner can call this function, treasuryAdr is valid and proper address and treasuryAddress is not already set.



```
226     //Set treasury address for connection
227     function setTreasuryAddress(address payable treasuryAdr)external {
228         require(treasuryAdr != address(0),"Address(0)");
229         require(msg.sender == owner,"Only owner can call this function");
230         require(treasuryAddress == address(0),"Treasury address already set");
```

- Here you are checking that only owner can call this function and commite state is not failed.

```
function updateCommitState(address acc,uint indexFirst) external {
    require(accountCommitsEther[acc][indexFirst].state == 0,"State Fail");
    require(msg.sender == owner,"Only owner can call this function");
```

- Here you are checking that only treasuryAddress can call this method.

```
function changeRewardClaimedStatus(address acc,uint commitIndex,uint paymentI
    require(msg.sender == treasuryAddress,"Only Treasury Contract can call thi
    accountCommitsEther[acc][commitIndex].defiProtocolValues[paymentIndex].is
    accountCommitsEther[acc][commitIndex].defiProtocolValues[paymentIndex].cl
```

❖ Filename:- ReitCoin.sol

- Here you are checking that treasuryAdr address is valid and proper, and contractOwner address can only call this function.

```
function setTreasuryAddress(address treasuryAdr)external {
    require(treasuryAdr != address(0),"Address(0)");
    require(msg.sender == contractOwner,"Only owner can call this function");
    treasuryAddress = treasuryAdr;
```

- Here you are checking that only treasuryAddress address can only call this function.

```
function mint(address account,uint amount)external override{
    require(msg.sender == treasuryAddress,"Only treasury contract can call thi
    _mint(account,amount);
```

- Here you are checking that sender and recipient addresses value are valid and proper.

```
function _transfer(address sender, address recipient, uint256 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 account address value is valid and proper.

```
function _mint(address account, uint256 amount) internal virtual {
    require(account != address(0), "ERC20: mint to the zero address");
```

```
function _burn(address account, uint256 amount) internal virtual {
    require(account != address(0), "ERC20: burn from the zero address");
```

- Here you are checking that owner and spender addresses value are valid and proper.

```
133     function _approve(address owner, address spender, uint256 amount) internal vir
134         require(owner != address(0), "ERC20: approve from the zero address");
135         require(spender != address(0), "ERC20: approve to the zero address");
```

❖ Filename:- ReitGCoin.sol

- Here you are checking that treasuryAdr is proper and valid address and only contract owner can call this function.

```
53
54      function setTreasuryAddress(address treasuryAdr)external {
55          require(treasuryAdr != address(0),"Adress(0)");
56          require(msg.sender == contractOwner,"Only owner can call this function");
57          treasuryAddress = treasuryAdr;
```

- Here you are checking that only owner of contract or treasuryAddress can call this function and _totalSupply should not be bigger than _maxSupply.

```
85
86      function mint(address account,uint amount)external override{
87          require(
88              msg.sender == treasuryAddress
89              ||
90              msg.sender == contractOwner,"Only treasury contract or owner can call
91              );
92          require(_maxSupply > _totalSupply,"Reached to maximum reitg supply");
```

- Here you are checking that sender and recipient addresses value are valid and proper.

```
106     function _transfer(address sender, address recipient, uint256 amount) internal
107         require(sender != address(0), "ERC20: transfer from the zero address");
108         require(recipient != address(0), "ERC20: transfer to the zero address");
```

- Here you are checking that account address value is valid and proper.

```
118     function _mint(address account, uint256 amount) internal virtual {
119         require(account != address(0), "ERC20: mint to the zero address");
120
```

```
126     }
127
128     function _burn(address account, uint256 amount) internal virtual {
129         require(account != address(0), "ERC20: burn from the zero address");
130
```

- Here you are checking that owner and spender addresses value are valid and proper.

```
138     function _approve(address owner, address spender, uint256 amount) internal vir
139         require(owner != address(0), "ERC20: approve from the zero address");
140         require(spender != address(0), "ERC20: approve to the zero address");
```

❖ Filename:- FeeManager.sol

- Here you are checking that only comitterAddress or depositorAddress can call this function.

```
56      //update collected txfee every commit payment or deposit
57      function txFeeProcess(uint txFee) external override{
58          //only depo and comitter
59          require(msg.sender == address(comitterAddress) || msg.sender == address(de
60          totalCollectedTxFee += txFee;
```

- Here you are checking that only treasuryAddress can call this function.

```
63      function managmentFeeProcess(uint fee) external override {
64          //only treasury
65          require(msg.sender == treasuryAddress,"Only owner can call this function");
66          totalCollectedManagmentFee += fee;
```

- Here you are checking that comitterAdr should be valid and proper, comitterAddress should be not already set.

```
68      //Set comitter address for connection
69      function setComitterAddress(address payable comitterAdr)external onlyAdmins{
70          require(comitterAdr != address(0),"Address(0)");
71          require(comitterAddress == address(0),"Comitter address already set");
```

- Here you are checking that depositorAdr value is valid and proper, depositorAddress is not already set.

```
75      //Set depositor address for connection
76      function setDepositorAddress(address payable depositorAdr)external onlyAdmins{
77          require(depositorAdr != address(0),"Address(0)");
78          require(depositorAddress == address(0),"Depositor address already set");
79          depositorAddress = depositorAdr;
```

- Here you are checking that treasuryAdr value is valid and proper, treasuryAddress is not already set.

```
82      //Set treasury address for connection
83      function setTreasuryAddress(address payable treasuryAdr)external onlyAdmins {
84          require(treasuryAdr != address(0),"Address(0)");
85          require(treasuryAddress == address(0),"Treasury address already set");
86          treasuryx = ITreasuryx(treasuryAdr);
```

- Here you are checking that acc should be valid and proper address, treasuryAddress can call this function.

```
89      //send txfeereward to user
90      function sendTxFeeReward(uint value,address payable acc) external override{
91          require(acc!=address(0),"Adress(0)");
92          require(msg.sender == treasuryAddress);
93          uint feeVal = value * feeRate / 1000;
```

❖ Filename:- Depositor.sol

- Here you are checking that treasury transfer is active or not and msg.value should be bigger or equal to minDepositValue.

```
56      //Deposit function for user
57      function enterDeposit(uint256 claimDate,uint[] memory dateForBuyBack) external
58          (bool isAllowed,)=reitAdminIns.isTreasuryTransferAllowed();
59          require(!isAllowed,"Treasury will be transfered");
60          uint txFeeRate = feeManagerInstance.getFeeRate();
61          require(msg.value >= minDepositValue ,"Deposit value must be higher or equa
```

- Here you are checking that only owner can call this function, treasuryAdr is valid and proper address and treasuryAddress is not already set.

```
97      //Treasury Connection
98      function setTreasuryAddress(address payable treasuryAdr)external {
99          require(treasuryAdr != address(0),"Address(0)");
100         require(msg.sender == owner,"Only owner can call this function");
101         require(treasuryAddress == address(0),"Treasury address already set");
102         treasuryAddress = treasuryAdr;
```


- Here you are checking that only owner can call this function, treasuryAdr is valid and proper address and treasuryAddress is not already set.


```
97      //Treasury Connection
98      function setTreasuryAddress(address payable treasuryAdr)external {
99          require(treasuryAdr != address(0),"Address(0)");
100         require(msg.sender == owner,"Only owner can call this function");
101         require(treasuryAddress == address(0),"Treasury address already set");
102         treasuryAddress = treasuryAdr;
```

- Here you are checking that only feeManager address can call this function. Please check val should be bigger than 0.

```
105     //Set minimum value for enter deposit
106     function setMinDepositValue(uint val)external override{
107         require(msg.sender == feeManager,"Only Fee Manager can call this function"
108         minDepositValue = val;
```

- Here you are checking that only treasuryAddress can call this method.

```
136     }
137     // if user claim his/her reward this function will update their state
138     function changeRewardClaimedStatus(address acc,uint index,bool isClaim,uint re
139         require(msg.sender == treasuryAddress,"Only Treasury Contract can call thi
140         accountAllDeposits[acc][index].isClaimed = isClaim;
```

• 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: Suggestions to add code 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.

    ❖ **Filename:- ReitAdmin.sol**

    Function: - allowSetLimit, allowTreasuryTransfer,

```
78      //SetSign allowance
79      function allowSetLimit(bool newState)external onlyAdmins{
80          setLimitSigns[msg.sender]=newState;
81      }
```

```
170     /* TREASURY TRANSFER ALLOWANCE */
171     function allowTreasuryTransfer(bool state)external onlyAdmins{
172         treasuryAllowance[msg.sender]=state;
173     }
```

    - In allowSetLimit function you need to check that msg.sender has not already same value set as newState. If newState value is already same for msg.sender then don't run this function.

    - In allowTreasuryTransfer you need to check that same state value is not set to msg.sender if it is same then don't run this function.

7.2: Uncheck return response of transfer method:-

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

=> 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.

❖ Filename:- Comitter.sol

- **Function: - enterEtherCommitment, enterPayment**

```
104
105         treasuryAddress.transfer(msg.value-feeValue);
106         treasury.collateralValueProcess(colleteralValue,monthlyColleteralAmount);
107
```

```
137         treasury.buyBackAdd(commitValHolder.commitmentBuyBackValue);
138         treasury.sendReitToUser(msg.sender,commitmentValue);
139         // treasury.setMonthlyCollateral(monthlyColleteralAmount);
140         feeManager.transfer(feeValue);
141         feeManagerInstance.txFeeProcess(feeValue);
```

```
186         // uint paymentBuyBackValue = amount.mul(70).div(100).div(24);
187
188         treasuryAddress.transfer(msg.value-feeValue);
189
```

```
198         accountCommitsEther[msg.sender][commitmentIndex].defiProtocolValues.push(n
199
200         defiComitterTotalEthAtIndex[lendingProtocolIndex] =  defiComitterTotalEthA
201         feeManager.transfer(feeValue);
202         feeManagerInstance.txFeeProcess(feeValue);
```

  - Here you are calling transfer method 4 times. It is good to check that the transfer is successfully done or not.

❖ Filename:- FeeManager.sol

- **Function: - sendTxFeeReward**

```
94          uint txFeeReward = feeVal * feeRePaymentRate / txFeeStabil;
95          totalCollectedTxFee = totalCollectedTxFee.sub(txFeeReward);
96          acc.transfer(txFeeReward);
```

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

7.3: 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.

❖ Filename:- ReitCoin.sol

Function: - _approve

```
132
133     function _approve(address owner, address spender, uint256 amount) internal vi
134         require(owner != address(0), "ERC20: approve from the zero address");
135         require(spender != address(0), "ERC20: approve to the zero address");
136
137         _allowances[owner][spender] = amount;
138         emit Approval(owner, spender, amount);
```

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

❖ Filename:- ReitGCoin.sol

Function: - _approve

```
137
138     function _approve(address owner, address spender, uint256 amount) internal vi
139         require(owner != address(0), "ERC20: approve from the zero address");
140         require(spender != address(0), "ERC20: approve to the zero address");
141
142         _allowances[owner][spender] = amount;
143         emit Approval(owner, spender, amount);
144     }
```

- Here you can check that balance of owner should be bigger or equal to amount 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:** You are paying attention to the visibility of the functions. Contract is nit perfect size with good validation. Code performance and quality is good. Address validation and value validation is done properly.
- **Suggestions:** Please try to add suggested code validations, check return response of transfer method, check user balance in approve function.