1. **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.

1. **Overview of the audit**
2. **Attack made to the contract**

### **Over and under flows**

An overflow happens when the limit of the type varibale uint256 , 2\*\*256, is exceeded. What happens is that the value resets to zero instead of incrementing more.

Attack noted: MulticoinICO Line 124: totalEtherRaised += msg.value;

1. **Critical vulnerabilites found in the contract**

There aren’t critical issues in the smart contract audited.

1. **Medium vulnerabilites found in the contract**

Avoid expensive gas executions.

1. **Low severity vulnerabilites found**

* You are using require() instead of assert() in most cases.

Assert and require behave almost identically but the assert function is used to validate contract state after making changes, while require is normally used at the top of the functions to verify the input of the function.

* Use the latest version of solidity to enforce the compiler and prevent from any previous bugs/flaws.

The current version is 0.4.21

1. **Line by line comments**

**ERC20 Implementation** (Multicoin.sol)

pragma solidity ^0.4.19;

import "./lib/Owned.sol";

import "./lib/SafeMath.sol";

import "./lib/ERC20Interface.sol";

import "./lib/ApproveAndCallFallBack.sol";

contract Multicoin is Owned, ERC20Interface {

/\* Avoiding overflows at all costs :) \*/

using SafeMath for uint;

/\* ERC20 Attributes \*/

mapping(address => uint256) balances;

mapping(address => mapping (address => uint256)) allowed;

mapping(address => bool) public freezeBypassing;

string public symbol = 'MTC';

string public name = 'Multicoin';

uint8 public decimals = 18;

uint public \_totalSupply = 2000000000 \* 10\*\*uint(decimals);

uint public \_circulatingSupply = 0;

bool public tradingLive = false;

function distributeSupply(address to, uint tokens) public onlyOwner returns (bool success) {

uint tokenAmount = tokens.mul(10\*\*uint(decimals));

require(\_circulatingSupply.add(tokenAmount) <= \_totalSupply);

\_circulatingSupply = \_circulatingSupply.add(tokenAmount);

balances[to] = tokenAmount;

return true;

}

function allowFreezeBypass(address sender) public onlyOwner returns (bool success) {

freezeBypassing[sender] = true;

return true;

}

function setTradingLive() public onlyOwner returns (bool tradingStatus) {

tradingLive = true;

return tradingLive;

}

modifier tokenTradingMustBeLive(address sender) {

require(tradingLive || freezeBypassing[sender]);

\_;

}

/\* ERC20 Implementation \*/

function totalSupply() public constant returns (uint) {

return \_totalSupply;

}

function balanceOf(address tokenOwner) public constant returns (uint balance) {

return balances[tokenOwner];

}

function transfer(address to, uint tokens) public tokenTradingMustBeLive(msg.sender) returns (bool success) {

balances[msg.sender] = balances[msg.sender].sub(tokens);

balances[to] = balances[to].add(tokens);

Transfer(msg.sender, to, tokens);

return true;

}

function transferFrom(address from, address to, uint tokens) public tokenTradingMustBeLive(from) returns (bool success) {

balances[from] = balances[from].sub(tokens);

allowed[from][msg.sender] = allowed[from][msg.sender].sub(tokens);

balances[to] = balances[to].add(tokens);

Transfer(from, to, tokens);

return true;

}

function approve(address spender, uint tokens) public returns (bool success) {

allowed[msg.sender][spender] = tokens;

Approval(msg.sender, spender, tokens);

return true;

}

function allowance(address tokenOwner, address spender) public constant returns (uint remaining) {

return allowed[tokenOwner][spender];

}

/\* End of default ERC20 Implementation \*/

/\* trigger the receiveApproval(...) on spender contract \*/

function approveAndCall(address spender, uint tokens, bytes data) public returns (bool success) {

allowed[msg.sender][spender] = tokens;

Approval(msg.sender, spender, tokens);

ApproveAndCallFallBack(spender).receiveApproval(msg.sender, tokens, this, data);

return true;

}

/\* Owner can transfer out any accidentally sent ERC20 tokens \*/

function transferAnyERC20Token(address tokenAddress, uint tokens) public onlyOwner returns (bool success) {

return ERC20Interface(tokenAddress).transfer(owner, tokens);

}

}

**1. Line 14 the allowed mapping**

Based on the last version of the ERC20 (available here on the EIP <https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md>), allowed mapping should be set to **internal**

**2. Line 10 SafeMath library**

We recommend activating the library for uint256 to protect from overflows.

**using SafeMath for uint256;**

**3. Line 20 totalSupply declaration**

The maximum number of tokens is the maximum amount a uint256 variable can store:

(2\*\*256) – 1 with 0 decimals. You use 18 decimals so the maximum amount of tokens will be: Max amount = (2^256-1)/10^18.

We recommend the use of uint256 for the totalSupply.

For the initialization of this variable, we recommend using a library like the OpenZeppelin’s [SafeMath.sol](https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/math/SafeMath.sol). It’ll help make secure calculations without the risk of under or over flows. The way you use it is by importing the library, activating it for uint256 and then using the function .mul()

**4. Line 48, Line 52 & Line 77 Function totalSupply, balanceOf & allowance**

As of [**Solc 0.4.17**](https://github.com/ethereum/solidity/releases/tag/v0.4.17), two new function modifiers have been introduced in lieu of **constant**-- they are **view** and **pure**. These are exciting additions that offer us the chance to write more expressive contracts.

So to be compliant ERC20 and to adopt the last version of this standard (available here on the [EIP](https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md)), we recommend to replace the **constant** modifier by **view** like below:

function totalSupply() public **view** returns (uint256) {

return totalSupply\_;

}

function balanceOf(address \_owner) public **view** returns (uint256 balance) {

return balances[\_owner];

}

function allowance(address \_owner, address \_spender) public view returns (uint256) {

return allowed[\_owner][\_spender];

}

**5. Line 56 Function transfer**

The transfer function is not preventing transfer of tokens to the 0x0 address.

At the time of writing, the “[zero](https://etherscan.io/address/0x0000000000000000000000000000000000000000)” address holds tokens with the value of **$1,051,574,608.71.**To prevent that, add this requirement:

**require(\_to != address(0));**

Consider also preventing the transfer of tokens to the same address of the smart contract.

An example of the potential for loss by leaving this open is the [EOS token smart contract](https://etherscan.io/address/0x86fa049857e0209aa7d9e616f7eb3b3b78ecfdb0) where more than 90,000 tokens are stuck at the contract address.

To prevent that, add this requirement:

**require(\_to != address(this));**

Consider also the check of the sender balance, by adding the requirement below:

**require(\_value <= balances[msg.sender]);**

**6. Line 63 Function transferFrom**

Same case of the previous function. We recommend the add of this requirements :

**require(\_to != address(0));** //prevent transferring tokens to the 0x0 address

**require(\_value <= balances[\_from]); //**check the balance of the sender

**require(\_value <= allowed[\_from][msg.sender]);//**check the allowance

Inverse the line 65 and line 66 (based on the last ERC20 version) to be sure that the setting of balances was successful. Once it’s ok we can update the allowed value.

**Library Implementation** (SafeMath.sol)

pragma solidity ^0.4.19;

library SafeMath {

function add(uint a, uint b) internal pure returns (uint c) {

c = a + b;

require(c >= a);

}

function sub(uint a, uint b) internal pure returns (uint c) {

require(b <= a);

c = a - b;

}

function mul(uint a, uint b) internal pure returns (uint c) {

c = a \* b;

require(a == 0 || c / a == b);

}

function div(uint a, uint b) internal pure returns (uint c) {

require(b > 0);

c = a / b;

}

}

This version is not the last one. So to prevent bugs and errors, we recommend the use of the openZeppelin library available [here](https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/math/SafeMath.sol).

**ERC20Interface.sol**

pragma solidity ^0.4.19;

contract ERC20Interface {

function totalSupply() public constant returns (uint);

function balanceOf(address tokenOwner) public constant returns (uint balance);

function allowance(address tokenOwner, address spender) public constant returns (uint remaining);

function transfer(address to, uint tokens) public returns (bool success);

function approve(address spender, uint tokens) public returns (bool success);

function transferFrom(address from, address to, uint tokens) public returns (bool success);

event Transfer(address indexed from, address indexed to, uint tokens);

event Approval(address indexed tokenOwner, address indexed spender, uint tokens);

}

**1. Line 4, line 5 & line 6**

Based on the change made on the top of Muticoin.sol (Line 48, Line 52 & Line 77). You should replace **constant** modifier by **view** (line 4, line 5 & line 6)to have a coherent interface.

**Owned.sol**

pragma solidity ^0.4.19;

contract Owned {

address public owner;

address public newOwner;

event OwnershipTransferred(address indexed \_from, address indexed \_to);

function Owned() public {

owner = msg.sender;

}

modifier onlyOwner {

require(msg.sender == owner);

\_;

}

function transferOwnership(address \_newOwner) public onlyOwner {

newOwner = \_newOwner;

}

function acceptOwnership() public {

require(msg.sender == newOwner);

OwnershipTransferred(owner, newOwner);

owner = newOwner;

newOwner = address(0);

}

}

No relevant issue on this contract.

The openZeppelin's [ownable.sol](https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/ownership/Ownable.sol) standard was not chosen as the basis for development. Therefore, it will be necessary to ensure the updating of future recommendations if this contract is brought to be reused.

**PelikanIco.sol**

pragma solidity ^0.4.19;

contract PelikanIco {

/\* Pelikan compatibility : Event to raise when address contributes \*/

event AddressDeposited (

address indexed depositor,

uint depositedAt,

uint amount,

uint tokenAmount,

bool indexed boughtOnBehalf

);

/\* Current distribution information \*/

function distributionInfo() public constant returns (

uint minContrib,

uint maxContrib,

uint currentTokenPrice,

uint currentBonus,

uint remainingSupply

);}

**1. Line 14 Function distributionInfo**

As of [**Solc 0.4.17**](https://github.com/ethereum/solidity/releases/tag/v0.4.17), two new function modifiers have been introduced in lieu of **constant**-- they are **view** and **pure**. These are exciting additions that offer us the chance to write more expressive contracts.

So, we recommend to replace the **constant** modifier by **view** like below:

function distributionInfo() public **view** returns

**KycRegistryInterface.sol**

pragma solidity ^0.4.19;

contract KycRegistryInterface {

event kycStatusChanged(address indexed \_address, uint changeTimestamp, bool toStatus);

function kycStatusSet(address \_address, bool \_newKycStatus) public;

function isAddressCleared(address \_address) public constant returns (bool);

}

**1. Line 6**

Like described above, we recommend to replace the constant modifier by view modifier like below:

function isAddressCleared(address \_address) public **view** returns (bool);

**KYC Implementation** (KycRegistry.sol)

pragma solidity ^0.4.19;

import "./lib/KycRegistryInterface.sol";

contract KycRegistry is KycRegistryInterface {

address public oracleAddress;

mapping (address => bool) kycClearances;

function KycRegistry(address \_oracleAddress) public {

oracleAddress = \_oracleAddress;

}

modifier onlyOracle() {

require(msg.sender == oracleAddress);

\_;

}

function kycStatusSet(address \_address, bool \_newKycStatus) public onlyOracle() {

kycClearances[\_address] = \_newKycStatus;

kycStatusChanged(\_address, block.timestamp, \_newKycStatus);

}

function updateOracleAddress(address \_newOracleAddress) public onlyOracle() {

oracleAddress = \_newOracleAddress;

}

function isAddressCleared(address \_address) public constant returns (bool) {

return kycClearances[\_address];

}

}

**1. Line 27 function isAddressCleared**

Based on modifications made above. We should replace the modifier constant by view like below:

function isAddressCleared(address \_address) public **view** returns (bool)

**The ICO Implementation** (MultivenICO.sol)

pragma solidity ^0.4.19;

import "./lib/Owned.sol";

import "./lib/SafeMath.sol";

import "./lib/ERC20Interface.sol";

import "./lib/PelikanIco.sol";

import "./lib/KycRegistryInterface.sol";

contract MultivenIco is Owned, PelikanIco {

using SafeMath for uint;

struct IcoRound {

uint minContribution;

uint maxContribution;

uint supplyAllowed;

uint supplyDistributed;

uint bonusAllocation;

}

event RemainingTokensSent(uint tokenSent, uint timestamp);

mapping (uint8 => IcoRound) icoRounds;

mapping (address => mapping(uint8 => uint)) totalCountributed;

uint8 public currentIcoRound = 0;

uint public tokenPrice = 0.01 ether;

uint public totalEtherRaised = 0;

ERC20Interface multicoin;

KycRegistryInterface kycRegistry;

address public multivenWallet;

modifier mustBeKycCleared(address \_address) {

require(kycRegistry.isAddressCleared(\_address));

\_;

}

/\* Pelikan Ico API : Method to implement \*/

function distributionInfo() public constant returns (

uint minContrib,

uint maxContrib,

uint currentTokenPrice,

uint currentBonus,

uint remainingSupply

) {

minContrib = icoRounds[currentIcoRound].minContribution;

maxContrib = icoRounds[currentIcoRound].maxContribution;

currentTokenPrice = tokenPrice;

currentBonus = icoRounds[currentIcoRound].bonusAllocation;

remainingSupply = icoRounds[currentIcoRound].supplyAllowed.sub(icoRounds[currentIcoRound].supplyDistributed);

}

function MultivenIco(

address kycRegistryAddress,

address multicoinAddress,

address multivenWalletAddress,

uint round1Supply,

uint round2supply,

uint round3Supply,

uint round4Supply

) public {

kycRegistry = KycRegistryInterface(kycRegistryAddress);

multicoin = ERC20Interface(multicoinAddress);

multivenWallet = multivenWalletAddress;

icoRounds[1] = IcoRound(

100 finney,

1000 ether,

round1Supply\*(10\*\*18),

0,

125

);

icoRounds[2] = IcoRound(

100 finney,

500 ether,

round2supply\*(10\*\*18),

0,

120

);

icoRounds[3] = IcoRound(

100 finney,

250 ether,

round3Supply\*(10\*\*18),

0,

115

);

icoRounds[4] = IcoRound(

100 finney,

50 ether,

round4Supply\*(10\*\*18),

0,

100

);

}

function() public mustBeKycCleared(msg.sender) payable {

// Checks if distribution round

require(currentIcoRound > 0 && currentIcoRound <= 4);

require(

msg.value >= icoRounds[currentIcoRound].minContribution &&

totalCountributed[msg.sender][currentIcoRound].add(msg.value) <= icoRounds[currentIcoRound].maxContribution

);

totalCountributed[msg.sender][currentIcoRound] = totalCountributed[msg.sender][currentIcoRound].add(msg.value);

// Gets token amount to count and release

uint countedAmount;

uint givenAmount;

(countedAmount, givenAmount) = tokenAmount(msg.value, icoRounds[currentIcoRound].bonusAllocation);

// Anticipate new supply of round

uint newSupply = icoRounds[currentIcoRound].supplyDistributed.add(countedAmount);

// Checks if this supply can be given

require(newSupply <= icoRounds[currentIcoRound].supplyAllowed);

icoRounds[currentIcoRound].supplyDistributed = newSupply;

// Add the received value to the total received

totalEtherRaised += msg.value;

// Transfer the value to the holder wallet

multivenWallet.transfer(msg.value);

// Do the transfer of token

multicoin.transfer(msg.sender, givenAmount);

// If the supply is empty, go to next round automatically

if(icoRounds[currentIcoRound].supplyDistributed == icoRounds[currentIcoRound].supplyAllowed) {

currentIcoRound += 1;

// Deverse ICO token remaining supply if reached the end

if(currentIcoRound == 5) {

withdrawRemainingMulticoins();

}

}

// Raise the AddressDeposited (Pelikan Ico API) event

AddressDeposited(

msg.sender,

block.timestamp,

msg.value,

givenAmount,

false

);

}

function tokenAmount(uint value, uint bonusAllocation) public constant returns (uint countedAmount, uint givenAmount) {

// Amount given for book keeping (theorical amount, without bonus)

countedAmount = value / tokenPrice \* (10\*\*18);

// Real amount given (including bonus)

givenAmount = countedAmount \* bonusAllocation / 100;

}

function goToNextRound() public onlyOwner returns (bool success) {

// Checks if possible next round

require(currentIcoRound >= 0 && currentIcoRound < 5);

// Load current round details

uint remainingSupply =

icoRounds[currentIcoRound].supplyAllowed.sub(icoRounds[currentIcoRound].supplyDistributed);

// Increase currentRound integer

currentIcoRound += 1;

if(currentIcoRound == 5) {

// Deverse ICO token remaining supply if reached the end

return withdrawRemainingMulticoins();

} else {

// Load next round by giving current round remaining supply to the next round

icoRounds[currentIcoRound].supplyAllowed =

icoRounds[currentIcoRound].supplyAllowed.add(remainingSupply);

return true;

}

}

function withdrawRemainingMulticoins() private returns (bool success) {

uint icoRemainingSupply = multicoin.balanceOf(address(this));

RemainingTokensSent(icoRemainingSupply, block.timestamp);

return multicoin.transfer(multivenWallet, icoRemainingSupply);

}

}

**1. Line 22 icoRounds**

We recommend to make the mapping icoRounds internal to prevent round details.

**2. Line 29** **multicoin declaration**

It’s more recommended to use directly the multicoin.sol to declare the token, like below:

**Multicoin multicoin;**

and import the Multicoin.sol file.

**3. Line 30 kycRegistry declaration**

Same of multicoin, we should use directly the implemented contract KycRegistry.sol and import it.

**4. Line 39 Function distributionInfo**

Based on change made on the top of the PelikanIco interface, you should replace the modifier constant by view like below:

function distributionInfo() public **view** returns

**5. Line 63 kycRegistry initialization**

Based on point 2, is more convenience to use the kycRegistry file and initialize like below:

**kycRegistry = KycRegistry(kycRegistryAddress);**

**6. Line 64 multicoin initialization**

Based on point 3, is more convenience to use the Multicoin file and initialize like below:

**multicoin = Multicoin(multicoinAddress);**

**7. Line 70, line 78, line 86 & line 94**

We recommend the use of the function mul() of the openZeppelin SafeMath library. Also we need to reconsider the use of gas.

**8. Line 124 totalEtherRaised**

### **Over and under flows**

An overflow happens when the limit of the type varibale uint256 , 2\*\*256, is exceeded. What happens is that the value resets to zero instead of incrementing more.

For instance, if I want to assign a value to a uint bigger than 2\*\*256 it will simple go to 0 — this is dangerous.On the other hand, an underflow happens when you try to substract 0 minus a number bigger than 0.

For example, if you substract 0 -1 the result will be = 2\*\*256 instead of -1.

This is quite dangerous when dealing with ether. Hovewer in this contract there’s no substraction anywhere so there’s no risk of underflows.

The only time an overflow could happen is when the fallback function is called and the received value to the total received is increased

**Line 124: totalEtherRaised += msg.value;**

Someone could send a huge amount of ether that would exceed the limit of 2\*\*256 and therefore making the totalEtherRaised 0. This is improbable but the risk is there.

Therefore we recommend using the SafeMath.sol library.It’ll help you make secure calculations without the risk of under or over flows.

**9. Line 152 Function tokenAmount**

Replace the modifier constant to view.

**10. Line 154 & line 157**

Use the function of the SafeMath library.

**11. Line 182**

Replace private by internal.

Internal functions can only be called inside the current contract (more specifically, inside the current code unit, which also includes internal library functions and inherited functions) because they cannot be executed outside of the context of the current contract. Calling an internal function is realized by jumping to its entry label, just like when calling a function of the current contract internally.

1. **Testing**

**Unit tests**

Unit tests are a critical part of testing any project. The contracts described above currently have unit tests, which are marked below.

**ERC20 Testing (Multicoin.js)**

const assert = require('assert')

const {assertReverts, assertLog, assertEq} = require('./lib')

const BigNumber = require('bignumber.js')

const Multicoin = artifacts.require('Multicoin')

const ApproveAndCallFallBackTest = artifacts.require('ApproveAndCallFallBackTest')

const decimalPrecision = new BigNumber(10).pow(18)

function tokenNumber(num) {

return new BigNumber(num).mul(decimalPrecision)

}

contract('Multicoin', ([admin, user1, user2, user3, user4]) => {

let multicoin

async function setupContracts() {

const multicoin = await Multicoin.new({from: admin})

return {multicoin}

}

beforeEach('redeploy', async function () {

const contracts = await setupContracts()

multicoin = contracts.multicoin

// User 1 : Has supply, has freeze bypass

await multicoin.distributeSupply(user1, 2000, {from: admin})

await multicoin.allowFreezeBypass(user1, {from: admin})

// User 2 : Has supply, has not freeze bypass

await multicoin.distributeSupply(user2, 2000, {from: admin})

// User 3 : Has no supply, has freeze bypass

await multicoin.allowFreezeBypass(user3, {from: admin})

// User 4 : Has no supply, has no freeze bypass

// No Tx : default state

})

it('sets the parameters correctly when admin calls distribution and freeze methods', async function() {

assertEq(await multicoin.balanceOf(user1), tokenNumber(2000))

assertEq(await multicoin.balanceOf(user2), tokenNumber(2000))

assertEq(await multicoin.freezeBypassing(user1), true)

assertEq(await multicoin.freezeBypassing(user3), true)

})

it('returns the right amount in the totalSupply() method', async function() {

assertEq(await multicoin.totalSupply(), tokenNumber(2000000000))

})

it('refuses to give more token than the totalSupply', async function() {

await assertReverts(

multicoin.distributeSupply(user4, tokenNumber(2000000001), {from: admin})

)

})

it('permits to a freeze-bypasser to send tokens', async function() {

assertEq(await multicoin.tradingLive(), false)

await multicoin.transfer(user4, tokenNumber(1000), {from: user1})

assertEq(await multicoin.balanceOf(user1), tokenNumber(1000))

assertEq(await multicoin.balanceOf(user4), tokenNumber(1000))

})

it('does not permit to a non-freeze-bypasser to send tokens if tradinf is not live', async function() {

assertEq(await multicoin.tradingLive(), false)

await assertReverts(

multicoin.transfer(user4, tokenNumber(1000), {from: user2})

)

assertEq(await multicoin.balanceOf(user2), tokenNumber(2000))

assertEq(await multicoin.balanceOf(user4), tokenNumber(0))

})

it('permits to a non-freeze-bypasser to send tokens if trading is live', async function() {

await multicoin.setTradingLive({from: admin})

assertEq(await multicoin.tradingLive(), true)

assertLog(await multicoin.transfer(user4, tokenNumber(1000), {from: user2}), 'Transfer', {

from: user2,

to: user4,

tokens: tokenNumber(1000)

})

assertEq(await multicoin.balanceOf(user2), tokenNumber(1000))

assertEq(await multicoin.balanceOf(user4), tokenNumber(1000))

})

it('creates and returns approvals correctly', async function() {

assertLog(await multicoin.approve(user4, tokenNumber(100), {from: user1}), 'Approval', {

tokenOwner: user1,

spender: user4,

tokens: tokenNumber(100)

})

assertEq(await multicoin.allowance(user1, user4), tokenNumber(100))

})

it('creates an allowance and permit to spend the token, if trading is live', async function() {

await multicoin.setTradingLive({from: admin})

assertEq(await multicoin.tradingLive(), true)

assertLog(await multicoin.approve(user4, tokenNumber(100), {from: user1}), 'Approval', {

tokenOwner: user1,

spender: user4,

tokens: tokenNumber(100)

})

assertEq(await multicoin.allowance(user1, user4), tokenNumber(100))

assertLog(await multicoin.transferFrom(user1, user3, tokenNumber(100), {from: user4}), 'Transfer', {

from: user1,

to: user3,

tokens: tokenNumber(100)

})

assertEq(await multicoin.balanceOf(user3), tokenNumber(100))

})

it('creates an allowance and permit to spend the token, if trading is not live but tokenOwner is a freeze-bypasser', async function() {

assertLog(await multicoin.approve(user4, tokenNumber(100), {from: user1}), 'Approval', {

tokenOwner: user1,

spender: user4,

tokens: tokenNumber(100)

})

assertEq(await multicoin.allowance(user1, user4), tokenNumber(100))

assertLog(await multicoin.transferFrom(user1, user3, tokenNumber(100), {from: user4}), 'Transfer', {

from: user1,

to: user3,

tokens: tokenNumber(100)

})

assertEq(await multicoin.balanceOf(user3), tokenNumber(100))

})

it('handles token transfer approval to a contract', async function() {

const destinationContract = await ApproveAndCallFallBackTest.new()

const tokenAmount = tokenNumber(100)

assertLog(await multicoin.approveAndCall(destinationContract.address, tokenAmount, 'Hello World !', {from: user1}), 'Approval', {

tokenOwner: user1,

spender: destinationContract.address,

tokens: tokenAmount,

})

assertEq(await destinationContract.from(), user1)

assertEq(await destinationContract.token(), multicoin.address)

assertEq(await destinationContract.tokens(), tokenAmount)

assertEq(await destinationContract.data(), web3.fromAscii('Hello World !'))

})

it('permits to withdraw any lost ERC20 token from the contract', async function() {

otherMultiCoin = await Multicoin.new({from: admin})

await otherMultiCoin.setTradingLive({from: admin})

await otherMultiCoin.distributeSupply(multicoin.address, 100, {from: admin})

assertEq(await otherMultiCoin.balanceOf(multicoin.address), tokenNumber(100))

await multicoin.transferAnyERC20Token(otherMultiCoin.address, tokenNumber(100), {from: admin})

assertEq(await otherMultiCoin.balanceOf(multicoin.address), 0)

assertEq(await otherMultiCoin.balanceOf(admin), tokenNumber(100))

})

it('permits to transfert its ownership', async function() {

assertEq(await multicoin.owner(), admin)

await multicoin.transferOwnership(user4, {from: admin})

assertEq(await multicoin.owner(), admin)

await multicoin.acceptOwnership({from: user4})

assertEq(await multicoin.owner(), user4)

})

it('checks if the new owner is allowed to accept it', async function() {

assertEq(await multicoin.owner(), admin)

await multicoin.transferOwnership(user4, {from: admin})

assertEq(await multicoin.owner(), admin)

await assertReverts(

multicoin.acceptOwnership({from: user3})

)

assertEq(await multicoin.owner(), admin)

})

})

1. Line 38 function to sets the parameters correctly when admin calls distribution and freeze methods.

**Test OK**

2. Line 45 function totalSupply(): returns the right amount.

**Test OK**

3. Line 55 function transfer(): check if a freeze-bypasser is allowed to send tokens. Transaction successful.

**Test OK**

4. Line 62 function transfer(): check if a non-freeze-bypasser is allowed to send tokens if trading is not live. Transaction failed.

**Test OK**

5. Line 73 function transfer(): check if a non-freeze-bypasser is allowed to send tokens if trading is live. Transaction successful.

**Test OK**

6. Line 87 function approve(): creates and returns approvals.

**Test OK**

7. Line 96 function transferFrom(): creates an allowance and permit to spend the token, if trading is live. Transaction successful.

**Test OK**

8. Line 114 function transferFrom(): creates an allowance and permit to spend the token, if trading is not live but tokenOwner is a freeze-bypasser. Transaction successful.

**Test OK**

9. Line 129 function approveAndCall(): handles token transfer approval to a contract. Transaction successful.

**Test OK**

10. Line 145 function transferAnyERC20Token(): permits to withdraw any lost ERC20 token from the contract. Transaction successful.

**Test OK**

11. Line 157 function transferOwnership() & acceptOwnership(): permits to transfer its ownership and to accept it by the new owner. Transaction successful.

**Test OK**

12. Line 165 function transferOwnership() & acceptOwnership(): checks if the new owner is allowed to accept it. Transaction failed.

**Test OK**

* **All unit tests are running successfully and there aren’t any issue with these unit tests. However, we recommend to add two functions:**

**Add a function to test if the transfer (transferFrom) to the zero address is allowed.**

it('transfer when the recipient is the zero address', function () {

const to = ZERO\_ADDRESS;

await assertReverts(multicoin.transfer(to, tokenAmount(100), { from: user1 }))

})

**Add a function to test if the transfer is allowed when the sender hasn’t enough balance.**

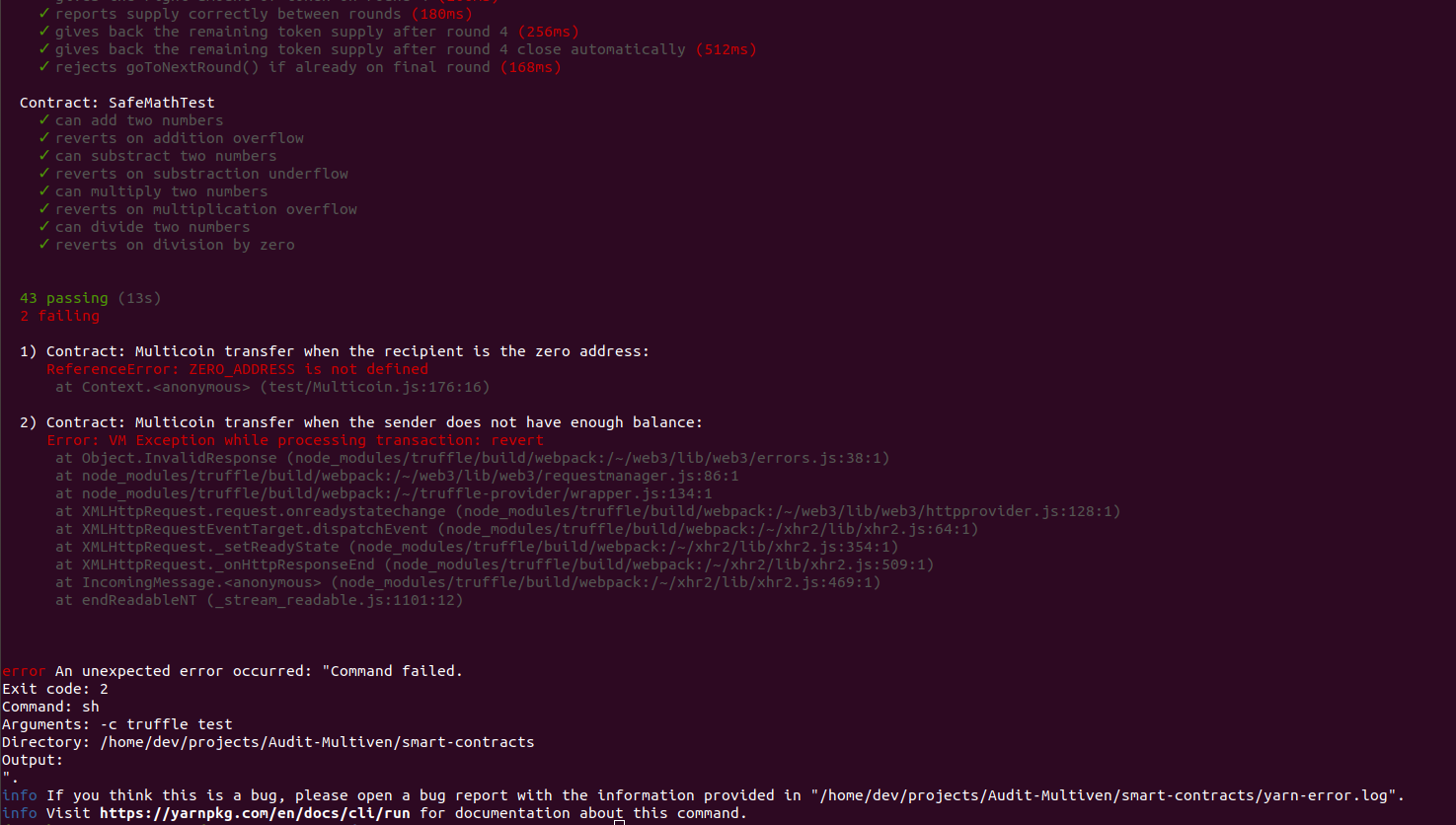
it('transfer when the sender does not have enough balance', function () { const amount = 2001;

await assertReverts(multicoin.transfer(user2, tokenNumber(amount), { from: user1 }))

})

* **These two functions was coded and tested and here is the result :**

**Transactions failed**

****

* **We recommend to add the requires like described on the section** above**.**

**Library Testing (SafeMath.js)**

const assert = require('assert')

const {assertReverts} = require('./lib');

const BigNumber = require('bignumber.js');

const SafeMathTest = artifacts.require('./lib/SafeMathTest');

const hugeNumber = new BigNumber(2).pow(256).sub(1);

contract('SafeMathTest', () => {

let safeMathTest;

beforeEach('redeploy', async function() {

safeMathTest = await SafeMathTest.new();

});

it('can add two numbers', async function() {

assert.equal(await safeMathTest.add(42, 69), 111);

});

it('reverts on addition overflow', async function() {

await assertReverts(safeMathTest.add(hugeNumber, 1));

});

it('can substract two numbers', async function() {

assert.equal(await safeMathTest.sub(69, 42), 27);

});

it('reverts on substraction underflow', async function() {

await assertReverts(safeMathTest.sub(42, 69));

});

it('can multiply two numbers', async function() {

assert.equal(await safeMathTest.mul(42, 69), 2898);

});

it('reverts on multiplication overflow', async function() {

await assertReverts(safeMathTest.mul(hugeNumber, 2));

});

it('can divide two numbers', async function() {

assert.equal(await safeMathTest.div(420, 69), 6);

});

it('reverts on division by zero', async function() {

await assertReverts(safeMathTest.div(42, 0));

});

});

**=> All unit tests are running successfully and there aren’t any issue with these unit tests. But, we recommend the SafeMath library of** [**openZeppelin**](https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/math/SafeMath.sol) **like described** above **with this** [**file**](https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/test/math/SafeMath.test.js) **for testing.**

**KYC Testing (KycRegistry.js)**

const assert = require('assert')

const {assertReverts} = require('./lib');

const KycRegistry = artifacts.require('KycRegistry');

contract('KycRegistry', ([oracle, newOracle, address1, address2]) => {

let kycRegistry;

beforeEach('redeploy', async function() {

kycRegistry = await KycRegistry.new(oracle);

});

it('has the correct oracle address', async function() {

assert.equal(await kycRegistry.oracleAddress(), oracle);

});

it('has no address cleared by default', async function() {

assert.equal(await kycRegistry.isAddressCleared(address1), false);

assert.equal(await kycRegistry.isAddressCleared(address2), false);

});

it('clears an address', async function() {

await kycRegistry.kycStatusSet(address1, true, {from: oracle});

assert.equal(await kycRegistry.isAddressCleared(address1), true);

assert.equal(await kycRegistry.isAddressCleared(address2), false);

});

it('unclears an address', async function() {

await kycRegistry.kycStatusSet(address1, true, {from: oracle});

await kycRegistry.kycStatusSet(address2, true, {from: oracle});

await kycRegistry.kycStatusSet(address1, false, {from: oracle});

assert.equal(await kycRegistry.isAddressCleared(address1), false);

assert.equal(await kycRegistry.isAddressCleared(address2), true);

});

it('can only be cleared by oracle', async function() {

await assertReverts(kycRegistry.kycStatusSet(address2, true, {from: address1}));

});

it('can update oracle address', async function() {

await kycRegistry.updateOracleAddress(newOracle, {from: oracle});

await assertReverts(kycRegistry.kycStatusSet(address1, true, {from: oracle}));

await assertReverts(kycRegistry.kycStatusSet(address1, true, {from: address1}));

await kycRegistry.kycStatusSet(address1, true, {from: newOracle});

assert.equal(await kycRegistry.isAddressCleared(address1), true);

});

it('cannot update oracle address from another address', async function() {

await assertReverts(kycRegistry.updateOracleAddress(newOracle, {from: address1}));

});

});

1. Line 13 function oracleAdrress() to check is kycRegistry has the correct oracle address

**Test Ok**

2. Line 17 function isAddressCleared() to check that kycRegistry has no address cleared by default

**Test OK**

3. Line 22 function kycStatusSet() to clear an address by oracle address

**Test OK**

4. Line 29 function kycStatusSet() to unclear an address by oracle address

**Test OK**

5. Line 38 function kycStatusSet() to test clearing from address1 != oracle.address

**Test OK**

6. Line 42 function updateOracleAddress() to update oracle address by oracle address

**Test OK**

7. Line 53 function updateOracleAddress() to try updating oracle address from another address

**Test OK**

**=> All unit tests are running successfully and there aren’t any issue with these unit tests.**

**The ICO Testing (MultivenICO.js)**

const assert = require('assert')

const {assertReverts, assertLog, assertEq} = require('./lib')

const BigNumber = require('bignumber.js')

const Multicoin = artifacts.require('Multicoin')

const MultivenIco = artifacts.require('MultivenIco')

const KycRegistry = artifacts.require('KycRegistry')

const Web3Utils = require('web3-utils')

const pricePerToken = Web3Utils.toWei('0.01', 'ether')

const decimalPrecision = new BigNumber(10).pow(18)

contract('MultivenIco', ([oracle, admin, multivenWallet, user1, user2, user3, user4, user5, user6, user7]) => {

let multicoin

let multivenIco

let kycRegistry

async function setupContracts() {

const kycRegistry = await KycRegistry.new(oracle)

const multicoin = await Multicoin.new({from: admin})

const multivenIco = await MultivenIco.new(

kycRegistry.address,

multicoin.address,

multivenWallet,

7000,

2000,

1500,

3000, // Low supply to make test easier

{from: admin}

)

return {kycRegistry, multivenIco, multicoin}

}

beforeEach('redeploy', async function () {

const contracts = await setupContracts()

kycRegistry = contracts.kycRegistry

multivenIco = contracts.multivenIco

multicoin = contracts.multicoin

await multicoin.distributeSupply(multivenIco.address, new BigNumber('15875'), {from: admin})

await multicoin.allowFreezeBypass(multivenIco.address, {from: admin})

await kycRegistry.kycStatusSet(user1, true, {from: oracle})

})

it('rejects the contribution if on Ico Round 0', async function() {

assertEq(await multivenIco.currentIcoRound(), 0)

await assertReverts(

multivenIco.sendTransaction( { from: user1, value: Web3Utils.toWei('1', 'ether') } )

)

})

it('rejects goToNextRound() calls if it is not the owner', async function() {

await assertReverts(

multivenIco.goToNextRound({from: user1})

)

})

it('accepts goToNextRound() calls if it is the owner', async function () {

await multivenIco.goToNextRound({from: admin})

assertEq(await multivenIco.currentIcoRound(), 1)

})

it('rejects a contribution made below the limit for a given round', async function () {

const paymentAmount = Web3Utils.toWei('0.001', 'ether')

await multivenIco.goToNextRound({from: admin})

assertEq(await multivenIco.currentIcoRound(), 1)

await assertReverts(

multivenIco.sendTransaction( { from: user1, value: paymentAmount } )

)

assertEq(await multicoin.balanceOf(user1), 0)

})

it('rejects a contribution if it goes behind the remaining supply of the round', async function() {

const paymentAmount = Web3Utils.toWei('71', 'ether')

await multivenIco.goToNextRound({from: admin})

assertEq(await multivenIco.currentIcoRound(), 1)

await assertReverts(

multivenIco.sendTransaction({ from: user1, value: paymentAmount })

)

})

it('goes to next round automaticaly at the end of the supply', async function() {

const paymentAmount = Web3Utils.toWei('70', 'ether')

await kycRegistry.kycStatusSet(user2, true, {from: oracle})

await multivenIco.goToNextRound({from: admin})

assertEq(await multivenIco.currentIcoRound(), 1)

await multivenIco.sendTransaction({ from: user2, value: paymentAmount })

assertEq(await multivenIco.currentIcoRound(), 2)

})

it('rejects a contribution from a non-cleared address', async function () {

const paymentAmount = Web3Utils.toWei('0.001', 'ether')

await multivenIco.goToNextRound({from: admin})

assertEq(await multivenIco.currentIcoRound(), 1)

assertEq(await kycRegistry.isAddressCleared(user2), false)

await assertReverts(

multivenIco.sendTransaction( { from: user2, value: paymentAmount } )

)

assertEq(await multicoin.balanceOf(user1), 0)

})

it('gives the right amount of token on round 1', async function () {

const paymentAmount = Web3Utils.toWei('1', 'ether')

const tokenAmount = new BigNumber(paymentAmount).dividedBy(pricePerToken).times('1.25').mul(decimalPrecision)

await multivenIco.goToNextRound({from: admin})

assertEq(await multivenIco.currentIcoRound(), 1)

assertEq(await multicoin.balanceOf(user1), 0)

await multivenIco.sendTransaction( { from: user1, value: paymentAmount } )

assertEq(await multicoin.balanceOf(user1), tokenAmount)

})

it('gives the right amount of token on round 2', async function () {

const paymentAmount = Web3Utils.toWei('1', 'ether')

const tokenAmount = new BigNumber(paymentAmount).dividedBy(pricePerToken).times('1.20').mul(decimalPrecision)

await multivenIco.goToNextRound({from: admin})

await multivenIco.goToNextRound({from: admin})

assertEq(await multivenIco.currentIcoRound(), 2)

assertEq(await multicoin.balanceOf(user1), 0)

await multivenIco.sendTransaction( { from: user1, value: paymentAmount } )

assertEq(await multicoin.balanceOf(user1), tokenAmount)

})

it('gives the right amount of token on round 3', async function () {

const paymentAmount = Web3Utils.toWei('1', 'ether')

const tokenAmount = new BigNumber(paymentAmount).dividedBy(pricePerToken).times('1.15').mul(decimalPrecision)

await multivenIco.goToNextRound({from: admin})

await multivenIco.goToNextRound({from: admin})

await multivenIco.goToNextRound({from: admin})

assertEq(await multivenIco.currentIcoRound(), 3)

assertEq(await multicoin.balanceOf(user1), 0)

await multivenIco.sendTransaction( { from: user1, value: paymentAmount } )

assertEq(await multicoin.balanceOf(user1), tokenAmount)

})

it('gives the right amount of token on round 4', async function () {

const paymentAmount = Web3Utils.toWei('1', 'ether')

const tokenAmount = new BigNumber(paymentAmount).dividedBy(pricePerToken).mul(decimalPrecision)

await multivenIco.goToNextRound({from: admin})

await multivenIco.goToNextRound({from: admin})

await multivenIco.goToNextRound({from: admin})

await multivenIco.goToNextRound({from: admin})

assertEq(await multivenIco.currentIcoRound(), 4)

assertEq(await multicoin.balanceOf(user1), 0)

await multivenIco.sendTransaction( { from: user1, value: paymentAmount } )

assertEq(await multicoin.balanceOf(user1), tokenAmount)

})

it('reports supply correctly between rounds', async function () {

const paymentAmount = Web3Utils.toWei('30', 'ether')

await multivenIco.goToNextRound({from: admin})

assertEq(await multivenIco.currentIcoRound(), 1)

await multivenIco.sendTransaction( { from: user1, value: paymentAmount } )

let prevDistributionInfo = await multivenIco.distributionInfo()

await multivenIco.goToNextRound({from: admin})

assertEq(await multivenIco.currentIcoRound(), 2)

let nextDistributionInfo = await multivenIco.distributionInfo()

const defaultRound2DistributionSupply = new BigNumber('2000').mul(decimalPrecision)

assertEq(defaultRound2DistributionSupply.add(prevDistributionInfo[4]), nextDistributionInfo[4])

})

it('gives back the remaining token supply after round 4', async function () {

const paymentAmount = Web3Utils.toWei('1', 'ether')

const tokenAmount = new BigNumber(paymentAmount).dividedBy(pricePerToken).mul(decimalPrecision)

await multivenIco.goToNextRound({from: admin})

await multivenIco.goToNextRound({from: admin})

await multivenIco.goToNextRound({from: admin})

await multivenIco.goToNextRound({from: admin})

assertEq(await multivenIco.currentIcoRound(), 4)

await multivenIco.sendTransaction( { from: user1, value: paymentAmount } )

let remainingSupply = new BigNumber('15875').mul(decimalPrecision).sub(tokenAmount)

assertLog(await multivenIco.goToNextRound({from: admin}), 'RemainingTokensSent', {

tokenSent: remainingSupply

})

assertEq(await multicoin.balanceOf(multivenWallet), remainingSupply)

assertEq(await multivenIco.currentIcoRound(), 5)

})

it('gives back the remaining token supply after round 4 close automatically', async function () {

const endRound1PaymentAmount = Web3Utils.toWei('70', 'ether')

const paymentAmountA = Web3Utils.toWei('25', 'ether')

const paymentAmountB = Web3Utils.toWei('15', 'ether')

// User that will finish round 1

await kycRegistry.kycStatusSet(user4, true, {from: oracle})

// Users that will finish round 4

await kycRegistry.kycStatusSet(user5, true, {from: oracle})

await kycRegistry.kycStatusSet(user6, true, {from: oracle})

await kycRegistry.kycStatusSet(user7, true, {from: oracle})

// Go to round 1

await multivenIco.goToNextRound({from: admin})

// Go to round 2 by exhausting round 1 supply

await multivenIco.sendTransaction({from: user4, value: endRound1PaymentAmount})

assertEq(await multivenIco.currentIcoRound(), 2)

// Go to round 3

await multivenIco.goToNextRound({from: admin})

// Go to round 4

await multivenIco.goToNextRound({from: admin})

assertEq(await multivenIco.currentIcoRound(), 4)

// remaining supplies

await multivenIco.sendTransaction({from: user5, value: paymentAmountA})

await multivenIco.sendTransaction({from: user6, value: paymentAmountA})

let remainingSupply = new BigNumber(await multicoin.balanceOf(multivenIco.address))

.sub(new BigNumber(paymentAmountB).dividedBy(pricePerToken).mul(decimalPrecision))

assertLog(await multivenIco.sendTransaction({from: user7, value: paymentAmountB}), 'RemainingTokensSent', {

tokenSent: remainingSupply

})

assertEq(await multicoin.balanceOf(multivenWallet), remainingSupply)

assertEq(await multivenIco.currentIcoRound(), 5)

})

it('rejects goToNextRound() if already on final round', async function() {

await multivenIco.goToNextRound({from: admin}) // 0 -> 1 : Go to Round 1

await multivenIco.goToNextRound({from: admin}) // 1 -> 2 : Go to Round 2

await multivenIco.goToNextRound({from: admin}) // 2 -> 3 : Go to Round 3

await multivenIco.goToNextRound({from: admin}) // 3 -> 4.: Go to Round 4

await multivenIco.goToNextRound({from: admin}) // 4 -> 5 : Final

await assertReverts(

multivenIco.goToNextRound({from: admin})

)

})

})

1. Line 47 function to check if a contribution on ICO round 0 is allowed.

**Test Ok**

2. Line 55 function goToNextRound() called by user1 != owner. Transaction rejected.

**Test OK**

3. Line 61 function goToNextRound() called by the owner. Transaction accepted.

**Test OK**

4. Line 66 function to check if a contribution made below the limit for a given round is rejected. Transaction failed.

**Test OK**

5. Line 78 function to check if a contribution can goes behind the remaining supply of the round. Transaction failed.

**Test OK**

6. Line 88 function to check if going to next round at the end of the supply is made automatically. Transaction successful.

**Test OK**

7. Line 97 function to check if a contribution is rejected if it comes from a non-cleared address. Transaction failed.

**Test OK**

8. Line 110 function to check the right amount of token to give on round 1.

**Test OK**

9. Line 122 function to check the right amount of token to give on round 2.

**Test OK**

10. Line 135 function to check the right amount of token to give on round 3.

**Test OK**

11. Line 149 function to check the right amount of token to give on round 4.

**Test OK**

12. Line 164 function to check if the reported supply is correct between rounds.

**Test OK**

13. Line 181 function to check if the remaining token supply is given back after the round 4.

**Test OK**

14. Line 201 function to check if the remaining token supply is given back after the round 4 and closed automatically.

**Test OK**

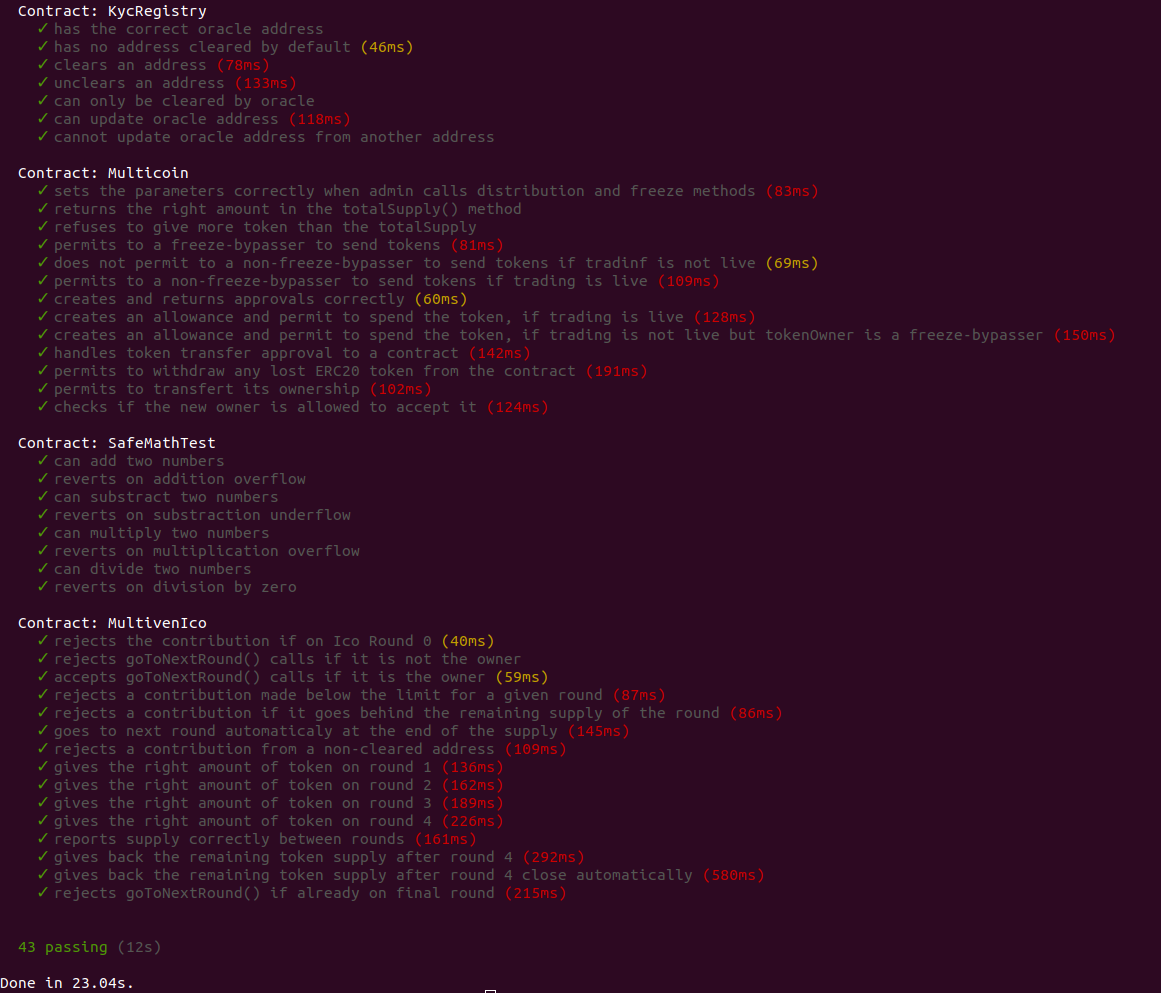
15. Line 240 function to check if goToNextRound() is rejected if already on final round. Transaction failed.

**Test OK**

**=> All unit tests are running successfully and there aren’t any issue with these unit tests.**

**Unit testing conclusion**

All unit tests are running successfully.

****

Only the ERC20 tests miss some checks that’s why we recommend to add the described requires. Also, we recommend the use of the openZeppelin SafeMath library and run the file test like described above.

**9. Summary of the audit**

**10. Conclusion**