16/04/2018

**CHAINEUM – PROJECT**

Trecento

April 16th 2018

TABLE OF CONTENTS

[Audit context 2](#_Toc510125808)

[High security flaws 3](#_Toc510125809)

[Medium vulnerabilities 4](#_Toc510125810)

[Low vulnerabilities 4](#_Toc510125811)

[Smart Contracts 4](#_Toc510125812)

[Unit tests 28](#_Toc510125813)

[Deployment tests 54](#_Toc510125814)

# Audit context

Ethereum Blockchain provides a decentralized, trustless, transparent, traceable and secure execution of smart contract. All those characteristics are respected only if the implementation is done in the state of the art.

In this way, smart contracts are the implementation of the logic/trust between the parties.

We analyzed in this report the effectiveness and safety of the code in order to bring you in a shortest time our technical recommendations and implementations suggests.

Besides, we have not received any functional and technical documentations which describe the architecture implementation about your system. We made our possible to figure out through the code and the section “8.1.4 ICO STRUCTURE” of the Multiven [Whitepaper](https://multiven.io/wp-content/uploads/2018/02/multiven_WhitePaper-15Feb2018.pdf) provided by your [GitHub](https://github.com/Multiven-Group-BV/smart-contracts/tree/master/contracts). In consideration, we cannot be liable for any security flaws related to interoperability between the Front-end and Back-end system.

Moreover, we notice that the ICO was launched before the realization of the audit and we hope that the existing process was not already deployed to take in consideration of our recommendations.

Smart contract auditing cannot unveil all existing flaws and/or vulnerabilities in the same way as an audit in which no flaws and/or vulnerabilities are found is not a guarantee for a secure smart contract. The objective of the audit is to find out flaws and/or vulnerabilities that were unobserved during development a. Various types of issues can be detected: some may affect the smart contract application while others might induce lack protection in certain areas. In this perspective, we carry out a source code analysis to identify code that need to be patched. We have performed widespread auditing in order to discern as many flaws and/or vulnerabilities as possible.

The smart contract 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.

# High security flaws

# Medium vulnerabilities

Avoid expensive gas executions by using SafetMath operations.

We recommend you to use a [pausable](https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/lifecycle/Pausable.sol) function provided by OpenZepplin to pause the running of the system if any problem or flaw is detected.

# Low vulnerabilities

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 latest available version is 0.4.21

# Smart Contracts

#### TrecentoToken.SOL

pragma solidity ^0.4.19;

import "./lib/Owned.sol";

import "./lib/SafeMath.sol";

import "./lib/ERC20Interface.sol";

import "./lib/ApproveAndCallFallBack.sol";

contract TrecentoToken is Owned, ERC20Interface {

using SafeMath for uint256;

/\* ERC20 Attributes \*/

mapping(address => uint256) balances;

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

mapping(address => bool) public freezeBypassing;

mapping(address => bool) public forgers;

string public symbol = 'TBCT';

string public name = 'TrecentoToken';

uint8 public decimals = 18;

uint256 public forgedSupply;

uint256 public circulatingSupply;

bool public tradingLive = false;

event ForgedTokens(uint256 amount, uint256 timestamp);

/\* Forging capability implementation \*/

modifier onlyForgers(address forger) {

require(forgers[forger]);

\_;

}

function setForgerStatus(address forger, bool able) public onlyOwner returns (bool success) {

forgers[forger] = able;

return true;

}

function forge(address to, uint256 tokens) public onlyForgers(msg.sender) returns (bool success) {

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

forgedSupply = forgedSupply.add(tokens);

emit ForgedTokens(tokens, block.timestamp);

emit Transfer(address(0), to, tokens);

return true;

}

/\* -- \*/

/\* Freezing capability Implementation \*/

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 Standard Implementation \*/

function totalSupply() public constant returns (uint) {

return forgedSupply;

}

function balanceOf(address tokenOwner) public constant returns (uint256 balance) {

return balances[tokenOwner];

}

function transfer(address to, uint256 tokens) public tokenTradingMustBeLive(msg.sender) returns (bool success) {

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

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

emit Transfer(msg.sender, to, tokens);

return true;

}

function transferFrom(address from, address to, uint256 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);

emit Transfer(from, to, tokens);

return true;

}

function approve(address spender, uint256 tokens) public returns (bool success) {

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

emit Approval(msg.sender, spender, tokens);

return true;

}

function allowance(address tokenOwner, address spender) public constant returns (uint256 remaining) {

return allowed[tokenOwner][spender];

}

/\* -- \*/

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

function approveAndCall(address spender, uint256 tokens, bytes data) public returns (bool success) {

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

emit 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, uint256 tokens) public onlyOwner returns (bool success) {

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

}

}

 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** for security reasons.

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

 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 constant returns (uint) {

return \_totalSupply;

}

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

return balances[tokenOwner];

}

function allowance(address tokenOwner, address spender) public constant returns (uint256 remaining) {

return allowed[tokenOwner][spender];

}

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]);**

function transferFrom(address from, address to, uint256 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);

emit Transfer(from, to, tokens);

return true;

}

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

**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

N.B : 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 you can update the allowed value.

#### SafeMath.sol

pragma solidity ^0.4.21;

/\*\*

\* @title SafeMath

\* @dev Math operations with safety checks that throw on error

\*/

library SafeMath {

/\*\*

\* @dev Multiplies two numbers, throws on overflow.

\*/

function mul(uint256 a, uint256 b) internal pure returns (uint256) {

if (a == 0) {

return 0;

}

uint256 c = a \* b;

assert(c / a == b);

return c;

}

/\*\*

\* @dev Integer division of two numbers, truncating the quotient.

\*/

function div(uint256 a, uint256 b) internal pure returns (uint256) {

// assert(b > 0); // Solidity automatically throws when dividing by 0

// uint256 c = a / b;

// assert(a == b \* c + a % b); // There is no case in which this doesn't hold

return a / b;

}

/\*\*

\* @dev Subtracts two numbers, throws on overflow (i.e. if subtrahend is greater than minuend).

\*/

function sub(uint256 a, uint256 b) internal pure returns (uint256) {

assert(b <= a);

return a - b;

}

/\*\*

\* @dev Adds two numbers, throws on overflow.

\*/

function add(uint256 a, uint256 b) internal pure returns (uint256) {

uint256 c = a + b;

assert(c >= a);

return c;

}

/\*\*

\* @dev Divides two numbers with 18 decimals, represented as uints (e.g. ether or token values)

\*/

uint constant ETHER\_PRECISION = 10 \*\* 18;

function ethdiv(uint x, uint y) internal pure returns (uint z) {

// Put x to the 36th order of magnitude, so natural division will put it back to the 18th

// Adding y/2 before putting x back to the 18th order of magnitude is necessary to force the EVM to round up instead of down

z = add(mul(x, ETHER\_PRECISION), y / 2) / y;

}

/\*\*

\* @dev Divides two numbers with 2 decimals, represented as uints (e.g. ether or token values)

\*/

uint constant FIAT\_PRECISION = 10 \*\* 2;

function fiatdiv(uint x, uint y) internal pure returns (uint z) {

// Put x to the 4th order of magnitude, so natural division will put it back to the 2th

// Adding y/2 before putting x back to the 18th order of magnitude is necessary to force the EVM to round up instead of down

z = add(mul(x, FIAT\_PRECISION), y / 2) / y;

}

}

No relevant issue on this contract.

#### ERC20Interface.sol

 Based on the change made on the top of TrecentoToken.sol (Line 64, Line 68 & Line 93). You should replace **constant** modifier by **view** (line 4, line 5 & line 6)to have a coherent interface.

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. **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 use of future update and recommendation by the Ethereum community if this contract is brought to be reused.

1. **ThesaurioIco.sol**

pragma solidity ^0.4.19;

contract ThesaurioIco {

event AddressDeposited (

address indexed depositor,

uint depositedAt,

uint amount,

uint tokenAmount,

bool indexed boughtOnBehalf

);

function distributionInfo() public constant returns (

uint minContrib,

uint maxContrib,

uint currentTokenPrice,

uint currentBonus,

uint remainingSupply

);

}

No relevant issue on this contract.

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

1. **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);

}

No relevant issue on this contract.

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

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

1. **KYC Implementation (KycRegistry.sol)**

pragma solidity ^0.4.21;

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;

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

}

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

oracleAddress = \_newOracleAddress;

}

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

return kycClearances[\_address];

}

}

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

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

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

return kycClearances[\_address];

}

1. **The ICO Implementation (TrecentoCrowdsale.sol)**

pragma solidity ^0.4.19;

import "./TrecentoToken.sol";

import "./lib/Owned.sol";

import "./lib/SafeMath.sol";

import "./lib/PricesInterface.sol";

import "./lib/ThesaurioIco.sol";

import "./lib/KycRegistryInterface.sol";

contract TrecentoCrowdsale is Owned, ThesaurioIco {

using SafeMath for uint256;

TrecentoToken trecentoToken;

PricesInterface pricesContract;

KycRegistryInterface kycRegistry;

address trecentoWallet;

uint256 public bonusPerThousandMultiplier = 1200;

uint256 public tokenPriceInEuroCents = 100;

uint256 public minimumContributionEuroCents = 1000000;

uint256 public currentlyRaisedEuroCents;

uint256 public constant HardCapEuroCents = 5000000000;

bool public crowdsaleEnabled = false;

function distributionInfo() public constant returns (

uint minContrib,

uint maxContrib,

uint currentTokenPrice,

uint currentBonus,

uint remainingSupply

) {

minContrib = minimumContribution();

maxContrib = 0;

currentTokenPrice = tokenPriceInWei();

currentBonus = bonusPerThousandMultiplier;

remainingSupply = 0;

}

function TrecentoCrowdsale(

address trecentoTokenAddr,

address pricesContractAddr,

address kycRegistryAddr,

address trecentoWalletAddr

) public {

trecentoToken = TrecentoToken(trecentoTokenAddr);

pricesContract = PricesInterface(pricesContractAddr);

kycRegistry = KycRegistryInterface(kycRegistryAddr);

trecentoWallet = trecentoWalletAddr;

}

function changeCrowdsaleSettings(

uint256 \_tokenPriceInEuroCents,

uint256 \_minimumContributionInEuroCents,

uint256 \_bonusPerThousandMultiplier,

bool \_crowdsaleEnabled

) public onlyOwner {

tokenPriceInEuroCents = \_tokenPriceInEuroCents;

minimumContributionEuroCents = \_minimumContributionInEuroCents;

bonusPerThousandMultiplier = \_bonusPerThousandMultiplier;

crowdsaleEnabled = \_crowdsaleEnabled;

}

function minimumContribution() public constant returns (uint256) {

return minimumContributionEuroCents \* pricesContract.eurPrice();

}

modifier mustBeKycCleared(address \_address) {

require(kycRegistry.isAddressCleared(\_address));

\_;

}

modifier saleMustBeEnabled() {

require(crowdsaleEnabled);

\_;

}

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

require(msg.value >= minimumContribution());

// Checking hard-cap limits

uint256 euroAmountCents = msg.value.div(pricesContract.eurPrice());

require(currentlyRaisedEuroCents.add(euroAmountCents) <= HardCapEuroCents);

currentlyRaisedEuroCents = currentlyRaisedEuroCents.add(euroAmountCents);

// Distributing token

uint256 givenAmount = tokenAmount(msg.value);

makeAllocation(givenAmount, msg.sender);

// Transfer the money to Trecento Wallet

trecentoWallet.transfer(msg.value);

emit AddressDeposited(

msg.sender,

block.timestamp,

msg.value,

givenAmount,

false

);

}

function otherCurrencyPayment(

uint256 euroAmountCents,

address tokenReceiver

) public mustBeKycCleared(tokenReceiver) saleMustBeEnabled onlyOwner {

// Calculating token amount

uint256 givenAmount = euroAmountCents

.fiatdiv(tokenPriceInEuroCents) // Token amount multiplied by 1e2

.mul(bonusPerThousandMultiplier) // Token amount, bonus included, multiplied by 1e5

.div(1000) // Token amount, bonus included, multiplied by 1e2

.mul(10\*\*16); // Token amount, bonus included, multiplied by 1e18

// Checking hard-cap limits

require(currentlyRaisedEuroCents.add(euroAmountCents) <= HardCapEuroCents);

currentlyRaisedEuroCents = currentlyRaisedEuroCents.add(euroAmountCents);

// Distributing token

makeAllocation(givenAmount, tokenReceiver);

emit AddressDeposited(

tokenReceiver,

block.timestamp,

euroAmountCents.div(tokenPriceInEuroCents).mul(tokenPriceInWei()), // Calculates the equivalent amount of ETH based on known EUR price

givenAmount,

true

);

}

function tokenAmount(uint256 etherAmount) public constant returns (uint256) {

return etherAmount

.ethdiv(tokenPriceInWei()) // Token amount multiplied by 1e18

.mul(bonusPerThousandMultiplier) // Token amount, bonus included, multiplied by 1e21

.div(1000); // Token amount, bonus included, multiplied by 1e18

}

function tokenPriceInWei() public constant returns (uint256) {

return pricesContract.eurPrice().mul(tokenPriceInEuroCents);

}

function makeAllocation(uint256 givenTokenAmount, address tokenReceiver) private {

trecentoToken.forge(tokenReceiver, givenTokenAmount);

}

}

1. **Line 15** **pricesInterface declaration**

PricesInterface pricesContract;

It’s more recommended to use directly a smart contract to implement all the functions and to declare this contract.

1. **Line 16 kycRegistry declaration**

KycRegistryInterface kycRegistry;

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

1. **Line 39 Function distributionInfo**

function distributionInfo() public constant returns (

uint minContrib,

uint maxContrib,

uint currentTokenPrice,

uint currentBonus,

uint remainingSupply

) {

minContrib = minimumContribution();

maxContrib = 0;

currentTokenPrice = tokenPriceInWei();

currentBonus = bonusPerThousandMultiplier;

remainingSupply = 0;

}

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

**function distributionInfo() public view returns**

1. **Line 47 PricesInterface initialization**

pricesContract = PricesInterface(pricesContractAddr);

Based on the point 2 suggestion and clarification it’s more convenience to

use the kycRegistry file and initialize it like below:

**pricesContract = Prices(pricesContractAddr); //Prices.sol will be the contract implementing PricesInterface.sol**

1. **Line 48 kycRegistry initialization**

kycRegistry = KycRegistryInterface(kycRegistryAddress);

Based on the point 2 suggestion and clarification it’s more convenience to

use the kycRegistry file and initialize it like below:

**kycRegistry = KycRegistry(kycRegistryAddress);**

1. **Line 64 function minimumContribution()**

function minimumContribution() public constant

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 minimumContribution() public view returns**

1. **Line 64 function tokenAmount(uint256 etherAmount)**

function tokenAmount(uint256 etherAmount) public constant

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 tokenAmount(uint256 etherAmount) public view returns**

1. **Line 136 function tokenPriceInWei()**

function tokenPriceInWei() public constant returns (uint256)

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 tokenPriceInWei() public view returns (uint256)**

1. **Line 140 makeAllocation (uint256 givenTokenAmount, address tokenReceiver)**

function makeAllocation(uint256 givenTokenAmount, address tokenReceiver) private {

trecentoToken.forge(tokenReceiver, givenTokenAmount);

}

We recommend to replace the **private** modifier by **internal** like below:

**function makeAllocation(uint256 givenTokenAmount, address tokenReceiver) internal**

**Conclusion**

**We have found many issues regarding the existing code related which impacts drastically the whole process of the Token distribution.**

**During the unit test phase, we have found also a number of failed results that not validate in our side the expected behavior of the token sale distribution. We recommend you strongly to review your code to prevent any theft or problem during your ICO phase.**

**Besides, we noticed that the ICO is running and we hope that the current token sale contract is not deployed and interfaced with your ICO website allowing to received contributions.**

# Unit tests

Unit tests are a critical part of testing any project. The contracts described above currently have unit tests, which are marked below. We have executed the all javascripts files by adding 2 functions to cover 100% of the behavior.

**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. (please see the audit section recommendation**

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.**

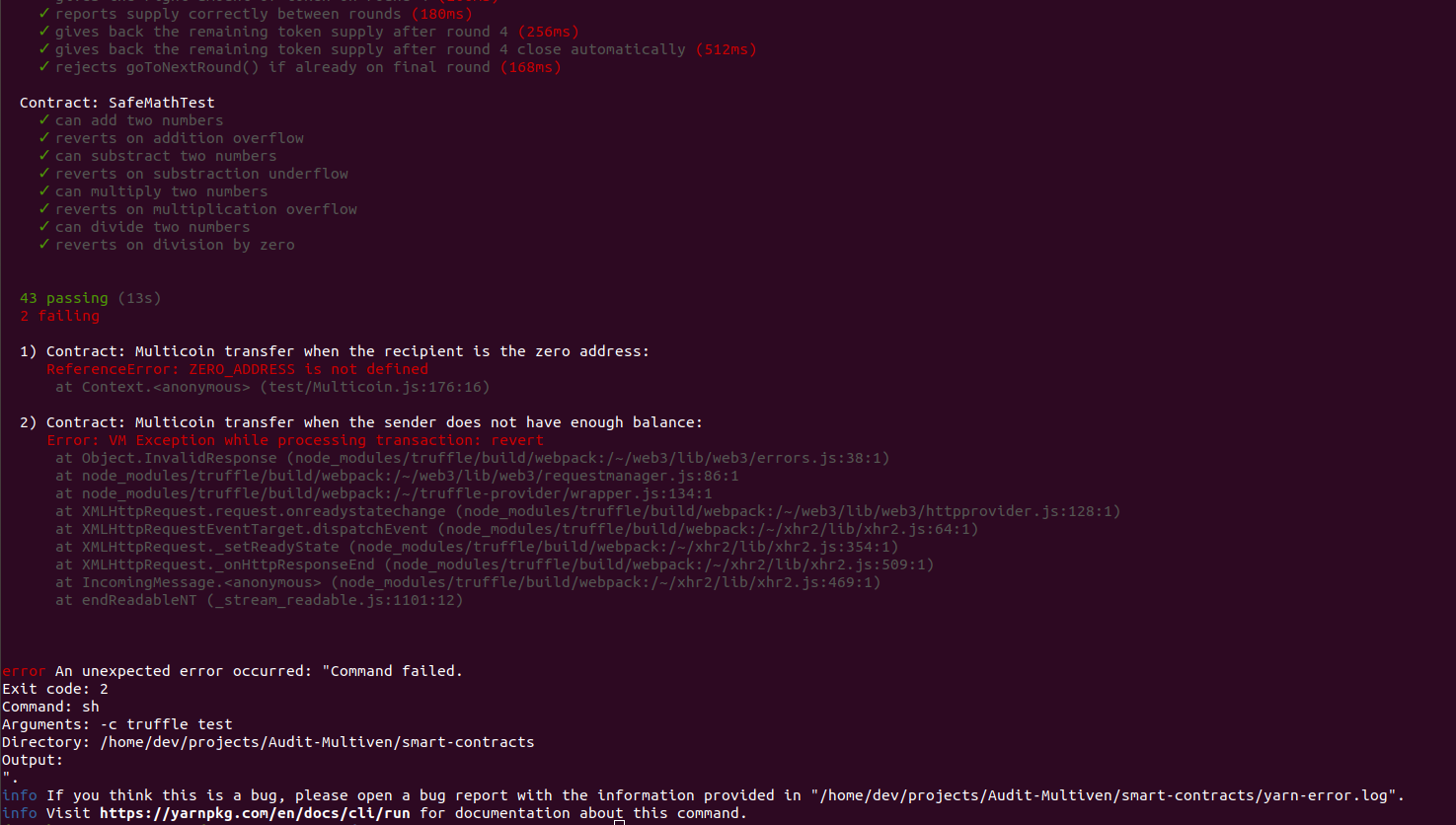
it('transfer when the sender does not have enough balance', function () { const amount = 2001;

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**

**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 we didn’t find any issue with these unit tests.**

**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 we didn’t find any issue with these unit tests coverage.**

**MultivenIco.js modified with correct value**

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,

100,

600,

600,

700, // 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('0.81', '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('0.8', '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('0.8', 'ether')

const paymentAmountA = Web3Utils.toWei('0.1', 'ether')

const paymentAmountB = Web3Utils.toWei('0.15', '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

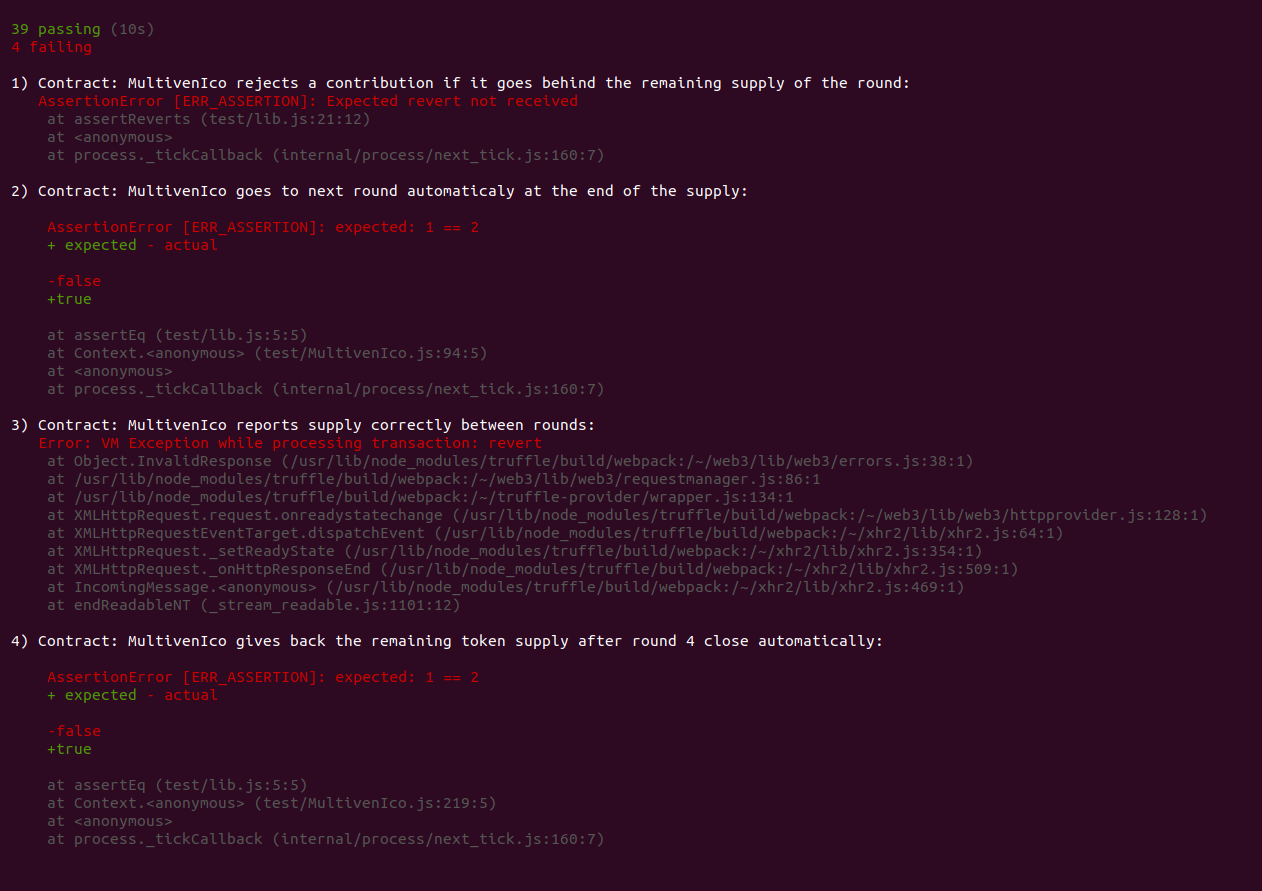
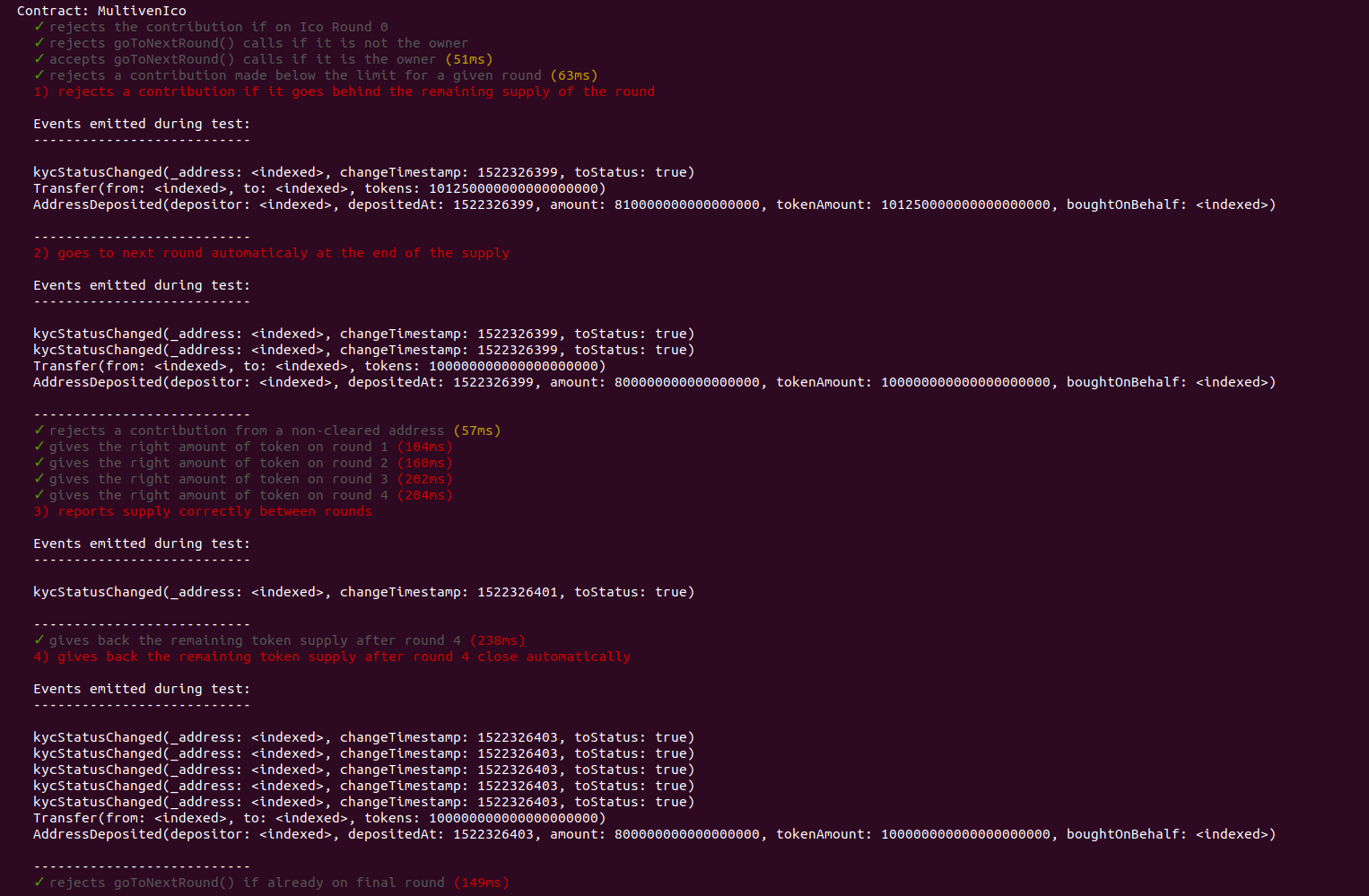
await assertReverts(

multivenIco.goToNextRound({from: admin})

)

})

})

****

**=> The tests failed because of the wrong calculation of the newsupply value as described on the audit section.**

**Unit testing conclusion**

**All unit tests are running successfully based on the existing test files provided in your Github. However, the success of these unit tests is not relevant because the testing parameters are not reflecting the correct expectations with the right amount of the distributed tokens.**

# Deployment tests

All contracts are deployed with success on the top of the ropsten network. You can check on etherscan :

* MultivenIco.sol [**https://ropsten.etherscan.io/tx/0xde598bbc40005457b510b3938a3baeda29c17d7e9d4ee7c74b91527db06aff1b**](https://ropsten.etherscan.io/tx/0xde598bbc40005457b510b3938a3baeda29c17d7e9d4ee7c74b91527db06aff1b)
* Multicoin.sol [**https://ropsten.etherscan.io/tx/0x6e52988aaab1ec4b187b422b2aa94c9073a1cd496dd8ab073b4f0e38f1809665**](https://ropsten.etherscan.io/tx/0x6e52988aaab1ec4b187b422b2aa94c9073a1cd496dd8ab073b4f0e38f1809665)
* KycRegistry.sol [**https://ropsten.etherscan.io/tx/0x7e5b8b045aba9fe0d1cea51be416801076b04e96a467aa4a7b188447f143f872**](https://ropsten.etherscan.io/tx/0x7e5b8b045aba9fe0d1cea51be416801076b04e96a467aa4a7b188447f143f872)

**Interaction tests**

Some of the interaction tests failed. The contract is failing and not accepting contribution from a cleared address. If this function is not successful, the contract can’t be used for the ICO. **This is a critical point.**

Here we will describe the testing process :

1. Deploy KycRegistry contract **OK** [**https://ropsten.etherscan.io/tx/0x7e5b8b045aba9fe0d1cea51be416801076b04e96a467aa4a7b188447f143f872**](https://ropsten.etherscan.io/tx/0x7e5b8b045aba9fe0d1cea51be416801076b04e96a467aa4a7b188447f143f872)
2. Clear a participant address **OK** [**https://ropsten.etherscan.io/tx/0xab7716b632d475c2d757c4bdf8ec9a13b3f1ffc12455dbae63d507a2926afc42**](https://ropsten.etherscan.io/tx/0xab7716b632d475c2d757c4bdf8ec9a13b3f1ffc12455dbae63d507a2926afc42)
3. Deploy Multicoin contract **OK** [**https://ropsten.etherscan.io/tx/0x6e52988aaab1ec4b187b422b2aa94c9073a1cd496dd8ab073b4f0e38f1809665**](https://ropsten.etherscan.io/tx/0x6e52988aaab1ec4b187b422b2aa94c9073a1cd496dd8ab073b4f0e38f1809665)
4. Deploy the MultivenIco contract **OK** [**https://ropsten.etherscan.io/tx/0xde598bbc40005457b510b3938a3baeda29c17d7e9d4ee7c74b91527db06aff1b**](https://ropsten.etherscan.io/tx/0xde598bbc40005457b510b3938a3baeda29c17d7e9d4ee7c74b91527db06aff1b)

MultivenIco(

KYCregistry.address "0x195d6a7765569c5d4b48d0e9591f935f232193de",

Multicoin.aDdress "0x95b9ce8cc7d8402163bcf6be4294e69f2b06833c",

MultivenWallet.address"0x77dFaE728f9CFB06D0f025dd1026575Ad5eb1587",

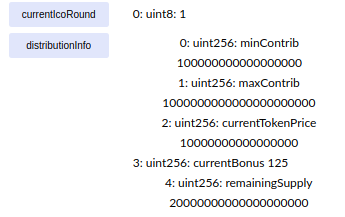
Round1supply 100,

Round2supply 600,

Round3supply 600,

Round4supply 700)

1. Distribute supply to the MultivenICO contract **OK** [**https://ropsten.etherscan.io/tx/0xd4a68e4e25b532a78e7eec6e48af6307366e47abf2da3f8ffc37ddb029ce2784**](https://ropsten.etherscan.io/tx/0xd4a68e4e25b532a78e7eec6e48af6307366e47abf2da3f8ffc37ddb029ce2784)
2. Allow freeze by pass to the MultiveICO **contract** OK[**https://ropsten.etherscan.io/tx/0xe5100172404180b786844a1b4d8e308eaa805fad3aaca417e05f11740e65b6cf**](https://ropsten.etherscan.io/tx/0xe5100172404180b786844a1b4d8e308eaa805fad3aaca417e05f11740e65b6cf)
3. Go to next round **OK** [**https://ropsten.etherscan.io/tx/0x0fe4a9febeccc922c186e7a92dcecdf0d52de412fb0b68907597e619daadc015**](https://ropsten.etherscan.io/tx/0x0fe4a9febeccc922c186e7a92dcecdf0d52de412fb0b68907597e619daadc015)
4. Contribute with 0.8 eth to get 100 token with bonus, which is the supply allowed of the first round OK [**https://ropsten.etherscan.io/tx/0xb6a0b2e569efebbdc56fa0adaaf738b241345f44479b5dff63cae270d80645c6**](https://ropsten.etherscan.io/tx/0xb6a0b2e569efebbdc56fa0adaaf738b241345f44479b5dff63cae270d80645c6)
5. Check result of the function distributeInfo **Fail**



 The expected value for currentIcoRound = 2 and remainingSupply = 0.

**Deployment & interaction tests conclusion**

**At this stage the process is not working properly even though the contributor will receive the right number of tokens. However, the entire process to calculate the remaining supply is not reflecting the expectation of the whitepaper and the required behavior. We recommend you to review the code with all suggestions and recommendations described above.**