1. Introduction

This Audit Report highlights the overall security of RIWIGO Smart Contracts. With this report, we have tried to ensure the reliability of their smart contract by complete assessment of their smart contract codebase.

2. Methodologies applied

Our team has performed thorough testing of the project starting with analyzing the code design patterns in which we reviewed the smart contract architecture to ensure it is structured and safe use of third party smart contracts and libraries.

Our team then performed a formal line by line inspection of the Smart Contract in order to find any potential issue like race conditions, transaction-ordering dependence, timestamp dependence, and denial of service attacks.

3. Source File

The files audited have been published at <Link>

4. Disclaimer

A. The audit makes no statements or warranties about the utility of the code, suitability of the business model, regulatory regime for the business model, or any other statements about the fitness of the contracts to purpose, or their full bugfree status.
B. The audit documentation is to evaluate the safety of the code.

5. Executive Summary

Overall this is a super ambitious ERC20 contract that is well conceived and well executed.

The contract provides for the following interesting features and use cases:

- Ability to create tokens upto total supply as mentioned in whitepaper.
- Ability to allow and give approval for delegated transfer.
- A ‘burn’ functionality allowing extra tokens to be burnt to make balance of demand and supply
- Approval mechanics for ownership transfers, minimizing ‘fat finger’ risks.

- Time-lock tokens within an ERC20 smart contract for multiple utilities without the need of transferring tokens to an external escrow smart contract.

6. Line by Line Comments

Included below are the line by line comments and notes as part of the audit process.

Line 1: Solidity Version

Latest version of Solidity is used 0.5.12.

Line 18 : SafeMath Contract

Latest zeppelin Safemath Library is used from the open source project named 'OpenZeppelin'.
https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/math/SafeMath.

So It is highly appreciable to use SafeMath contract and reuse it. Most of the successful ICO contracts implement the safety using the same. It helps in the following way:

1. Reusable to all basic 4 operations in the contract.
2. It mitigates Overflow and Underflow attacks.

Line 29 and 36 : ERC20Basic and ERC20 contract

This contract is implemented as an interface type contract, where the functions are declared but not implemented. It is a good approach to make the contract modular and readable. All ERC20 compatible methods and events are present in the contract.

Line 43 : BasicToken contract

This contract contains implementation of ERC20Basic interface contract functions.

Line 55 : Transfer function

Transfer given number of tokens from the message sender to given recipient.

Function Checks:
notSender(_to) : This modifier prevents user to send transaction on his own address.

require(_to != address(0)) : Can not send to a empty or null address.

wallets[msg.sender].tokensAmount >= _value : Check to mitigate spending more amount than existing in the account.
checkIfCanUseTokens(msg.sender, _value) : Prevents user to send freeze tokens.

All the checks protects from many errors and attack.

Line 83: StandartToken contract

This contract contains implementation of the functions which are declared in ERC20Basic and BasicToken interface contract.

Line 104: Transfer from function

Transfer given number of tokens from given owner to given recipient.

Function checks:
notSender(_to) : This modifier prevents user to send transaction on his own address.

require(_to != address()) : Can not send to a empty or null address.

wallets[msg.sender].tokensAmount >= _value : Check to mitigate spending more amount than existing in the account.

checkIfCanUseTokens(msg.sender, _value) : Prevents user to send freeze tokens.

_value <= allowed[_from][msg.sender] : Spender can not send more amount than allocated via approval method.

Line 86 : Approve function

Allow given spender to transfer given number of tokens from the message sender.

Line 117 : Ownable contract

This contract initialized owner account at the time of contract creation. Owner variable mitigates major task in of ownership change.
In this contract, all the events and functions are declared for the Ownership Transfer in this contract.

Line 130 : FreezableToken contract

This contract provides basic functionality to time-lock tokens within an ERC20 smart contract for multiple utilities without the need of transferring tokens to an external escrow smart contract. It also allows fetching balance of locked and transferable tokens.
Time-locking can also be achieved via staking, but that requires transfer of tokens to an escrow contract / stake manager, resulting in the following six concerns:

- additional trust on escrow contract / stake manager
- additional approval process for token transfer
- increased ops costs due to gas requirements in transfers
- tough user experience as the user needs to claim the amount back from external escrows.
- inability for the user to track their true token balance / token activity
- inability for the user to utilize their locked tokens within the token ecosystem.

Line 148: Freeze tokens function

This function provides basic functionality to time-lock tokens within an ERC20 smart contract.

Function checks :
isFreezeAllowed : This modifier only allow to user to freeze the token then only permission given by the owner.

wallets[msg.sender].freezedAmount == 0 : We can freeze tokens only if there are no frozen tokens on the wallet.

wallets[msg.sender].tokensAmount >= _amount : Check freeze amount can not more then balance amount .

Line 187 : RIWIGO contract :

It is the main contract to generate tokens. It can only be called by the owner which is obvious best security and necessary.

Line 190, 191, 192,193: Decimal, Symbol , Name and totalSupply

The fields mentioned are same as mentioned in Whitepaper.

Line 201 : Burn Token function

It is a good step to make a provision of token burning functionality for future use. As the RIWIGO ecosystem has token flow around all of its subsystems, so the application will need to burn excess

tokens once the full fledged version will be implemented. It can only be called by any token holder which provides real transparency and full decentralized control to user. Only be called by smart contract owner

Function checks :
onlyOwner : This modifier allow only owner to burn the tokens.

wallets[msg.sender].tokensAmount >= _value : Check to mitigate spending more amount than existing in the account.

checkIfCanUseTokens(msg.sender, _value) : Prevents user to burn freeze tokens.

Line 211 : Transfer Ownership function

This Would help us to stop a foot fault / attack by making sure at least that newOwner isn't 0x0. That doesn't stop someone from making up a malicious newOwner but it would stop a fat finger function call. Only be called by smart contract owner

Function checks :
_newOwner != address(0) : Prevents to set a empty or null address.

notSender(_newOwner) : This modifier prevents to assign same owner again.

7. Notes on Some Vulnerabilities

1. Approval racing condition

The standard ERC20 implementation contains a widely-known racing condition in its approve function, wherein a spender is able to witness the token owner broadcast a transaction altering their approval, and quickly sign and broadcast a transaction using transferFrom to move the current approved amount from the owner's balance to the spender. If the spender's transaction is validated before the owner's, the spender is able to spend their entire approval amount twice.

Line 86 : approve() has a race condition problem.

approve() doesn't check if the value of the allowance is equal to 0 before performing operation.

We recommend disabling users from calling this function if the value of allowance is not equal to 0.

We also recommend adding this code before performing operation:

```
require((_value == 0) || (allowed[msg.sender][_spender] == 0));
```

Status : Fixed

2. Negative tokens approval

Contract should not be able to approve negative value tokens but this test case is failing.

```
require((_value == 0) || (allowed[msg.sender][_spender] == 0));
  if(allowed[msg.sender][_spender] == 0){
        require(_value > 0);
        allowed[msg.sender][_spender] = _value;emit Approval(msg.sender, _spender, _value);
        return true;
}
        else {
        allowed[msg.sender][_spender] = _value;emit Approval(msg.sender, _spender, _value);
        return true;
}
```

Status : Fixed

3. Remove MultisendableToken contract

We are having MultisendableToken contract which is having functions for the mass token distribution. This function taking array as input parameters, Which may lead to "Extra gas consumption".

But, Ethereum miners impose a limit on the total number of gas consumed in a block. If _addresses.length is large enough, the function exceeds the block gas limit, and transactions calling it will never be confirmed.
This becomes a security issue, if an external actor influences array.length. E.g., if array enumerates all registered addresses, an adversary can register many addresses, causing the problem.

So that most of ERC20 the contract are not having mass token Transfer functions on the Solidity level to be compliant with the solidity coding standard. So we have to remove that "MultisendableToken" contract.

Status: Fixed.

4. Add _value > 0 check in transfer function

Contract should not be able to transfer negative value tokens.

Status: Fixed.

5. Add _value > 0 check in transfer from function

Contract should not be able to transfer negative value tokens.

Status: Fixed.

6. Add _value > 0 check in freezeAllowance function

Contract should not be able to freeze negative value tokens.

Status: Fixed.

7. Remove hard coded master key:

Remove the master address and allow the owner to transfer the ownership as per the ERC20 standard.

Status: Fixed.