Smart Contract Security Audit Report

The SlowMist Security Team received the DRC team's application for smart contract security audit of the DRC on Dec. 21, 2020. The following are the details and results of this smart contract security audit:

Token name :

DRC

The Contract address :

https://github.com/DRC-asia/DRC-mobility-contracts

commit: 5ef9d3215b271d4534a7248636ad08da88a61e59

The audit items and results :

(Other unknown security vulnerabilities are not included in the audit responsibility scope)

No.	Audit Items	Audit Subclass	Audit Subclass Result
1	Overflow Audit	-	Passed
2	Race Conditions Audit	-	Passed
3	Authority Control Audit	Permission vulnerability audit	Passed
		Excessive authority audit	Passed
4	Safety Design Audit	Zeppelin module safe use	Passed
		Compiler version security	Passed
		Hard-coded address security	Passed
		Fallback function safe use	Passed
		Show coding security	Passed
		Function return value security	Passed
		Call function security	Passed
5	Denial of Service Audit	-	Passed
6	Gas Optimization Audit	-	Passed
7	Design Logic Audit	-	Passed
8	“False top-up” vulnerability Audit	-	Passed
9	Malicious Event Log Audit	-	Passed

10	Scoping and Declarations Audit	-	Passed
11	Replay Attack Audit	ECDSA's Signature Replay Audit	Passed
12	Uninitialized Storage Pointers Audit	-	Passed
13	Arithmetic Accuracy Deviation Audit	-	Passed

Audit Result : Passed

Audit Number : 0X002012220001

Audit Date : Dec. 22, 2020

Audit Team : SlowMist Security Team

Summary: This is a token contract that contains the tokenVault and crowdsale section. The total amount of contract tokens can be changed. The contract does not have the Overflow and the Race Conditions issue. During the audit, we found the following:

1. The admin role can set the lock period arbitrarily through the setLockPeriods function. When setting the lock period, it is not checked whether the periods are greater than the current time.

2. The admin role can time lock any user's tokens through the lock function.

3. The admin role can increase the number of tokens locked by the user through the increaseLockAmount function.

4. The admin role can increase the period that user tokens are locked through the adjustLockPeriod function.

5. The admin role can increase the period that user tokens are locked through the adjustLockPeriod function.

6. The admin role can period lock the user's tokens through the vestDedicatedTokens function.

7. In the _withdrawToken function, if the transfer function of the target contract has no return value or the writing does not meet the EIP20 standard, this require check will never pass

The source code:

AdminRole.sol:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue

pragma solidity ^0.4.24;

import "openzeppelin-solidity/contracts/access/Roles.sol";

contract AdminRole {
  using Roles for Roles.Role;

  event AdminAdded(address indexed account);
  event AdminRemoved(address indexed account);

  Roles.Role private admins;

  constructor() internal {
    _addAdmin(msg.sender);
  }

  modifier onlyAdmin() {
    require(isAdmin(msg.sender));
    _;
  }

  function isAdmin(address account) public view returns (bool) {
    return admins.has(account);
  }

  function addAdmin(address account) public onlyAdmin {
    _addAdmin(account);
```

```
  }

  function renounceAdmin() public {
    _removeAdmin(msg.sender);
  }

  function _addAdmin(address account) internal {
    admins.add(account);
    emit AdminAdded(account);
  }

  function _removeAdmin(address account) internal {
    admins.remove(account);
    emit AdminRemoved(account);
  }
}
```

Whitelisted.sol:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue

pragma solidity ^0.4.24;

import "openzeppelin-solidity/contracts/access/Roles.sol";
import "./AdminRole.sol";

contract Whitelisted is AdminRole {
  using Roles for Roles.Role;

  event AddedToWhitelist(address indexed account);
  event RemovedFromWhitelist(address indexed account);

  Roles.Role private whitelist;


  modifier onlyWhitelisted() {
    require(isWhitelisted(msg.sender));
    _;
  }

  constructor() internal {}
```

```
  function isWhitelisted(address account) public view returns (bool) {
    return whitelist.has(account);
  }

  function addToWhitelist(address account) public onlyAdmin {
    whitelist.add(account);
    emit AddedToWhitelist(account);
  }

  function addWhitelist(address[] accounts) external onlyAdmin {
    for (uint i = 0; i < accounts.length; i++) {
      addToWhitelist(accounts[i]);
    }
  }

  function removeFromWhitelist(address account) public onlyAdmin {
    whitelist.remove(account);
    emit RemovedFromWhitelist(account);
  }
}
```

CappedCrowdsale.sol:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue

pragma solidity ^0.4.24;

import "openzeppelin-solidity/contracts/math/SafeMath.sol";
import "./Crowdsale.sol";
import "../accessControl/AdminRole.sol";



/**
 * @title CappedCrowdsale
 * @dev Crowdsale capped individually and totally
 */
contract CappedCrowdsale is Crowdsale, AdminRole {
  using SafeMath for uint256;

  mapping(address => uint256) private _contributions;
  uint256 private _totalSaleCap;
  uint256 private _maxIndividualCap;
```

```
uint256 private _minIndividualCap;

event IndividualCapsSet(uint256 minCap, uint256 maxCap);



constructor(uint256 totalSaleCap) internal {
    require(totalSaleCap > 0);
    _totalSaleCap = totalSaleCap;
    // No individual caps for the deployment time
    _maxIndividualCap = ~uint256(0);
    _minIndividualCap = 0;
}

/**
 * @dev Sets a specific beneficiary's maximum contribution.
 * @param minCap Minimum wei limit for individual contribution
 * @param maxCap Maximum wei limit for individual contribution
 */
function setIndividualCaps(
    uint256 minCap,
    uint256 maxCap
)
    external
    onlyAdmin
{
    _minIndividualCap = minCap;
    _maxIndividualCap = maxCap;

    emit IndividualCapsSet(minCap, maxCap);
}

/**
 * @dev Sets total sale cap in wei
 * @param totalSaleCap Sale cap in wei
 */
function setTotalSaleCap(uint256 totalSaleCap)
    external
    onlyAdmin
{
    require(totalSaleCap >= 0, "Total sale cannot be negative");
    _totalSaleCap = totalSaleCap;
}
```

```
/**
 * @dev Returns the total sale cap
 * @return total sale cap
 */
function getTotalSaleCap()
  public
  view
  returns (uint256)
{
  return _totalSaleCap;
}

/**
 * @dev Checks whether the total sale cap has been reached.
 * @return Whether the cap was reached
 */
function isTotalSaleCapReached()
  public
  view
  returns (bool)
{
  return weiRaised() >= _totalSaleCap;
}

/**
 * @dev Returns the individual caps
 * @return individual caps
 */
function getIndividualCaps()
  public
  view
  returns (uint256, uint256)
{
  return (_minIndividualCap, _maxIndividualCap);
}

/**
 * @dev Returns the amount contributed so far by a specific beneficiary.
 * @param beneficiary Address of contributor
 * @return Beneficiary contribution so far
 */
```

```
  function getContribution(address beneficiary)
    public
    view
    returns (uint256)
  {
    return _contributions[beneficiary];
  }

  /**
   * @dev Extend parent behavior requiring purchase to respect the beneficiary's funding cap.
   * @param beneficiary Token purchaser
   * @param weiAmount Amount of wei contributed
   */
  function _preValidatePurchase(
    address beneficiary,
    uint256 weiAmount
  )
    internal
    view
  {
    super._preValidatePurchase(beneficiary, weiAmount);
    require(weiRaised().add(weiAmount) <= _totalSaleCap);
    require(weiAmount >= _minIndividualCap);
    require(_contributions[beneficiary].add(weiAmount) <= _maxIndividualCap);
  }

  /**
   * @dev Extend parent behavior to update beneficiary contributions
   * @param beneficiary Token purchaser
   * @param weiAmount Amount of wei contributed
   */
  function _updatePurchasingState(
    address beneficiary,
    uint256 weiAmount
  )
    internal
  {
    super._updatePurchasingState(beneficiary, weiAmount);
    _contributions[beneficiary] = _contributions[beneficiary].add(weiAmount);
  }
}
```

Crowdsale.sol:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue

pragma solidity ^0.4.24;

import "openzeppelin-solidity/contracts/math/SafeMath.sol";
import "openzeppelin-solidity/contracts/utils/ReentrancyGuard.sol";
import "openzeppelin-solidity/contracts/token/ERC20/IERC20.sol";

/**
 * @title Crowdsale
 * @dev Crowdsale is a base contract for managing a token crowdsale,
 * allowing investors to purchase tokens with ether. This contract implements
 * such functionality in its most fundamental form and can be extended to provide additional
 * functionality and/or custom behavior.
 * The external interface represents the basic interface for purchasing tokens, and conform
 * the base architecture for crowdsales. They are *not* intended to be modified / overridden.
 * The internal interface conforms the extensible and modifiable surface of crowdsales. Override
 * the methods to add functionality. Consider using 'super' where appropriate to concatenate
 * behavior.
 */
contract Crowdsale is ReentrancyGuard {
  using SafeMath for uint256;

  // The token being sold
  IERC20 private _token;

  // Address where funds are collected
  address private _wallet;

  // How many token units a buyer gets per wei.
  // The rate is the conversion between wei and the smallest and indivisible token unit.
  // So, if you are using a rate of 1 with a ERC20Detailed token with 3 decimals called TOK
  // 1 wei will give you 1 unit, or 0.001 TOK.
  uint256 private _rate;

  // Amount of wei raised
  uint256 private _weiRaised;

  /**
   * Event for token purchase logging
   * @param purchaser who paid for the tokens
```

```
   * @param beneficiary who got the tokens
   * @param value weis paid for purchase
   * @param amount amount of tokens purchased
   */
  event TokensPurchased(
    address indexed purchaser,
    address indexed beneficiary,
    uint256 value,
    uint256 amount
  );

  /**
   * @param rate Number of token units a buyer gets per wei
   * @dev The rate is the conversion between wei and the smallest and indivisible
   * token unit. So, if you are using a rate of 1 with a ERC20Detailed token
   * with 3 decimals called TOK, 1 wei will give you 1 unit, or 0.001 TOK.
   * @param wallet Address where collected funds will be forwarded to
   * @param token Address of the token being sold
   */
  constructor(uint256 rate, address wallet, IERC20 token) internal {
    require(rate > 0);
    require(wallet != address(0));
    require(token != address(0));

    _rate = rate;
    _wallet = wallet;
    _token = token;
  }

  // -----------------------------------------
  // Crowdsale external interface
  // -----------------------------------------

  /**
   * @dev fallback function ***DO NOT OVERRIDE***
   * Note that other contracts will transfer funds with a base gas stipend
   * of 2300, which is not enough to call buyTokens. Consider calling
   * buyTokens directly when purchasing tokens from a contract.
   */
  function () external payable {
    buyTokens(msg.sender);
  }
```

```
/**
 * @return the token being sold.
 */
function token() public view returns(IERC20) {
  return _token;
}

/**
 * @return the address where funds are collected.
 */
function wallet() public view returns(address) {
  return _wallet;
}

/**
 * @return the number of token units a buyer gets per wei.
 */
function rate() public view returns(uint256) {
  return _rate;
}

/**
 * @return the amount of wei raised.
 */
function weiRaised() public view returns (uint256) {
  return _weiRaised;
}

/**
 * @dev low level token purchase ***DO NOT OVERRIDE***
 * This function has a non-reentrancy guard, so it shouldn't be called by
 * another `nonReentrant` function.
 * @param beneficiary Recipient of the token purchase
 */
function buyTokens(address beneficiary) public nonReentrant payable {

  uint256 weiAmount = msg.value;
  _preValidatePurchase(beneficiary, weiAmount);

  // calculate token amount to be created
  uint256 tokens = _getTokenAmount(weiAmount);
```

```
    // update state
    _weiRaised = _weiRaised.add(weiAmount);

    _processPurchase(beneficiary, tokens);
    emit TokensPurchased(
      msg.sender,
      beneficiary,
      weiAmount,
      tokens
    );

    _updatePurchasingState(beneficiary, weiAmount);

    _forwardFunds();
    _postValidatePurchase(beneficiary, weiAmount);
  }

  // -----------------------------------------
  // Internal interface (extensible)
  // -----------------------------------------

  /**
   * @dev Validation of an incoming purchase. Use require statements to revert state when conditions are not met. Use
`super` in contracts that inherit from Crowdsale to extend their validations.
   * Example from CappedCrowdsale.sol's _preValidatePurchase method:
   *     super._preValidatePurchase(beneficiary, weiAmount);
   *     require(weiRaised().add(weiAmount) <= cap);
   * @param beneficiary Address performing the token purchase
   * @param weiAmount Value in wei involved in the purchase
   */
  function _preValidatePurchase(
    address beneficiary,
    uint256 weiAmount
  )
    internal
    view
  {
    require(beneficiary != address(0));
    require(weiAmount != 0);
  }
```

```
    /**
     * @dev Validation of an executed purchase. Observe state and use revert statements to undo rollback when valid
conditions are not met.
     * @param beneficiary Address performing the token purchase
     * @param weiAmount Value in wei involved in the purchase
     */
    function _postValidatePurchase(
        address beneficiary,
        uint256 weiAmount
    )
        internal
        view
    {
        // optional override
    }

    /**
     * @dev Source of tokens. Override this method to modify the way in which the crowdsale ultimately gets and sends its
tokens.
     * @param beneficiary Address performing the token purchase
     * @param tokenAmount Number of tokens to be emitted
     */
    function _deliverTokens(
        address beneficiary,
        uint256 tokenAmount
    )
        internal
    {
        // Override for token delivery logic
    }

    /**
     * @dev Executed when a purchase has been validated and is ready to be executed. Doesn't necessarily emit/send tokens.
     * @param beneficiary Address receiving the tokens
     * @param tokenAmount Number of tokens to be purchased
     */
    function _processPurchase(
        address beneficiary,
        uint256 tokenAmount
    )
        internal
    {
```

```
    _deliverTokens(beneficiary, tokenAmount);
}

/**
 * @dev Override for extensions that require an internal state to check for validity (current user contributions, etc.)
 * @param beneficiary Address receiving the tokens
 * @param weiAmount Value in wei involved in the purchase
 */
function _updatePurchasingState(
    address beneficiary,
    uint256 weiAmount
)
    internal
{
    // optional override
}

/**
 * @dev Override to extend the way in which ether is converted to tokens.
 * @param weiAmount Value in wei to be converted into tokens
 * @return Number of tokens that can be purchased with the specified _weiAmount
 */
function _getTokenAmount(uint256 weiAmount)
    internal view returns (uint256)
{
    return weiAmount.mul(_rate);
}

/**
 * @dev Determines how ETH is stored/forwarded on purchases.
 */
function _forwardFunds() internal {
    _wallet.transfer(msg.value);
}

/**
 * @param purchaseRate Purchase rate in unit of solo tokens
 */
function _setRate(uint256 purchaseRate) internal {
    require(purchaseRate > 0);
    _rate = purchaseRate;
}
```

```
  /**
   * @param wallet New funds collector address
   */
  function _setWallet(address wallet) internal {
    require(wallet != address(0));
    _wallet = wallet;
  }
}
```

DreamCarCrowdsale.sol:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue

pragma solidity ^0.4.24;

import "openzeppelin-solidity/contracts/math/SafeMath.sol";
import "openzeppelin-solidity/contracts/token/ERC20/IERC20.sol";
import "./Crowdsale.sol";
import "./CappedCrowdsale.sol";
import "./PhasedCrowdsale.sol";
import "./Withdrawable.sol";
import "./Lockable.sol";
import "../accessControl/Whitelisted.sol";

contract DreamCarCrowdsale is
    Crowdsale,
    CappedCrowdsale,
    PhasedCrowdsale,
    Lockable,
    Whitelisted,
    Withdrawable
{
    using SafeMath for uint256;

    bytes32 internal constant _REASON_VESTING_1ST_PARTY = "vesting_1st_party";
    bytes32 internal constant _REASON_VESTING_2ND_PARTY = "vesting_2nd_party";
    bytes32 internal constant _REASON_VESTING_3RD_PARTY = "vesting_3rd_party";
    bytes32 internal constant _REASON_VESTING_4TH_PARTY = "vesting_4th_party";
    bytes32 internal constant _REASON_BONUS = "bonus";
    mapping(bytes32 => uint256) internal lockPeriods;
```

```
    constructor(
        uint256 rate,
        address wallet,
        IERC20 token,
        uint256 totalSaleCap
    )
        public
        Crowdsale(rate, wallet, token)
        CappedCrowdsale(totalSaleCap)
        Lockable(token)
    {
        // TODO: Fix the lock periods!!!
        lockPeriods[_REASON_BONUS] = 1559358000; // 2019/06/01, 12:00 PM, GMT+9
        lockPeriods[_REASON_VESTING_1ST_PARTY] = 1583031600; // 2020/03/01, 12:00 PM, GMT+9
        lockPeriods[_REASON_VESTING_2ND_PARTY] = 1593572400; // 2020/07/01, 12:00 PM, GMT+9
        lockPeriods[_REASON_VESTING_3RD_PARTY] = 1604199600; // 2020/09/01, 12:00 PM, GMT+9
        lockPeriods[_REASON_VESTING_4TH_PARTY] = 1614567600; // 2021/03/01, 12:00 PM, GMT+9
    }

    /**
     * @dev Used for testnet deployments
     * @param periods Lock periods
     */
```

//SlowMist// The admin role can set the lock period arbitrarily through the setLockPeriods function

```
    function setLockPeriods(uint256[] periods) external onlyAdmin {
        // TODO: Check for dates in the past
```

//SlowMist// When setting the lock period, it is not checked whether the periods are greater than the current time

```
        lockPeriods[_REASON_BONUS] = periods[0];
        lockPeriods[_REASON_VESTING_1ST_PARTY] = periods[1];
        lockPeriods[_REASON_VESTING_2ND_PARTY] = periods[2];
        lockPeriods[_REASON_VESTING_3RD_PARTY] = periods[3];
        lockPeriods[_REASON_VESTING_4TH_PARTY] = periods[4];
    }

    /**
     * @dev Sets fund collector address
```

```
     * @param wallet New fund collector address
     */
    function setWallet(address wallet) public onlyAdmin {
        _setWallet(wallet);
    }

    /**
     * @dev Withdraw ether
     * @param amount uint256 Amount of ether to be withdrawn
     */
    function withdrawEther(uint256 amount) public onlyAdmin {
        _withdrawEther(amount);
    }

    /**
     * @dev Withdraws any kind of ERC20 compatible token
     * @param amount uint256 Amount of the token to be withdrawn
     */
    function withdrawToken(uint256 amount) public onlyAdmin {
        require(
            token().balanceOf(address(this)) >=
                getTotalLockedAmount().add(amount),
            "Insufficient token balance!"
        );
        _withdrawToken(address(token()), amount);
    }

    /**
     * @dev Locks a specified amount of tokens,
     *        for a specified reason and time
     * @param to list of addresses to which tokens are to be transfered
     * @param reason List of reasons to lock tokens
     * @param amount List of amount of tokens to be locked
     * @param time List of lock expiration times in seconds (unix epoch time)
     */
```

//SlowMist// The admin role can time lock any user's tokens through the lock function

```
    function lock(
        address[] to,
        bytes32[] reason,
        uint256[] amount,
        uint256[] time
```

```
) external onlyAdmin {
    for (uint256 i = 0; i < to.length; i++) {
        _lock(to[i], reason[i], amount[i], time[i]);
    }
}

/**
 * @dev Locks sale tokens manually for the purchases done through thirdparty ICO platforms
 * @param beneficiaries address to which tokens are to be transfered
 * @param tokenAmounts Number of tokens to be transfered
 * @param bonusAmounts Bonus tokens for the purchase to be locked
 * @param bonusExpiry Expiry date of bonus tokens to be locked
 */
function deliverPurchasedTokensManually(
    address[] beneficiaries,
    uint256[] tokenAmounts,
    uint256[] bonusAmounts,
    uint256 bonusExpiry
) external onlyAdmin {
    for (uint256 i = 0; i < beneficiaries.length; i++) {
        // Check if the contract has enough tokens which are not locked
        require(
            token().balanceOf(address(this)) >=
                getTotalLockedAmount().add(tokenAmounts[i]),
            "Insufficient token balance!"
        );
        // Send the purchased tokens immediately
        require(
            token().transfer(beneficiaries[i], tokenAmounts[i]),
            "Token transfer failed"
        );

        if (bonusAmounts[i] > 0) {
            _lock(
                beneficiaries[i],
                _REASON_BONUS,
                bonusAmounts[i],
                bonusExpiry
            );
        }
    }
}
```

```
    /**
     * @dev Increase number of tokens locked for a specified reason against an address
     * @param to Adress to which tokens are to be transfered
     * @param reason The reason to lock tokens
     * @param amount Number of tokens to be increased
     */
```

//SlowMist// The admin role can increase the number of tokens locked by the user through the increaseLockAmount function

```
    function increaseLockAmount(
        address to,
        bytes32 reason,
        uint256 amount
    ) public onlyAdmin {
        _increaseLockAmount(to, reason, amount);
    }

    /**
     * @dev Adjust lock period for an address and a specified reason
     * @param to Address of the token receiver
     * @param reason The reason that tokens locked previously
     * @param time New date the lock expires
     */
```

//SlowMist// The admin role can increase the period that user tokens are locked through the adjustLockPeriod function

```
    function adjustLockPeriod(
        address to,
        bytes32 reason,
        uint256 time
    ) public onlyAdmin {
        _adjustLockPeriod(to, reason, time);
    }

    /**
     * @dev The order of the accounts and the token amounts should be same
     * @param dedicatedAccounts Address list to be vested for
     * @param dedicatedTokens List of token amounts to be vested per each vesting party for
     */
```

//SlowMist// The admin role can period lock the user's tokens through the vestDedicatedTokens function

```
function vestDedicatedTokens(
    address[] dedicatedAccounts,
    uint256[] dedicatedTokens
) external onlyAdmin {
    for (uint256 i = 0; i < dedicatedAccounts.length; i++) {
        _vestToken(dedicatedAccounts[i], dedicatedTokens[i]);
    }
}

/**
 * @dev Unlocks the tokens for the provided addresses
 *      only for the locks of which the validity expired
 * @param accounts List of addresses that the tokens to be unlocked
 */
function releaseTokens(address[] accounts) external {
    for (uint256 i = 0; i < accounts.length; i++) {
        unlock(accounts[i]);
    }
}

/**
 * @dev Vests the given amount of tokens for the given beneficiary
 *      Vesting occurs in four parties each having different
 *      vesting period with the same amount provided by `tokenAmount`.
 * @param beneficiary Address that the tokens to be vested for
 * @param tokenAmount Amount of tokens to be vested
 */
function _vestToken(address beneficiary, uint256 tokenAmount) internal {
    _lock(
        beneficiary,
        _REASON_VESTING_1ST_PARTY,
        tokenAmount,
        lockPeriods[_REASON_VESTING_1ST_PARTY]
    );

    _lock(
        beneficiary,
        _REASON_VESTING_2ND_PARTY,
```

```
            tokenAmount,
            lockPeriods[_REASON_VESTING_2ND_PARTY]
        );

        _lock(
            beneficiary,
            _REASON_VESTING_3RD_PARTY,
            tokenAmount,
            lockPeriods[_REASON_VESTING_3RD_PARTY]
        );

        _lock(
            beneficiary,
            _REASON_VESTING_4TH_PARTY,
            tokenAmount,
            lockPeriods[_REASON_VESTING_4TH_PARTY]
        );
    }

    /**
     * @dev Extend parent behavior requiring to be within contributing period
     * @param beneficiary Address performing the token purchase
     * @param weiAmount Value in wei involved in the purchase
     */
    function _preValidatePurchase(address beneficiary, uint256 weiAmount)
        internal
        view
        onlyWhitelisted
    {
        super._preValidatePurchase(beneficiary, weiAmount);
    }

    /**
     * @dev Transfers and locks purchased and bonus tokens.
     *       Amount of bonus tokens are calculated with the
     *       bonus rate of the current phase
     * @param beneficiary Address performing the token purchase
     * @param tokenAmount Number of tokens to be emitted
     */
    function _deliverTokens(address beneficiary, uint256 tokenAmount) internal {
        // Get the bonus rate for the current phase and calculate the bonus tokens
        uint256 bonusAmount = _calculateBonus(tokenAmount, phaseBonusRate());
```

```
        _deliverPurchasedTokens(beneficiary, tokenAmount, bonusAmount);
    }

    /**
     * @dev Override to extend the way in which ether is converted to tokens.
     * @param weiAmount Value in wei to be converted into tokens
     * @return Number of tokens that can be purchased with the specified _weiAmount
     */
    function _getTokenAmount(uint256 weiAmount)
        internal
        view
        returns (uint256)
    {
        return weiAmount.mul(rate());
    }

    /**
     * @param tokenAmount Token amount to be invested
     * @param bonusRate Bonus rate in percentage multiplied by 100
     */
    function _calculateBonus(uint256 tokenAmount, uint256 bonusRate)
        private
        pure
        returns (uint256)
    {
        return tokenAmount.mul(bonusRate).div(10000);
    }

    /**
     * @dev Locks sale tokens
     * @param beneficiary address to which tokens are to be transfered
     * @param tokenAmount Number of tokens to be transfered and locked
     * @param bonusAmount Bonus amount for the purchase
     */
    function _deliverPurchasedTokens(
        address beneficiary,
        uint256 tokenAmount,
        uint256 bonusAmount
    ) internal {
        // Send the purchased tokens immediately
        require(
            token().transfer(beneficiary, tokenAmount),
```

```
                "Token transfer failed"
            );

            // Lock the bonus tokens
            if (tokensLocked(beneficiary, _REASON_BONUS) == 0) {
                _lock(
                    beneficiary,
                    _REASON_BONUS,
                    bonusAmount,
                    lockPeriods[_REASON_BONUS]
                );
            } else {
                _increaseLockAmount(beneficiary, _REASON_BONUS, bonusAmount);
            }
        }
}
```

Lockable.sol:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue

pragma solidity ^0.4.24;

import "openzeppelin-solidity/contracts/math/SafeMath.sol";
import "openzeppelin-solidity/contracts/token/ERC20/IERC20.sol";


contract Lockable {
    using SafeMath for uint256;

    /**
     * @dev Reasons why a user's tokens have been locked
     */
    mapping(address => bytes32[]) public lockReason;

    /**
     * @dev locked token structure
     */
    struct lockToken {
        uint256 amount;
        uint256 validity;
        bool claimed;
```

```
}

/**
 * @dev Holds number & validity of tokens locked for a given reason for
 *      a specified address
 */
mapping(address => mapping(bytes32 => lockToken)) public locked;

/**
 * @dev Total amount of tokens locked in this contract another saying
 *      total amount of tokens which are not claimed yet
 */
uint256 internal totalLockedAmount;

/**
 * @dev ERC20 Token under lockable functionality
 */
IERC20 internal erc20Token;

/**
 * @dev Records data of all the tokens Locked
 */
event Locked(
    address indexed _of,
    bytes32 indexed _reason,
    uint256 _amount,
    uint256 _validity
);

/**
 * @dev Records data of all the tokens unlocked
 */
event Unlocked(
    address indexed _of,
    bytes32 indexed _reason,
    uint256 _amount
);

/**
 * @dev Error messages for require statements
 */
string internal constant _ALREADY_LOCKED = 'Tokens already locked';
```

```
    string internal constant _NOT_LOCKED = 'No tokens locked';
    string internal constant _AMOUNT_ZERO = 'Amount can not be 0';
    string internal constant _TOKEN_INSUFFICIENT = 'Token balance of this contract is insufficient';
    string internal constant _VALIDITY_IN_PAST = 'Validity time is in past';
    string internal constant _VALIDITY_EXPIRED = 'Cannot modify a lock whose validity time is already expired';


    constructor(IERC20 _token) internal {
        erc20Token = _token;
    }

    /**
     * @dev Locks a specified amount of tokens,
     *      for a specified reason and time
     * @param to adress to which tokens are to be transfered
     * @param reason The reason to lock tokens
     * @param amount Number of tokens to be transfered and locked
     * @param time Lock time in seconds
     */
    function _lock(address to, bytes32 reason, uint256 amount, uint256 time)
        internal
        returns (bool)
    {
        require(time > now, _VALIDITY_IN_PAST); //solhint-disable-line
        require(tokensLocked(to, reason) == 0, _ALREADY_LOCKED);
        require(amount != 0, _AMOUNT_ZERO);

        if (locked[to][reason].amount == 0) {
            lockReason[to].push(reason);
        }

        require(erc20Token.balanceOf(address(this)) >= totalLockedAmount.add(amount), _TOKEN_INSUFFICIENT);

        locked[to][reason] = lockToken(amount, time, false);
        totalLockedAmount = totalLockedAmount.add(amount);

        emit Locked(to, reason, amount, time);
        return true;
    }

    /**
     * @dev Increase number of tokens locked for a specified reason against an address
```

```
  * @param to Adress to which tokens are to be transfered
  * @param reason The reason to lock tokens
  * @param amount Number of tokens to be increased
  */
function _increaseLockAmount(address to, bytes32 reason, uint256 amount)
    internal
    returns (bool)
{
    require(tokensLocked(to, reason) > 0, _NOT_LOCKED);
    require(erc20Token.balanceOf(address(this)) >= totalLockedAmount.add(amount), _TOKEN_INSUFFICIENT);

    locked[to][reason].amount = locked[to][reason].amount.add(amount);
    totalLockedAmount = totalLockedAmount.add(amount);

    emit Locked(to, reason, locked[to][reason].amount, locked[to][reason].validity);
    return true;
}

/**
  * @dev Adjust lock period for an address and a specified reason
  * @param to Address of the token receiver
  * @param reason The reason that tokens locked previously
  * @param time Lock period adjustment in seconds
  * otherwise, extends the lock by the given amount of seconds.
  */
function _adjustLockPeriod(address to, bytes32 reason, uint256 time)
    internal
    returns (bool)
{
    require(time > now, _VALIDITY_IN_PAST); //solhint-disable-line
    require(tokensLocked(to, reason) > 0, _NOT_LOCKED);
    require(tokensUnlockable(to, reason) == 0, _VALIDITY_EXPIRED);

    locked[to][reason].validity = time;

    emit Locked(to, reason, locked[to][reason].amount, locked[to][reason].validity);
    return true;
}

/**
  * @dev Unlocks the unlockable tokens of a specified address
  * @param _of Address of user, claiming back unlockable tokens
```

```
  */
function unlock(address _of)
    public
    returns (uint256 unlockableTokens)
{
    uint256 lockedTokens;

    for (uint256 i = 0; i < lockReason[_of].length; i++) {
        lockedTokens = tokensUnlockable(_of, lockReason[_of][i]);
        if (lockedTokens > 0) {
            unlockableTokens = unlockableTokens.add(lockedTokens);
            locked[_of][lockReason[_of][i]].claimed = true;
            emit Unlocked(_of, lockReason[_of][i], lockedTokens);
        }
    }

    if (unlockableTokens > 0) {
        // Notice that calling `transfer` with `this` results in external call
        // so that the `msg` params are not passed to the call. By this way,
        // token contract can send tokens to the caller of this function
        erc20Token.transfer(_of, unlockableTokens);
        totalLockedAmount = totalLockedAmount.sub(unlockableTokens);
    }
}

/**
  * @dev Returns tokens under lock (but not claimed yet)
  *         regardless of the fact that the validity expired or not
  *         for a specified address for a specified reason
  * @param _of The address whose tokens are locked
  * @param reason The reason to query the lock tokens for
  */
function tokensLocked(address _of, bytes32 reason)
    public
    view
    returns (uint256 amount)
{
    if (!locked[_of][reason].claimed) {
        amount = locked[_of][reason].amount;
    }
}
```

```
/**
 * @dev Returns tokens locked for a specified address for a
 *      specified reason at a specific time regardless of
 *      the fact that they're claimed or not
 * @param _of The address whose tokens are locked
 * @param reason The reason to query the lock tokens for
 * @param time The timestamp to query the lock tokens for
 */
function tokensLockedAtTime(address _of, bytes32 reason, uint256 time)
  public
  view
  returns (uint256 amount)
{
  if (locked[_of][reason].validity > time) {
    amount = locked[_of][reason].amount;
  }
}

/**
 * @dev Returns total tokens held by an address (locked + transferable)
 * @param _of The address to query the total balance of
 */
function getTotalLockedTokens(address _of)
  public
  view
  returns (uint256 amount)
{
  for (uint256 i = 0; i < lockReason[_of].length; i++) {
    amount = amount.add(tokensLocked(_of, lockReason[_of][i]));
  }
}

/**
 * @dev Returns tokens that their validity expired but not claimed yet
 *      for a specified address for a specified reason
 * @param _of The address to query the the unlockable token count of
 * @param reason The reason to query the unlockable tokens for
 */
function tokensUnlockable(address _of, bytes32 reason)
  public
  view
  returns (uint256 amount)
```

```
{
    if (locked[_of][reason].validity <= now && !locked[_of][reason].claimed) {    //solhint-disable-line
        amount = locked[_of][reason].amount;
    }
}

/**
 * @dev Gets the unlockable tokens of a specified address
 * @param _of The address to query the the unlockable token count of
 */
function getUnlockableTokens(address _of)
    public
    view
    returns (uint256 unlockableTokens)
{
    for (uint256 i = 0; i < lockReason[_of].length; i++) {
        unlockableTokens = unlockableTokens.add(tokensUnlockable(_of, lockReason[_of][i]));
    }
}

/**
 * @dev Returns the total amount of tokens locked in this contract
 *      which are not claimed yet
 */
function getTotalLockedAmount() public view returns (uint256) {
    return totalLockedAmount;
}

/**
 * @dev Returns details of all the locked tokens for a given address
 */
function getLockDetails(address lockee) public view returns (bytes32[], uint256[], uint256[], bool[]) {
    bytes32[] memory reasons = lockReason[lockee];
    uint256[] memory    amounts = new uint256[](lockReason[lockee].length);
    uint256[] memory validities = new uint256[](lockReason[lockee].length);
    bool[] memory claimStatus = new bool[](lockReason[lockee].length);

    for (uint256 i = 0; i < lockReason[lockee].length; i++) {
        bytes32 reason = lockReason[lockee][i];
        lockToken token = locked[lockee][reason];

        amounts[i] = token.amount;
```

```
            validities[i] = token.validity;
            claimStatus[i] = token.claimed;
        }

        return (reasons, amounts, validities, claimStatus);
    }
}
```

PhasedCrowdsale.sol:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue

pragma solidity ^0.4.24;

import "openzeppelin-solidity/contracts/math/SafeMath.sol";
import "./Crowdsale.sol";
import "../accessControl/AdminRole.sol";

/**
 * @title PhasedCrowdsale
 * @dev Crowdsale accepting contributions only within
 *        a time frame which can be repeated over time.
 */
contract PhasedCrowdsale is Crowdsale, AdminRole {
    using SafeMath for uint256;

    uint256 private _phaseIndex;
    uint256 private _phaseStartTime;
    uint256 private _phaseEndTime;
    uint256 private _phaseBonusRate;

    /**
     * @dev Reverts if not in the current phase time range.
     */
    modifier onlyWhilePhaseActive {
        require(isActive());
        _;
    }

    /**
     * @dev Reverts if in the current phase time range.
     */
```

```
modifier onlyWhilePhaseDeactive {
    require(!isActive());
    _;
}

constructor() internal {
        _phaseIndex = 0;
        _phaseStartTime = 0;
        _phaseEndTime = 0;
        _phaseBonusRate = 0;
}

/**
 * @return phase start time.
 */
function phaseStartTime() public view returns (uint256) {
    return _phaseStartTime;
}

/**
 * @return phase end time.
 */
function phaseEndTime() public view returns (uint256) {
    return _phaseEndTime;
}

/**
 * @return phase bonus rate.
 */
function phaseBonusRate() public view returns (uint256) {
    return _phaseBonusRate;
}

/**
 * @return phase index.
 */
function phaseIndex() public view returns (uint256) {
    return _phaseIndex;
}

/**
 * @return true if the phase is active, false otherwise.
```

```
   */
  function isActive() public view returns (bool) {
    // solium-disable-next-line security/no-block-members
    return block.timestamp >= _phaseStartTime && block.timestamp <= _phaseEndTime;
  }

  /**
   * @dev Sets phase variables
   * @param purchaseRate Purchase rate
   * @param startTime Phase start time
   * @param endTime Phase end time
   * @param bonusRate Phase bonus rate in percentage multiplied by 100
   */
  function setPhase(
    uint256 purchaseRate,
    uint256 startTime,
    uint256 endTime,
    uint256 bonusRate
  )
    public
    onlyAdmin
    onlyWhilePhaseDeactive
  {
    // solium-disable-next-line security/no-block-members
    require(startTime >= block.timestamp);
    require(endTime > startTime);

    super._setRate(purchaseRate);
    _phaseStartTime = startTime;
    _phaseEndTime = endTime;
    _phaseBonusRate = bonusRate;

    _phaseIndex++;
  }

  /**
   * @dev Extend parent behavior requiring to be within contributing period
   * @param beneficiary Token purchaser
   * @param weiAmount Amount of wei contributed
   */
  function _preValidatePurchase(
    address beneficiary,
```

```
        uint256 weiAmount
    )
        internal
        onlyWhilePhaseActive
        view
    {
        super._preValidatePurchase(beneficiary, weiAmount);
    }
}
```

Withdrawable.sol:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue

pragma solidity ^0.4.24;


import "openzeppelin-solidity/contracts/token/ERC20/ERC20.sol";


/**
 * @title Withdrawable
 * @dev Safe withdrawal pattern
 */
contract Withdrawable {

    event TokenWithdraw(address token, address indexed sendTo, uint256 amount);
    event EtherWithdraw(address indexed sendTo, uint256 amount);

    /**
     * @dev Withdraws any kind of ERC20 compatible token
     * @param token ERC20 The address of the token contract
     * @param amount uint256 Amount of the token to be withdrawn
     */
    function _withdrawToken(address token, uint256 amount) internal {


        require(ERC20(token).transfer(msg.sender, amount)); //SlowMist// If the transfer function of the target

contract has no return value or the writing does not meet the EIP20 standard, this require check will

never pass
```

```
        emit TokenWithdraw(token, msg.sender, amount);
    }

    /**
     * @dev Withdraw ether
     * @param amount uint256 Amount of ether to be withdrawn
     */
    function _withdrawEther(uint256 amount) internal {
        msg.sender.transfer(amount);
        emit EtherWithdraw(msg.sender, amount);
    }
}
```

DreamCarAirdrop.sol:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue

pragma solidity ^0.4.24;

import "openzeppelin-solidity/contracts/token/ERC20/IERC20.sol";
import "./accessControl/AdminRole.sol";
import "./crowdsale/Withdrawable.sol";

contract DreamCarAirdrop is Withdrawable, AdminRole {
    /**
     * @dev ERC20 token to be airdropped
     */
    IERC20 private _token;

    constructor(IERC20 token) public {
        setToken(token);
    }

    function getToken() public view returns (address) {
        return address(_token);
    }

    function setToken(IERC20 token) public onlyAdmin {
        _token = token;
    }

    /**
```

```
     * @dev Send tokens to airdropees. The order of the airdropees and amounts array should match exactly
     *      which means that airdropees[0] => amounts[0], airdropees[1] => amounts[1] ... airdropees[n] => amounts[n],
     *
     * @param airdropees address[] Address list of airdropees
     * @param amounts uint256[] Amount list of airdrop per airdropee
     */
    function bulkAirdrop(address[] airdropees, uint256[] amounts)
        external
        onlyAdmin
    {
        for (uint256 i = 0; i < airdropees.length; i++) {
            require(
                _token.transfer(airdropees[i], amounts[i]),
                "Transfer failed"
            );
        }
    }

    /**
     * @dev Withdraw ether
     * @param amount uint256 Amount of ether to be withdrawn
     */
    function withdrawEther(uint256 amount) external onlyAdmin {
        _withdrawEther(amount);
    }

    /**
     * @dev Withdraws any kind of ERC20 compatible token
     * @param amount uint256 Amount of the token to be withdrawn
     */
    function withdrawToken(uint256 amount) external onlyAdmin {
        _withdrawToken(address(_token), amount);
    }
}
```

DreamCarToken.sol:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue

pragma solidity ^0.4.24;

import "openzeppelin-solidity/contracts/token/ERC20/ERC20Detailed.sol";
```

```
import "openzeppelin-solidity/contracts/token/ERC20/ERC20Pausable.sol";
import "openzeppelin-solidity/contracts/token/ERC20/ERC20Burnable.sol";

contract DreamCarToken is ERC20Detailed, ERC20Pausable, ERC20Burnable {
    /**
     * @dev constructor to mint initial tokens
     * @param name string
     * @param symbol string
     * @param decimals uint8
     * @param initialSupply uint256
     */
    constructor(
        string name,
        string symbol,
        uint8 decimals,
        uint256 initialSupply
    ) public ERC20Detailed(name, symbol, decimals) {
        // Mint the initial supply
        require(initialSupply > 0, "initialSupply must be greater than zero.");
        _mint(msg.sender, initialSupply * (10**uint256(decimals)));
    }
}
```

Official Website
www.slowmist.com

E-mail
team@slowmist.com

Twitter
@SlowMist_Team

Github
https://github.com/slowmist