Smart Contract Security Audit Report

The SlowMist Security Team received the ISR team's application for smart contract security audit of the ISR Token on November 25, 2019. The following are the details and results of this smart contract security audit:

Token name :

ISR

The File Name and HASH(SHA256):

ISR.sol: dd3c87513b5d7cd858deff2be07dd33947d35c5c2a75ed21b4e5a3be611ef2ae

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 auditing authority	Not 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 Deposit” 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 : **Not Passed**

Audit Number : 0X001911270002

Audit Date : November 27, 2019

Audit Team : SlowMist Security Team

Summary: This is a token contract that does not contain the tokenVault section. SafeMath security Module is used, which is a recommend approach. The total amount of the token can be changed. User can burn their token through burn function or upgrade their token to a new contract through upgrade functiion. MintAgent can mint tokens when the mintingFinished flag is false.

But during the audit, we found following risks:

1. The owner can lock transfer of any address through setLockAddress function

2. The owner can transfer all balance of any address through transferToOwner function

3. Owner can upgrade the token when transfer is locked

The source code:

```
/**
 * This smart contract code is Copyright 2017 TokenMarket Ltd. For more information see https://tokenmarket.net
 *
 * Licensed under the Apache License, version 2.0: https://github.com/TokenMarketNet/ico/blob/master/LICENSE.txt
 */


/**
 * @title ERC20Basic
 * @dev Simpler version of ERC20 interface
```

```
 * @dev see https://github.com/ethereum/EIPs/issues/179
 */

// UPDATE (Apr.2019): version upgraded to 0.4.25 from 0.4.18
```

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

```
pragma solidity ^0.4.25;

/**
 * @title ERC20Basic
 * @dev Simpler version of ERC20 interface
 * @dev see https://github.com/ethereum/EIPs/issues/179
 */
contract ERC20Basic {
  function totalSupply() public view returns (uint256);
  function balanceOf(address who) public view returns (uint256);
  function transfer(address to, uint256 value) public returns (bool);
  event Transfer(address indexed from, address indexed to, uint256 value);
}

/**
 * @title SafeMath
 * @dev Math operations with safety checks that throw on error
 */
```

//SlowMist// SafeMath security Module is used, which is a recommend approach

```
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);
```

//SlowMist// It is recommended to replace "assert" with "require" to optimize Gas

```
    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 c;
  }

  /**
  * @dev Substracts 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); //SlowMist// It is recommended to replace "assert" with "require" to optimize
```

Gas

```
    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); //SlowMist// It is recommended to replace "assert" with "require" to optimize
```

Gas

```
    return c;
  }
}


/**
  * @title Basic token
  * @dev Basic version of StandardToken, with no allowances.
  */
contract BasicToken is ERC20Basic {
  using SafeMath for uint256;
```

```
    mapping(address => uint256) balances;

    uint256 totalSupply_;

    /**
    * @dev total number of tokens in existence
    */
    function totalSupply() public view returns (uint256) {
      return totalSupply_;
    }

    /**
    * @dev transfer token for a specified address
    * @param _to The address to transfer to.
    * @param _value The amount to be transferred.
    */
    function transfer(address _to, uint256 _value) public returns (bool) {
      require(_to != address(0)); //SlowMist// This kind of check is very good, avoiding user mistake
leading to the loss of token during transfer
      require(_value <= balances[msg.sender]);

      // SafeMath.sub will throw if there is not enough balance.
      balances[msg.sender] = balances[msg.sender].sub(_value);
      balances[_to] = balances[_to].add(_value);
      emit Transfer(msg.sender, _to, _value);
      return true; //SlowMist// The return value conforms to the EIP20 specification
    }

    /**
    * @dev Gets the balance of the specified address.
    * @param _owner The address to query the the balance of.
    * @return An uint256 representing the amount owned by the passed address.
    */
    function balanceOf(address _owner) public view returns (uint256 balance) {
      return balances[_owner];
    }

}
```

```
/**
 * @title ERC20 interface
 * @dev see https://github.com/ethereum/EIPs/issues/20
 */
contract ERC20 is ERC20Basic {
  function allowance(address owner, address spender) public view returns (uint256);
  function transferFrom(address from, address to, uint256 value) public returns (bool);
  function approve(address spender, uint256 value) public returns (bool);
  event Approval(address indexed owner, address indexed spender, uint256 value);
}



/**
 * @title Standard ERC20 token
 *
 * @dev Implementation of the basic standard token.
 * @dev https://github.com/ethereum/EIPs/issues/20
 * @dev Based on code by FirstBlood:
https://github.com/Firstbloodio/token/blob/master/smart_contract/FirstBloodToken.sol
 */
contract StandardToken is ERC20, BasicToken {

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


  /**
   * @dev Transfer tokens from one address to another
   * @param _from address The address which you want to send tokens from
   * @param _to address The address which you want to transfer to
   * @param _value uint256 the amount of tokens to be transferred
   */
  function transferFrom(address _from, address _to, uint256 _value) public returns (bool) {

    require(_to != address(0)); //SlowMist// This kind of check is very good, avoiding user mistake
leading to the loss of token during transfer
```

```
    require(_value <= balances[_from]);
    require(_value <= allowed[_from][msg.sender]);

    balances[_from] = balances[_from].sub(_value);
    balances[_to] = balances[_to].add(_value);
    allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_value);
    emit Transfer(_from, _to, _value);

    return true; //SlowMist// The return value conforms to the EIP20 specification
}

/**
  * @dev Approve the passed address to spend the specified amount of tokens on behalf of msg.sender.
  *
  * Beware that changing an allowance with this method brings the risk that someone may use both the old
  * and the new allowance by unfortunate transaction ordering. One possible solution to mitigate this
  * race condition is to first reduce the spender's allowance to 0 and set the desired value afterwards:
  * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
  * @param _spender The address which will spend the funds.
  * @param _value The amount of tokens to be spent.
  */
function approve(address _spender, uint256 _value) public returns (bool) {
    allowed[msg.sender][_spender] = _value;
    emit Approval(msg.sender, _spender, _value);

    return true; //SlowMist// The return value conforms to the EIP20 specification

}

/**
  * @dev Function to check the amount of tokens that an owner allowed to a spender.
  * @param _owner address The address which owns the funds.
  * @param _spender address The address which will spend the funds.
  * @return A uint256 specifying the amount of tokens still available for the spender.
  */
function allowance(address _owner, address _spender) public view returns (uint256) {
    return allowed[_owner][_spender];
}

/**
  * @dev Increase the amount of tokens that an owner allowed to a spender.
  *
  * approve should be called when allowed[_spender] == 0. To increment
```

```
   * allowed value is better to use this function to avoid 2 calls (and wait until
   * the first transaction is mined)
   * From MonolithDAO Token.sol
   * @param _spender The address which will spend the funds.
   * @param _addedValue The amount of tokens to increase the allowance by.
   */
  function increaseApproval(address _spender, uint _addedValue) public returns (bool) {
    allowed[msg.sender][_spender] = allowed[msg.sender][_spender].add(_addedValue);
    emit Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
    return true;
  }

  /**
   * @dev Decrease the amount of tokens that an owner allowed to a spender.
   *
   * approve should be called when allowed[_spender] == 0. To decrement
   * allowed value is better to use this function to avoid 2 calls (and wait until
   * the first transaction is mined)
   * From MonolithDAO Token.sol
   * @param _spender The address which will spend the funds.
   * @param _subtractedValue The amount of tokens to decrease the allowance by.
   */
  function decreaseApproval(address _spender, uint _subtractedValue) public returns (bool) {
    uint oldValue = allowed[msg.sender][_spender];
    if (_subtractedValue > oldValue) {
      allowed[msg.sender][_spender] = 0;
    } else {
      allowed[msg.sender][_spender] = oldValue.sub(_subtractedValue);
    }
    emit Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
    return true;
  }

}



/**
 * Standard EIP-20 token with an interface marker.
 *
 * @notice Interface marker is used by crowdsale contracts to validate that addresses point a good token contract.
 *
```

```
 */
contract StandardTokenExt is StandardToken {

  /* Interface declaration */
  function isToken() public pure returns (bool weAre) {
    return true;
  }
}



contract BurnableToken is StandardTokenExt {

  // @notice An address for the transfer event where the burned tokens are transferred in a faux Transfer event
  address public constant BURN_ADDRESS = 0;

  /** How many tokens we burned */
  event Burned(address burner, uint burnedAmount);

  /**
   * Burn extra tokens from a balance.
   *
   */
  function burn(uint burnAmount) public {
    address burner = msg.sender;
    balances[burner] = balances[burner].sub(burnAmount);
    totalSupply_ = totalSupply_.sub(burnAmount);
    emit Burned(burner, burnAmount);

    // Inform the blockchain explores that track the
    // balances only by a transfer event that the balance in this
    // address has decreased
    emit Transfer(burner, BURN_ADDRESS, burnAmount);
  }
}

/**
 * Upgrade agent interface inspired by Lunyr.
 *
 * Upgrade agent transfers tokens to a new contract.
 * Upgrade agent itself can be the token contract, or just a middle man contract doing the heavy lifting.
 */
contract UpgradeAgent {
```

```
  uint public originalSupply;

  /** Interface marker */
  function isUpgradeAgent() public pure returns (bool) {
    return true;
  }

  function upgradeFrom(address _from, uint256 _value) public;

}



/**
 * A token upgrade mechanism where users can opt-in amount of tokens to the next smart contract revision.
 *
 * First envisioned by Golem and Lunyr projects.
 */
contract UpgradeableToken is StandardTokenExt {

  /** Contract / person who can set the upgrade path. This can be the same as team multisig wallet, as what it is with its default value. */
  address public upgradeMaster;

  /** The next contract where the tokens will be migrated. */
  UpgradeAgent public upgradeAgent;

  /** How many tokens we have upgraded by now. */
  uint256 public totalUpgraded;

  /**
   * Upgrade states.
   *
   * - NotAllowed: The child contract has not reached a condition where the upgrade can bgun
   * - WaitingForAgent: Token allows upgrade, but we don't have a new agent yet
   * - ReadyToUpgrade: The agent is set, but not a single token has been upgraded yet
   * - Upgrading: Upgrade agent is set and the balance holders can upgrade their tokens
   *
   */
  enum UpgradeState {Unknown, NotAllowed, WaitingForAgent, ReadyToUpgrade, Upgrading}

  /**
```

```
  * Somebody has upgraded some of his tokens.
  */
event Upgrade(address indexed _from, address indexed _to, uint256 _value);

/**
  * New upgrade agent available.
  */
event UpgradeAgentSet(address agent);

/**
  * Do not allow construction without upgrade master set.
  */
constructor(address _upgradeMaster) public {
   upgradeMaster = _upgradeMaster;
}

/**
  * Allow the token holder to upgrade some of their tokens to a new contract.
  */
function upgrade(uint256 value) public {

      UpgradeState state = getUpgradeState();
      if(!(state == UpgradeState.ReadyToUpgrade || state == UpgradeState.Upgrading)) {
        // Called in a bad state
        revert();
      }

      // Validate input value.
      if (value == 0) revert();

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

      // Take tokens out from circulation
      totalSupply_ = totalSupply_.sub(value);
      totalUpgraded = totalUpgraded.add(value);

      // Upgrade agent reissues the tokens
      upgradeAgent.upgradeFrom(msg.sender, value);
      emit Upgrade(msg.sender, upgradeAgent, value);
}

/**
```

```
  * Set an upgrade agent that handles
  */
function setUpgradeAgent(address agent) external {

    if(!canUpgrade()) {
        // The token is not yet in a state that we could think upgrading
        revert();
    }

    if (agent == 0x0) revert();
    // Only a master can designate the next agent
    if (msg.sender != upgradeMaster) revert();
    // Upgrade has already begun for an agent
    if (getUpgradeState() == UpgradeState.Upgrading) revert();

    upgradeAgent = UpgradeAgent(agent);

    // Bad interface
    if(!upgradeAgent.isUpgradeAgent()) revert();
    // Make sure that token supplies match in source and target
    if (upgradeAgent.originalSupply() != totalSupply_) revert();

    emit UpgradeAgentSet(upgradeAgent);
}

/**
  * Get the state of the token upgrade.
  */
function getUpgradeState() public view returns(UpgradeState) {
  if(!canUpgrade()) return UpgradeState.NotAllowed;
  else if(address(upgradeAgent) == 0x00) return UpgradeState.WaitingForAgent;
  else if(totalUpgraded == 0) return UpgradeState.ReadyToUpgrade;
  else return UpgradeState.Upgrading;
}

/**
  * Change the upgrade master.
  *
  * This allows us to set a new owner for the upgrade mechanism.
  */
function setUpgradeMaster(address master) public {
    if (master == 0x0) revert();
```

```
        if (msg.sender != upgradeMaster) revert();
        upgradeMaster = master;
    }

    /**
     * Child contract can enable to provide the condition when the upgrade can begun.
     */
    function canUpgrade() public view returns(bool);

}

/**
 * @title Ownable
 * @dev The Ownable contract has an owner address, and provides basic authorization control
 * functions, this simplifies the implementation of "user permissions".
 */
contract Ownable {
    address public owner;


    /**
     * @dev The Ownable constructor sets the original `owner` of the contract to the sender
     * account.
     */
    constructor() public {
        owner = msg.sender;
    }


    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(msg.sender == owner);
        _;
    }


    /**
     * @dev Allows the current owner to transfer control of the contract to a newOwner.
     * @param newOwner The address to transfer ownership to.
     */
```

```
  function transferOwnership(address newOwner) public onlyOwner {
    require(newOwner != address(0)); //SlowMist// This check is quite good in avoiding losing control of the contract caused by user mistakes
    owner = newOwner;
  }

}



/**
 * Define interface for releasing the token transfer after a successful crowdsale.
 */
contract ReleasableToken is ERC20, Ownable {

  /* The finalizer contract that allows unlift the transfer limits on this token */
  address public releaseAgent;

  /** A crowdsale contract can release us to the wild if ICO success. If false we are are in transfer lock up period.*/
  bool public released = false;

  /** Map of agents that are allowed to transfer tokens regardless of the lock down period. These are crowdsale contracts and possible the team multisig itself. */
  mapping (address => bool) public transferAgents;

  /** Map of addresses that are locked to transfer tokens   */
  mapping (address => bool) public lockAddresses;

  /**
   * Limit token transfer until the crowdsale is over.
   *
   */
  modifier canTransfer(address _sender) {
    if(lockAddresses[_sender]) {
      revert();
    }
    if(!released) {
        if(!transferAgents[_sender]) {
            revert();
```

```
        }
    }

    _;
}

/**
  * Set the contract that can call release and make the token transferable.
  * Design choice. Allow reset the release agent to fix fat finger mistakes.
  * UPDATE (Apr.2019): Now owner can permit some account as transfer agent any time.
  */
function setReleaseAgent(address addr) onlyOwner /* inReleaseState(false) */ public {

    // We don't do interface check here as we might want to a normal wallet address to act as a release agent
    releaseAgent = addr;
}

/**
  * Owner can allow a particular address (a crowdsale contract) to transfer tokens despite the lock up period.
  * UPDATE (Apr.2019): Now owner can permit some account as transfer agent any time.
  */
function setTransferAgent(address addr, bool state) onlyOwner /* inReleaseState(false) */ public {
    transferAgents[addr] = state;
}

/**
  * Owner can lock a particular address (a crowdsale contract)
  * UPDATE (Apr.2019): Now owner can lock specific account in some cases.
  */
```

//SlowMist// Owner can lock any address

```
function setLockAddress(address addr, bool state) onlyOwner /*inReleaseState(false)*/ public {
    lockAddresses[addr] = state;
}

/**
  * One way function to release the tokens to the wild.
  *
  * Can be called only from the release agent that is the final ICO contract. It is only called if the crowdsale has been success
(first milestone reached).
  */
function releaseTokenTransfer() public onlyReleaseAgent {
```

```
    released = true;
}

/**
 * UPDATE (Apr.2019): Now owner can lock whole transfer in some cases (i.e security accidents)
 */
```

//SlowMist// Suspending all transactions upon major abnormalities is a recommended approach.

```
function lockTokenTransfer() public onlyOwner {
    released = false;
}

/** The function can be called only before or after the tokens have been releasesd
 * UPDATE (Apr.201): Obsolete code, not used any more.
 */
    // modifier inReleaseState(bool releaseState) {
    //     if(releaseState != released) {
    //         revert();
    //     }
    //     _;
    // }

/** The function can be called only by a whitelisted release agent. */
modifier onlyReleaseAgent() {
    if(msg.sender != releaseAgent) {
        revert();
    }
    _;
}

function transfer(address _to, uint _value) public canTransfer(msg.sender) returns (bool success) {
    // Call StandardToken.transfer()
  return super.transfer(_to, _value);
}

function transferFrom(address _from, address _to, uint _value) public canTransfer(_from) returns (bool success) {
    // Call StandardToken.transferForm()
    return super.transferFrom(_from, _to, _value);
}
```

```
}

/**
 * Safe unsigned safe math.
 *
 * https://blog.aragon.one/library-driven-development-in-solidity-2bebcaf88736#.750gwtwli
 *
 * Originally from https://raw.githubusercontent.com/AragonOne/zeppelin-solidity/master/contracts/SafeMathLib.sol
 *
 * Maintained here until merged to mainline zeppelin-solidity.
 *
 */
library SafeMathLib {

  function times(uint a, uint b) internal pure returns (uint) {
    uint c = a * b;

    assert(a == 0 || c / a == b); //SlowMist// It is recommended to replace "assert" with "require" to optimize Gas

    return c;
  }

  function minus(uint a, uint b) internal pure returns (uint) {

    assert(b <= a); //SlowMist// It is recommended to replace "assert" with "require" to optimize Gas

    return a - b;
  }

  function plus(uint a, uint b)   internal pure returns (uint) {
    uint c = a + b;

    assert(c>=a); //SlowMist// It is recommended to replace "assert" with "require" to optimize Gas

    return c;
  }

}
```

```
/**
 * A token that can increase its supply by another contract.
 *
 * This allows uncapped crowdsale by dynamically increasing the supply when money pours in.
 * Only mint agents, contracts whitelisted by owner, can mint new tokens.
 *
 */
contract MintableToken is StandardTokenExt, Ownable {

  using SafeMathLib for uint;

  bool public mintingFinished = false;

  /** List of agents that are allowed to create new tokens */
  mapping (address => bool) public mintAgents;

  event MintingAgentChanged(address addr, bool state);
  event Minted(address receiver, uint amount);

  /**
   * Create new tokens and allocate them to an address..
   *
   * Only callably by a crowdsale contract (mint agent).
   */
  function mint(address receiver, uint amount) onlyMintAgent canMint public {
    totalSupply_ = totalSupply_.plus(amount);
    balances[receiver] = balances[receiver].plus(amount);

    // This will make the mint transaction apper in EtherScan.io
    // We can remove this after there is a standardized minting event
    emit Transfer(0, receiver, amount);
  }

  /**
   * Owner can allow a crowdsale contract to mint new tokens.
   */
  function setMintAgent(address addr, bool state) onlyOwner canMint public {
    mintAgents[addr] = state;
    emit MintingAgentChanged(addr, state);
  }
```

```
  modifier onlyMintAgent() {
    // Only crowdsale contracts are allowed to mint new tokens
    if(!mintAgents[msg.sender]) {
          revert();
    }
    _;
  }

  /** Make sure we are not done yet. */
  modifier canMint() {
    if(mintingFinished) revert();
    _;
  }
}


/**
 * A crowdsaled token.
 *
 * An ERC-20 token designed specifically for crowdsales with investor protection and further development path.
 *
 * - The token transfer() is disabled until the crowdsale is over
 * - The token contract gives an opt-in upgrade path to a new contract
 * - The same token can be part of several crowdsales through approve() mechanism
 * - The token can be capped (supply set in the constructor) or uncapped (crowdsale contract can mint new tokens)
 *
 */
contract CrowdsaleToken is ReleasableToken, MintableToken, UpgradeableToken {

  /** Name and symbol were updated. */
  event UpdatedTokenInformation(string newName, string newSymbol);

  string public name;

  string public symbol;

  uint public decimals;

  /**
   * Construct the token.
   *
```

```
  * This token must be created through a team multisig wallet, so that it is owned by that wallet.
  *
  * @param _name Token name
  * @param _symbol Token symbol - should be all caps
  * @param _initialSupply How many tokens we start with
  * @param _decimals Number of decimal places
  * @param _mintable Are new tokens created over the crowdsale or do we distribute only the initial supply? Note that when the token becomes transferable the minting always ends.
  */
 constructor(string _name, string _symbol, uint _initialSupply, uint _decimals, bool _mintable) public
   UpgradeableToken(msg.sender) {

   // Create any address, can be transferred
   // to team multisig via changeOwner(),
   // also remember to call setUpgradeMaster()
   owner = msg.sender;

   name = _name;
   symbol = _symbol;

   totalSupply_ = _initialSupply;

   decimals = _decimals;

   // Create initially all balance on the team multisig
   balances[owner] = totalSupply_;

   if(totalSupply_ > 0) {
     emit Minted(owner, totalSupply_);
   }

   // No more new supply allowed after the token creation
   if(!_mintable) {
     mintingFinished = true;
     if(totalSupply_ == 0) {
       revert(); // Cannot create a token without supply and no minting
     }
   }
 }

 /**
  * When token is released to be transferable, enforce no new tokens can be created.
```

```
   */
  function releaseTokenTransfer() public onlyReleaseAgent {
    mintingFinished = true;
    super.releaseTokenTransfer();
  }

  /**
   * Allow upgrade agent functionality kick in only if the crowdsale was success.
   */
  function canUpgrade() public view returns(bool) {
    return released;
  }

  /**
   * Owner can update token information here.
   *
   * It is often useful to conceal the actual token association, until
   * the token operations, like central issuance or reissuance have been completed.
   *
   * This function allows the token owner to rename the token after the operations
   * have been completed and then point the audience to use the token contract.
   */
  function setTokenInformation(string _name, string _symbol) public onlyOwner {
    name = _name;
    symbol = _symbol;

    emit UpdatedTokenInformation(name, symbol);
  }

}


/**
 * A crowdsaled token that you can also burn.
 *
 */
contract BurnableCrowdsaleToken is BurnableToken, CrowdsaleToken {

  constructor(string _name, string _symbol, uint _initialSupply, uint _decimals, bool _mintable) public
    CrowdsaleToken(_name, _symbol, _initialSupply, _decimals, _mintable) {

  }
```

```
}



/**
 * The AML Token
 *
 * This subset of BurnableCrowdsaleToken gives the Owner a possibility to
 * reclaim tokens from a participant before the token is released
 * after a participant has failed a prolonged AML process.
 *
 * It is assumed that the anti-money laundering process depends on blockchain data.
 * The data is not available before the transaction and not for the smart contract.
 * Thus, we need to implement logic to handle AML failure cases post payment.
 * We give a time window before the token release for the token sale owners to
 * complete the AML and claw back all token transactions that were
 * caused by rejected purchases.
 */
contract AMLToken is BurnableCrowdsaleToken {

  // An event when the owner has reclaimed non-released tokens
  event OwnerReclaim(address fromWhom, uint amount);

  constructor(string _name, string _symbol, uint _initialSupply, uint _decimals, bool _mintable) public
BurnableCrowdsaleToken(_name, _symbol, _initialSupply, _decimals, _mintable) {

  }
  /// @dev Here the owner can reclaim the tokens from a participant if
  ///         the token is not released yet. Refund will be handled offband.
  /// @param fromWhom address of the participant whose tokens we want to claim

  //SlowMist// Owner can transfer balance from any address

  function transferToOwner(address fromWhom) public onlyOwner {
    // UPDATE (Apr.2019): Onwer can reclaim any time in some cases (i.e. security accidents)
    // if (released) revert();
    uint amount = balanceOf(fromWhom);
    balances[fromWhom] = balances[fromWhom].sub(amount);
    balances[owner] = balances[owner].add(amount);
    emit Transfer(fromWhom, owner, amount);
    emit OwnerReclaim(fromWhom, amount);
  }
}
```

慢雾科技
slow mist

Official Website
www.slowmist.com

E-mail
team@slowmist.com

Twitter
@SlowMist_Team

WeChat Official Account