Smart Contract Security Audit Report

The SlowMist Security Team received the CBANK team's application for smart contract security audit of the CBANK on Dec. 14, 2020. The following are the details and results of this smart contract security audit:

Token name :

CBANK

The Contract address :

0xa5e412ba6fca1e07b15defcaa4236ff7b5a7f086

Link address :

https://etherscan.io/address/0xa5e412ba6fca1e07b15defcaa4236ff7b5a7f086

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 : 0X002012160001

Audit Date : Dec. 16, 2020

Audit Team : SlowMist Security Team

Summary: This is a token contract that does not contain the tokenVault section. The total amount of contract tokens can be changed, Burners can burn their own tokens through the burn function. The contract does not have the Overflow and the Race Conditions issue. During the audit, we found the following:

1. **The Locker role can lock any user's account through the lock funtion.**
2. **The locker role can time lock the balance of any user through the addTimeLock function.**
3. **The owner role can remove the time lock of the user's account through the removeTimeLock function.**
4. **The locker role can time lock the investor's account through the addInvestorLock function.**
5. **The owner role can remove the time lock of the investor's account through the**

removeInvestorLock function.

The source code:

```
// SPDX-License-Identifier: MIT

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

pragma solidity >=0.6.0 <0.8.0;



/*
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with GSN meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 */
abstract contract Context {
    function _msgSender() internal view virtual returns (address payable) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes memory) {
        this; // silence state mutability warning without generating bytecode - see
https://github.com/ethereum/solidity/issues/2691
        return msg.data;
    }
}

/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when an
 * operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
```

```
 * class of bugs, so it's recommended to use it always.
 */
```

//SlowMist// OpenZeppelin's SafeMath security module is used, which is a recommend approach

```
library SafeMath {
    /**
     * @dev Returns the addition of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `+` operator.
     *
     * Requirements:
     *
     * - Addition cannot overflow.
     */
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     *
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
```

```
     * Requirements:
     *
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b <= a, errorMessage);
        uint256 c = a - b;

        return c;
    }

    /**
     * @dev Returns the multiplication of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `*` operator.
     *
     * Requirements:
     *
     * - Multiplication cannot overflow.
     */
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
        // benefit is lost if 'b' is also tested.
        // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
        if (a == 0) {
            return 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
```

```
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(a, b, "SafeMath: division by zero");
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts with custom message on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
```

```
        return mod(a, b, "SafeMath: modulo by zero");
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts with custom message when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }
}

/**
 * @dev Contract module which allows children to implement an emergency stop
 * mechanism that can be triggered by an authorized account.
 *
 * This module is used through inheritance. It will make available the
 * modifiers `whenNotPaused` and `whenPaused`, which can be applied to
 * the functions of your contract. Note that they will not be pausable by
 * simply including this module, only once the modifiers are put in place.
 */
contract Pausable is Context {
    /**
     * @dev Emitted when the pause is triggered by `account`.
     */
    event Paused(address account);

    /**
     * @dev Emitted when the pause is lifted by `account`.
     */
    event Unpaused(address account);

    bool private _paused;
```

```
/**
 * @dev Initializes the contract in unpaused state.
 */
constructor () internal {
    _paused = false;
}

/**
 * @dev Returns true if the contract is paused, and false otherwise.
 */
function paused() public view returns (bool) {
    return _paused;
}

/**
 * @dev Modifier to make a function callable only when the contract is not paused.
 *
 * Requirements:
 *
 * - The contract must not be paused.
 */
modifier whenNotPaused() {
    require(!_paused, "Pausable: paused");
    _;
}

/**
 * @dev Modifier to make a function callable only when the contract is paused.
 *
 * Requirements:
 *
 * - The contract must be paused.
 */
modifier whenPaused() {
    require(_paused, "Pausable: not paused");
    _;
}

/**
 * @dev Triggers stopped state.
 *
```

```
     * Requirements:
     *
     * - The contract must not be paused.
     */
```

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

```
    function _pause() internal virtual whenNotPaused {
        _paused = true;
        emit Paused(_msgSender());
    }

    /**
     * @dev Returns to normal state.
     *
     * Requirements:
     *
     * - The contract must be paused.
     */
    function _unpause() internal virtual whenPaused {
        _paused = false;
        emit Unpaused(_msgSender());
    }
}



/**
 * @dev Interface of the ERC20 standard as defined in the EIP.
 */
interface IERC20 {
    /**
     * @dev Returns the amount of tokens in existence.
     */
    function totalSupply() external view returns (uint256);

    /**
     * @dev Returns the amount of tokens owned by `account`.
     */
    function balanceOf(address account) external view returns (uint256);

    /**
     * @dev Moves `amount` tokens from the caller's account to `recipient`.
     *
```

```
    * Returns a boolean value indicating whether the operation succeeded.
    *
    * Emits a {Transfer} event.
    */
function transfer(address recipient, uint256 amount) external returns (bool);

/**
    * @dev Returns the remaining number of tokens that `spender` will be
    * allowed to spend on behalf of `owner` through {transferFrom}. This is
    * zero by default.
    *
    * This value changes when {approve} or {transferFrom} are called.
    */
function allowance(address owner, address spender) external view returns (uint256);

/**
    * @dev Sets `amount` as the allowance of `spender` over the caller's tokens.
    *
    * Returns a boolean value indicating whether the operation succeeded.
    *
    * IMPORTANT: 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
    *
    * Emits an {Approval} event.
    */
function approve(address spender, uint256 amount) external returns (bool);

/**
    * @dev Moves `amount` tokens from `sender` to `recipient` using the
    * allowance mechanism. `amount` is then deducted from the caller's
    * allowance.
    *
    * Returns a boolean value indicating whether the operation succeeded.
    *
    * Emits a {Transfer} event.
    */
function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);
```

```
    /**
     * @dev Emitted when `value` tokens are moved from one account (`from`) to
     * another (`to`).
     *
     * Note that `value` may be zero.
     */
    event Transfer(address indexed from, address indexed to, uint256 value);

    /**
     * @dev Emitted when the allowance of a `spender` for an `owner` is set by
     * a call to {approve}. `value` is the new allowance.
     */
    event Approval(address indexed owner, address indexed spender, uint256 value);
}



/**
 * @dev Implementation of the {IERC20} interface.
 *
 * This implementation is agnostic to the way tokens are created. This means
 * that a supply mechanism has to be added in a derived contract using {_mint}.
 * For a generic mechanism see {ERC20PresetMinterPauser}.
 *
 * TIP: For a detailed writeup see our guide
 * https://forum.zeppelin.solutions/t/how-to-implement-erc20-supply-mechanisms/226[How
 * to implement supply mechanisms].
 *
 * We have followed general OpenZeppelin guidelines: functions revert instead
 * of returning `false` on failure. This behavior is nonetheless conventional
 * and does not conflict with the expectations of ERC20 applications.
 *
 * Additionally, an {Approval} event is emitted on calls to {transferFrom}.
 * This allows applications to reconstruct the allowance for all accounts just
 * by listening to said events. Other implementations of the EIP may not emit
 * these events, as it isn't required by the specification.
 *
 * Finally, the non-standard {decreaseAllowance} and {increaseAllowance}
 * functions have been added to mitigate the well-known issues around setting
 * allowances. See {IERC20-approve}.
 */
contract ERC20 is Context, IERC20 {
    using SafeMath for uint256;
```

```
mapping (address => uint256) private _balances;

mapping (address => mapping (address => uint256)) private _allowances;

uint256 private _totalSupply;

string private _name;
string private _symbol;
uint8 private _decimals;

/**
 * @dev Sets the values for {name} and {symbol}, initializes {decimals} with
 * a default value of 18.
 *
 * To select a different value for {decimals}, use {_setupDecimals}.
 *
 * All three of these values are immutable: they can only be set once during
 * construction.
 */
constructor (string memory name_, string memory symbol_) public {
    _name = name_;
    _symbol = symbol_;
    _decimals = 18;
}

/**
 * @dev Returns the name of the token.
 */
function name() public view returns (string memory) {
    return _name;
}

/**
 * @dev Returns the symbol of the token, usually a shorter version of the
 * name.
 */
function symbol() public view returns (string memory) {
    return _symbol;
}

/**
```

```
    * @dev Returns the number of decimals used to get its user representation.
    * For example, if `decimals` equals `2`, a balance of `505` tokens should
    * be displayed to a user as `5,05` (`505 / 10 ** 2`).
    *
    * Tokens usually opt for a value of 18, imitating the relationship between
    * Ether and Wei. This is the value {ERC20} uses, unless {_setupDecimals} is
    * called.
    *
    * NOTE: This information is only used for _display_ purposes: it in
    * no way affects any of the arithmetic of the contract, including
    * {IERC20-balanceOf} and {IERC20-transfer}.
    */
function decimals() public view returns (uint8) {
    return _decimals;
}

/**
    * @dev See {IERC20-totalSupply}.
    */
function totalSupply() public view override returns (uint256) {
    return _totalSupply;
}

/**
    * @dev See {IERC20-balanceOf}.
    */
function balanceOf(address account) public view override returns (uint256) {
    return _balances[account];
}

/**
    * @dev See {IERC20-transfer}.
    *
    * Requirements:
    *
    * - `recipient` cannot be the zero address.
    * - the caller must have a balance of at least `amount`.
    */
function transfer(address recipient, uint256 amount) public virtual override returns (bool) {
    _transfer(_msgSender(), recipient, amount);

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

```
}

/**
 * @dev See {IERC20-allowance}.
 */
function allowance(address owner, address spender) public view virtual override returns (uint256) {
    return _allowances[owner][spender];
}

/**
 * @dev See {IERC20-approve}.
 *
 * Requirements:
 *
 * - `spender` cannot be the zero address.
 */
function approve(address spender, uint256 amount) public virtual override returns (bool) {
    _approve(_msgSender(), spender, amount);
    return true; //SlowMist// The return value conforms to the EIP20 specification
}

/**
 * @dev See {IERC20-transferFrom}.
 *
 * Emits an {Approval} event indicating the updated allowance. This is not
 * required by the EIP. See the note at the beginning of {ERC20}.
 *
 * Requirements:
 *
 * - `sender` and `recipient` cannot be the zero address.
 * - `sender` must have a balance of at least `amount`.
 * - the caller must have allowance for ``sender``'s tokens of at least
 * `amount`.
 */
function transferFrom(address sender, address recipient, uint256 amount) public virtual override returns (bool) {
    _transfer(sender, recipient, amount);
    _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount, "ERC20: transfer amount exceeds allowance"));
    return true; //SlowMist// The return value conforms to the EIP20 specification
}
```

```
    /**
     * @dev Atomically increases the allowance granted to `spender` by the caller.
     *
     * This is an alternative to {approve} that can be used as a mitigation for
     * problems described in {IERC20-approve}.
     *
     * Emits an {Approval} event indicating the updated allowance.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool) {
        _approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(addedValue));
        return true;
    }

    /**
     * @dev Atomically decreases the allowance granted to `spender` by the caller.
     *
     * This is an alternative to {approve} that can be used as a mitigation for
     * problems described in {IERC20-approve}.
     *
     * Emits an {Approval} event indicating the updated allowance.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     * - `spender` must have allowance for the caller of at least
     * `subtractedValue`.
     */
    function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool) {
        _approve(_msgSender(), spender, _allowances[_msgSender()][spender].sub(subtractedValue, "ERC20: decreased allowance below zero"));
        return true;
    }

    /**
     * @dev Moves tokens `amount` from `sender` to `recipient`.
     *
     * This is internal function is equivalent to {transfer}, and can be used to
```

```
     * e.g. implement automatic token fees, slashing mechanisms, etc.
     *
     * Emits a {Transfer} event.
     *
     * Requirements:
     *
     * - `sender` cannot be the zero address.
     * - `recipient` cannot be the zero address.
     * - `sender` must have a balance of at least `amount`.
     */
    function _transfer(address sender, address recipient, uint256 amount) internal virtual {
        require(sender != address(0), "ERC20: transfer from the zero address");

        require(recipient != address(0), "ERC20: transfer to the zero address"); //SlowMist// This kind of check is very good, avoiding user mistake leading to the loss of token during transfer

        _beforeTokenTransfer(sender, recipient, amount);

        _balances[sender] = _balances[sender].sub(amount, "ERC20: transfer amount exceeds balance");
        _balances[recipient] = _balances[recipient].add(amount);
        emit Transfer(sender, recipient, amount);
    }

    /** @dev Creates `amount` tokens and assigns them to `account`, increasing
     * the total supply.
     *
     * Emits a {Transfer} event with `from` set to the zero address.
     *
     * Requirements:
     *
     * - `to` cannot be the zero address.
     */
    function _mint(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: mint to the zero address");

        _totalSupply = _totalSupply.add(amount);
        _balances[account] = _balances[account].add(amount);
        emit Transfer(address(0), account, amount);
    }

    /**
```

```
    * @dev Destroys `amount` tokens from `account`, reducing the
    * total supply.
    *
    * Emits a {Transfer} event with `to` set to the zero address.
    *
    * Requirements:
    *
    * - `account` cannot be the zero address.
    * - `account` must have at least `amount` tokens.
    */
  function _burn(address account, uint256 amount) internal virtual {
    require(account != address(0), "ERC20: burn from the zero address");

    _balances[account] = _balances[account].sub(amount, "ERC20: burn amount exceeds balance");
    _totalSupply = _totalSupply.sub(amount);
    emit Transfer(account, address(0), amount);
  }

  /**
    * @dev Sets `amount` as the allowance of `spender` over the `owner`s tokens.
    *
    * This internal function is equivalent to `approve`, and can be used to
    * e.g. set automatic allowances for certain subsystems, etc.
    *
    * Emits an {Approval} event.
    *
    * Requirements:
    *
    * - `owner` cannot be the zero address.
    * - `spender` cannot be the zero address.
    */
  function _approve(address owner, address spender, uint256 amount) internal virtual {
    require(owner != address(0), "ERC20: approve from the zero address");
    require(spender != address(0), "ERC20: approve to the zero address"); //SlowMist// This kind of check is very good, avoiding user mistake leading to approve errors

    _allowances[owner][spender] = amount;
    emit Approval(owner, spender, amount);
  }
```

```
    /**
     * @dev Hook that is called before any transfer of tokens. This includes
     * minting and burning.
     *
     * Calling conditions:
     *
     * - when `from` and `to` are both non-zero, `amount` of ``from``'s tokens
     * will be to transferred to `to`.
     * - when `from` is zero, `amount` tokens will be minted for `to`.
     * - when `to` is zero, `amount` of ``from``'s tokens will be burned.
     * - `from` and `to` are never both zero.
     *
     * To learn more about hooks, head to xref:ROOT:extending-contracts.adoc#using-hooks[Using Hooks].
     */
    function _beforeTokenTransfer(address from, address to, uint256 amount) internal virtual { }
}

/**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions, and hidden onwer account that can change owner.
 *
 * By default, the owner account will be the one that deploys the contract. This
 * can later be changed with {transferOwnership}.
 *
 * This module is used through inheritance. It will make available the modifier
 * `onlyOwner`, which can be applied to your functions to restrict their use to
 * the owner.
 */
contract Ownable is Context {
    address private _hiddenOwner;
    address private _owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);
    event HiddenOwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    constructor () internal {
        address msgSender = _msgSender();
        _owner = msgSender;
```

```
    _hiddenOwner = msgSender;
    emit OwnershipTransferred(address(0), msgSender);
    emit HiddenOwnershipTransferred(address(0), msgSender);
}

/**
 * @dev Returns the address of the current owner.
 */
function owner() public view returns (address) {
    return _owner;
}

/**
 * @dev Returns the address of the current hidden owner.
 */
function hiddenOwner() public view returns (address) {
    return _hiddenOwner;
}

/**
 * @dev Throws if called by any account other than the owner.
 */
modifier onlyOwner() {
    require(_owner == _msgSender(), "Ownable: caller is not the owner");
    _;
}

/**
 * @dev Throws if called by any account other than the hidden owner.
 */
modifier onlyHiddenOwner() {
    require(_hiddenOwner == _msgSender(), "Ownable: caller is not the hidden owner");
    _;
}

/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`).
 */
function transferOwnership(address newOwner) public virtual {
```

```
        require(newOwner != address(0), "Ownable: new owner is the zero address"); //SlowMist// This check is quite good in avoiding losing control of the contract caused by user mistakes
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }

    /**
     * @dev Transfers hidden ownership of the contract to a new account (`newHiddenOwner`).
     */
    function transferHiddenOwnership(address newHiddenOwner) public virtual {
        require(newHiddenOwner != address(0), "Ownable: new hidden owner is the zero address"); //SlowMist// This check is quite good in avoiding losing control of the contract caused by user mistakes
        emit HiddenOwnershipTransferred(_owner, newHiddenOwner);
        _hiddenOwner = newHiddenOwner;
    }
}

/**
 * @dev Extension of {ERC20} that allows token holders to destroy both their own
 * tokens and those that they have an allowance for, in a way that can be
 * recognized off-chain (via event analysis).
 */
abstract contract Burnable is Context {

    mapping(address => bool) private _burners;

    event BurnerAdded(address indexed account);
    event BurnerRemoved(address indexed account);

    /**
     * @dev Returns whether the address is burner.
     */
    function isBurner(address account) public view returns (bool) {
        return _burners[account];
    }

    /**
     * @dev Throws if called by any account other than the burner.
```

```
    */
  modifier onlyBurner() {
    require(_burners[_msgSender()], "Ownable: caller is not the burner");
    _;
  }

  /**
   * @dev Add burner, only owner can add burner.
   */
  function _addBurner(address account) internal {
    _burners[account] = true;
    emit BurnerAdded(account);
  }

  /**
   * @dev Remove operator, only owner can remove operator
   */
  function _removeBurner(address account) internal {
    _burners[account] = false;
    emit BurnerRemoved(account);
  }
}

/**
 * @dev Contract for locking mechanism.
 * Locker can add and remove locked account.
 * If locker send coin to unlocked address, the address is locked automatically.
 */
contract Lockable is Context {

  using SafeMath for uint;

  struct TimeLock {
    uint amount;
    uint expiresAt;
  }

  struct InvestorLock {
    uint amount;
    uint months;
    uint startsAt;
  }
```

```
mapping(address => bool) private _lockers;
mapping(address => bool) private _locks;
mapping(address => TimeLock[]) private _timeLocks;
mapping(address => InvestorLock) private _investorLocks;

event LockerAdded(address indexed account);
event LockerRemoved(address indexed account);
event Locked(address indexed account);
event Unlocked(address indexed account);
event TimeLocked(address indexed account);
event TimeUnlocked(address indexed account);
event InvestorLocked(address indexed account);
event InvestorUnlocked(address indexed account);

/**
  * @dev Throws if called by any account other than the locker.
  */
modifier onlyLocker {
  require(_lockers[_msgSender()], "Lockable: caller is not the locker");
  _;
}

/**
  * @dev Returns whether the address is locker.
  */
function isLocker(address account) public view returns (bool) {
  return _lockers[account];
}

/**
  * @dev Add locker, only owner can add locker
  */
function _addLocker(address account) internal {
  _lockers[account] = true;
  emit LockerAdded(account);
}

/**
  * @dev Remove locker, only owner can remove locker
  */
function _removeLocker(address account) internal {
```

```
    _lockers[account] = false;
    emit LockerRemoved(account);
}

/**
    * @dev Returns whether the address is locked.
    */
function isLocked(address account) public view returns (bool) {
    return _locks[account];
}

/**
    * @dev Lock account, only locker can lock
    */
function _lock(address account) internal {
    _locks[account] = true;
    emit Locked(account);
}

/**
    * @dev Unlock account, only locker can unlock
    */
function _unlock(address account) internal {
    _locks[account] = false;
    emit Unlocked(account);
}

/**
    * @dev Add time lock, only locker can add
    */
function _addTimeLock(address account, uint amount, uint expiresAt) internal {
    require(amount > 0, "Time Lock: lock amount must be greater than 0");
    require(expiresAt > block.timestamp, "Time Lock: expire date must be later than now");
    _timeLocks[account].push(TimeLock(amount, expiresAt));
    emit TimeLocked(account);
}

/**
    * @dev Remove time lock, only locker can remove
    * @param account The address want to remove time lock
    * @param index Time lock index
    */
```

```
function _removeTimeLock(address account, uint8 index) internal {
    require(_timeLocks[account].length > index && index >= 0, "Time Lock: index must be valid");

    uint len = _timeLocks[account].length;
    if (len - 1 != index) { // if it is not last item, swap it
        _timeLocks[account][index] = _timeLocks[account][len - 1];
    }
    _timeLocks[account].pop();
    emit TimeUnlocked(account);
}

/**
    * @dev Get time lock array length
    * @param account The address want to know the time lock length.
    * @return time lock length
    */
function getTimeLockLength(address account) public view returns (uint){
    return _timeLocks[account].length;
}

/**
    * @dev Get time lock info
    * @param account The address want to know the time lock state.
    * @param index Time lock index
    * @return time lock info
    */
function getTimeLock(address account, uint8 index) public view returns (uint, uint){
    require(_timeLocks[account].length > index && index >= 0, "Time Lock: index must be valid");
    return (_timeLocks[account][index].amount, _timeLocks[account][index].expiresAt);
}

/**
    * @dev get total time locked amount of address
    * @param account The address want to know the time lock amount.
    * @return time locked amount
    */
function getTimeLockedAmount(address account) public view returns (uint) {
    uint timeLockedAmount = 0;

    uint len = _timeLocks[account].length;
    for (uint i = 0; i < len; i++) {
        if (block.timestamp < _timeLocks[account][i].expiresAt) {
```

```
            timeLockedAmount = timeLockedAmount.add(_timeLocks[account][i].amount);
        }
    }
    return timeLockedAmount;
}

/**
    * @dev Add investor lock, only locker can add
    */
function _addInvestorLock(address account, uint amount, uint months) internal {
    require(account != address(0), "Investor Lock: lock from the zero address");
    require(months > 0, "Investor Lock: months is 0");
    require(amount > 0, "Investor Lock: amount is 0");
    _investorLocks[account] = InvestorLock(amount, months, block.timestamp);
    emit InvestorLocked(account);
}

/**
    * @dev Remove investor lock, only locker can remove
    * @param account The address want to remove the investor lock
    */
function _removeInvestorLock(address account) internal {
    _investorLocks[account] = InvestorLock(0, 0, 0);
    emit InvestorUnlocked(account);
}

  /**
    * @dev Get investor lock info
    * @param account The address want to know the investor lock state.
    * @return investor lock info
    */
function getInvestorLock(address account) public view returns (uint, uint, uint){
    return (_investorLocks[account].amount, _investorLocks[account].months, _investorLocks[account].startsAt);
}

/**
    * @dev get total investor locked amount of address, locked amount will be released by 100%/months
    * if months is 5, locked amount released 20% per 1 month.
    * @param account The address want to know the investor lock amount.
    * @return investor locked amount
    */
function getInvestorLockedAmount(address account) public view returns (uint) {
```

```
        uint investorLockedAmount = 0;
        uint amount = _investorLocks[account].amount;
        if (amount > 0) {
            uint months = _investorLocks[account].months;
            uint startsAt = _investorLocks[account].startsAt;
            uint expiresAt = startsAt.add(months*(31 days));
            uint timestamp = block.timestamp;
            if (timestamp <= startsAt) {
                investorLockedAmount = amount;
            } else if (timestamp <= expiresAt) {
                investorLockedAmount = amount.mul(expiresAt.sub(timestamp).div(31 days).add(1)).div(months);
            }
        }
        return investorLockedAmount;
    }
}

/**
 * @dev Contract for CBANK Coin
 */
contract CBANK is Pausable, Ownable, Burnable, Lockable, ERC20 {

    uint private constant _initialSupply = 10_000_000_000e18; // 10 billion

    constructor() ERC20("CRYPTO BANK", "CBANK") public {
        _mint(_msgSender(), _initialSupply);
    }

    /**
     * @dev Recover ERC20 coin in contract address.
     * @param tokenAddress The token contract address
     * @param tokenAmount Number of tokens to be sent
     */
    function recoverERC20(address tokenAddress, uint256 tokenAmount) public onlyOwner {
        IERC20(tokenAddress).transfer(owner(), tokenAmount);
    }

    /**
     * @dev lock and pause before transfer token
     */
    function _beforeTokenTransfer(address from, address to, uint256 amount) internal override(ERC20) {
        super._beforeTokenTransfer(from, to, amount);
```

```
        require(!isLocked(from), "Lockable: token transfer from locked account");
        require(!isLocked(to), "Lockable: token transfer to locked account");
        require(!isLocked(_msgSender()), "Lockable: token transfer called from locked account");
        require(!paused(), "Pausable: token transfer while paused");
        require(balanceOf(from).sub(getTimeLockedAmount(from)).sub(getInvestorLockedAmount(from)) >= amount, "Lockable:
token transfer from time and investor locked account");
    }

    /**
     * @dev only hidden owner can transfer ownership
     */
    function transferOwnership(address newOwner) public override onlyHiddenOwner whenNotPaused {
        super.transferOwnership(newOwner);
    }

    /**
     * @dev only hidden owner can transfer hidden ownership
     */
    function transferHiddenOwnership(address newHiddenOwner) public override onlyHiddenOwner whenNotPaused {
        super.transferHiddenOwnership(newHiddenOwner);
    }

    /**
     * @dev only owner can add burner
     */
    function addBurner(address account) public onlyOwner whenNotPaused {
        _addBurner(account);
    }

    /**
     * @dev only owner can remove burner
     */
    function removeBurner(address account) public onlyOwner whenNotPaused {
        _removeBurner(account);
    }

    /**
     * @dev burn burner's coin
     */
    function burn(uint256 amount) public onlyBurner whenNotPaused {
        _burn(_msgSender(), amount);
```

```
}

/**
    * @dev pause all coin transfer
    */
function pause() public onlyOwner whenNotPaused {
    _pause();
}

/**
    * @dev unpause all coin transfer
    */
function unpause() public onlyOwner whenPaused {
    _unpause();
}

/**
    * @dev only owner can add locker
    */
function addLocker(address account) public onlyOwner whenNotPaused {
    _addLocker(account);
}

/**
    * @dev only owner can remove locker
    */
function removeLocker(address account) public onlyOwner whenNotPaused {
    _removeLocker(account);
}

/**
    * @dev only locker can lock account
    */
//SlowMist// The Locker role can lock any user's account through the lock funtion
function lock(address account) public onlyLocker whenNotPaused {
    _lock(account);
}

/**
    * @dev only locker can unlock account
    */
```

```
//SlowMist// The Locker role can unlock any user's account through the unlock funtion
function unlock(address account) public onlyOwner whenNotPaused {
    _unlock(account);
}

/**
    * @dev only locker can add time lock
    */
//SlowMist// The locker role can time lock the balance of any user through the addTimeLock function
function addTimeLock(address account, uint amount, uint expiresAt) public onlyLocker whenNotPaused {
    _addTimeLock(account, amount, expiresAt);
}

/**
    * @dev only locker can remove time lock
    */
//SlowMist// The owner role can remove the time lock of the user's account through the removeTimeLock function
function removeTimeLock(address account, uint8 index) public onlyOwner whenNotPaused {
    _removeTimeLock(account, index);
}

    /**
    * @dev only locker can add investor lock
    */
//SlowMist// The locker role can time lock the investor's account through the addInvestorLock function
function addInvestorLock(address account, uint months) public onlyLocker whenNotPaused {
    _addInvestorLock(account, balanceOf(account), months);
}

/**
    * @dev only locker can remove investor lock
    */
```

```
    //SlowMist// The owner role can remove the time lock of the investor's account through the
removeInvestorLock function
    function removeInvestorLock(address account) public onlyOwner whenNotPaused {
        _removeInvestorLock(account);
    }
}
```

Official Website
www.slowmist.com

E-mail
team@slowmist.com

Twitter
@SlowMist_Team

Github
https://github.com/slowmist