Smart Contract
Security Audit Report

The SlowMist Security Team received the team's application for smart contract security audit of the KStarCoin(KSC) on 2023.01.19. The following are the details and results of this smart contract security audit:

Token Name :

KStarCoin(KSC)

The contract address :

https://etherscan.io/address/0x7f5488c507aB3b0cF18c9d0b10B1decB723CeF3D

The audit items and results :

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

NO.	Audit Items	Result
1	Replay Vulnerability	Passed
2	Denial of Service Vulnerability	Passed
3	Race Conditions Vulnerability	Passed
4	Authority Control Vulnerability	Passed
5	Integer Overflow and Underflow Vulnerability	Passed
6	Gas Optimization Audit	Passed
7	Design Logic Audit	Passed
8	Uninitialized Storage Pointers Vulnerability	Passed
9	Arithmetic Accuracy Deviation Vulnerability	Passed
10	"False top-up" Vulnerability	Passed
11	Malicious Event Log Audit	Passed
12	Scoping and Declarations Audit	Passed

NO.	Audit Items	Result
13	Safety Design Audit	Passed
14	Non-privacy/Non-dark Coin Audit	Passed

Audit Result : Passed

Audit Number : 0X002301210001

Audit Date : 2023.01.19 - 2023.01.21

Audit Team : SlowMist Security Team

Summary conclusion : This is a token contract that contains the tokenVault section and does not contain the dark coin functions. The total amount of contract tokens remains unchangeable. SafeMath security module is used, which is a recommended approach. The contract does not have the Overflow and the Race Conditions issue.

During the audit, we found the following information:

1. Only the owner role can freeze users' accounts through the freezeAccout function.
2. Only the owner role can unfreeze users' accounts through the unfreezeAccout function.
3. Only the owner role can lock users' accounts through the lock function.
4. Only the owner role can unlock users' accounts through the unlock function.

The source code:

```
/**
 *Submitted for verification at Etherscan.io on 2023-01-02
*/
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.5.0;
//SlowMist// SafeMath security module is used, which is a recommended approach
library SafeMath {
    /**
    * @dev Multiplies two unsigned integers, reverts on 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-solidity/pull/522
        if (a == 0) {
            return 0;
        }

        uint256 c = a * b;
        require(c / a == b);

        return c;
    }

    /**
    * @dev Integer division of two unsigned integers truncating the quotient, reverts
on division by zero.
    */
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        // Solidity only automatically asserts when dividing by 0
        require(b > 0);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
    * @dev Subtracts two unsigned integers, reverts on overflow (i.e. if subtrahend
is greater than minuend).
    */
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b <= a);
        uint256 c = a - b;

        return c;
    }

    /**
    * @dev Adds two unsigned integers, reverts on overflow.
    */
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
```

```
        require(c >= a);

        return c;
    }

    /**
    * @dev Divides two unsigned integers and returns the remainder (unsigned integer
modulo),
    * reverts when dividing by zero.
    */
    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b != 0);
        return a % b;
    }
}

contract Ownable {
    address private _owner;

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

    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    constructor () internal {
        address msgSender = msg.sender;
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }

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

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

```
    }

    /**
     * @dev Leaves the contract without owner. It will not be possible to call
     * `onlyOwner` functions anymore. Can only be called by the current owner.
     *
     * NOTE: Renouncing ownership will leave the contract without an owner,
     * thereby removing any functionality that is only available to the owner.
     */
    function renounceOwnership() public onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public onlyOwner {
        //SlowMist// This check is quite good in avoiding losing control of the
contract caused by user mistakes
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}

contract Pausable is Ownable {
    event Paused(address account);
    event Unpaused(address account);

    bool private _paused;

    constructor () internal {
        _paused = false;
    }

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

```
    /**
     * @dev Modifier to make a function callable only when the contract is not
paused.
     */
    modifier whenNotPaused() {
        require(!_paused);
        _;
    }

    /**
     * @dev Modifier to make a function callable only when the contract is paused.
     */
    modifier whenPaused() {
        require(_paused);
        _;
    }

    /**
     * @dev called by the owner to pause, triggers stopped state
     */
    //SlowMist// Suspending all transactions upon major abnormalities is a
recommended approach
    function pause() public onlyOwner whenNotPaused {
        _paused = true;
        emit Paused(msg.sender);
    }

    /**
     * @dev called by the owner to unpause, returns to normal state
     */
    function unpause() public onlyOwner whenPaused {
        _paused = false;
        emit Unpaused(msg.sender);
    }
}

interface IERC20 {
    function transfer(address to, uint256 value) external returns (bool);

    function approve(address spender, uint256 value) external returns (bool);

    function transferFrom(address from, address to, uint256 value) external returns
(bool);
```

```
    function totalSupply() external view returns (uint256);

    function balanceOf(address who) external view returns (uint256);

    function allowance(address owner, address spender) external view returns
(uint256);

    event Transfer(address indexed from, address indexed to, uint256 value);

    event Approval(address indexed owner, address indexed spender, uint256 value);
}

contract ERC20 is IERC20 {
    using SafeMath for uint256;

    mapping (address => uint256) internal _balances;

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

    uint256 private _totalSupply;

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

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

    /**
     * @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 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) {
        _transfer(msg.sender, to, value);
        //SlowMist// The return value conforms to the ERC20 specification
        return true;
    }

    /**
     * @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) {
        //SlowMist// This kind of check is very good, avoiding user mistake leading
to approve errors
        require(spender != address(0));

        _allowed[msg.sender][spender] = value;
        emit Approval(msg.sender, spender, value);
        //SlowMist// The return value conforms to the ERC20 specification
        return true;
    }

    /**
     * @dev Transfer tokens from one address to another.
     * Note that while this function emits an Approval event, this is not required as
```

```
per the specification,
     * and other compliant implementations may not emit the event.
     * @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) {
        _allowed[from][msg.sender] = _allowed[from][msg.sender].sub(value);
        _transfer(from, to, value);
        emit Approval(from, msg.sender, _allowed[from][msg.sender]);
        //SlowMist// The return value conforms to the ERC20 specification
        return true;
    }

    /**
     * @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
     * Emits an Approval event.
     * @param spender The address which will spend the funds.
     * @param addedValue The amount of tokens to increase the allowance by.
     */
    function increaseAllowance(address spender, uint256 addedValue) public returns
(bool) {
        require(spender != address(0));

        _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
     * Emits an Approval event.
     * @param spender The address which will spend the funds.
     * @param subtractedValue The amount of tokens to decrease the allowance by.
```

```
     */
    function decreaseAllowance(address spender, uint256 subtractedValue) public
returns (bool) {
        require(spender != address(0));

        _allowed[msg.sender][spender] = _allowed[msg.sender]
[spender].sub(subtractedValue);
        emit Approval(msg.sender, spender, _allowed[msg.sender][spender]);
        return true;
    }

    /**
    * @dev Transfer token for a specified addresses
    * @param from The address to transfer from.
    * @param to The address to transfer to.
    * @param value The amount to be transferred.
    */
    function _transfer(address from, address to, uint256 value) internal {
        //SlowMist// This kind of check is very good, avoiding user mistake leading
to the loss of token during transfer
        require(to != address(0));

        _balances[from] = _balances[from].sub(value);
        _balances[to] = _balances[to].add(value);
        emit Transfer(from, to, value);
    }

    /**
     * @dev Internal function that mints an amount of the token and assigns it to
     * an account. This encapsulates the modification of balances such that the
     * proper events are emitted.
     * @param account The account that will receive the created tokens.
     * @param value The amount that will be created.
     */
    function _mint(address account, uint256 value) internal {
        require(account != address(0));

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

    /**
     * @dev Internal function that burns an amount of the token of a given
```

```
     * account.
     * @param account The account whose tokens will be burnt.
     * @param value The amount that will be burnt.
     */
    function _burn(address account, uint256 value) internal {
        require(account != address(0));

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

    /**
     * @dev Internal function that burns an amount of the token of a given
     * account, deducting from the sender's allowance for said account. Uses the
     * internal burn function.
     * Emits an Approval event (reflecting the reduced allowance).
     * @param account The account whose tokens will be burnt.
     * @param value The amount that will be burnt.
     */
    function _burnFrom(address account, uint256 value) internal {
        _allowed[account][msg.sender] = _allowed[account][msg.sender].sub(value);
        _burn(account, value);
        emit Approval(account, msg.sender, _allowed[account][msg.sender]);
    }
}

contract ERC20Pausable is ERC20, Pausable {
    function transfer(address to, uint256 value) public whenNotPaused returns (bool)
{
        return super.transfer(to, value);
    }

    function transferFrom(address from, address to, uint256 value) public
whenNotPaused returns (bool) {
        return super.transferFrom(from, to, value);
    }

    /*
     * approve/increaseApprove/decreaseApprove can be set when Paused state
     */

    /*
     * function approve(address spender, uint256 value) public whenNotPaused returns
```

```
(bool) {
     *     return super.approve(spender, value);
     * }
     *
     * function increaseAllowance(address spender, uint addedValue) public
whenNotPaused returns (bool success) {
     *     return super.increaseAllowance(spender, addedValue);
     * }
     *
     * function decreaseAllowance(address spender, uint subtractedValue) public
whenNotPaused returns (bool success) {
     *     return super.decreaseAllowance(spender, subtractedValue);
     * }
     */
}

contract ERC20Detailed is IERC20 {
    string private _name;
    string private _symbol;
    uint8 private _decimals;

    constructor (string memory name, string memory symbol, uint8 decimals) public {
        _name = name;
        _symbol = symbol;
        _decimals = decimals;
    }

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

    /**
     * @return the symbol of the token.
     */
    function symbol() public view returns (string memory) {
        return _symbol;
    }

    /**
     * @return the number of decimals of the token.
     */
```

```
    function decimals() public view returns (uint8) {
        return _decimals;
    }
}

contract KStarCoin is ERC20Detailed, ERC20Pausable {

    struct LockInfo {
        uint256 _releaseTime;
        uint256 _amount;
    }

    mapping (address => LockInfo[]) public timelockList;
        mapping (address => bool) public frozenAccount;

    event Freeze(address indexed holder);
    event Unfreeze(address indexed holder);
    event Lock(address indexed holder, uint256 value, uint256 releaseTime);
    event Unlock(address indexed holder, uint256 value);

    modifier notFrozen(address _holder) {
        require(!frozenAccount[_holder]);
        _;
    }

    constructor() ERC20Detailed("KStarCoin", "KSC", 18) public {
        _mint(msg.sender, 1000000000 * (10 ** 18));
    }

    function balanceOf(address owner) public view returns (uint256) {
        uint256 totalBalance = super.balanceOf(owner);
        if( timelockList[owner].length >0 ){
            for(uint i=0; i<timelockList[owner].length;i++){
                totalBalance = totalBalance.add(timelockList[owner][i]._amount);
            }
        }
        return totalBalance;
    }

    function transfer(address to, uint256 value) public notFrozen(msg.sender) returns
(bool) {
        if (timelockList[msg.sender].length > 0 ) {
            _autoUnlock(msg.sender);
        }
```

```
        return super.transfer(to, value);
    }

    function transferFrom(address from, address to, uint256 value) public
notFrozen(from) returns (bool) {
        if (timelockList[from].length > 0) {
            _autoUnlock(from);
        }
        return super.transferFrom(from, to, value);
    }
    //SlowMist// Only the owner role can freeze users' account through the
freezeAccout function
    function freezeAccount(address holder) public onlyOwner returns (bool) {
        require(!frozenAccount[holder]);
        frozenAccount[holder] = true;
        emit Freeze(holder);
        return true;
    }
    //SlowMist// Only the owner role can unfreeze users' account through the
unfreezeAccout function
    function unfreezeAccount(address holder) public onlyOwner returns (bool) {
        require(frozenAccount[holder]);
        frozenAccount[holder] = false;
        emit Unfreeze(holder);
        return true;
    }
    //SlowMist// Only the owner role can lock users' account through the lock
function
    function lock(address holder, uint256 value, uint256 releaseTime) public
onlyOwner returns (bool) {
        require(_balances[holder] >= value,"There is not enough balances of
holder.");
        _lock(holder,value,releaseTime);

        return true;
    }

    function transferWithLock(address holder, uint256 value, uint256 releaseTime)
public onlyOwner returns (bool) {
        _transfer(msg.sender, holder, value);
        _lock(holder,value,releaseTime);
        return true;
    }
    //SlowMist// Only the owner role can unlock users' account through the unlock
```

function

```
    function unlock(address holder, uint256 idx) public onlyOwner returns (bool) {
        require( timelockList[holder].length > idx, "There is not lock info.");
        _unlock(holder,idx);
        return true;
    }

    function _lock(address holder, uint256 value, uint256 releaseTime) internal
returns(bool) {
        _balances[holder] = _balances[holder].sub(value);
        timelockList[holder].push( LockInfo(releaseTime, value) );

        emit Lock(holder, value, releaseTime);
        return true;
    }

    function _unlock(address holder, uint256 idx) internal returns(bool) {
        LockInfo storage lockinfo = timelockList[holder][idx];
        uint256 releaseAmount = lockinfo._amount;

        delete timelockList[holder][idx];
        timelockList[holder][idx] = timelockList[holder]
[timelockList[holder].length.sub(1)];
        timelockList[holder].length -=1;

        emit Unlock(holder, releaseAmount);
        _balances[holder] = _balances[holder].add(releaseAmount);

        return true;
    }

    function _autoUnlock(address holder) internal returns(bool) {
        for(uint256 idx =0; idx < timelockList[holder].length ; idx++ ) {
            if (timelockList[holder][idx]._releaseTime <= now) {
                // If lockupinfo was deleted, loop restart at same position.
                if( _unlock(holder, idx) ) {
                    idx -=1;
                }
            }
        }
        return true;
    }
}
```

Statement

SlowMist issues this report with reference to the facts that have occurred or existed before the issuance of this report, and only assumes corresponding responsibility based on these.

For the facts that occurred or existed after the issuance, SlowMist is not able to judge the security status of this project, and is not responsible for them. The security audit analysis and other contents of this report are based on the documents and materials provided to SlowMist by the information provider till the date of the insurance report (referred to as "provided information"). SlowMist assumes: The information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed, or inconsistent with the actual situation, the SlowMist shall not be liable for any loss or adverse effect resulting therefrom. SlowMist only conducts the agreed security audit on the security situation of the project and issues this report. SlowMist is not responsible for the background and other conditions of the project.

Official Website

www.slowmist.com

E-mail

team@slowmist.com

Twitter

@SlowMist_Team

Github

https://github.com/slowmist