Smart Contract Security Audit Report

The SlowMist Security Team received the IZE Fintech Blockchain team's application for smart contract security audit of the IZE on June 09, 2020. The following are the details and results of this smart contract security audit:

Token name :

IZE

The Contract address :

0x6944d3e38973c4831da24e954fbd790c7e688bdd

Link address :

https://etherscan.io/address/0x6944d3e38973c4831da24e954fbd790c7e688bdd

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	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 : **Passed**

Audit Number : 0X002006110001

Audit Date : June 11, 2020

Audit Team : SlowMist Security Team

(Statement : SlowMist only issues this report based on the fact that has occurred or existed before the report is issued, and bears the corresponding responsibility in this regard. For the facts occur or exist later after the report, SlowMist cannot judge the security status of its smart contract. SlowMist is not responsible for it. The security audit analysis and other contents of this report are based on the documents and materials provided by the information provider to SlowMist as of the date of this report (referred to as "the provided information"). SlowMist assumes that: there has been no information missing, tampered, deleted, or concealed. If the information provided has been missed, modified, deleted, concealed or reflected and is inconsistent with the actual situation, SlowMist will not bear any responsibility for the resulting loss and adverse effects. SlowMist will not bear any responsibility for the background or other circumstances of the project.)

Summary: This is a token contract that contains the tokenVault section. The total amount of contract tokens can be changed, owner can burn his tokens through the burn function. The owner can mint tokens through the mint function. The owner or operator can lock any account balance through the addLock function. OpenZeppelin’s SafeMath security module is used, which is a commendable approach. The contract does not have the Overflow and the Race Conditions issue. The comprehensive evaluation contract is no risk.

The source code:

```
/**
 *Submitted for verification at Etherscan.io on 2020-01-16
*/

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

pragma solidity ^0.5.0;
```

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


/**
  * @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 (uint);
  function transferFrom(address from, address to, uint value) public returns (bool);
  function approve(address spender, uint value) public returns (bool);
  event Approval(address indexed owner, address indexed spender, uint value);
}

/**
  * @title Ownable
  * @dev Owner validator
  */
contract Ownable {
  address private _owner;
  address private _operator;

  event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);
  event OperatorTransferred(address indexed previousOperator, address indexed newOperator);

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

```
    _operator = msg.sender;
    emit OwnershipTransferred(address(0), _owner);
    emit OperatorTransferred(address(0), _operator);
}

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

/**
    * @return the address of the operator.
    */
function operator() public view returns (address) {
    return _operator;
}

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

/**
    * @dev Throws if called by any account other than the owner or operator.
    */
modifier onlyOwnerOrOperator() {
    require(isOwner() || isOperator());
    _;
}


/**
    * @return true if `msg.sender` is the owner of the contract.
    */
function isOwner() public view returns (bool) {
    return msg.sender == _owner;
}
```

```
    /**
     * @return true if `msg.sender` is the operator of the contract.
     */
    function isOperator() public view returns (bool) {
        return msg.sender == _operator;
    }

    /**
     * @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

        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }

    /**
     * @dev Allows the current operator to transfer control of the contract to a newOperator.
     * @param newOperator The address to transfer ownership to.
     */
    function transferOperator(address newOperator) public onlyOwner {
        require(newOperator != address(0));

        emit OperatorTransferred(_operator, newOperator);
        _operator = newOperator;
    }


}


/**
 * @title Pausable
 * @dev Base contract which allows children to implement an emergency stop mechanism.
 */
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 a pauser to pause, triggers stopped state.
 */
//SlowMist// Suspending all transactions upon major abnormalities is a recommended approach
function pause() public onlyOwnerOrOperator whenNotPaused {
    _paused = true;
    emit Paused(msg.sender);
}
```

```
  /**
    * @dev Called by a pauser to unpause, returns to normal state.
    */
  function unpause() public onlyOwnerOrOperator whenPaused {
    _paused = false;
    emit Unpaused(msg.sender);
  }
}

/**
  * @title SafeMath
  * @dev Unsigned math operations with safety checks that revert on error.
  */
```

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

```
library SafeMath {

  /**
    * @dev Multiplies two unsigned integers, reverts on overflow.
    */
  function mul(uint a, uint b) internal pure returns (uint) {
    if (a == 0) {
      return 0;
    }

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

    return c;
  }

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

    return c;
  }
```

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

    return c;
  }

  /**
   * @dev Adds two numbers, throws on overflow.
   */
  function add(uint a, uint b) internal pure returns (uint) {
    uint 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(uint a, uint b) internal pure returns (uint) {
    require(b != 0);
    return a % b;
  }

}


/**
 * @title StandardToken
 * @dev Base OI token
 */
contract StandardToken is ERC20, Pausable {
  using SafeMath for uint;

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

```
mapping (address => mapping (address => uint)) private _allowed;

uint private _totalSupply;

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

/**
  * @dev Gets the balance of the specified address.
  * @param owner The address to query the balance of.
  * @return A uint representing the amount owned by the passed address.
  */
function balanceOf(address owner) public view returns (uint) {
  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 uint specifying the amount of tokens still available for the spender.
  */
function allowance(address owner, address spender) public view returns (uint) {
  return _allowed[owner][spender];
}

/**
  * @dev Transfer token to a specified address.
  * @param to The address to transfer to.
  * @param value The amount to be transferred.
  */
function transfer(address to, uint value) public whenNotPaused returns (bool) {
  _transfer(msg.sender, 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, uint value) public whenNotPaused returns (bool) {
    _approve(msg.sender, spender, value);

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

}

/**
    * @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 uint the amount of tokens to be transferred
    */
function transferFrom(address from, address to, uint value) public whenNotPaused returns (bool) {
    _transferFrom(from, to, value);

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

}

/**
    * @dev Increase the amount of tokens that an owner allowed to a spender.
    * approve should be called when _allowed[msg.sender][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, uint addedValue) public whenNotPaused returns (bool) {
    _approve(msg.sender, spender, _allowed[msg.sender][spender].add(addedValue));
    return true;
}
```

```
    /**
     * @dev Decrease the amount of tokens that an owner allowed to a spender.
     * approve should be called when _allowed[msg.sender][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, uint subtractedValue) public whenNotPaused returns (bool) {
        _approve(msg.sender, spender, _allowed[msg.sender][spender].sub(subtractedValue));
        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, uint value) internal {
        require(to != address(0)); //SlowMist// This kind of check is very good, avoiding user mistake leading
to the loss of token during transfer

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

    /**
     * @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 uint the amount of tokens to be transferred
     */
    function _transferFrom(address from, address to, uint value) internal {
```

```
    _transfer(from, to, value);
    _approve(from, msg.sender, _allowed[from][msg.sender].sub(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, uint 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 the owner
   * account.
   * @param value The amount that will be burnt.
   */
  function _burn(uint value) internal {
    _totalSupply = _totalSupply.sub(value);
    _balances[msg.sender] = _balances[msg.sender].sub(value);
    emit Transfer(msg.sender, address(0), value);
  }

  /**
   * @dev Approve an address to spend another addresses' tokens.
   * @param owner The address that owns the tokens.
   * @param spender The address that will spend the tokens.
   * @param value The number of tokens that can be spent.
   */
  function _approve(address owner, address spender, uint value) internal {

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

    require(owner != address(0));
```

```
        _allowed[owner][spender] = value;
        emit Approval(owner, spender, value);
    }
}

/**
 * @title MintableToken
 * @dev Mintable token
 */
contract MintableToken is StandardToken {
    event MintFinished();

    bool public mintingFinished = false;

    modifier canMint() {
        require(!mintingFinished);
        _;
    }

    /**
     * @dev Function to mint tokens
     * @param to The address that will receive the minted tokens.
     * @param amount The amount of tokens to mint
     * @return A boolean that indicates if the operation was successful.
     */
    function mint(address to, uint amount) public whenNotPaused onlyOwner canMint returns (bool) {
        _mint(to, amount);
        return true;
    }

    /**
     * @dev Function to stop minting new tokens.
     * @return True if the operation was successful.
     */
    function finishMinting() public whenNotPaused onlyOwner canMint returns (bool) {
        mintingFinished = true;
        emit MintFinished();
        return true;
    }
}
```

```
/**
 * @title Burnable Token
 * @dev Token that can be irreversibly burned (destroyed).
 */
contract BurnableToken is MintableToken {
  /**
   * @dev Burns a specific amount of tokens.
   * @param value The amount of token to be burned.
   */
  function burn(uint value) public whenNotPaused onlyOwner returns (bool) {
    _burn(value);
    return true;
  }
}



/**
 * @title LockableToken
 * @dev locking of granted balance
 */
contract LockableToken is BurnableToken {

  using SafeMath for uint;

  /**
   * @dev Lock defines a lock of token
   */
  struct Lock {
    uint amount;
    uint expiresAt;
  }

  mapping (address => Lock[]) public grantedLocks;

  /**
   * @dev Transfer tokens to another
   * @param to address the address which you want to transfer to
   * @param value uint the amount of tokens to be transferred
   */
  function transfer(address to, uint value) public whenNotPaused returns (bool) {
    _verifyTransferLock(msg.sender, value);
```

```
    _transfer(msg.sender, to, value);
    return true;
}

/**
    * @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 uint the amount of tokens to be transferred
    */
function transferFrom(address from, address to, uint value) public whenNotPaused returns (bool) {
    _verifyTransferLock(from, value);
    _transferFrom(from, to, value);
    return true;
}

/**
    * @dev Function to add lock
    * @param granted The address that will be locked.
    * @param amount The amount of tokens to be locked
    * @param expiresAt The expired date as unix timestamp
    */
```

//SlowMist// The owner or operator can lock any account balance through the addLock function

```
function addLock(address granted, uint amount, uint expiresAt) public whenNotPaused onlyOwnerOrOperator {
    require(amount > 0);
    require(expiresAt > now);

    grantedLocks[granted].push(Lock(amount, expiresAt));
}

/**
    * @dev Function to delete lock
    * @param granted The address that was locked
    * @param index The index of lock
    */
function deleteLock(address granted, uint8 index) public whenNotPaused onlyOwnerOrOperator {
    require(grantedLocks[granted].length > index);

    uint len = grantedLocks[granted].length;
    if (len == 1) {
        delete grantedLocks[granted];
```

```
    } else {
      if (len - 1 != index) {
        grantedLocks[granted][index] = grantedLocks[granted][len - 1];
      }
      delete grantedLocks[granted][len - 1];
    }
  }

  /**
   * @dev Verify transfer is possible
   * @param from - granted
   * @param value - amount of transfer
   */
  function _verifyTransferLock(address from, uint value) internal view {
    uint lockedAmount = getLockedAmount(from);
    uint balanceAmount = balanceOf(from);

    require(balanceAmount.sub(lockedAmount) >= value);
  }

  /**
   * @dev get locked amount of address
   * @param granted The address want to know the lock state.
   * @return locked amount
   */
  function getLockedAmount(address granted) public view returns(uint) {
    uint lockedAmount = 0;

    uint len = grantedLocks[granted].length;
    for (uint i = 0; i < len; i++) {
      if (now < grantedLocks[granted][i].expiresAt) {
        lockedAmount = lockedAmount.add(grantedLocks[granted][i].amount);
      }
    }
    return lockedAmount;
  }
}

/**
 * @title IzeToken
 * @dev ERC20 Token
 */
```

```
contract IzeToken is LockableToken {

    string public constant name = "IZE Fintech Blockchain";
    string public constant symbol = "IZE";
    uint32 public constant decimals = 18;

    uint public constant INITIAL_SUPPLY = 100000000000e18;

    /**
     * @dev Constructor that gives msg.sender all of existing tokens.
     */
    constructor() public {
        _mint(msg.sender, INITIAL_SUPPLY);
        emit Transfer(address(0), msg.sender, INITIAL_SUPPLY);
    }
}
```

慢雾科技
slow mist

Official Website
www.slowmist.com

E-mail
team@slowmist.com

Twitter
@SlowMist_Team

WeChat Official Account