Smart Contract Security Audit Report

[2021]

The SlowMist Security Team received the CRTR team's application for smart contract security audit of the FANDOM on 2021.07.13. The following are the details and results of this smart contract security audit:

Token Name :

FANDOM

The contract address :

https://etherscan.io/address/0x0ac65666f502a6a9de0a1393f42c72d3ff62c40f

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

Audit Result : Passed

Audit Number : 0x002107160002

Audit Date : 2021.07.13 - 2021.07.16

Audit Team : SlowMist Security Team

Summary conclusion : This is a token contract that contains the tokenVault section. The total amount of contract tokens can be changed, the owner can burn his own tokens through the burn function. 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. The Owner can freeze any user account through the freeze function.
2. The Owner can unfreeze any user account through the unfreeze function.
3. The Owner can time lock any user account through the lock function.
4. The Owner can unlock the time lock of any user account through the unlock function.

The source code:

```
/**
 *Submitted for verification at Etherscan.io on 2021-07-15
*/
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.5.4;

// File: node_modules/openzeppelin-solidity/contracts/token/ERC20/IERC20.sol

/**
 * @title ERC20 interface
 * @dev see https://github.com/ethereum/EIPs/issues/20
 */
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);
}

// File: node_modules/openzeppelin-solidity/contracts/math/SafeMath.sol

/**
 * @title SafeMath
 * @dev Unsigned math operations with safety checks that revert on error
 */
//SlowMist// SafeMath security module is used, which is a recommend 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;
    }
}

// File: node_modules/openzeppelin-solidity/contracts/token/ERC20/ERC20.sol
```

```
/**
 * @title Standard ERC20 token
 *
 * @dev Implementation of the basic standard token.
 * https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md
 * Originally based on code by FirstBlood:
 *
https://github.com/Firstbloodio/token/blob/master/smart_contract/FirstBloodToken.sol
 *
 * This implementation emits additional Approval events, allowing applications to
reconstruct the allowance status for
 * all accounts just by listening to said events. Note that this isn't required by
the specification, and other
 * compliant implementations may not do it.
 */
contract ERC20 is IERC20 {
    using SafeMath for uint256;

    mapping (address => uint256) internal _balances;

    mapping (address => mapping (address => uint256)) private _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 EIP20 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 EIP20 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 EIP20 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.
     */
    //SlowMist// Because burnFrom() and transferFrom() share the allowed amount of
approve(), if the agent be evil, there is the possibility of malicious burn
    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 CRTRToken is ERC20 {
    string public constant name = "FANDOM";
    string public constant symbol = "CRTR";
    uint8 public constant decimals = 18;
    uint256 public constant initialSupply = 4000000000 * (10 ** uint256(decimals));

    constructor() public {
        super._mint(msg.sender, initialSupply);
        owner = msg.sender;
    }

    //ownership
    address public owner;
    event OwnershipRenounced(address indexed previousOwner);
    event OwnershipTransferred(
    address indexed previousOwner,
    address indexed newOwner
    );

    modifier onlyOwner() {
        require(msg.sender == owner, "Not owner");
        _;
    }

  /**
   * @dev Allows the current owner to relinquish control of the contract.
   * @notice Renouncing to ownership will leave the contract without an owner.
   * It will not be possible to call the functions with the `onlyOwner`
   * modifier anymore.
   */
    function renounceOwnership() public onlyOwner {
        emit OwnershipRenounced(owner);
        owner = address(0);
    }

  /**
   * @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 {
        _transferOwnership(_newOwner);
    }

  /**
   * @dev Transfers control of the contract to a newOwner.
   * @param _newOwner The address to transfer ownership to.
   */
    function _transferOwnership(address _newOwner) internal {
        //SlowMist// This check is quite good in avoiding losing control of the
contract caused by user mistakes
        require(_newOwner != address(0), "Already owner");
        emit OwnershipTransferred(owner, _newOwner);
        owner = _newOwner;
    }

    //pausable
    event Pause();
    event Unpause();
    bool public paused = false;

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

    /**
    * @dev Modifier to make a function callable only when the contract is paused.
    */
    modifier whenPaused() {
        require(paused, "Not paused now");
        _;
    }

    /**
    * @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 Pause();
    }

    /**
    * @dev called by the owner to unpause, returns to normal state
    */
    function unpause() public onlyOwner whenPaused {
        paused = false;
        emit Unpause();
    }

    //freezable
    event Frozen(address target);
    event Unfrozen(address target);

    mapping(address => bool) internal freezes;

    modifier whenNotFrozen() {
        require(!freezes[msg.sender], "Sender account is locked.");
        _;
    }
    //SlowMist// The Owner can freeze any user account through the freeze function
    function freeze(address _target) public onlyOwner {
        freezes[_target] = true;
        emit Frozen(_target);
    }
    //SlowMist// The Owner can unfreeze any user account through the unfreeze
function
    function unfreeze(address _target) public onlyOwner {
        freezes[_target] = false;
        emit Unfrozen(_target);
    }

    function isFrozen(address _target) public view returns (bool) {
        return freezes[_target];
    }

    function transfer(
        address _to,
        uint256 _value
    )
      public
      whenNotFrozen
      whenNotPaused
      returns (bool)
    {
        releaseLock(msg.sender);
        return super.transfer(_to, _value);
    }

    function transferFrom(
        address _from,
        address _to,
        uint256 _value
    )
      public
      whenNotPaused
      returns (bool)
    {
        require(!freezes[_from], "From account is locked.");
        releaseLock(_from);
        return super.transferFrom(_from, _to, _value);
    }
    //burnable
    event Burn(address indexed burner, uint256 value);

    function burn(uint256 _value) public onlyOwner {
        require(_value <= super.balanceOf(msg.sender), "Balance is too small.");

        address _who = msg.sender;
        _burn(_who, _value);
        emit Burn(_who, _value);
    }


    //lockable
    struct LockInfo {
        uint256 releaseTime;
        uint256 balance;
    }
    mapping(address => LockInfo[]) internal lockInfo;

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

    function balanceOf(address _holder) public view returns (uint256 balance) {
        uint256 lockedBalance = 0;
        for(uint256 i = 0; i < lockInfo[_holder].length ; i++ ) {
            lockedBalance = lockedBalance.add(lockInfo[_holder][i].balance);
        }
        return super.balanceOf(_holder).add(lockedBalance);
    }

    function releaseLock(address _holder) internal {

        for(uint256 i = 0; i < lockInfo[_holder].length ; i++ ) {
            if (lockInfo[_holder][i].releaseTime <= now) {
                _balances[_holder] = _balances[_holder].add(lockInfo[_holder]
[i].balance);
                emit Unlock(_holder, lockInfo[_holder][i].balance);
                lockInfo[_holder][i].balance = 0;

                if (i != lockInfo[_holder].length - 1) {
                    lockInfo[_holder][i] = lockInfo[_holder][lockInfo[_holder].length
- 1];

                    i--;
                }
                lockInfo[_holder].length--;

            }
        }
    }
    function lockCount(address _holder) public view returns (uint256) {
        return lockInfo[_holder].length;
    }
    function lockState(address _holder, uint256 _idx) public view returns (uint256,
uint256) {
        return (lockInfo[_holder][_idx].releaseTime, lockInfo[_holder]
[_idx].balance);
    }
    //SlowMist// The Owner can time lock any user account through the lock function
    function lock(address _holder, uint256 _amount, uint256 _releaseTime) public
onlyOwner {
        require(super.balanceOf(_holder) >= _amount, "Balance is too small.");
        _balances[_holder] = _balances[_holder].sub(_amount);
        lockInfo[_holder].push(
            LockInfo(_releaseTime, _amount)
        );
        emit Lock(_holder, _amount, _releaseTime);
    }

    function lockAfter(address _holder, uint256 _amount, uint256 _afterTime) public
onlyOwner {
        require(super.balanceOf(_holder) >= _amount, "Balance is too small.");
        _balances[_holder] = _balances[_holder].sub(_amount);
        lockInfo[_holder].push(
            LockInfo(now + _afterTime, _amount)
        );
        emit Lock(_holder, _amount, now + _afterTime);
    }
    //SlowMist// The Owner can unlock the time lock of any user account through the
unlock function
    function unlock(address _holder, uint256 i) public onlyOwner {
        require(i < lockInfo[_holder].length, "No lock information.");

        _balances[_holder] = _balances[_holder].add(lockInfo[_holder][i].balance);
        emit Unlock(_holder, lockInfo[_holder][i].balance);
        lockInfo[_holder][i].balance = 0;

        if (i != lockInfo[_holder].length - 1) {
            lockInfo[_holder][i] = lockInfo[_holder][lockInfo[_holder].length - 1];
        }
        lockInfo[_holder].length--;
    }

    function transferWithLock(address _to, uint256 _value, uint256 _releaseTime)
public onlyOwner returns (bool) {
        require(_to != address(0), "wrong address");
        require(_value <= super.balanceOf(owner), "Not enough balance");

        _balances[owner] = _balances[owner].sub(_value);
        lockInfo[_to].push(
            LockInfo(_releaseTime, _value)
        );
        emit Transfer(owner, _to, _value);
        emit Lock(_to, _value, _releaseTime);

        return true;
    }

    function transferWithLockAfter(address _to, uint256 _value, uint256 _afterTime)
public onlyOwner returns (bool) {
        require(_to != address(0), "wrong address");
        require(_value <= super.balanceOf(owner), "Not enough balance");

        _balances[owner] = _balances[owner].sub(_value);
        lockInfo[_to].push(
            LockInfo(now + _afterTime, _value)
        );
        emit Transfer(owner, _to, _value);
        emit Lock(_to, _value, now + _afterTime);

        return true;
    }

    function currentTime() public view returns (uint256) {
        return now;
    }

    function afterTime(uint256 _value) public view returns (uint256) {
        return now + _value;
    }
}
```

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