Smart Contract Security Audit Report

[2021]

The SlowMist Security Team received the ATT team's application for smart contract security audit of the ARTUBE TOKEN on 2021.11.08. The following are the details and results of this smart contract security audit:

Token Name :

ARTUBE TOKEN

The contract address :

https://scope.klaytn.com/account/0x07aa7ae19b17579f7237ad72c616fecf4ccc787b?tabId=contractCode

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 : 0X002111090001

Audit Date : 2021.11.08 - 2021.11.09

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, 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 role can stop all token transfers by setting the stopped value to true.
2. The owner role can start all token transfers by setting the stopped value to false.
3. The owner role can lock users' tokens amount at specified time.

The source code:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.5.0;

/**
 * @title ERC20 interface
 * @dev see https://github.com/ethereum/EIPs/issues/20
 */
interface IERC20 {
  function totalSupply() external view returns (uint256);

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

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

  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);

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

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

/**
 * @title ERC20Detailed token
 * @dev The decimals are only for visualization purposes.
 * All the operations are done using the smallest and indivisible token unit,
 * just as on Ethereum all the operations are done in wei.
 */
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;
    }
}

/**
 * @title SafeMath
 * @dev Math operations with safety checks that revert on error
 */
//SlowMist// SafeMath security module is used, which is a recommended approach
library SafeMath {

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

      return c;
    }

    /**
    * @dev Subtracts two numbers, 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 numbers, 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 numbers 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;
    }
  }

  /**
   * @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
   */
  contract ERC20 is IERC20 {
    using SafeMath for uint256;

    mapping (address => uint256) private _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 KIP7 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 KIP7 specification
      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 uint256 the amount of tokens to be transferred
     */
    function transferFrom(
      address from,
      address to,
      uint256 value
    )
      public
      returns (bool)
```

```
{
  require(value <= _allowed[from][msg.sender]);

  _allowed[from][msg.sender] = _allowed[from][msg.sender].sub(value);
  _transfer(from, to, value);
  //SlowMist// The return value conforms to the KIP7 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
 * @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
 * @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 {
    require(value <= _balances[from]);
    //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));
    require(value <= _balances[account]);

    _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.
   * @param account The account whose tokens will be burnt.
   * @param value The amount that will be burnt.
   */
  function _burnFrom(address account, uint256 value) internal {
    require(value <= _allowed[account][msg.sender]);

    // Should https://github.com/OpenZeppelin/zeppelin-solidity/issues/707 be
accepted,
    // this function needs to emit an event with the updated approval.
    _allowed[account][msg.sender] = _allowed[account][msg.sender].sub(
      value);
    _burn(account, value);
  }
}


/**
 * @title Burnable Token
 * @dev Token that can be irreversibly burned (destroyed).
 */
contract ERC20Burnable is ERC20 {

  /**
   * @dev Burns a specific amount of tokens.
   * @param value The amount of token to be burned.
   */
  function burn(uint256 value) public {
    _burn(msg.sender, value);
  }

  /**
    * @dev Burns a specific amount of tokens from the target address and decrements
allowance
    * @param from address The address which you want to send tokens from
    * @param value uint256 The amount of token to be burned
    */
  /*
  --Do not use
  function burnFrom(address from, uint256 value) public {
    _burnFrom(from, value);
    //Only the owner's address can be burned.  @ejkang
  }
  */
}



pragma solidity ^0.5.0;

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

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

  /**
    * @dev The Ownable constructor sets the original `owner` of the contract to the
sender
    * account.
    */
  constructor() internal {
    _owner = msg.sender;
    emit OwnershipTransferred(address(0), _owner);
  }
```

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

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

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

    /**
     * @dev Allows the current owner to transfer control of the contract to a newOwner.
     * @param newOwner The address to transfer ownership to.
     */
    function transferOwnership(address newOwner) public onlyOwner {
      _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));
      emit OwnershipTransferred(_owner, newOwner);
      _owner = newOwner;
    }
}

/**
 * @title Stoppable
 * @dev Change status public value for contract operation.
 */
contract Stoppable is Ownable{
      bool public stopped = false;

      modifier enabled {
          require (!stopped);
          _;
      }
      /**
          * @dev Run only owner. For vaalue change
      */
      //SlowMist// The owner role can stop all token transfers by setting the stopped
value to true
      function stop() external onlyOwner {
          stopped = true;
      }
      /**
          * @dev Run only owner. For vaalue change
      */
      //SlowMist// The owner role can start all token transfers by setting the stopped
value to false
      function start() external onlyOwner {
          stopped = false;
      }
}


/// @author
/// @title Token contract
contract ATTToken is ERC20Detailed, ERC20Burnable, Stoppable {

      constructor (
              string memory name,
              string memory symbol,
              uint256 totalSupply,
              uint8 decimals
      ) ERC20Detailed(name, symbol, decimals)
      public {
          _mint(owner(), totalSupply * 10**uint(decimals));
      }

      // Don't accept ETH
      function () payable external {
          revert();
      }

      //-------------------------
      // Lock account transfer

      mapping (address => uint256) private _lockTimes;
      mapping (address => uint256) private _lockAmounts;

      event LockChanged(address indexed account, uint256 releaseTime, uint256 amount);

      /// Lock user amount.  (run only owner)
      /// @param account account to lock
      /// @param releaseTime Time to release from lock state.
      /// @param amount  amount to lock.
      /// @return Boolean
      //SlowMist// The owner role can lock users' tokens amount at specified time
      function setLock(address account, uint256 releaseTime, uint256 amount) onlyOwner
  public {
          //require(now < releaseTime, "ERC20 : Current time is greater than release
  time");
          require(block.timestamp < releaseTime, "ERC20 : Current time is greater than
  release time");
          require(amount != 0, "ERC20: Amount error");
          _lockTimes[account] = releaseTime;
          _lockAmounts[account] = amount;
          emit LockChanged( account, releaseTime, amount );
      }

      /// Get Lock information  (run anyone)
      /// @param account user acount
      /// @return lokced time and locked amount.
      function getLock(address account) public view returns (uint256 lockTime, uint256
  lockAmount) {
          return (_lockTimes[account], _lockAmounts[account]);
      }

      /// Check lock state  (run anyone)
      /// @param account user acount
      /// @param amount amount to check.
      /// @return Boolean : Don't use balance (true)
      function _isLocked(address account, uint256 amount) internal view returns (bool)
  {
          return _lockAmounts[account] != 0 &&
              _lockTimes[account] > block.timestamp &&
              (
                  balanceOf(account) <= _lockAmounts[account] ||
                  balanceOf(account).sub(_lockAmounts[account]) < amount
              );
      }

      /// Transfer token  (run anyone)
      /// @param recipient Token trasfer destination acount.
      /// @param amount Token transfer amount.
      /// @return Boolean
      function transfer(address recipient, uint256 amount) enabled public returns
  (bool) {
          require( !_isLocked( msg.sender, amount ) , "ERC20: Locked balance");
          return super.transfer(recipient, amount);
      }

      /// Transfer token  (run anyone)
      /// @param sender Token trasfer source acount.
      /// @param recipient Token transfer destination acount.
      /// @param amount Token transfer amount.
      /// @return Boolean
      function transferFrom(address sender, address recipient, uint256 amount) enabled
  public returns (bool) {
          require( !_isLocked( sender, amount ) , "ERC20: Locked balance");
          return super.transferFrom(sender, recipient, amount);
      }

      /// Decrease token balance (run only owner)
      /// @param value Amount to decrease.
      function burn(uint256 value) onlyOwner public {
          require( !_isLocked( msg.sender, value ) , "ERC20: Locked balance");
          super.burn(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.