Smart Contract Security Audit Report

The SlowMist Security Team received the ARPA team's application for smart contract security audit of the ARPA Token on October 25, 2019. The following are the details and results of this smart contract security audit:

Token name :

ARPA

The Contract address :

0xBA50933C268F567BDC86E1aC131BE072C6B0b71a

Link address :

https://etherscan.io/address/0xBA50933C268F567BDC86E1aC131BE072C6B0b71a

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 : 0X001910280001

Audit Date : October 28, 2019

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 but there is a cap. The contract owner can use the mint function to mint tokens, allowing users to burn their own tokens, which affects the total amount of tokens. The burned tokens can be re-coin by the mint 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 2019-04-22
*/

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

pragma solidity 0.5.4;

/**
```

```
 * @title ERC20 interface
 * @dev see https://eips.ethereum.org/EIPS/eip-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);
}

/**
 * @title Standard ERC20 token
 *
 * @dev Implementation of the basic standard token.
 * https://eips.ethereum.org/EIPS/eip-20
 * 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)) internal _allowed;

    uint256 internal _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 A 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 to 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);

    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, uint256 value) public 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 uint256 the amount of tokens to be transferred
  */
function transferFrom(address from, address to, uint256 value) public returns (bool) {
    _transfer(from, to, value);
    _approve(from, msg.sender, _allowed[from][msg.sender].sub(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, uint256 addedValue) public 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, uint256 subtractedValue) public 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, uint256 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 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)); //SlowMist// This kind of check is very good, avoiding user
mistake leading to the loss of token during mint
```

```
        _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 {
```

//SlowMist// This kind of check is very good, avoiding user mistake leading to burn errors

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

        _totalSupply = _totalSupply.sub(value);
        _balances[account] = _balances[account].sub(value);
        emit Transfer(account, 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, uint256 value) internal {
```

//SlowMist// This kind of check is very good, avoiding user mistake leading to approve errors

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

        _allowed[owner][spender] = value;
        emit Approval(owner, spender, 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 {
        _burn(account, value);
        _approve(account, msg.sender, _allowed[account][msg.sender].sub(value));
    }
}

/**
 * @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 a
     * specified account.
     * @param initalOwner The address of the inital owner.
     */
    constructor (address initalOwner) internal {
        _owner = initalOwner;
        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 relinquish control of the contract.
 * It will not be possible to call the functions with the `onlyOwner`
 * modifier anymore.
 * @notice 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 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 {
```

```
        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;
    }
}

/**
 * @title ARPAToken
 * @dev ARPA is an ownable, mintable, pausable and burnable ERC20 token
 */
contract ARPAToken is ERC20, Ownable {
    using SafeMath for uint;

    string public constant name = "ARPA Token";
    uint8 public constant decimals = 18;
    string public constant symbol = "ARPA";
    uint public constant maxSupply = 2 * 10**9 * 10**uint(decimals); // 2 billion
    uint public constant initalSupply = 14 * 10**8 * 10**uint(decimals); // 1.4 billion

    bool public paused; // True when circulation is paused.

    mapping (address => bool) public minter;

    /**
     * @dev Throws if called by any account that is not a minter.
     */
    modifier onlyMinter() {
        require(minter[msg.sender]);
        _;
    }

    /**
     * @dev Throws if called when the circulation is paused.
     */
    modifier whenNotPaused() {
        require(paused == false);
        _;
    }
```

```
/**
  * @dev The ARPAToken constructor sets the original manager of the contract to the a
  * specified account, and send all the inital supply to it.
  * @param manager The address of the first manager of this contract.
  */
constructor(address manager) public Ownable(manager) {
      _balances[manager] = initalSupply;
      _totalSupply = initalSupply;
}

/**
  * @dev Add an address to the minter list.
  * @param minterAddress The address to be added as a minter.
  */
function addMinter(address minterAddress) public onlyOwner {
      minter[minterAddress] = true;
}

/**
  * @dev Remove an address from the minter list.
  * @param minterAddress The address to be removed from minters.
  */
function removeMinter(address minterAddress) public onlyOwner {
      minter[minterAddress] = false;
}

/**
  * @dev Function to mint tokens by a minter
  * @param to The address that will receive the minted tokens.
  * @param value The amount of tokens to mint.
  * @return A boolean that indicates if the operation was successful.
  * @notice 30% of the ARPA token are issued by the mining process.
  */
```

//SlowMist// Mint function

```
function mint(address to, uint value) public onlyMinter returns (bool) {
      require(_totalSupply.add(value) <= maxSupply);
      _mint(to, value);
      return true;
}
```

```
    /**
     * @dev Function to pause all the circulation in the case of emergency.
     */

    //SlowMist// Suspending all transactions upon major abnormalities is a recommended approach
    function pause() public onlyOwner {
        paused = true;
    }

    /**
     * @dev Function to recover all the circulation from emergency.
     */
    function unpause() public onlyOwner {
        paused = false;
    }

    /**
     * @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 account whose tokens will be burned.
     * @param value uint256 The amount of token to be burned.
     */
    function burnFrom(address from, uint256 value) public {
        _burnFrom(from, value);
    }


    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);
    }

    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) {
        return super.increaseAllowance(spender, addedValue);
    }

    function decreaseAllowance(address spender, uint subtractedValue) public whenNotPaused returns (bool) {
        return super.decreaseAllowance(spender, subtractedValue);
    }

}

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

//SlowMist// OpenZeppelin's SafeMath security module is used, which is a commendable 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;
    }
}
```

慢雾科技
slow mist

Official Website
www.slowmist.com

E-mail
team@slowmist.com

Twitter
@SlowMist_Team

WeChat Official Account