密级 ★ | 项目组公开

[AOA-DMSJ-20180719]

Smart Contract Audit Report

AOA

知道创宇

Document information

文档名称	文档版本号	文档编号	保密级别
AOA Smart Contract Audit Report	V2.0	【AOA-DMSJ-20180719】	Project team

Copyright Notice

目录

1. Overview

The effective test time of this report is from July 13, 2018 to July 19, 2018. During this period, the security and standardization of the AOA smart contract code will be audited and used as the statistical basis for the report.

The overflow problem in 3.2 and the transaction sequence problem in 3.6 in the document, due to the difficulty of use, are comprehensively assessed as passed. (For detailed description, see 3.2 and 3.6)

Regarding the issue of fake recharge in Document 3.9, the contract has this risk point. Considering all the perfect inspection mechanisms of the transaction, the comprehensive assessment is passed. (For detailed description, see 3.9)

The results of this smart contract security audit: passed

Since this testing process is carried out in a non-production environment, all codes are up-to-date backups, the testing process is communicated with the relevant interface person, and relevant testing operations are carried out under the controllable operational risk to avoid production during the testing process Operational risk, code security risk.

Target information of this test:

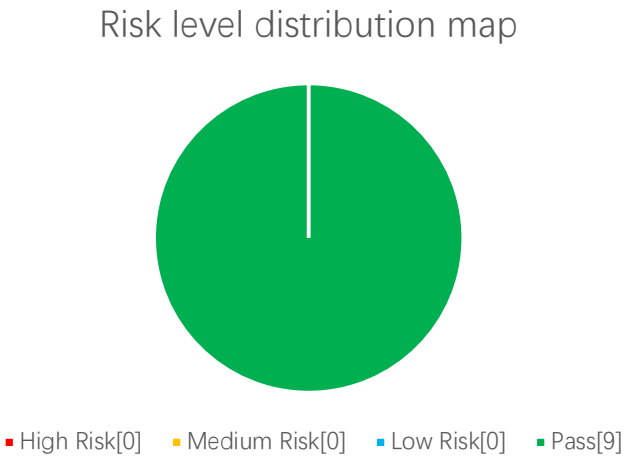
Module name	
Token name	AOA
Code type	Token code
Contract address	0x9ab165d795019b6d8b3e971dda91071421305e5a
link address	https://etherscan.io/address/0x9ab165d795019b6d8b3e971dda91071421305e5a#code
Code language	solidity

2. Code vulnerability analysis

2.1. Vulnerability level distribution

This vulnerability risk is counted by level:

Vulnerability risk level statistics table			
High Risk	Medium Risk	Low Risk	Pass
0	0	0	9



2.2. Summary of audit results

Test items	Test content	description
Smart contract	Reentry attack detection	Check the safe use of call.value() function
	Numerical overflow detection	Check the safety of add and sub functions
	Access control defect detection	Check access control for each operation
	Calls with unverified return values	Check the transfer method to see if the return value is verified
	Wrong use of random number detection	Check whether there is a unified content filter
	Transaction order dependency detection	Check transaction order dependency
	Denial of service attack detection	Check whether there are resource abuse problems when the code uses resources
	Logical design defect detection	Check the security issues related to business design in the smart contract code
	Fake recharge vulnerability detection	Check whether there is a false recharge vulnerability in the smart contract code

3. Analysis of code audit results

3.1. Reentrant attack detection

Reentrance vulnerability is the most famous Ethereum smart contract vulnerability, which once led to the fork of Ethereum (The DAO hack).

The call.value() function in Solidity consumes all the gas it receives when it is used to send Ether. When the call.value() function to send Ether occurs before the actual reduction of the sender' s account balance, There is a risk of reentry attacks.

Test result: After detection, there is no related call external contract call in the smart contract code.

Safety advice: None.

3.2. Numerical overflow detection

The arithmetic problems in smart contracts refer to integer overflow and integer underflow.

Solidity can handle up to 256-bit numbers (2^256-1). If the maximum number increases by 1, it will overflow to 0. Similarly, when the number is an unsigned type, 0 minus 1 will underflow to get the maximum digital value.

Integer overflow and underflow are not a new type of vulnerability, but they are especially dangerous in smart contracts. Overflow conditions can lead to incorrect results, especially if the possibility is not expected, which may affect the

reliability and safety of the program.

Test result: After testing, SafeMath is not used in the transfer function and transferFrom function of the AOA contract code, and there is no overflow check before the addition. There is a risk of value overflow, as shown in the figure:


```
41  function transfer(address _to, uint256 _value) public returns (bool success) {
42      if (balances[msg.sender] >= _value && _value > 0) {
43          balances[msg.sender] -= _value;
44          balances[_to] += _value;
45          Transfer(msg.sender, _to, _value);
46          return true;
47      } else { return false; }
48  }
49
50  function transferFrom(address _from, address _to, uint256 _value) public returns (bool success) {
51      //same as above. Replace this line with the following if you want to protect against wrapping uints.
52      //if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value && balances[_to] + _value > balances[_to]) {
53      if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value && _value > 0) {
54          balances[_to] += _value;
55          balances[_from] -= _value;
56          allowed[_from][msg.sender] -= _value;
57          Transfer(_from, _to, _value);
58          return true;
59      } else { return false; }
60  }
```

However, because there is no way to issue additional tokens, overflow will not occur under normal circumstances. Therefore, it was assessed as passed.

Security advice: add overflow check code at the beginning of the function, such as: assert(balances[_to] + value > balances[_to]);

3.3. Access control detection

Access control defects are security risks that may exist in all programs. Smart contracts also have similar problems. The well-known Parity Wallet smart contract has been affected by this problem.

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

3.4. Return value call verification

This problem mostly occurs in smart contracts related to currency transfer, so it is also called silent failed delivery or unchecked delivery.

There are transfer(), send(), call.value() and other currency transfer methods in Solidity, which can all be used to send Ether to an address. The difference is: when the transfer fails, it will be thrown and the state will be rolled back; Only 2300gas will be passed for calling to prevent reentry attacks; false will be returned when send fails; only 2300gas will be passed for calling to prevent reentry attacks; false will be returned when call.value fails to be sent; all available gas will be passed for calling (can be Limit by passing in the gas_value parameter), which cannot effectively prevent reentry attacks.

If the return value of the above send and call.value transfer functions is not checked in the code, the contract will continue to execute the following code, which may lead to unexpected results due to Ether sending failure.

Test result: After testing, there are no related vulnerabilities in the smart contract code.

Safety advice: None.

3.5. Incorrect use of random numbers

Smart contracts may need to use random numbers. Although the functions and variables provided by Solidity can access obviously unpredictable values, such

as block.number and block.timestamp, they are usually either more public than they appear or are affected by miners, namely These random numbers are predictable to a certain extent, so malicious users can usually copy it and rely on its unpredictability to attack the function.

Test result: After testing, the problem does not exist in the smart contract code.

Safety advice: None.

3.6. Transaction order dependency

Since miners always obtain gas fees through codes that represent externally owned addresses (EOA), users can specify higher fees for faster transactions. Since the Ethereum blockchain is public, everyone can see the content of other people's pending transactions. This means that if a user submits a valuable solution, a malicious user can steal the solution and copy its transaction at a higher fee to preempt the original solution.

Test result: After detection, the approve function in the AOA token contract has a transaction sequence dependent attack risk. The code is as follows:

```
66    function approve(address _spender, uint256 _value) public returns (bool success) {
67        allowed[msg.sender][_spender] = _value;
68        Approval(msg.sender, _spender, _value);
69        return true;
70    }
```

The possible security risks are described as follows:

1. By calling the approve function, user A allows user B to transfer money on his behalf to N (N>0);

2. After a period of time, user A decides to change N to M (M>0), so call the approve function again;

3. User B quickly calls the transferFrom function to transfer N number of tokens before the second call is processed by the miner;

After user A' s second call to approve is successful, user B can obtain M' s transfer quota again, that is, user B obtains N+M' s transfer quota through the transaction sequence attack.

Due to the difficulty of its use, the comprehensive assessment is passed.

Safety advice:

1. Front-end restriction, when user A changes the quota from N to M, he can first change from N to 0, and then from 0 to M.

2. Add the following code at the beginning of the approve function:：

```
require((_value == 0) || (allowed[msg.sender][_spender] == 0));
```

3.7. Denial of service attack

In the world of Ethereum, denial of service is fatal, and a smart contract that has suffered this type of attack may never be able to return to its normal working state. There may be many reasons for the denial of service of a smart contract, including malicious behavior as a transaction receiver, artificially increasing the gas required for computing functions to cause gas exhaustion, abusing access control to access private components of the smart contract, using confusion and negligence, etc. Wait.

Test result: After testing, there are no related vulnerabilities in the smart contract code.

Safety advice: None.

3.8. Logical design flaws

Detect security issues related to business design in smart contract code.

Test result: After testing, there are no related vulnerabilities in the smart contract code.

Safety advice: none.

3.9. Fake recharge vulnerability

The transfer function of the token contract uses the if judgment method to check the balance of the transfer initiator (msg.sender). When balances[msg.sender] <value, enter the else logic part and return false, and finally no exception is thrown. We think that only if/else this kind of gentle judgment method is an imprecise coding method in sensitive function scenarios such as transfer.

Test result: After testing, if/else is used in the transfer and transferFrom functions of the AOA contract to check the balance, there is a false recharge vulnerability, as shown in the figure:

```
41    function transfer(address _to, uint256 _value) public returns (bool success) {
42        if (balances[msg.sender] >= _value && _value > 0) {
43            balances[msg.sender] -= _value;
44            balances[_to] += _value;
45            Transfer(msg.sender, _to, _value);
46            return true;
47        } else { return false; }
48    }
49
50    function transferFrom(address _from, address _to, uint256 _value) public returns (bool success) {
51        //same as above. Replace this line with the following if you want to protect against wrapping uints.
52        //if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value && balances[_to] + _value > balances[_to]) {
53        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value && _value > 0) {
54            balances[_to] += _value;
55            balances[_from] -= _value;
56            allowed[_from][msg.sender] -= _value;
57            Transfer(_from, _to, _value);
58            return true;
59        } else { return false; }
60    }
61
```

Attackers can initiate recharge operations to centralized exchanges, wallets and other service platforms. If the exchange only judges that the token recharge is successful if TxReceipt Status is not successful, the attacker can profit from this.

But to recharge successfully, the following conditions must be met:

1. The receipt status is successful;

2. There is transfer in the log;

3. Check whether the balances of the two addresses are consistent;

At the same time, if/else is used in the transfer function of the contract to determine the balance. When the balance is insufficient, the function returns false. At this time, the TxReceipt Status is success. If the exchange only determines the transfer success based on this value, it will form a "fake recharge". The exchange caused losses. If it is confirmed that the exchange has used a complete transfer verification method, there is no such risk.

Therefore, the overall rating is low risk.

Security advice: Use require/assert in the transfer and transferFrom functions to determine the balance, do not use if/else.

4. Appendix A: Contract Code

Source of the test code:

https://etherscan.io/address/0x9ab165d795019b6d8b3e971dda91071421305e5a#code

```
pragma solidity ^0.4.19;

contract Token {
    /// total amount of tokens
    uint256 public totalSupply;

    /// @param _owner The address from which the balance will be retrieved
    /// @return The balance
    function balanceOf(address _owner) public constant returns (uint256 balance);

    /// @notice send `_value` token to `_to` from `msg.sender`
    /// @param _to The address of the recipient
    /// @param _value The amount of token to be transferred
    /// @return Whether the transfer was successful or not
    function transfer(address _to, uint256 _value)public  returns (bool success);

    /// @notice send `_value` token to `_to` from `_from` on the condition it is
approved by `_from`
    /// @param _from The address of the sender
    /// @param _to The address of the recipient
    /// @param _value The amount of token to be transferred
    /// @return Whether the transfer was successful or not
    function transferFrom(address _from, address _to, uint256 _value) public returns
(bool success);

    /// @notice `msg.sender` approves `_spender` to spend `_value` tokens
    /// @param _spender The address of the account able to transfer the tokens
    /// @param _value The amount of tokens to be approved for transfer
    /// @return Whether the approval was successful or not
    function approve(address _spender, uint256 _value) public returns (bool success);

    /// @param _owner The address of the account owning tokens
    /// @param _spender The address of the account able to transfer the tokens
    /// @return Amount of remaining tokens allowed to spent
    function allowance(address _owner, address _spender) public constant returns
(uint256 remaining);

    event Transfer(address indexed _from, address indexed _to, uint256 _value);
    event Approval(address indexed _owner, address indexed _spender, uint256 _value);
}

contract StandardToken is Token {
    //knownsec//存在假充值漏洞，详见3.9章节
    function transfer(address _to, uint256 _value) public returns (bool success) {
        if (balances[msg.sender] >= _value && _value > 0) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;  //knownsec//存在数值溢出风险
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint256 _value) public returns
(bool success) {
        //same as above. Replace this line with the following if you want to protect
against wrapping uints.
        //if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
balances[_to] + _value > balances[_to]) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
_value > 0) {
            balances[_to] += _value;  //knownsec//存在数值溢出风险
            balances[_from] -= _value;  //knownsec//建议先减后加
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
```

```
            return true;
        } else { return false; }
    }

    function balanceOf(address _owner) public constant returns (uint256 balance) {
        return balances[_owner];
    }
    //knownsec//approve函数存在事务顺序依赖风险，详见3.6章节
    function approve(address _spender, uint256 _value) public returns (bool success) {
        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
        return true;
    }

    function allowance(address _owner, address _spender) public constant returns
(uint256 remaining) {
      return allowed[_owner][_spender];
    }

    mapping (address => uint256) balances;
    mapping (address => mapping (address => uint256)) allowed;
}

contract ArthurStandardToken is StandardToken {
    string public name;
    uint8 public decimals;
    string public symbol;

    function ArthurStandardToken(
        uint256 _initialAmount,
        string _tokenName,
        uint8 _decimalUnits,
        string _tokenSymbol
        ) public {
        balances[msg.sender] = _initialAmount;              // Give the creator all
initial tokens
        totalSupply = _initialAmount;                       // Update total supply
        name = _tokenName;                                  // Set the name for display
purposes
        decimals = _decimalUnits;                           // Amount of decimals for
display purposes
        symbol = _tokenSymbol;                              // Set the symbol for display
purposes
    }

    function approveAndCall(address _spender, uint256 _value, bytes _extraData) public
returns (bool success) {
        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);


if(!_spender.call(bytes4(bytes32(keccak256("receiveApproval(address,uint256,address,by
tes)"))), msg.sender, _value, this, _extraData)) { return false; }
        return true;
    }
}
```

5. Appendix B: Vulnerability Risk Rating Standard

Smart contract vulnerability rating standards	
Vulnerability rating	Vulnerability rating description
High-risk vulnerabilities	Vulnerabilities that can directly cause the loss of token contracts or user funds, such as: value overflow vulnerability that can cause the value of tokens to zero, false recharge vulnerability that can cause exchanges to lose tokens, and can cause contract accounts to lose ETH or tokens. Access loopholes, etc.;
Mid-risk vulnerability	Vulnerabilities that can cause the loss of ownership of token contracts, such as: access control defects of key functions, call injection leading to key function access control bypass, etc.;
Low-risk vulnerabilities	Vulnerabilities that can cause token contracts to not work properly, such as: denial of service vulnerability caused by sending ETH to malicious addresses, and denial of service vulnerability caused by exhaustion of gas.

6. Appendix C: Introduction to Vulnerability Testing Tools

6.1. Manticore

Manticore is a symbolic execution tool for analyzing binary files and smart contracts. Manticore includes a symbolic Ethereum Virtual Machine (EVM), an EVM disassembler/assembler, and a convenient interface for automatic compilation and analysis of Solidity. It also integrates Ethersplay, the Bit of Traits of Bits visual disassembler for EVM bytecode, for visual analysis. Like binary files, Manticore provides a simple command-line interface and a Python API for analyzing EVM bytecode.

6.2. Oyente

Oyente is a smart contract analysis tool. Oyente can be used to detect common bugs in smart contracts, such as reentrancy, transaction ordering dependencies, and so on. What' s more convenient is that Oyente' s design is modular, so this allows advanced users to implement and insert their own detection logic to check custom attributes in their contracts.

6.3. securify.sh

Securify can verify common security issues of Ethereum smart contracts, such as disorder of transactions and lack of input verification. It analyzes all possible execution paths of the program while fully automated. In addition, Securify also has a specific language for specifying vulnerabilities, which makes Securify can keep an eye on current security and other reliability issues.

6.4. Echidna

Echidna is a Haskell library designed for fuzzing EVM code.

6.5. MAIAN

MAIAN is an automated tool used to find vulnerabilities in Ethereum smart contracts. Maian processes the bytecode of the contract and tries to establish a series of transactions to find and confirm errors.

6.6. ethersplay

ethersplay is an EVM disassembler, which contains relevant analysis tools.

6.7. ida-evm

ida-evm is an IDA processor module for the Ethereum Virtual Machine (EVM).

6.8. Remix-ide

Remix is a browser-based compiler and IDE that allows users to build Ethereum contracts and debug transactions in the Solidity language.

6.9. Knownsec penetration tester special toolkit

Knownsec Penetration Tester's special toolkit is developed, collected and used by Knownsec penetration test engineers. It contains batch automatic testing tools dedicated to testers, self-developed tools, scripts or utilization tools, etc.

知道创宇

[咨询电话] +86(10)400 060 9587
[投诉电话] 13811527185
[邮　　箱] sec@knownsec.com
[网　　址] www.knownsec.com
[地　　址] 北京市朝阳区望京SOHO T3 A座15层

北京知道创宇信息技术有限公司