Degree of secrets ★ | Project team open

Smart contract audit report

LYN

Security status

Security

★ ★ ★ ★ ★

Chief test Officer: *Knownsec blockchain security team*

Version Summary

Content	Date	Version
Editing Document	20201013	V1.0

Report Information

Title	Version	Document Number	Type
LYN contract audit report	V1.0	【LYN-ZNNY-20201013】	Open to project team

Copyright Notice

Table of Contents

1. Introduction

The effective test time of this report is from October 13, 2020 to October 14, 2020. During this period, the security and standardization of **the smart contract code of the LYN** will be audited and used as the statistical basis for the report.

In this audit report, engineers conducted a comprehensive analysis of the common vulnerabilities of smart contracts (Chapter 3). There is a risk of transaction sequence dependence, but its use is more difficult, **the smart contract code of the LYN** is comprehensively assessed as **SAFE**.

Results of this smart contract security audit： SAFE

Since the testing is under non-production environment, all codes are the latest version. In addition, the testing process is communicated with the relevant engineer, and testing operations are carried out under the controllable operational risk to avoid production during the testing process, such as: Operational risk, code security risk.

Target information of the LYN audit:

Target information	
Token name	Lynchpin(LYN)
Contract address	0xB0B1685f55843D03739c7D9b0A230F1B7DcF03D5
Code type	ETH smart contract code、token code
Code language	solidity

Contract documents and hash:

Contract documents	MD5
LynchpinToken.sol	421adf91bcd28800167fe5cc3b7bc41b

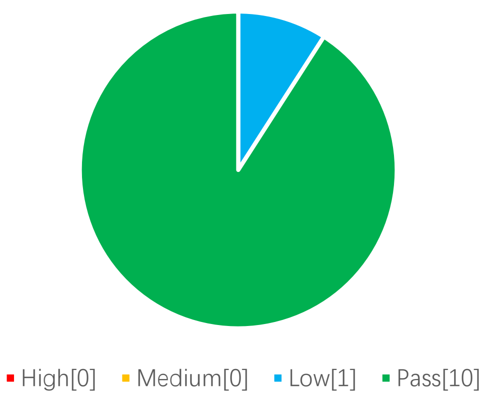
2. Code vulnerability analysis

2.1 Vulnerability Level Distribution

Vulnerability risk statistics by level：

Vulnerability risk level statistics table			
High	Medium	Low	Pass
0	0	1	10

Risk level distribution



2.2 Audit Result

Result of audit			
Audit Target	**Audit**	**Status**	**Audit Description**
Basic code vulnerability detection	**Reentry attack detection**	Pass	After testing, there is no such safety vulnerability.
	Numerical overflow detection	Pass	After testing, there is no such safety vulnerability.
	Access control defect detection	Pass	After testing, there is no such safety vulnerability.
	Calls with unverified return value	Pass	After testing, there is no such safety vulnerability.
	Wrong use of random number detection	Pass	After testing, there is no such safety vulnerability.
	Transaction order dependency detection	Low (Pass)	After testing, there is a risk of transaction sequence dependence in the code, but because it is too difficult to use, it is comprehensively assessed as passed.
	Denial of service attack detection	Pass	After testing, there is no such safety vulnerability.
	Logical design defect detection	Pass	After testing, there is no such safety vulnerability.
	Fake recharge vulnerability detection	Pass	After testing, there is no such safety vulnerability.
	Additional token issuance	Pass	After testing, there is no such safety vulnerability.

	vulnerability detection		
	Frozen account bypass detection	Pass	After testing, there is no such safety vulnerability.

3. Basic code vulnerability detection

3.1. Reentry attack detection 【PASS】

Re-entry 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 is called to send Ether before it actually reduces the balance of the sender's account, There is a risk of reentry attacks.

Audit analysis: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

3.2. Numerical overflow detection 【PASS】

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.

Audit analysis: After testing, the security problem does not exist in the smart contract code.

Recommendation: nothing.

3.3. Access control detection 【PASS】

Access control flaws 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.

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

Recommendation: nothing.

3.4. Calls with unverified return value 【PASS】

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 gas_value parameters), 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.

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

Recommendation： nothing.

3.5. Wrong use of random number detection 【PASS】

Smart contracts may need to use random numbers. Although the functions and variables provided by Solidity can access values that are obviously unpredictable, such as block.number and block.timestamp, they are usually more public than they appear or are affected by miners. 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.

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

Recommendation： nothing.

3.6. Transaction order dependency detection 【LOW】

Since miners always get 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.

Audit result: After testing, the approve function on line 117 of the LynchpinToken.sol contract file has the risk of transaction sequence dependency attack. The code is as follows:

```
function approve(address _spender, uint _value) public returns (bool success)
{
        allowance[msg.sender][_spender] = _value;
        emit Approval(msg.sender, _spender, _value);
        return true;
}
```

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;

4. 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.

Recommendation:

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) || (allowance[msg.sender][_spender] == 0));

3.7. Denial of service attack detection【PASS】

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 the smart contract, including malicious behavior as the transaction recipient, artificially increasing the gas required for computing functions to cause gas exhaustion, abusing access control to access the private component of the smart contract, using confusion and negligence, etc.

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

Recommendation: nothing.

3.8. Logical design defect detection【PASS】

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

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

Recommendation: nothing.

3.9. Fake recharge vulnerability detection【PASS】

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, it enters the else logic part and returns false, and finally no exception is thrown. We believe that only if/else this kind of gentle judgment method is an imprecise coding method in the scene of sensitive functions such as transfer.

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

Recommendation: nothing.

3.10. Additional token issuance vulnerability detection 【PASS】

Check whether there is a function that may increase the total amount of tokens in the token contract after initializing the total amount of tokens.

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

Recommendation: nothing.

3.11. Frozen account bypass detection 【PASS】

Detect whether there is an operation that does not verify the source account, originating account, and target account of the token when transferring tokens in the token contract.

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

Recommendation： nothing.

4. Appendix A: Contract code

Source code:

```
LynchpinToken.sol

/**
 *Submitted for verification at Etherscan.io on 2018-11-12
*/

pragma solidity ^0.4.24;

library SafeMath
{
        function mul(uint256 a, uint256 b) internal pure
        returns (uint256)
        {
                uint256 c = a * b;

                assert(a == 0 || c / a == b);

                return c;
        }

        function div(uint256 a, uint256 b) internal pure
        returns (uint256)
        {
                // assert(b > 0); // Solidity automatically throws 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;
        }

        function sub(uint256 a, uint256 b) internal pure
        returns (uint256)
        {
                assert(b <= a);

                return a - b;
        }

        function add(uint256 a, uint256 b) internal pure
        returns (uint256)
        {
                uint256 c = a + b;

                assert(c >= a);

                return c;
        }
}

interface ERC20
{
        function totalSupply() view external returns (uint _totalSupply);
        function balanceOf(address _owner) view external returns (uint balance);
        function transfer(address _to, uint _value) external returns (bool success);
        function transferFrom(address _from, address _to, uint _value) external returns (bool success);
        function approve(address _spender, uint _value) external returns (bool success);
        function allowance(address _owner, address _spender) view external returns (uint remaining);

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

contract LynchpinToken is ERC20
{
        using SafeMath for uint256;

        string          public name             = "Lynchpin";
        string          public symbol           = "LYN";
        uint8           public decimals         = 18;
        uint   public totalSupply = 5000000 * (10 ** uint(decimals));/
        address public owner            = 0xAc983022185b95eF2B2C7219143483BD0C65Ecda;

        mapping (address => uint) public balanceOf;
        mapping (address => mapping (address => uint)) public allowance;

        constructor() public
        {
                balanceOf[owner] = totalSupply;
```

```
    }

    function totalSupply() view external returns (uint _totalSupply)
    {
        return totalSupply;
    }

    function balanceOf(address _owner) view external returns (uint balance)
    {
        return balanceOf[_owner];
    }

    function allowance(address _owner, address _spender) view external returns (uint remaining)
    {
        return allowance[_owner][_spender];
    }
    function _transfer(address _from, address _to, uint _value) internal
    {
        require(_to != 0x0);

        uint previousBalances = balanceOf[_from].add(balanceOf[_to]);
        balanceOf[_from] = balanceOf[_from].sub(_value);
        balanceOf[_to] = balanceOf[_to].add(_value);

        emit Transfer(_from, _to, _value);
        assert(balanceOf[_from].add(balanceOf[_to]) == previousBalances);
    }

    function transfer(address _to, uint _value) public returns (bool success)
    {
        _transfer(msg.sender, _to, _value);
        return true;
    }

    function transferFrom(address _from, address _to, uint _value) public returns (bool success)
    {
        allowance[_from][msg.sender] = allowance[_from][msg.sender].sub(_value);
        _transfer(_from, _to, _value);
        return true;
    }

    function approve(address _spender, uint _value) public returns (bool success)
    {

        allowance[msg.sender][_spender] = _value;
        emit Approval(msg.sender, _spender, _value);
        return true;
    }

    // disallow incoming ether to this contract
    function () public
    {
        revert();
    }
}
```

5. Appendix B：Vulnerability rating standard

Smart contract vulnerability rating standards

Level	Level Description
High	Vulnerabilities that can directly cause the loss of token contracts or user funds, such as: value overflow loopholes that can cause the value of tokens to zero, fake recharge loopholes that can cause exchanges to lose tokens, and can cause contract accounts to lose ETH or tokens. Access loopholes, etc.; Vulnerabilities that can cause loss of ownership of token contracts, such as: access control defects of key functions, call injection leading to bypassing of access control of key functions, etc.; Vulnerabilities that can cause the token contract 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.
Medium	High-risk vulnerabilities that require specific addresses to trigger, such as value overflow vulnerabilities that can be triggered by token contract owners; access control defects for non-critical functions, and logical design defects that cannot cause direct capital losses, etc.
Low	Vulnerabilities that are difficult to be triggered, vulnerabilities with limited damage after triggering, such as value overflow vulnerabilities that require a large amount of ETH or tokens to trigger, vulnerabilities where attackers cannot directly profit after triggering value overflow, and the transaction sequence triggered by specifying high gas depends on the risk Wait.

6. Appendix C：Introduction to auditing 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, Bit of Traits of Bits visual disassembler for EVM bytecode, used for visual analysis. Like binary files, Manticore provides a simple command line interface and a Python for analyzing EVM bytecode API.

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 sequencing dependencies, etc. More convenient, Oyente's design is modular, so this allows advanced users to implement and Insert their own detection logic to check the custom attributes in their contract.

6.3 securify.sh

Securify can verify common security issues of Ethereum smart contracts, such as disordered 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 at any time.

6.4 Echidna

Echidna is a Haskell library designed for fuzzing EVM code.

6.5 MAIAN

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

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

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

6.9 Knownsec Penetration Tester Special Toolkit

Pen-Tester tools collection is created by KnownSec team. It contains plenty of

Pen-Testing tools such as automatic testing tool, scripting tool, Self-developed tools etc.

KNOWNSEC

Beijing KnownSec Information Technology Co., Ltd.

Advisory telephone	+86(10)400 060 9587
E-mail	sec@knownsec.com
Website	www.knownsec.com
Address	wangjing soho T2-B2509,Chaoyang District, Beijing