LUNARAY
BLOCKCHAINSECURITY

SMART CONTRACT SECURITY AUDIT REPORT

For AIOTNetWorkToken

22 March 2022

lunaray.co

Table of Contents

1. Overview

On Mar 16, 2022, the security team of Lunaray Technology received the security audit request of the **AIOTNetWorkToken project**. The team completed the audit of the **AIOTNetWorkToken smart contract** on Mar 22, 2022. During the audit process, the security audit experts of Lunaray Technology and the AIOTNetWorkToken project interface Personnel communicate and maintain symmetry of information, conduct security audits under controllable operational risks, and avoid risks to project generation and operations during the testing process.

Through communicat and feedback with AIOTNetWorkToken project party, it is confirmed that the loopholes and risks found in the audit process have been repaired or within the acceptable range. The result of this AIOTNetWorkToken smart contract security audit: **Passed**

Audit Report HASH：

1615B63B241A06B5F2200D5833E27E5D9FF4ABE5CA3951E19C205E95DD025349

2. Background

2.1 Project Description

Project name	AIOTNetWorkToken
Contract type	Token, DeFi
Code language	Solidity
Total issuance	65,000,000 AIOT
Public chain	Binance Smart Chain
Contract file	AIOTNetWorkToken.sol

2.2 Audit Range

AIOTNetWorkToken Officially Provides Binance Smart Chain Contract Address：

Name	Address
AIOTNetWorkToken	0xd0A3E36eFf78333A421950A8008231Da656E7689

2.3 Findings Summary

Severity	Found	Resolved	Acknowledged
High	0	0	0
Medium	0	0	0
Low	0	0	0
Info	1	0	1

3. Project contract details

3.1 Contract Overview

Ownable Contract

owner address-related settings, the existence of the owner address will be thrown to the empty address, there is a lock address function, the administrator will control the time to temporarily lock the address in control, when the address after the time, you can continue to get the owner address administrator privileges, the current administrator owner address has been transferred to the empty address.

AIOTNetWorkToken Contract

To distribute Token funds, the main function is the transfer and authorization function, when the contract itself assets reach fixed conditions, the transfer method can be exchanged for funds and add liquidity to obtain LP Token This part of the function is mainly to call the UniswapV2 contract interface to achieve, part of the judgment value can also be set, because the administrator has been transferred to an empty address, so there is no possibility of modifying the amount.

3.2 Contract details

AIOTNetWorkToken

Name	Parameter	Attributes
setWrap	IWrap _wrap	onlyOwner
setNumTokensSellTo AddToLiquidity	uint256 _num	onlyOwner
name	none	public
symbol	none	public
decimals	none	public
totalSupply	none	public
balanceOf	address account	public
transfer	address recipient uint256 amount	public
allowance	address owner address spender	public
approve	address spender uint256 amount	public
transferFrom	address sender address recipient uint256 amount	public
increaseAllowance	address spender uint256 addedValue	public
decreaseAllowance	address spender uint256 subtractedValue	public
isExcludedFromReward	address account	public
totalFees	none	public
tokenFromReflection	uint256 rAmount	public
excludeFromReward	address account	onlyOwner
includeInReward	address account	onlyOwner
excludeFromFee	address account	onlyOwner
includeInFee	address account	onlyOwner

Name	Parameter	Attributes
setSwapAndLiquifyEnabled	bool _enabled	onlyOwner
_reflectFee	uint256 rFee uint256 tFee	private
_getValues	uint256 tAmount	private
_getTValues	uint256 tAmount	private
_getRValues	uint256 tAmount uint256 tFee uint256 tLiquidity uint256 tDestory uint256 currentRate	private
_getRate	none	private
_getCurrentSupply	none	private
_takeLiquidity	uint256 tLiquidity	private
_takeDestory	uint256 tDestory	private
calculateTaxFee	uint256 _amount	private
calculateLiquidityFee	uint256 _amount	private
calculateDestroyFee	uint256 _amount	private
removeAllFee	none	private
restoreAllFee	none	private
isExcludedFromFee	address account	public
_approve	address owner address spender uint256 amount	private
_transfer	address from address to uint256 amount	private
swapAndLiquify	uint256 contractTokenBalance	private
swapTokensForUsdt	uint256 tokenAmount	private
addLiquidityUsdt	uint256 tokenAmount uint256 usdtAmount	private
_tokenTransfer	address sender address recipient uint256 amount bool takeFee	private
_transferBothExcluded	address sender address recipient uint256 tAmount	private

Name	Parameter	Attributes
_transferStandard	address sender address recipient uint256 tAmount	private
_transferToExcluded	address sender address recipient uint256 tAmount	private
_transferFromExcluded	address sender address recipient uint256 tAmount	private

Ownable

Name	Parameter	Attributes
renounceOwnership	none	onlyOwner
transferOwnership	address newOwner	onlyOwner
geUnlockTime	none	public
lock	uint256 time	onlyOwner
unlock	none	public

4. Audit details

4.1 Risk distribution

Name	Risk level	Repair status
transferFrom funds judgment	info	Acknowledged
transfer method	No	normal
Numerical accuracy	No	normal
Default visibility	No	normal
tx.origin authentication	No	normal
Wrong constructor	No	normal
Unverified return value	No	normal
Insecure random number	No	normal
Timestamp dependent	No	normal
Transaction order dependence	No	normal
Delegatecall	No	normal
Call	No	normal
Denial of service	No	normal
Logical design flaws	No	normal
Fake recharge vulnerability	No	normal
Short address attack	No	normal
Uninitialized storage pointer	No	normal
Frozen account bypass	No	normal
Uninitialized	No	normal

Reentry attack	No	normal
Integer Overflow	No	normal

4.2 Risk audit details

4.2.1 transferFrom funds judgment

- **Risk description**

When the transferFrom method is used to transfer funds, it does not determine whether the authorized funds at the address are greater than the amount of funds passed in, which may cause the method to transfer funds by calling the _transfer method first and then calling the _approve method later with an error, causing the caller's Gas to be consumed.

```
    function transferFrom(address sender, address recipient, uint256 am
ount) public override returns (bool) {
        _transfer(sender, recipient, amount);
        _approve(sender, _msgSender(), _allowances[sender][_msgSender
()].sub(amount, "ERC20: transfer amount exceeds allowance"));
        return true;
    }
```

- **Safety advice**

  It is suggested that the transferFrom method should include a determination of the size of the authorized funds.

- **Repair Status**

AIOTNetWorkToken is officially confirmed.

4.2.2 transfer method

- **Risk description**

The transfer transfer method calls _transfer.

```
    function transfer(address recipient, uint256 amount) public overrid
e returns (bool) {
        _transfer(_msgSender(), recipient, amount);
        return true;
    }
```

Pass in the parameters _msgSender(), recipient, amount

```
    function _transfer(address from, address to, uint256 amount) privat
e {
        require(from != address(0), "ERC20: transfer from the zero addr
ess");
        require(to != address(0), "ERC20: transfer to the zero address
");
        require(amount > 0, "Transfer amount must be greater than zero
");
        uint256 contractTokenBalance = balanceOf(address(this));

        bool overMinTokenBalance = contractTokenBalance >= numTokensSel
lToAddToLiquidity;

        if (overMinTokenBalance &&! inSwapAndLiquify &&
            from != uniswapV2Pair && swapAndLiquifyEnabled
        ) {
            contractTokenBalance = numTokensSellToAddToLiquidity;
            swapAndLiquify(contractTokenBalance);
        }

        bool takeFee = false;

        if (from == uniswapV2Pair || to == uniswapV2Pair) {
            takeFee = true;
        }

        if (_isExcludedFromFee[from] || _isExcludedFromFee[to]) {
            takeFee = false;
        }
        _tokenTransfer(from, to, amount, takeFee);
    }
```

First determine that the two addresses passed in are not zero addresses, and second determine that the transfer amount must be greater than zero.

The contract balance is then assigned to the contractTokenBalance variable and it is judged whether this variable is larger than the numTokensSellToAddToLiquidity variable.

After that, if other conditions such as overMinTokenBalance is true are met, the numTokensSellToAddToLiquidity variable is assigned to contractTokenBalance and passed into the swapAndLiquify method to do logical operations.

```
    function swapAndLiquify(uint256 contractTokenBalance) private lockT
heSwap {
        uint256 half = contractTokenBalance.div(2);
        uint256 otherHalf = contractTokenBalance.sub(half, "sub half");

        uint256 initialBalance = usdt.balanceOf(address(this));

        swapTokensForUsdt(half);

        uint256 newBalance = usdt.balanceOf(address(this)).sub(initialB
alance);

        addLiquidityUsdt(otherHalf, newBalance);

        emit SwapAndLiquify(half, newBalance, otherHalf);
    }
```

In the swapAndLiquify method, first half of the contractTokenBalance variable is assigned to the half variable, then the remaining half is assigned to the otherHalf variable, then the USDT balance in the contract is assigned to the initialBalance variable, then the half variable is passed to the swapTokensForUsdt method for logical operations.

```
    function swapTokensForUsdt(uint256 tokenAmount) private {
        address[] memory path = new address[](2);
        path[0] = address(this);
        path[1] = address(usdt);

        _approve(address(this), address(uniswapV2Router), tokenAmount);

        // make the swap
        uniswapV2Router.swapExactTokensForTokensSupportingFeeOnTransfer
Tokens(
            tokenAmount,
            0, // accept any amount of ETH
            path,
```

```
            address(wrap),
            block.timestamp
        );
        wrap.withdraw();
    }
```

The swapTokensForUsdt method will get the swap path for this time, then transfer the exchanged funds to the wrap contract, back to the swapAndLiquify method, where the USDT funds of the current contract address minus the USDT funds of the previous contract address will be assigned to the newBalance variable, which is the USDT funds. Then the otherHalf and newBalance variables are passed into the addLiquidityUsdt method for logical operation.

```
    function addLiquidityUsdt(uint256 tokenAmount, uint256 usdtAmount)
private {
        _approve(address(this), address(uniswapV2Router), tokenAmount);
        usdt.approve(address(uniswapV2Router), usdtAmount);

        uniswapV2Router.addLiquidity(
            address(this),
            address(usdt),
            tokenAmount,
            usdtAmount,
            0,
            0,
            _lpAddress,
            block.timestamp
        );
    }
```

Ultimately, the LP Token is obtained by adding liquidity to the contract address and USDT funds.

- **Audit results : passed**

4.2.3 Floating Point and Numeric Precision

- **Risk description**

In Solidity, the floating-point type is not supported, and the fixed-length floating-point type is not fully supported. The result of the division operation will be rounded off, and if there is a decimal number, the part after the decimal point will be discarded and only the integer part will be taken, for example, dividing 5 pass 2 directly will result in 2. If the result of the operation is less than 1 in the token operation, for example, 4.9 tokens will be approximately equal to 4, bringing a certain degree of The tokens are not only the tokens of the same size, but also the tokens of the same size. Due to the economic properties of tokens, the loss of precision is equivalent to the loss of assets, so this is a cumulative problem in tokens that are frequently traded.

- **Audit results: passed**

4.2.4 Default Visibility

- **Risk description**

In Solidity, the visibility of contract functions is public pass default. therefore, functions that do not specify any visibility can be called externally pass the user. This can lead to serious vulnerabilities when developers incorrectly ignore visibility specifiers for functions that should be private, or visibility specifiers that can only be called from within the contract itself. One of the first hacks on Parity's multi-signature wallet was the failure to set the visibility of a function, which defaults to public, leading to the theft of a large amount of money.

- **Audit results : passed**

4.2.5 tx.origin authentication

- **Risk description**

tx.origin is a global variable in Solidity that traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in a smart contract can make the contract vulnerable to phishing-like attacks.

- **Audit results : passed**

4.2.6 Faulty constructor

- **Risk description**

Prior to version 0.4.22 in solidity smart contracts, all contracts and constructors had the same name. When writing a contract, if the constructor name and the contract name are not the same, the contract will add a default constructor and the constructor you set up will be treated as a normal function, resulting in your original contract settings not being executed as expected, which can lead to terrible consequences, especially if the constructor is performing a privileged operation.

- **Audit results : passed**

4.2.7 Unverified return value

- **Risk description**

Three methods exist in Solidity for sending tokens to an address: transfer(), send(), call.value(). The difference between them is that the transfer function throws an exception throw when sending fails, rolls back the transaction state, and costs 2300gas; the send function returns false when sending fails and costs 2300gas; the call.value method returns false when sending fails and costs all gas to call, which will lead to the risk of reentrant attacks. If the send or call.value method is used in the contract code to send tokens without checking the return value of the method, if an error occurs, the contract will continue to execute the code later, which will lead to the thought result.

- **Audit results : passed**

4.2.8 Insecure random numbers

- **Risk description**

All transactions on the blockchain are deterministic state transition operations with no uncertainty, which ultimately means that there is no source of entropy or randomness within the blockchain ecosystem. Therefore, there is no random number function like rand() in Solidity. Many developers use future block variables such as block hashes, timestamps, block highs and lows or Gas caps to generate random numbers. These quantities are controlled pass the miners who mine them and are therefore not truly random, so using past or present block variables to generate random numbers could lead to a destructive vulnerability.

- **Audit results : passed**

4.2.9 Timestamp Dependency

- **Risk description**

In blockchains, data block timestamps (block.timestamp) are used in a variety of applications, such as functions for random numbers, locking funds for a period of time, and conditional statements for various time-related state changes. Miners have the ability to adjust the timestamp as needed, for example block.timestamp or the alias now can be manipulated pass the miner. This can lead to serious vulnerabilities if the wrong block timestamp is used in a smart contract. This may not be necessary if the contract is not particularly concerned with miner manipulation of block timestamps, but care should be taken when developing the contract.

- **Audit results : passed**

4.2.10 Transaction order dependency

- **Risk description**

In a blockchain, the miner chooses which transactions from that pool will be included in the block, which is usually determined pass the gasPrice transaction, and the miner will choose the transaction with the highest transaction fee to pack into the block. Since the information about the transactions in the block is publicly available, an attacker can watch the transaction pool for transactions that may contain problematic solutions, modify or revoke the attacker's privileges or change the state of the contract to the attacker's detriment. The attacker can then take data from this transaction and create a higher-level transaction gasPrice and include its transactions in a block before the original, which will preempt the original transaction solution.

- **Audit results : passed**

4.2.11 Delegatecall function call

- **Risk description**

In Solidity, the delegatecall function is the standard message call method, but the code in the target address runs in the context of the calling contract, i.e., keeping msg.sender and msg.value unchanged. This feature supports implementation libraries, where developers can create reusable code for future contracts. The code in the library itself can be secure and bug-free, but when run in another application's environment, new vulnerabilities may arise, so using the delegatecall function may lead to unexpected code execution.

- **Audit results : passed**

4.2.12 Call function call

- **Risk description**

The call function is similar to the delegatecall function in that it is an underlying function provided pass Solidity, a smart contract writing language, to interact with external contracts or libraries, but when the call function method is used to handle an external Standard Message Call to a contract, the code runs in the environment of the external contract/function The call function is used to interact with an external contract or library. The use of such functions requires a determination of the security of the call parameters, and caution is recommended. An attacker could easily borrow the identity of the current contract to perform other malicious operations, leading to serious vulnerabilities.

- **Audit results : passed**

4.2.13 Denial of Service

- **Risk description**

Denial of service attacks have a broad category of causes and are designed to keep the user from making the contract work properly for a period of time or permanently in certain situations, including malicious behavior while acting as the recipient of a transaction, artificially increasing the gas required to compute a function causing gas exhaustion (such as controlling the size of variables in a for loop), misuse of access control to access the private component of the contract, in which the Owners with privileges are modified, progress state based on external calls, use of obfuscation and oversight, etc. can lead to denial of service attacks.

- **Audit results : passed**

4.2.14 Logic Design Flaw

- **Risk description**

In smart contracts, developers design special features for their contracts intended to stabilize the market value of tokens or the life of the project and increase the highlight of the project, however, the more complex the system, the more likely it is to have the possibility of errors. It is in these logic and functions that a minor mistake can lead to serious depasstions from the whole logic and expectations, leaving fatal hidden dangers, such as errors in logic judgment, functional implementation and design and so on.

- **Audit results : passed**

4.2.15 Fake recharge vulnerability

- **Risk description**

The success or failure (true or false) status of a token transaction depends on whether an exception is thrown during the execution of the transaction (e.g., using mechanisms such as require/assert/revert/throw). When a user calls the transfer function of a token contract to transfer funds, if the transfer function runs normally without throwing an exception, the transaction will be successful or not, and the status of the transaction will be true. When balances[msg.sender] < _value goes to the else logic and returns false, no exception is thrown, but the transaction acknowledgement is successful, then we believe that a mild if/else judgment is an undisciplined way of coding in sensitive function scenarios like transfer, which will lead to Fake top-up vulnerability in centralized exchanges, centralized wallets, and token contracts.

- **Audit results : passed**

4.2.16 Short Address Attack Vulnerability

- **Risk description**

In Solidity smart contracts, when passing parameters to a smart contract, the parameters are encoded according to the ABI specification. the EVM runs the attacker to send encoded parameters that are shorter than the expected parameter length. For example, when transferring money on an exchange or wallet, you need to send the transfer address address and the transfer amount value. The attacker could send a 19-passte address instead of the standard 20-passte address, in which case the EVM would fill in the 0 at the end of the encoded parameter to make up the expected length, which would result in an overflow of the final transfer amount parameter value, thus changing the original transfer amount.

- **Audit results : passed**

4.2.17 Uninitialized storage pointer

- **Risk description**

EVM uses both storage and memory to store variables. Local variables within functions are stored in storage or memory pass default, depending on their type. uninitialized local storage variables could point to other unexpected storage variables in the contract, leading to intentional or unintentional vulnerabilities.

- **Audit results: passed**

4.2.18 Frozen Account passpass

- **Risk Description**

In the transfer operation code in the contract, detect the risk that the logical functionality to check the freeze status of the transfer account exists in the contract code and can be passpassed if the transfer account has been frozen.

- **Audit results : passed**

4.2.19 Contract caller not initialized

- **Risk description**

The initialize function in the contract can be called pass another attacker before the owner, thus initializing the administrator address.

- **Audit results : passed**

4.2.20 Re-entry Attack

- **Risk description**

An attacker constructs a contract containing malicious code at an external address in the Fallback function When the contract sends tokens to this address, it will call the malicious code. The call.value() function in Solidity will consume all the gas he receives when it is used to send tokens, so a re-entry attack will occur when the call to the call.value() function to send tokens occurs before the actual reduction of the sender's account balance. The re-entry vulnerability led to the famous The DAO attack.

- **Audit results : passed**

4.2.21 Integer Overflow

- **Risk description**

Integer overflows are generally classified as overflows and underflows. The types of integer overflows that occur in smart contracts include three types: multiplicative overflows, additive overflows, and subtractive overflows. In Solidity language, variables support integer types in steps of 8, from uint8 to uint256, and int8 to int256, integers specify fixed size data types and are unsigned, for example, a uint8 type , can only be stored in the range 0 to 2^8-1, that is, [0,255] numbers, a uint256 type can only store numbers in the range 0 to 2^256-1. This means that an integer variable can only have a certain range of numbers represented, and cannot exceed this formulated range. Exceeding the range of values expressed pass the variable type will result in an integer overflow vulnerability.

- **Audit results : passed**

5. Security Audit Tool

Tool name	Tool Features
Oyente	Can be used to detect common bugs in smart contracts
securify	Common types of smart contracts that can be verified
MAIAN	Multiple smart contract vulnerabilities can be found and classified
Lunaray Toolkit	self-developed toolkit

Disclaimer：

Lunaray Technology only issues a report and assumes corresponding responsibilities for the facts that occurred or existed before the issuance of this report, Since the facts that occurred after the issuance of the report cannot determine the security status of the smart contract, it is not responsible for this.

Lunaray Technology conducts security audits on the security audit items in the project agreement, and is not responsible for the project background and other circumstances, The subsequent on-chain deployment and operation methods of the project party are beyond the scope of this audit.

This report only conducts a security audit based on the information provided by the information provider to Lunaray at the time the report is issued, If the information of this project is concealed or the situation reflected is inconsistent with the actual situation, Lunaray Technology shall not be liable for any losses and adverse effects caused thereby.

There are risks in the market, and investment needs to be cautious. This report only conducts security audits and results announcements on smart contract codes, and does not make investment recommendations and basis.

LUNARAY

BLOCKCHAINSECURITY


https://lunaray.co


https://github.com/lunaraySec


https://twitter.com/lunaray_Sec


http://t.me/lunaraySec