LUNARAY
BLOCKCHAINSECURITY

SMART CONTRACT SECURITY AUDIT REPORT

For RatelSwap

27 April 2022

lunaray.co

Table of Contents

1. Overview

On April 19, 2022, the security team of Lunaray Technology received the security audit request of the **RatelSwap project**. The team completed the audit of the **RatelSwap smart contract** on April 27, 2022. During the audit process, the security audit experts of Lunaray Technology and the RatelSwap 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 RatelSwap 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 RatelSwap smart contract security audit: **Passed**

Audit Report Hash:
95EBCE6E96065FDB0368240922C292B7FC6B1D90B0362420393152E43EA3C43A

2. Background

2.1 Project Description

Project name	RatelSwap
Contract type	Token , DeFi
Code language	Solidity
Public chain	Binance
Project address	https://ratelswap.io/
Contract file	RTStake_flat.sol, RTToken_flat.sol, Mulsign.sol

2.2 Audit Range

The smart contract file provided by RatelSwap and the corresponding MD5：

Name	address
RTStake_flat.sol	60DB17A5EF8FC7811BC28797C420EA35
RTToken_flat.sol	802DB63FEDAEBE4807E12E46F86A2B3E
Mulsign.sol	F81CFF40D20D53E77E0E3CE0EC12B245

3. Project contract details

3.1 Contract Overview

RTToken Contract

It mainly implements the functions of minting, destroying and transferring RT Token, and additionally implements the blacklist function.

RTStake Contract

The main implementation of the relevant business logic, as well as the basic deposit and withdrawal of money function.

MulSign Contract

Implemented the logic related to multi-signature contracts, which is used to restrain the problem of excessive administrator privileges, and all events must be confirmed by all managers before they can be executed.

3.2 Contract details

RTToken Contract

Name	Parameter	Attributes
transfer	address to uint256 amount	public
transferFrom	address from address to uint256 amount	public
mint	address to uint256 amount	onlyOwner
ratelBlackAdd	address addr	onlyOwner
ratelBlackRemove	address addr	onlyOwner

RTStake Contract

Name	Parameter	Attributes
owner	none	public
renounceOwnership	none	onlyOwner
transferOwnership	address newOwner	onlyOwner
_transferOwnership	address newOwner	internal
name	none	public
symbol	none	public
decimals	none	public
totalSupply	none	public
balanceOf	address account	public
transfer	address to uint256 amount	public
allowance	address owner address spender	public

approve	address spender uint256 amount	public
transferFrom	address from address to uint256 amount	public
increaseAllowance	address spender uint256 addedValue	public
decreaseAllowance	address spender uint256 subtractedValue	public
_transfer	address from address to uint256 amount	internal
_mint	address account uint256 amount	internal
_burn	address account uint256 amount	internal
_approve	address owner address spender uint256 amount	internal
_spendAllowance	address owner address spender uint256 amount	internal
_beforeTokenTransfer	address from address to uint256 amount	internal
_afterTokenTransfer	address from address to uint256 amount	internal
updateTestNumber	uint256 _testNumber	onlyOwner
updateInitPoolRoot	address _addr	onlyOwner
updatePerReward	uint256 _reward	onlyOwner
updateFoundationPerReward	uint256 _reward	onlyOwner
updatePoolPerReward	uint256 _reward	onlyOwner
updateMarketPerReward	uint256 _reward	onlyOwner
updateEndTime	uint256 _endTime	onlyOwner
updateInitPoolMaxUsdt	uint256 _maxUsdt	onlyOwner
updateDataStorage	address _dataStorage	onlyOwner

removeWhiteList	address _addr	onlyOwner
ratelBlackAdd	address addr	onlyOwner
ratelBlackRemove	address addr	onlyOwner
addStakeInfo	uint32 _day uint256 _rate	onlyOwner
setStakeInfo	uint32 _sid uint32 _day uint256 _rate	onlyOwner
transfer	address _to address _token uint256 _amount	onlyOwner
giveBackTokenOwner	address _newOwner	onlyOwner
mintRT	address _addr uint256 _amount	onlyOwner
whiteListContains	address _addr	public
viewWhiteList	none	public
ableToAddInInit	address _user	public
_stateDepositCheck	uint32 _sid UserInfo _user uint256 _amount	internal
deposit	uint32 _sid uint256 _amount	public
_stateWithdrawCheck	UserInfo _user uint256 _amount uint256 _maxWithdrawLp	internal
withdraw	uint256 _amount	public
updatePool	none	public
updateBase	none	public
pendingRT	address _user	public
pendingMarketRT	address _user	public
pendingFoundationRT	none	public
pendingPoolRT	none	public

viewAddressList	none	public
viewAddress	uint256 _index	public
userLength	none	public
getPrice	none	public
getUserTokenAmount	address _user	public
getTotalTokenAmount	none	public
getLiquidityAmount	uint256 liquidity	public
getStakeInfos	address _user	public
_updateStakeState	none	internal
getInitAddrWithdrawMaxAmount	address _addr	public
_addUser	none	internal
_updateSharePow	uint256 _oldSharePow uint256 _newSharePow	internal
_initDeposit	uint32 _sid uint256 _amount	internal
_addToSwap	none	internal
_removeLiquidity	uint256 _liquidity	internal
_swapRT	uint256 _amount	internal
_swapToken	uint256 _amount address _tokenA address _tokenB	internal
_addLiquidity	address _tokenA address _tokenB uint256 _amountA uint256 _amountB	internal
_getAmountOut	uint256 amountIn uint256 reserveIn uint256 reserveOut	internal

_getLpAmount	uint256 _amount	internal
_getInitLpAmount	none	internal
_safeTokenTransfer	address _to address token uint256 _amount	internal
_getRewardTime	uint256 _blockTime	internal
_getMultiplier	uint256 _from uint256 _to	internal
_getPowRate	uint32 _sid uint256 _usdt bool isInit	internal
_getSharePow	uint32 _sid uint256 _usdt	internal
_getRate	uint32 _sid uint256 _usdt bool isInit	internal
_getShareRate	uint32 _sid uint256 _usdt	internal

MulSign Contract

Name	Parameter	Attributes
_equal	string a string b	internal
transferAuth	uint256 eventId address addr	onlyManage
auth_transferAuth	uint256 eventId	onlyManage
updateLock	uint256 eventId address lockAddr	onlyManage
auth_updateLock	uint256 eventId	onlyManage
ratelBlackAdd	uint256 eventId address addr	onlyManage
auth_ratelBlackAdd	uint256 eventId	onlyManage
ratelBlackRemove	uint256 eventId address addr	onlyManage
auth_ratelBlackRemove	uint256 eventId	onlyManage
updateRt	uint256 eventId address addr	onlyManage

auth_updateRt	uint256 eventId	onlyManage
updateLpToken	uint256 eventId address addr	onlyManage
auth_updateLpToken	uint256 eventId	onlyManage
updateSwapRouter	uint256 eventId address addr	onlyManage
auth_updateSwapRouter	uint256 eventId	onlyManage
updateSwapFactory	uint256 eventId address addr	onlyManage
auth_updateSwapFactory	uint256 eventId	onlyManage
updateTestNumber	uint256 eventId uint256 _number	onlyManage
auth_updateTestNumber	uint256 eventId	onlyManage
updatePerReward	uint256 eventId uint256 perReward	onlyManage
auth_updatePerReward	uint256 eventId	onlyManage
updateFoundationPerReward	uint256 eventId uint256 foundationPerReward	onlyManage
auth_updateFoundationPerReward	uint256 eventId	onlyManage
updatePoolPerReward	uint256 eventId uint256 poolPerReward	onlyManage
auth_updatePoolPerReward	uint256 eventId	onlyManage
updateMarketPerReward	uint256 eventId uint256 marketPerReward	onlyManage
auth_updateMarketPerReward	uint256 eventId	onlyManage
updateEndTime	uint256 eventId uint256 endTime	onlyManage
auth_updateEndTime	uint256 eventId	onlyManage

updateDataStorage	uint256 eventId address ds	onlyManage
auth_updateDataStorage	uint256 eventId	onlyManage
transfer	uint256 eventId address to address token uint256 amount	onlyManage
auth_transfer	uint256 eventId	onlyManage
giveBackTokenOwner	uint256 eventId address newOwner	onlyManage
auth_giveBackTokenOwner	uint256 eventId	onlyManage
mintRT	uint256 eventId address addr uint256 amount	onlyManage
auth_mintRT	uint256 eventId	onlyManage
grantRole	uint256 eventId bytes32 role address account	onlyManage
auth_grantRole	uint256 eventId	onlyManage
revokeRole	uint256 eventId bytes32 role address account	onlyManage
auth_revokeRole	uint256 eventId	onlyManage

4. Audit details

4.1 Findings Summary

Severity	Found	Resolved	Acknowledged
High	0	0	0
Medium	0	0	0
Low	2	0	2
Info	3	0	3

4.2 Risk distribution

Name	Risk level	Repair status
Self Transfer	Low	Acknowledged
Possible denial of service	Low	Acknowledged
Potential for logical confusion	Info	Acknowledged
Redundant codes	Info	Acknowledged
Incomplete logic	Info	Acknowledged
Deposit logic issues	No	normal
Withdrawal logic issues	No	normal
Variables are updated	No	normal
Floating Point and Numeric Precision	No	normal
Default visibility	No	normal
tx.origin authentication	No	normal
Faulty constructor	No	normal
Unverified return value	No	normal
Insecure random numbers	No	normal
Timestamp Dependent	No	normal
Transaction order dependency	No	normal
Delegatecall	No	normal
Call	No	normal
Denial of Service	No	normal
Logical Design Flaw	No	normal
Fake recharge vulnerability	No	normal

Short address attack Vulnerability	No	normal
Uninitialized storage pointer	No	normal
Frozen account bypass	No	normal
Uninitialized	No	normal
Reentry attack	No	normal
Integer Overflow	No	normal

4.3 Risk audit details

4.3.1 Self Transfer

- **Risk description**

The transfer function only checks the blacklist and does not check if the to address is the same as _msgSender(), resulting in the user being able to consume the fee through the act of self-transfer.

```
function transfer(address to, uint256 amount) public virtual override r
eturns (bool) {
    require(!_black.contains(_msgSender()), "forbidden");
    uint256 burnValue = amount / 100;
    if (_burnAmount >= _MAX_BURN){
        burnValue = 0;
    } else if (_burnAmount + burnValue > _MAX_BURN){
        burnValue = _burnAmount + burnValue - _MAX_BURN;
    }
    _transfer(_msgSender(), to, amount - burnValue);

    if (burnValue != 0){
        _transfer(_msgSender(), _burnAddress, burnValue);
        _burnAmount = _burnAmount + burnValue;
    }
    if (amount >= 10 * 1e18) {
        IDataStorage(_dataStorage).bindReferer(_msgSender(), to);
    }
    return true;
}
```

- **Safety advice**

It is recommended to check the address consistency before transferring.

- **Repair Status**

RatelSwap has confirmed.

4.3.2 Possible denial of service

- **Risk description**

There are only two multi-signature control addresses in the contract, and when the key of either control address is lost, a denial-of-service problem occurs, which means that the method invoked by the administrator cannot be executed.

```
function auth_ratelBlackAdd(uint256 eventId) public onlyManage authId(e
ventId){
    CallEvent storage callEvent = callEvents[eventId];
    bool authed = callEvent.auth[msg.sender];
    require(!authed, "MulSign: Has authorized");
    require(_equal(callEvent.eventName, "ratelBlackAdd"), "MulSign: wro
ng auth");
    callEvent.auth[msg.sender] = true;
    callEvent.authCount++;
    if(callEvent.authCount == manageCount){
        stake.ratelBlackAdd(callEvent.param4);
        callEvent.state = 1;
    }
}
```

- **Safety advice**

It is recommended to use more than 3 control addresses and to pass the verification after meeting two addresses (according to about 70% of the number of multi-signature addresses to determine whether you can pass the verification).

- **Repair Status**

RatelSwap has confirmed.

4.3.3 Potential for logical confusion

- **Risk description**

Some key business logic judgments use the subscript of the stakeInfo list written dead, but the stakeInfo list is modifiable by the administrator, and new ones may be added, or old ones may be modified, which may cause the logic in the business code to be inconsistent with the actual situation if the administrator makes a misoperation.

```
function _stateDepositCheck(uint32 _sid, UserInfo memory _user, uint256
 _amount) internal view{
        if (curState > State.Init &&  _amount == 0){
            return;
        }
        require (block.timestamp < endTime, "RTStake: end");
        require(_amount >= 10 * _USDT_DECIMALS, "RTStake: The amount mu
st not be less than 10 USDT");

        if (curState == State.Init){
            require(_sid == stakeInfo.length - 1, "RTStake: (Init) wron
g sid");
            require(ableToAddInInit(msg.sender), "RTStake: (Init) forbi
dden");
            require(!_user.isInitAddr, "RTStake: (Init) forbidden");
        }else if (curState == State.Locked){
            require(whiteList.contains(msg.sender), "RTStake: (Locked)
forbidden");
            require(!_user.isInitAddr, "RTStake: (Locked) forbidden");
        }else if (curState == State.Half){
            require(_sid > 0, "RTStake: (Half) wrong sid");
            require(!_user.isInitAddr, "RTStake: (Locked) forbidden");
        }else{
            require(!_user.isInitAddr, "RTStake: (Locked) forbidden");
        }
        require(_sid < stakeInfo.length, "RTStake: no sid");
        require(_user.sid <= _sid, "RTStake: do not allow");
    }
```

- **Safety advice**

Make the code logic as consistent as possible with the business logic, using the unique key in the stakeInfo structure to make the determination.

- **Repair Status**

RatelSwap has confirmed.

4.3.4 Code Redundancy

- **Risk description**

There is more redundant code in the contract, for example, according to the logic, the bottom stage of the pool, initAddr_totalLpAmount == lpAmount, resulting in hasWithdraw always equal to 0, which is meaningless code; internal method _addLiquidity, _getAmountOut is not used; _getSharePow method only plays a passing role, can be deleted, directly call _getShareRate method;

```
function getInitAddrWithdrawMaxAmount(address _addr) public view return
s (uint256){
    UserInfo memory user = userInfo[_addr];
    if (!user.isInitAddr){
        return 0;
    }
    StakeInfo memory stake = stakeInfo[user.sid];
    uint256 interval = curTimeValue;
    uint256 stakeEndTime = startTime + stake.day * curTimeValue;
    if (block.timestamp <= stakeEndTime){
        return 0;
    }else if(block.timestamp >= stakeEndTime + (interval * 360)){
        return user.lpAmount;
    }
    uint256 times = block.timestamp.sub(stakeEndTime).div(interval);
    uint256 hasWithdraw = user.initAddr_totalLpAmount.sub(user.lpAmoun
t);
    return user.initAddr_totalLpAmount.mul(times).div(360).sub(hasWithd
raw);
}


function _getSharePow(uint32 _sid, uint256 _usdt)
    internal
    view
    returns (uint256)
{
    return _getShareRate(_sid, _usdt);
}

function _getShareRate(uint32 _sid, uint256 _usdt)
    internal
    view
    returns (uint256)
{
    require(_sid < stakeInfo.length, "RTStake: no sid");
    StakeInfo storage info = stakeInfo[_sid];
```

```
    uint256 rate = info.rate;
    uint256 usdt100 = _USDT_DECIMALS * 100;
    uint256 xUsdt100 = _usdt.div(usdt100);
    if (xUsdt100 >= 100) {
        return rate.mul(1000);
    } else if (xUsdt100 >= 50) {
        return rate.mul(500);
    } else if (xUsdt100 >= 20) {
        return rate.mul(200);
    }else if (xUsdt100 >= 10) {
        return rate.mul(100);
    } else if (xUsdt100 >= 5) {
        return rate.mul(50);
    } else {
        return 0;
    }
}
```

- **Safety advice**

Sort out the contract logic and remove useless code to avoid causing extra handling f ees.

- **Repair Status**

RatelSwap has confirmed.

4.3.5 Incomplete logic

- **Risk description**

The transfer function lacks the relevant check logic, and the validity of the amount of funds transferred is not judged when the transfer is made.

```
function transferFrom(address from, address to, uint256 amount) public
override returns (bool) {
    require(!_black.contains(_msgSender()), "forbidden");
    require(!_black.contains(from), "forbidden");

    _spendAllowance(from, _msgSender(), amount);

    uint256 burnValue = amount / 100;
    if (_burnAmount >= _MAX_BURN){
        burnValue = 0;
    } else if (_burnAmount + burnValue > _MAX_BURN){
            burnValue = _burnAmount + burnValue - _MAX_BURN;
    }
    _transfer(from, to, amount - burnValue);
    if (burnValue != 0){
        _transfer(from, _burnAddress, burnValue);
        _burnAmount = _burnAmount + burnValue;
    }
    return true;
}
```

- **Safety advice**

Strict checks for important operations in the contract, adding missing checks logic.

- **Repair Status**

RatelSwap has confirmed.

4.3.6 Deposit logic issues

- **Risk description**

This method is public and the caller passes in sid and _amount as parameters. First, the user information of the caller is taken out from storage and the legitimacy is checked with the current contract status.

If in the bottom pool stage, if the remaining minted quantity of rt is enough, the corresponding tokens are minted to the contract, and at the same time the user pays the corresponding usdt to the contract, if the remaining minted quantity is not enough, only the remaining supply is minted to the contract, and the user then transfers the corresponding usdt to the contract. When the total supply of rt is fully minted, the two tokens in the contract are added to the liquidity and the internal pledge phase begins.

If in the internal pledge phase, the mining proceeds of the liquidity pool are updated first, and then the proceeds of the committee are updated. For the caller's corresponding user, its mining proceeds are calculated and tokens are sent to the user. Next, the proceeds of the caller's promoter are updated and the promoter can access this part of the proceeds when it is accessed.

```
function deposit(uint32 _sid, uint256 _amount) public {
    UserInfo storage user = userInfo[msg.sender];
    _stateDepositCheck(_sid, user, _amount);
    if (_amount > 0){
        _addUser();
    }
    if (mintRTAmount != 0 && curState == State.Init) {
        _initDeposit(_sid, _amount);
        emit Deposit(msg.sender, _amount, _sid);
        return;
    }
    updatePool();
    updateBase();
    IERC20(usdt).safeTransferFrom(
        address(msg.sender),
        address(this),
        _amount
        );
    if (user.pow > 0) {
        uint256 pending = user.pow.mul(accTokenPerShare).div(MULVALUE).
sub(
            user.rewardDebt
        );
        if (pending > 0) {
            _safeTokenTransfer(msg.sender, rt, pending);
```

```
            user.totalTakeOut = user.totalTakeOut.add(pending);
        }
    }
    uint256 marketPow = IDataStorage(dataStorage).viewMarketAmount(msg.
sender);
    if (marketPow > 0) {
        uint256 pending = marketPow
            .mul(accMarketTokenPerShare)
            .div(MULVALUE)
            .sub(user.marketDebt);
        if (pending > 0 || user.marketUntakeOut > 0) {
            uint256 total = pending.add(user.marketUntakeOut);
            _safeTokenTransfer(msg.sender, rt, total);
            user.totalMarketTakeOut = user.totalMarketTakeOut.add(tota
l);
            user.marketUntakeOut = 0;
        }
    }
    if (_amount > 0) {
        address[] memory parents = IDataStorage(dataStorage).viewParent
s(msg.sender);
        _updateParent(parents);
        uint256 oldSharePow = _getSharePow(_sid, user.usdtAmount);
        user.lpAmount = user.lpAmount.add(_swapRT(_amount));
        user.usdtAmount = user.usdtAmount.add(_amount);
        user.stakeRate = _getPowRate(_sid, user.usdtAmount, user.isInit
Addr);
        uint256 newPow = user.lpAmount.mul(user.stakeRate);
        totalPow = totalPow.sub(user.pow).add(newPow);
        user.pow = newPow;
        user.sid = _sid;
        user.lastStakeTime = block.timestamp;
        user.shareRate = _getSharePow(_sid, user.usdtAmount);
        _updateSharePow(oldSharePow, user.shareRate);
        _updateParentDebt(parents);
    }
    user.rewardDebt = user.pow.mul(accTokenPerShare).div(MULVALUE);
    user.marketDebt = marketPow.mul(accMarketTokenPerShare).div(MULVALU
E);
    _updateStakeState();
    emit Deposit(msg.sender, _amount, _sid);
}
```

- **Audit Results : Passed**

4.3.7 Withdrawal logic issues

- **Risk description**

The method is public and the caller passes in _amount as a parameter. First, the user information of the caller is taken from storage and the mining revenue of the liquid pool is updated first, and then the revenue of the committee is updated. For the caller's corresponding user, its mining revenue is calculated and the tokens are sent to the user. Next, the proceeds of the caller's promoter are updated, and then the corresponding quantity is reduced from the liquidity according to the quantity to be taken out.

```
function withdraw(uint256 _amount) public {
    UserInfo storage user = userInfo[msg.sender];
    if (user.lastStakeTime < startTime) {
        user.lastStakeTime = startTime;
    }
    uint256 maxWithdrawLp = getInitAddrWithdrawMaxAmount(msg.sender);
        _stateWithdrawCheck(user, _amount, maxWithdrawLp);
    updatePool();
    updateBase();
    if (user.pow > 0) {
        uint256 pending = user.pow.mul(accTokenPerShare).div(MULVALUE).
sub(
            user.rewardDebt
        );
        if (pending > 0 || user.marketUntakeOut > 0) {
            _safeTokenTransfer(msg.sender, rt, pending);
            user.totalTakeOut = user.totalTakeOut.add(pending);
        }
    }
    uint256 marketPow = IDataStorage(dataStorage).viewMarketAmount(
        msg.sender
    );
    if (marketPow > 0) {
        uint256 pending = marketPow
            .mul(accMarketTokenPerShare)
            .div(MULVALUE)
            .sub(user.marketDebt);
       if (pending > 0 || user.marketUntakeOut > 0) {
            uint256 total = pending.add(user.marketUntakeOut);
            _safeTokenTransfer(msg.sender, rt, total);
            user.totalMarketTakeOut = user.totalMarketTakeOut.add(tota
l);
            user.marketUntakeOut = 0;
        }
```

```
    }
    if (_amount > 0) {
        address[] memory parents = IDataStorage(dataStorage).viewParent
s(msg.sender);
        _updateParent(parents);
        uint256 oldSharePow = _getSharePow(user.sid, user.usdtAmount);
        uint256 withdrawUsdt = _amount.mul(user.usdtAmount).div(user.lp
Amount);
        user.usdtAmount = user.usdtAmount.sub(withdrawUsdt);
        user.lpAmount = user.lpAmount.sub(_amount);
        user.stakeRate = _getPowRate(user.sid, user.usdtAmount, user.is
InitAddr);
        uint256 newPow = user.lpAmount.mul(user.stakeRate);
        totalPow = totalPow.sub(user.pow).add(newPow);
        user.pow = newPow;
        user.shareRate = _getSharePow(user.sid, user.usdtAmount);
        _removeLiquidity(_amount);
        _updateSharePow(oldSharePow, user.shareRate);
        _updateParentDebt(parents);
    }
    user.rewardDebt = user.pow.mul(accTokenPerShare).div(MULVALUE);
    user.marketDebt = marketPow.mul(accMarketTokenPerShare).div(MULVALU
E);
    emit Withdraw(msg.sender, _amount);
}
```

- **Audit Results : Passed**

4.3.8 Variables are updated

- **Risk description**

When there is a contract logic to obtain rewards or transfer funds, the coder mistakenly updates the value of the variable that sends the funds, so that the user can use the value of the variable that is not updated to obtain funds, thus affecting the normal operation of the project.

- **Audit Results : Passed**

4.3.9 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**

4.3.10 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**

4.3.11 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**

4.3.12 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**

4.3.13 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**

4.3.14 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**

4.3.15 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**

4.3.16 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**

4.3.17 Delegatecall

- **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**

4.3.18 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**

4.3.19 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**

4.3.20 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**

4.3.21 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**

4.3.22 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**

4.3.23 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**

4.3.24 Frozen Account bypass

- **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**

4.3.25 Uninitialized

- **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**

4.3.26 Reentry 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**

4.3.27 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