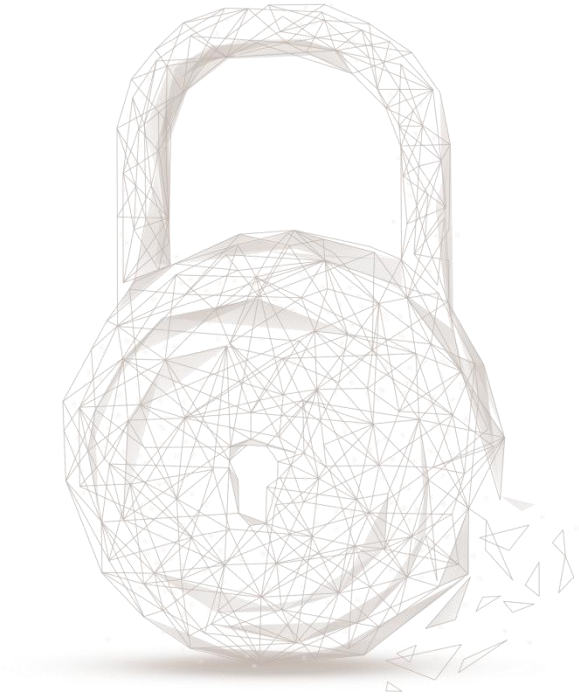
BEOSIN
Blockchain Security

Smart contract security audit report



Audit Number：202012221600

Report Query Name: Tomo

Smart Contract Address Link：

https://github.com/RAMP-DEFI/RAMP_TOMOCHAIN/tree/master/contracts

Initial Commit Hash：

d5419e085fdc731b4db7f31f2711835c6dd1dbc3

Finally Commit Hash：

6a9fd17789f96cafcfd5765ca043172e086d23b3

Start Date：2020.12.09

Completion Date：2020.12.22

Overall Result：Pass

Audit Team: Beosin (Chengdu LianAn) Technology Co. Ltd.

Audit Categories and Results:

No.	Categories	Subitems	Results
1	Coding Conventions	Compiler Version Security	Pass
		Deprecated Items	Pass
		Redundant Code	Pass
		SafeMath Features	Pass
		require/assert Usage	Pass
		Gas Consumption	Pass
		Visibility Specifiers	Pass
		Fallback Usage	Pass
2	General Vulnerability	Integer Overflow/Underflow	Pass
		Reentrancy	Pass
		Pseudo-random Number Generator (PRNG)	Pass
		Transaction-Ordering Dependence	Pass
		DoS (Denial of Service)	Pass

		Access Control of Owner	Pass
		Low-level Function (call/delegatecall) Security	Pass
		Returned Value Security	Pass
		tx.origin Usage	Pass
		Replay Attack	Pass
		Overriding Variables	Pass
3	Business Security	Business Logics	Pass
		Business Implementations	Pass

Note: Audit results and suggestions in code comments

Disclaimer: This audit is only applied to the type of auditing specified in this report and the scope of given in the results table. Other unknown security vulnerabilities are beyond auditing responsibility. Beosin (Chengdu LianAn) Technology only issues this report based on the attacks or vulnerabilities that already existed or occurred before the issuance of this report. For the emergence of new attacks or vulnerabilities that exist or occur in the future, Beosin (Chengdu LianAn) Technology lacks the capability to judge its possible impact on the security status of smart contracts, thus taking no responsibility for them. The security audit analysis and other contents of this report are based solely on the documents and materials that the contract provider has provided to Beosin (Chengdu LianAn) Technology before the issuance of this report, and the contract provider warrants that there are no missing, tampered, deleted; if the documents and materials provided by the contract provider are missing, tampered, deleted, concealed or reflected in a situation that is inconsistent with the actual situation, or if the documents and materials provided are changed after the issuance of this report, Beosin (Chengdu LianAn) Technology assumes no responsibility for the resulting loss or adverse effects. The audit report issued by Beosin (Chengdu LianAn) Technology is based on the documents and materials provided by the contract provider, and relies on the technology currently possessed by Beosin (Chengdu LianAn). Due to the technical limitations of any organization, this report conducted by Beosin (Chengdu LianAn) still has the possibility that the entire risk cannot be completely detected. Beosin (Chengdu LianAn) disclaims any liability for the resulting losses.

The final interpretation of this statement belongs to Beosin (Chengdu LianAn).

Audit Results Explained:

Beosin (Chengdu LianAn) Technology has used several methods including Formal Verification, Static Analysis, Typical Case Testing and Manual Review to audit three major aspects of smart contracts TOMO project, including Coding Standards, Security, and Business Logic. **The TOMO project contracts passed all audit items. The overall result is Pass. The smart contract is able to function properly.**

1. Coding Conventions

Check the code style that does not conform to Solidity code style.

1.1 Compiler Version Security

- Description: Check whether the code implementation of current contract contains the exposed solidity compiler bug.
- Result: Pass

1.2 Deprecated Items

- Description: Check whether the current contract has the deprecated items.
- Result: Pass

1.3 Redundant Code

- Description: Check whether the contract code has redundant codes.
- Result: Pass

1.4 SafeMath Features

- Description: Check whether the SafeMath has been used. Or prevents the integer overflow/underflow in mathematical operation.
- Result: Pass

1.5 require/assert Usage

- Description: Check the use reasonability of 'require' and 'assert' in the contract.
- Result: Pass

1.6 Gas Consumption

- Description: Check whether the gas consumption exceeds the block gas limitation.
- Result: Pass

1.7 Visibility Specifiers

- Description: Check whether the visibility conforms to design requirement.
- Result: Pass

1.8 Fallback Usage

- Description: Check whether the Fallback function has been used correctly in the current contract.
- Result: Pass

2. General Vulnerability

Check whether the general vulnerabilities exist in the contract.

2.1 Integer Overflow/Underflow

- Description: Check whether there is an integer overflow/underflow in the contract and the calculation result is abnormal.
- Result: Pass

2.2 Reentrancy

- Description: An issue when code can call back into your contract and change state, such as withdrawing ETH.
- Result: Pass

2.3 Pseudo-random Number Generator (PRNG)

- Description: Whether the results of random numbers can be predicted.
- Result: Pass

2.4 Transaction-Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Pass

2.5 DoS (Denial of Service)

- Description: Whether exist DoS attack in the contract which is vulnerable because of unexpected reason.
- Result: Pass

2.6 Access Control of Owner

- Description: Whether the owner has excessive permissions, such as malicious issue, modifying the balance of others.
- Result: Pass

2.7 Low-level Function (call/delegatecall) Security

- Description: Check whether the usage of low-level functions like call/delegatecall have vulnerabilities.
- Result: Pass

2.8 Returned Value Security

- Description: Check whether the function checks the return value and responds to it accordingly.
- Result: Pass

2.9 tx.origin Usage

- Description: Check the use secure risk of 'tx.origin' in the contract.
- Result: Pass

2.10 Replay Attack

- Description: Check the weather the implement possibility of Replay Attack exists in the contract.
- Result: Pass

2.11 Overriding Variables

- Description: Check whether the variables have been overridden and lead to wrong code execution.
- Result: Pass

3. Business Security

Check whether the business is secure.

3.1 Business analysis of Contract Token

(1) Basic Token Information

Token name	filled out after deployment
Token symbol	filled out after deployment
decimals	filled out after deployment
totalSupply	Initial supply is 0 (Mintable, Burnable)
Token type	TRC20

Table 1 Basic Token Information

(2) addToWhitelist/removeFromWhitelist Function

- Description: As shown in Figure below, the contract implements the *addToWhitelist* & *removeFromWhitelist* functions to add & remove the whitelist, only the contract owner can call this function, and only users on the whitelist can participate in transfer, transferFrom, burn, and mint tokens.

```
function addToWhitelist(address _address) public onlyOwner {
    _whitelist[_address] = true;
    emit AddedToWhitelist(_address);
}

/**
 * @dev Removes an address from the list
 * @param _address contract address
 */
function removeFromWhitelist(address _address) public onlyOwner {
    _whitelist[_address] = false;
    emit RemovedFromWhitelist(_address);
}
```

Figure 1 addToWhitelist/removeFromWhitelist Function source code

- Related functions: addToWhitelist, removeFromWhitelist
- Result: Pass

(3) mint Function

- Description: As shown in Figure below, the contract implements the *mint* function to mint tokens. After deployment, the StakingManager should be set to a whitelisted user. This token is used as staking point token. The impact of unlimited minting has not been found yet.

```
function mint(address account, uint256 value)
    public
    onlyWhitelisted
    returns (bool)
{
    _mint(account, value);
    return true;
}
```

Figure 2 mint Functions Source Code

- Related functions: *mint*
- Result: Pass

3.2 Business analysis of Contract StakingManager

(1) initialize Function

- Description: As shown in Figure below, the contract implements the *initialize* function to initialize the relevant parameters of the StakingManager contract. The initialize function is called through the proxy contract for initialization. The *initialize* function is only allowed to be called once.

```
function initialize(
    address _validator,
    address _token,
    address _tradeinAccount,
    address _teamfeeAccount,
    address _nodeAddress,
    address _owner
)
public
initializer
{
    validator = _validator;
    token = _token;

    tradeinAccount = _tradeinAccount;
    teamfeeAccount = _teamfeeAccount;
    nodeAddress = _nodeAddress;

    teamFee = 10;
    tradeinPercentage = 30;

    lastAddedRewardEpochIndex = 0;
    currentRewardingEpoch = 0;
    lastCalculateIndex = 0;
    minTokenDeposit = 100 ether;
    Ownable.ownableInitialize(_owner);
    Pausable.pausableInitialize();
}
```

Figure 3 initialize Function Source Code

- Related functions: *initialize*
- Result: Pass

(2) setTeamFee Function

- Description: As shown in Figure below, contract implements *setTeamFee* function to set team fee, contract owner can call this function to set teamfee rate. Note: If the sum of the teamfee rate and tradeinPercentage rate is 100%, Users will not get rewards if they deposited.

```
function setTeamFee(uint256 amount) public onlyOwner {
    require(amount >= 0 && amount <= 100, "Incorrect amount.");
    teamFee = amount;
}
```

Figure 4 setTeamFee Function Source code

- Related functions: *setTeamFee*

- Safety Suggestion: It is recommended to set the teamfee rate to a fixed value.
- Fix Result: Ignored.
- Result: Pass

(3) setTradeinPercentage Function

- Description: As shown in Figure below, contract implements *setTradeinPercentage* function to set tradeinPercentage rate. The contract owner can call this function to set tradeinPercentage rate. Note: If the sum of the teamfee rate and tradeinPercentage rate is 100%, Users will not get rewards if they deposited.

```
521     function getMyInvitedLength(address account)
522         external
523         view
524         returns (uint256, uint256)
525     {
526         User memory user = USERS[account];
527 
528         return (user.Invited1st.length, user.Invited2nd.length);
529     }
```

Figure 5 setTradeinPercentage Function Source Code

- Related functions: *getMyInvitedLength*
- Safety Suggestion: It is recommended to set the tradeinPercentage rate to a fixed value. The project party has ignored.
- Result: Pass

(4) setMinimumAmountToStake Function

- Description: As shown in Figure below, the contract implements the *setMinimumAmountToStake* function to set minimum amount of user deposit. The contract owner can call this function to set minimum amount of user deposit.

```
function setMinimumAmountToStake(uint256 amount) public onlyOwner {
    minTokenDeposit = amount;
}
```

Figure 6 setMinimumAmountToStake Function Source Code

- Related functions: *setMinimumAmountToStake*
- Result: Pass

(5) claimTeamFee Function

- Description: As shown in Figure below, the contract implements the *claimTeamFee* function to withdraw team fee. The address teamfeeAccount can call this function to withdraw team fee. Note: When the contract is paused, the function is not available.

```
function claimTeamFee() public whenNotPaused {
    require(msg.sender == teamfeeAccount, "Wrong address.");
    uint256 amount = totalTeamFee;

    // Subtract the amount from the payable
    payableAmount = payableAmount.sub(amount);

    teamfeeAccount.transfer(totalTeamFee);
    totalTeamFee = 0;
    emit ClaimTeamFee(teamfeeAccount, amount);
}
```

Figure 6 claimTeamFee Function Source Code

- Related functions: *claimTeamFee , transfer*
- Result: Pass

(6) claimTradeinPercentage Function

- Description: As shown in Figure below, the contract implements the *claimTradeinPercentage* function to withdraw TradeinPercentage fee. The tradeinAccount can call this function to withdraw TradeinPercentage fee. Note: When the contract is paused, the function is not available.

```
function claimTradeinPercentage() public whenNotPaused {
    require(msg.sender == tradeinAccount, "Wrong address.");
    uint256 amount = totalTradein;

    // Subtract the amount from the payable
    payableAmount = payableAmount.sub(amount);

    tradeinAccount.transfer(totalTradein);
    totalTradein = 0;
    emit ClaimTradeinPercentage(teamfeeAccount, amount);
}
```

Figure 7 claimTradeinPercentage Function Source Code

- Related functions: *claimTradeinPercentage, transfer*
- Result: Pass

(7) deposit Function

- Description: As shown in Figure below, the contract implements the *deposit* function to deposit tokens or delegate deposit tokens. The user's deposit amount is required to be greater than or equal to 100 ETH, and no reward settlement is currently in progress. After the user deposited, update the deposit information of the user and contract and call the *vote* function of the validator contract to vote, then call

the *mint* function of the token contract to mint the corresponding amount of point tokens to the user. Note: When the contract is paused, the function is not available.

```
function deposit(address user, address ethAddress) public payable whenNotPaused {
    require(msg.value >= minTokenDeposit, "Incorrect value.");
    require(user != address(0), "Incorrect user address.");
    require(lastCalculateIndex == 0, "Function not available, rewards are being calculated.");

    totalStake = totalStake.add(msg.value);
    deposits.push(
        Deposit(
            user, ethAddress,
            currentEpoch(), 0, msg.value
        )
    );
    userIndex[user].push(deposits.length - 1);

    IValidator(validator).vote.value(msg.value)(nodeAddress);
    IERC20(token).mint(user, msg.value);
    emit DepositEvent(user, msg.value);
}
```

deposit Function Source Code

- Related functions: *deposit, vote, mint*
- Result: Pass

(8) requestWithdraw Function

- Description: As shown in Figure below, the contract implements the *requestWithdraw* function to request withdrawal of deposited tokens. The user is required to have deposited and the withdrawal index is in the user's deposited list. The user's deposited has not requested withdrawal. After the user requests to withdraw deposited tokens, update the contract deposited information and user deposited information. Call the *unvote* function of the validator contract to cancel the vote, and then call the *burnFrom* function of the token contract to destroy the corresponding amount of point tokens for this request. Note: When the contract is paused, the function is not available.

```
function requestWithdraw(uint256 index) public whenNotPaused {
    require(userIndex[msg.sender].length > 0, "Staker doesn't exists.");
    require(userIndex[msg.sender].length > index, "Incorrect index.");

    uint256 depositIndex = userIndex[msg.sender][index];
    Deposit storage userDeposit = deposits[depositIndex];
    uint256 amount = userDeposit.amount;

    require(deposits[depositIndex].endEpoch == 0, "Withdrawal already requested.");

    totalStake = totalStake.sub(amount);
    IValidator(validator).unvote(nodeAddress, amount);

    deposits[depositIndex].endEpoch = currentEpoch();

    // burn the underlying wToken
    IERC20(token).burnFrom(msg.sender, amount);

    emit RequestWithdraw(
        msg.sender,
        index,
        amount,
        deposits[depositIndex].endEpoch
    );
}
```

Figure 8 requestWithdraw Function Source Code

- Related functions: *requestWithdraw, unvote, currentEpoch, burnFrom*
- Result: Pass

(9) withdraw Function

- Description: The contract implements the *withdraw* function for the user to withdraw the staked tokens requested to be withdrawn. The user is required to have deposited and the withdrawal index is in the user's deposit list. The withdrawal index has been requested to withdraw and 96 epochs (about 48 hours) have passed. After withdrawal, delete the deposit information corresponding to the user deposit list and the deposit information corresponding to the contract overall deposit list, and send the withdrawn staked tokens to the user address. Note: When the contract is paused, the function is not available. When withdrawing tokens, the contract Operator is required to first uniformly withdraw the staked tokens requested by the user from the validator contract, otherwise the withdrawal will fail.

```
function withdraw(uint256 index) public whenNotPaused {
    require(userIndex[msg.sender].length > 0, "Staker doesn't exists.");
    require(userIndex[msg.sender].length > index, "Incorrect index.");
    require(lastCalculateIndex == 0, "Function not available, rewards are being calculated.");

    uint256 depositIndex = userIndex[msg.sender][index];
    Deposit storage userDeposit = deposits[depositIndex];
    uint256 amount = userDeposit.amount;

    require(deposits[depositIndex].endEpoch != 0, "Withdrawal for the deposit is not requested.");
    require(deposits[depositIndex].endEpoch + 96 < currentEpoch(), "Funds are still frozen.");

    if (userIndex[msg.sender].length > 1) {
        userIndex[msg.sender][index] = userIndex[msg.sender][userIndex[msg.sender].length - 1];
    }
    userIndex[msg.sender].length--;

    if (deposits.length > 1) {
        if (depositIndex != deposits.length - 1){
            address tempUser = deposits[deposits.length - 1].user;
            for (uint256 i = 0; i < userIndex[tempUser].length; i++) {
                if (userIndex[tempUser][i] == deposits.length - 1) {
                    userIndex[tempUser][i] = depositIndex;
                    break;
                }
            }
            deposits[depositIndex] = deposits[deposits.length - 1];
        }
    }
    deposits.length--;

    // transfer funds to user
    msg.sender.transfer(amount);
    payableAmount = payableAmount.sub(amount);
    emit Withdraw(msg.sender, index, amount);
}
```

Figure 9 withdraw Function Source Code

- Related functions: *withdraw, currentEpoch, transfer*
- Result: Pass

(10) claimReward Function

- Description: As shown in Figure below, the contract implements the *claimReward* function to withdraw deposited rewards of users. Any users can call this function to withdraw deposited rewards to user address. Note: When the contract is paused, the function is not available.

```
function claimReward(address user) public whenNotPaused
    uint256 amount = rewards[user];

    // Subtract the amount from the payable
    payableAmount = payableAmount.sub(amount);

    require(amount > 0, "Staker doesn't exists.");
    user.transfer(amount);
    rewards[user] = 0;
    emit ClaimReward(user, amount);
}
```

Figure 10 claimReward Function Source Code

- Related functions: *claimReward* , *transfer*
- Safety Suggestion: It is recommended to set the set stake fee rate to a fixed value. The project party has ignored.
- Result: Pass

(11) updateNodeRewardAndTransferFrozenWithdrawal Function

- Description: The contract implements *updateNodeRewardAndTransferFrozenWithdrawal* function to update the user stake reward and uniformly withdraw the staked tokens requested by the user in the validator contract. The lastCalculateIndex is required to be 0, and then the stake reward coefficient of the current epoch is updated. Call the *getWithdrawBlockNumbers* and *withdraw* functions of the validator contract to uniformly withdraw staked tokens request by the user to this contract. Note: When the contract is paused, the function is not available.

```
function updateNodeRewardAndTransferFrozenWithdrawal(uint256 maxCount)
    public
    onlyOperator
    whenNotPaused
{
    // Check inputs
    require(lastCalculateIndex == 0, "Last cycle reward is not completed");
    uint256 epoch = currentEpoch();

    // Get contract balance: this is payableAmount + rewardsDistributable + tradeIn + teamfee
    uint256 contractBalance = address(this).balance;
    uint256 reward = contractBalance.sub(payableAmount);

    if (reward > 0 && totalStake > 0) {
        // Accumulate all rewards payable. This includes trade-in, teamfee and user rewards claima
        payableAmount = payableAmount.add(reward);

        // Update coef
        coefs[lastAddedRewardEpochIndex] = Coef(reward.mul(10 ** 18).div(totalStake), epoch);
        lastAddedRewardEpochIndex++;
    }

    uint256[] memory blockNumbers = IValidator(validator).getWithdrawBlockNumbers();
    if (blockNumbers.length > 0) {
        uint256 processCount;
        // Adjust number to process
        if( blockNumbers.length > maxCount) {
            processCount = maxCount;
        } else {
            processCount = blockNumbers.length;
        }
        uint256 currentContractBalance = address(this).balance;
        for (uint256 i = 0; i < processCount; i++) {
            if(blockNumbers[i] > 0 && block.number >= blockNumbers[i]) {
                IValidator(validator).withdraw(blockNumbers[i], i);
            }
        }

        payableAmount = payableAmount.add(address(this).balance).sub(currentContractBalance);
    }
```

Figure 11 updateNodeRewardAndTransferFrozenWithdrawal Function Source Code

- Related functions: *updateNodeRewardAndTransferFrozenWithdrawal, currentEpoch, getWithdrawBlockNumbers, withdraw*
- Result: Pass

(12) updateClaimableRewards Function

- Description: The contract implements *updateClaimableRewards* function to update the user's stake reward. The contract operator calls this function to update the user's stake reward in the current epoch, calculates totalTeamFee and totalTradein, and updates the relevant information about the contract reward calculation. Note: If the sum of the teamfee rate and tradeinPercentage rate is 100%, the users will not get rewards if they deposited.

```
function updateClaimableRewards(uint256 maxCount) public onlyOperator whenNotPaused {
    uint256 pendingToProcess =  deposits.length.sub(lastCalculateIndex);
    require(pendingToProcess > 0, "Nothing to process - updateClaimableRewards");
    uint256 processCount;
    bool finalRun = false;

    // Adjust number to process
    if( pendingToProcess > maxCount) {
        processCount = maxCount;
    } else {
        processCount = pendingToProcess;
        finalRun = true;
    }

    uint256 coef = coefs[currentRewardingEpoch].coef;

    for (uint256 i = 0; i < processCount; i++) {
        uint256 index = lastCalculateIndex.mod(deposits.length);
        if (
            coefs[currentRewardingEpoch].epoch > deposits[index].startEpoch &&
            (coefs[currentRewardingEpoch].epoch <= deposits[index].endEpoch || deposits[index].endEpoch == 0)
        ) {
            uint256 netClaimable = deposits[index].amount.mul(coef).div(10 ** 18);

            // calculate current team fee and update team fee
            uint256 currentTeamFee = netClaimable.mul(teamFee).div(100);
            totalTeamFee = totalTeamFee.add(currentTeamFee);
            netClaimable = netClaimable.sub(currentTeamFee);

            uint256 currentTradeInFee = netClaimable.mul(tradeinPercentage).div(100);
            totalTradein = totalTradein.add(currentTradeInFee);
            netClaimable = netClaimable.sub(currentTradeInFee);

            // update eth tradein amountTeamFee
            tradeinRewards[deposits[index].ethAddress] = tradeinRewards[deposits[index].ethAddress].add(currentTradeInFee);

            // update user claimable amount
            rewards[deposits[index].user] = rewards[deposits[index].user].add(netClaimable);
            lastCalculateIndex++;
        }
```

Figure 11 updateClaimableRewards Function Source Code

- Related functions: *updateClaimableRewards*
- Result: Pass

4. Conclusion

Beosin(ChengduLianAn) conducted a detailed audit on the design and code implementation of the smart contracts TOMO project. All problems found during the audit have been repaired after being informed to the project side. Note: After calling *requestWithdraw*, only the contract Operator calls *updateNodeRewardAndTransferFrozenWithdrawal* to uniformly withdraw back, the user can withdraw staked tokens. The stake rewards are recorded by the contract Operator calling related functions. If the sum of the teamfee rate and tradeinPercentage rate is 100%, Users will not get rewards if they deposited. The overall audit result of the smart contracts TOMO project is **Pass**.

BEOSIN

Blockchain Security

Official Website

https://lianantech.com

E-mail

vaas@lianantech.com

Twitter

https://twitter.com/Beosin_com