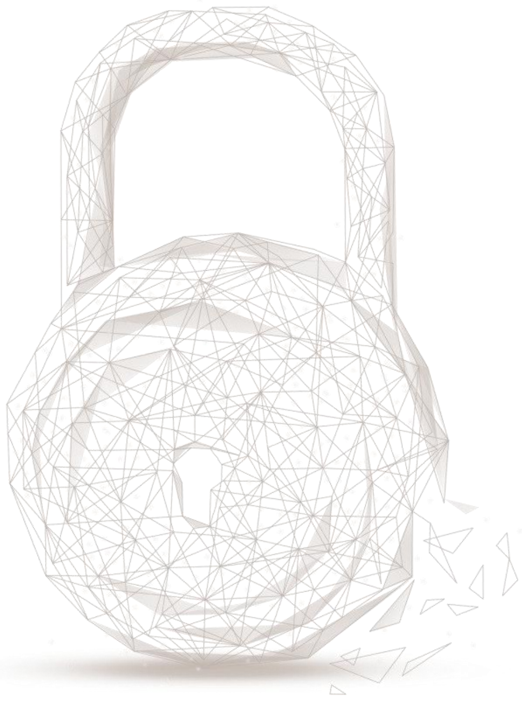
BEOSIN

Blockchain Security

Smart contract security audit report



Audit Number：202011091625

Report Query Name: RAMP_STAKING

Smart Contract Name:

RampStaking

Smart Contract Link:

https://github.com/RAMP-DEFI/RAMP_STAKING.git

Origin commit id: 6b3f438fc7dc8813bf5977363574fb5c9f245429

Final commit id: 5e45bceeaf1eb7b3750e62cda936266036e2511f

Start Date：2020.11.05

Completion Date：2020.11.09

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 project RAMP_STAKING, including Coding Standards, Security, and Business Logic. **The RAMP_STAKING project 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

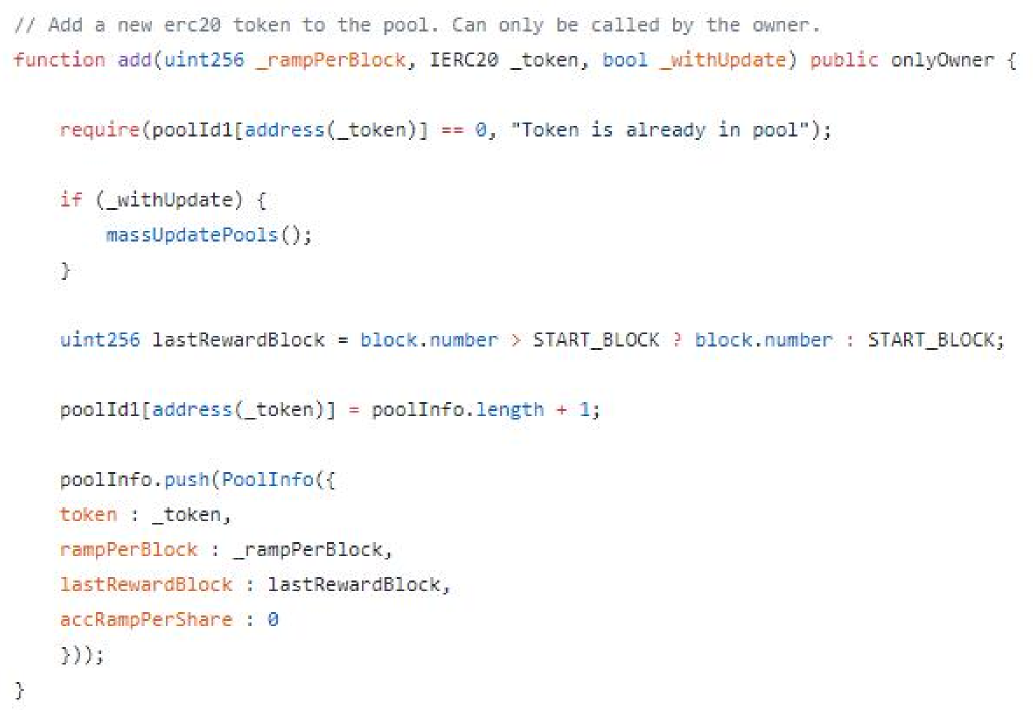
2.11 Overriding Variables

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

3. Business Security

(1) *add* function

- Description: As shown in Figure 1 below, the contract owner can call *add* function to add the Pool for the user to stake for getting the reward and store the pool-related information, requiring the staking token of the pool is the first time to be added.



```
// Add a new erc20 token to the pool. Can only be called by the owner.
function add(uint256 _rampPerBlock, IERC20 _token, bool _withUpdate) public onlyOwner {

    require(poolId1[address(_token)] == 0, "Token is already in pool");

    if (_withUpdate) {
        massUpdatePools();
    }

    uint256 lastRewardBlock = block.number > START_BLOCK ? block.number : START_BLOCK;

    poolId1[address(_token)] = poolInfo.length + 1;

    poolInfo.push(PoolInfo({
    token : _token,
    rampPerBlock : _rampPerBlock,
    lastRewardBlock : lastRewardBlock,
    accRampPerShare : 0
    }));
}
```

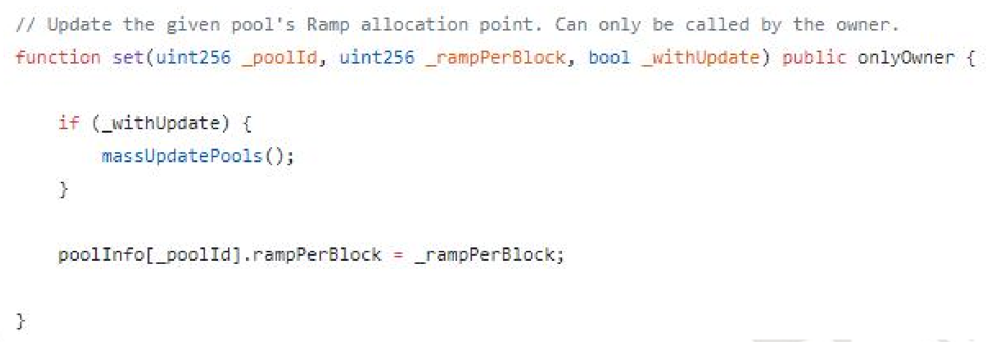
Figure 1 Source code of function *add*

- Related functions: *add, massUpdatePools*

- Result: Pass

(2) *set* function

- Description: As shown in Figure 2 below, the contract owner can set the number of rewards for each block of the specified pool. After the pool rewards for each block is modified, it will affect the value of RAMP rewards when users withdraw or deposit tokens.



```
// Update the given pool's Ramp allocation point. Can only be called by the owner.
function set(uint256 _poolId, uint256 _rampPerBlock, bool _withUpdate) public onlyOwner {

    if (_withUpdate) {
        massUpdatePools();
    }

    poolInfo[_poolId].rampPerBlock = _rampPerBlock;

}
```

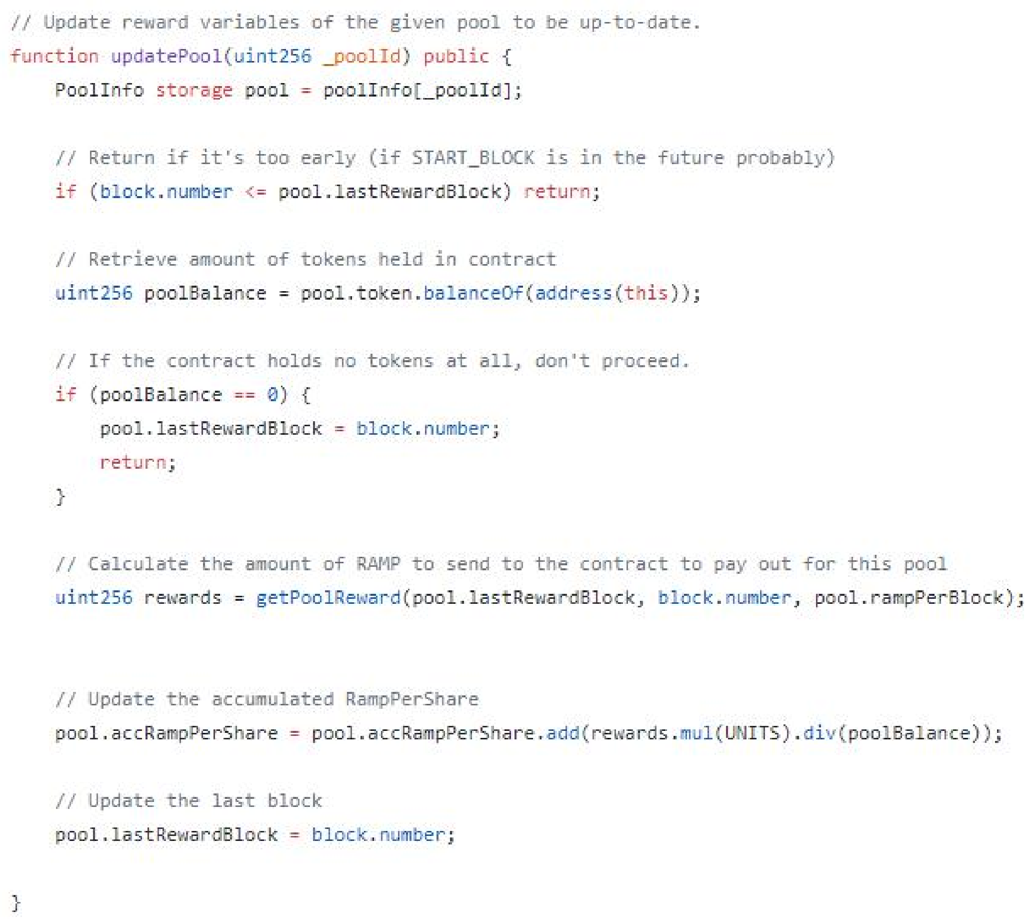
Figure 2 Source code of function *set*

- Related functions: *set, massUpdatePools*

- Result: Pass

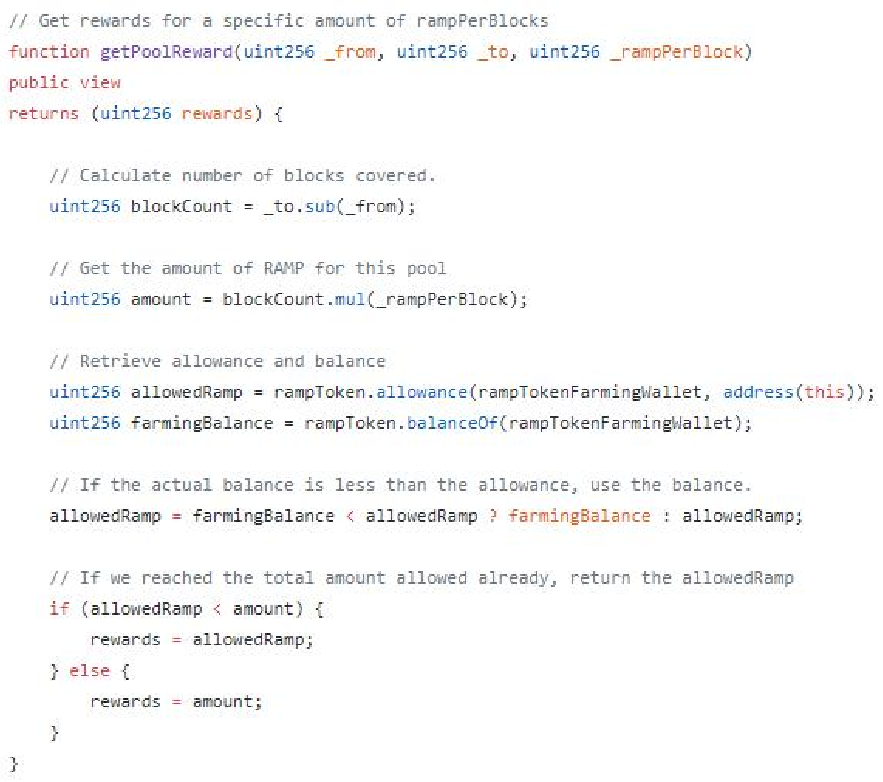
(3) *updatePool* function

- Description: As shown in Figure 3 below, any user can call *updatePool* function to update latest pool RAMP rewards and information of current block.



```
// Update reward variables of the given pool to be up-to-date.
function updatePool(uint256 _poolId) public {
    PoolInfo storage pool = poolInfo[_poolId];

    // Return if it's too early (if START_BLOCK is in the future probably)
    if (block.number <= pool.lastRewardBlock) return;

    // Retrieve amount of tokens held in contract
    uint256 poolBalance = pool.token.balanceOf(address(this));

    // If the contract holds no tokens at all, don't proceed.
    if (poolBalance == 0) {
        pool.lastRewardBlock = block.number;
        return;
    }

    // Calculate the amount of RAMP to send to the contract to pay out for this pool
    uint256 rewards = getPoolReward(pool.lastRewardBlock, block.number, pool.rampPerBlock);


    // Update the accumulated RampPerShare
    pool.accRampPerShare = pool.accRampPerShare.add(rewards.mul(UNITS).div(poolBalance));

    // Update the last block
    pool.lastRewardBlock = block.number;

}
```

Figure 3 Source code of function *updatePool*



```
// Get rewards for a specific amount of rampPerBlocks
function getPoolReward(uint256 _from, uint256 _to, uint256 _rampPerBlock)
public view
returns (uint256 rewards) {

    // Calculate number of blocks covered.
    uint256 blockCount = _to.sub(_from);

    // Get the amount of RAMP for this pool
    uint256 amount = blockCount.mul(_rampPerBlock);

    // Retrieve allowance and balance
    uint256 allowedRamp = rampToken.allowance(rampTokenFarmingWallet, address(this));
    uint256 farmingBalance = rampToken.balanceOf(rampTokenFarmingWallet);

    // If the actual balance is less than the allowance, use the balance.
    allowedRamp = farmingBalance < allowedRamp ? farmingBalance : allowedRamp;

    // If we reached the total amount allowed already, return the allowedRamp
    if (allowedRamp < amount) {
        rewards = allowedRamp;
    } else {
        rewards = amount;
    }
}
```

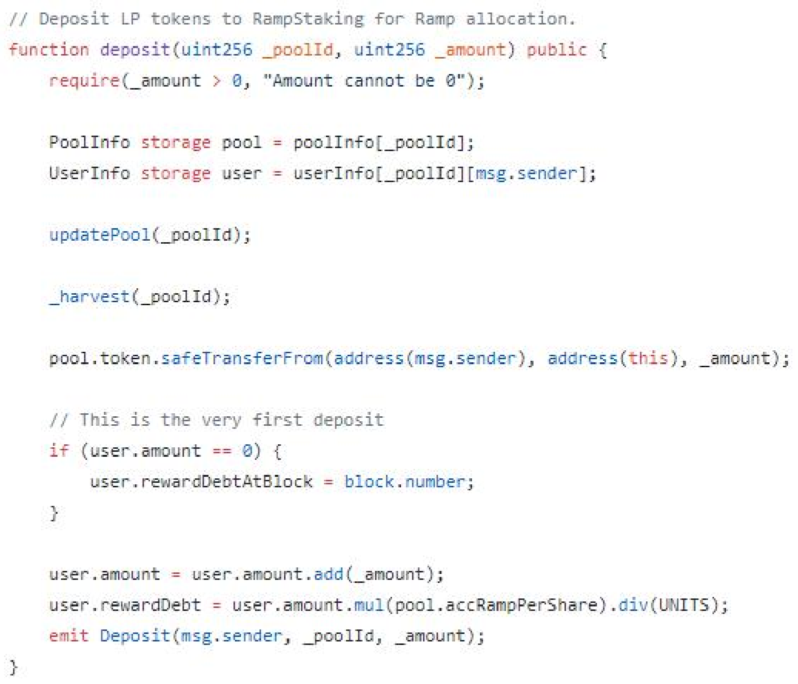
Figure 4 Source code of function *getPoolReward*

- Related functions: *updatePool, massUpdatePools, getPoolReward*

- Result: Pass

(4) *deposit* function

- Description: As shown in Figure 5 below, the contract implements the *deposit* function for users to stake tokens, the user pre-approves this contract address and then calls this function to deposit tokens (require the pool is existed). Update the pool information and the user deposit information when the user is deposited, if the user has previous deposit, calculate the user's previous deposit reward and send the reward to the user address.



```
// Deposit LP tokens to RampStaking for Ramp allocation.
function deposit(uint256 _poolId, uint256 _amount) public {
    require(_amount > 0, "Amount cannot be 0");

    PoolInfo storage pool = poolInfo[_poolId];
    UserInfo storage user = userInfo[_poolId][msg.sender];

    updatePool(_poolId);

    _harvest(_poolId);

    pool.token.safeTransferFrom(address(msg.sender), address(this), _amount);

    // This is the very first deposit
    if (user.amount == 0) {
        user.rewardDebtAtBlock = block.number;
    }

    user.amount = user.amount.add(_amount);
    user.rewardDebt = user.amount.mul(pool.accRampPerShare).div(UNITS);
    emit Deposit(msg.sender, _poolId, _amount);
}
```

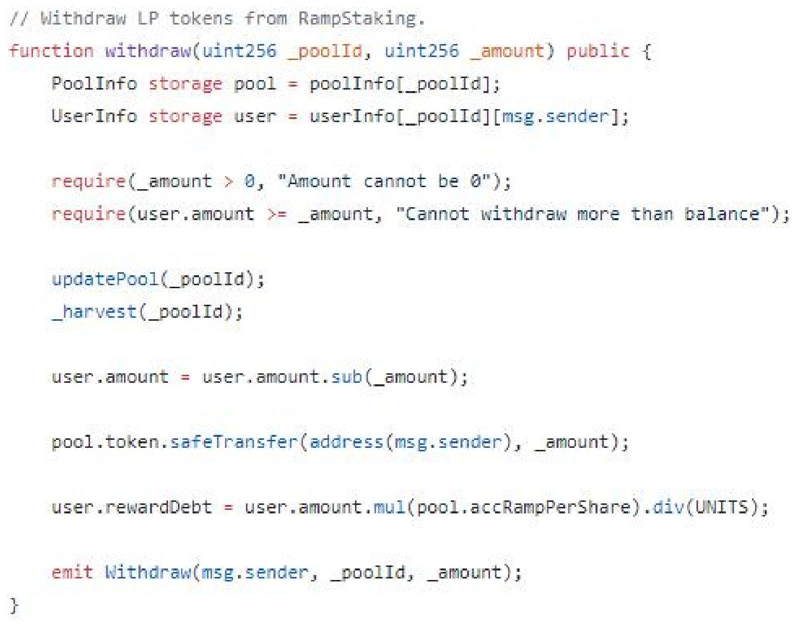
Figure 5 Source code of function *deposit*

- Related functions: *deposit, updatePool*

- Result: Pass

(5) *withdraw* function

- Description: As shown in Figure 6 below, the user can call *withdraw* function to withdraw the specified amount of deposit tokens and all RAMP reward in the current block (require the pool is existed). Update pool information when users withdraw deposit tokens and RAMP rewards, and transfer the specified deposit tokens and RAMP rewards to the user address and update the user deposit information.



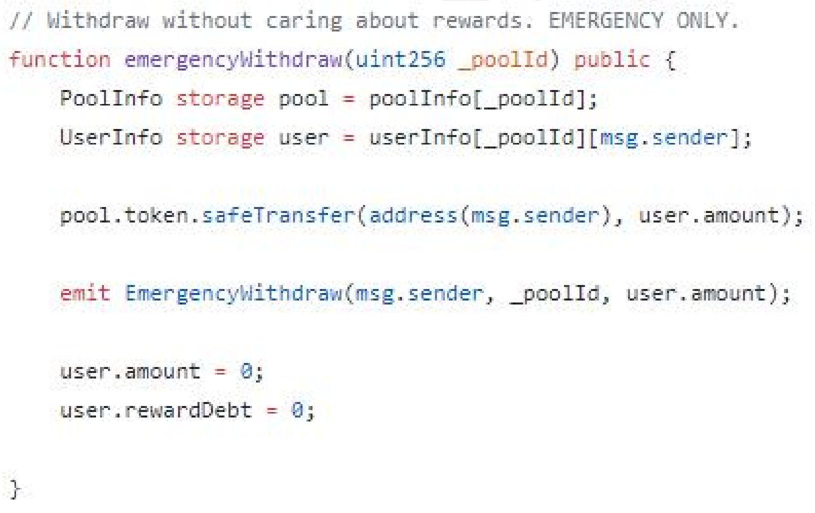
```
// Withdraw LP tokens from RampStaking.
function withdraw(uint256 _poolId, uint256 _amount) public {
    PoolInfo storage pool = poolInfo[_poolId];
    UserInfo storage user = userInfo[_poolId][msg.sender];

    require(_amount > 0, "Amount cannot be 0");
    require(user.amount >= _amount, "Cannot withdraw more than balance");

    updatePool(_poolId);
    _harvest(_poolId);

    user.amount = user.amount.sub(_amount);

    pool.token.safeTransfer(address(msg.sender), _amount);

    user.rewardDebt = user.amount.mul(pool.accRampPerShare).div(UNITS);

    emit Withdraw(msg.sender, _poolId, _amount);
}
```

Figure 6 Source code of function *withdraw*

- Related functions: *withdraw, updatePool*
- Result: Pass

(6) *emergencyWithdraw* function

- Description: As shown in Figure 7 below, the user can call *emergencyWithdraw* function to withdraw all deposited tokens (require the pool is existed). Update user deposit information and transfer all deposited tokens to the user address (Note: calling this function cannot get any deposit rewards).



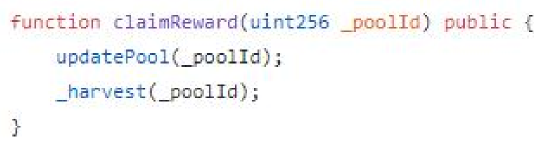
```
// Withdraw without caring about rewards. EMERGENCY ONLY.
function emergencyWithdraw(uint256 _poolId) public {
    PoolInfo storage pool = poolInfo[_poolId];
    UserInfo storage user = userInfo[_poolId][msg.sender];

    pool.token.safeTransfer(address(msg.sender), user.amount);

    emit EmergencyWithdraw(msg.sender, _poolId, user.amount);

    user.amount = 0;
    user.rewardDebt = 0;

}
```

Figure 7 Source code of function *emergencyWithdraw*

- Related functions: *emergencyWithdraw*
- Result: Pass

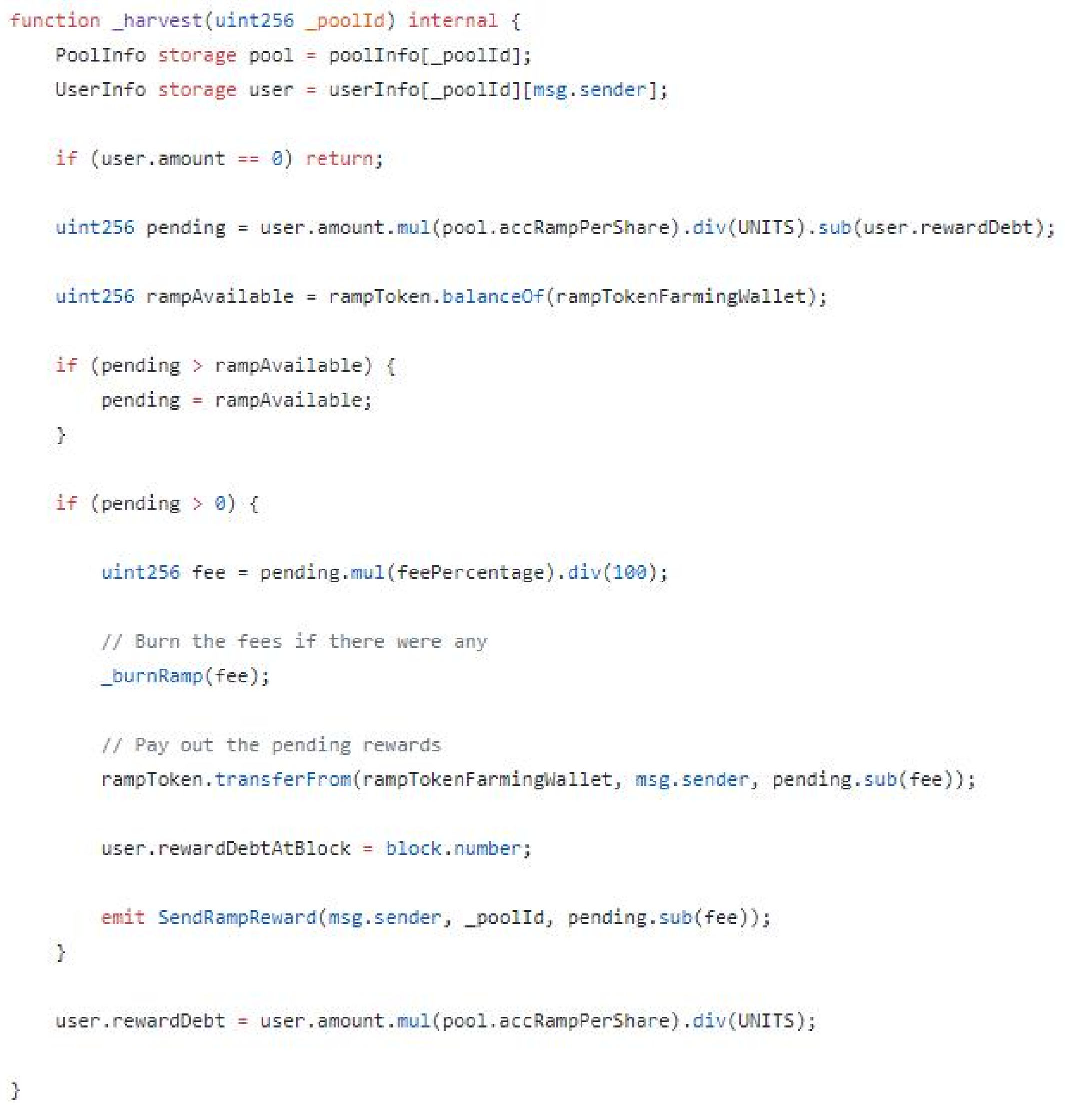
(7) *claimReward* function

- Description: As shown in Figure 8 below, the user can manually claim the RAMP reward through the *claimReward* function. Before claiming the reward, the *updatePool* function will be called to update the pool's cumulative reward information, and then the internal function *_harvest* (Figure9) will be called to claim the reward.



```
function claimReward(uint256 _poolId) public {
    updatePool(_poolId);
    _harvest(_poolId);
}
```

Figure 8 Source code of function *claimReward*



```
function _harvest(uint256 _poolId) internal {
    PoolInfo storage pool = poolInfo[_poolId];
    UserInfo storage user = userInfo[_poolId][msg.sender];

    if (user.amount == 0) return;

    uint256 pending = user.amount.mul(pool.accRampPerShare).div(UNITS).sub(user.rewardDebt);

    uint256 rampAvailable = rampToken.balanceOf(rampTokenFarmingWallet);

    if (pending > rampAvailable) {
        pending = rampAvailable;
    }

    if (pending > 0) {

        uint256 fee = pending.mul(feePercentage).div(100);

        // Burn the fees if there were any
        _burnRamp(fee);

        // Pay out the pending rewards
        rampToken.transferFrom(rampTokenFarmingWallet, msg.sender, pending.sub(fee));

        user.rewardDebtAtBlock = block.number;

        emit SendRampReward(msg.sender, _poolId, pending.sub(fee));
    }

    user.rewardDebt = user.amount.mul(pool.accRampPerShare).div(UNITS);

}
```

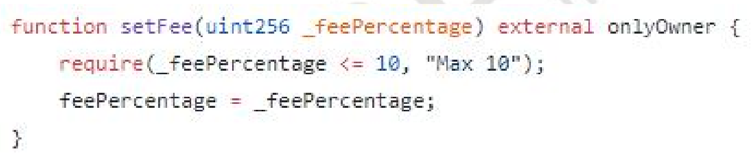
Figure 9 Source code of function *_harvest*

- Related functions: *claimReward, updatePool*

- Result: Pass

(8) *setFee* function

● Description: As shown in Figure 10 below, the contract owner can set the fee percentage for the reward. The current maximum fee percentage for the reward is 10%.



```
function setFee(uint256 _feePercentage) external onlyOwner {
    require(_feePercentage <= 10, "Max 10");
    feePercentage = _feePercentage;
}
```

Figure 10 Source code of function *setFee*

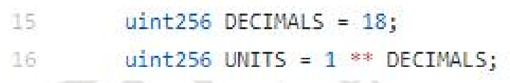
● Related functions: *set*

● Result: Pass

4. Details of audit results

4.1 UNITS

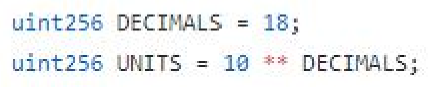
- Description: 1**18 is still 1.



```
uint256 DECIMALS = 18;
uint256 UNITS = 1 ** DECIMALS;
```

Figure 11 The origin source code of the definition of *UNITS* variable

- Suggestion for modification: It recommended to modify to 10**DECIMALS.
- Fix Result: Fixed. The final code is shown below.

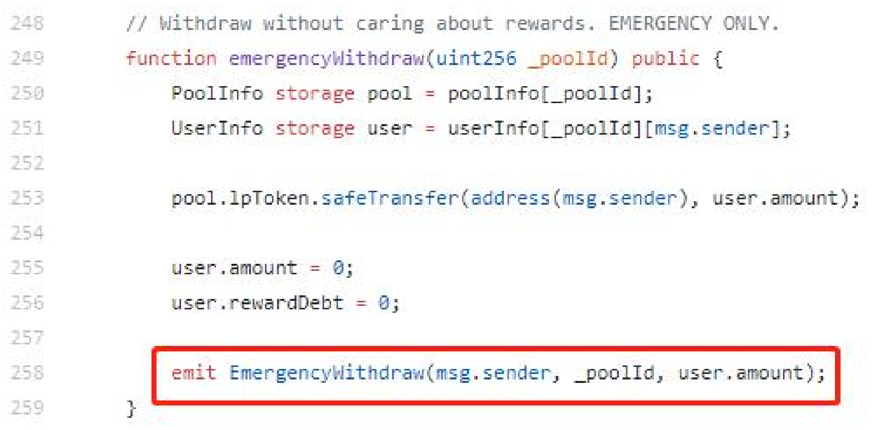


```
uint256 DECIMALS = 18;
uint256 UNITS = 10 ** DECIMALS;
```

Figure 12 The final source code of the definition of *UNITS* variable

4.2 emergencyWithdraw

- Description: When *EmergencyWithdraw* event is triggered, user.amount has been reset to 0.



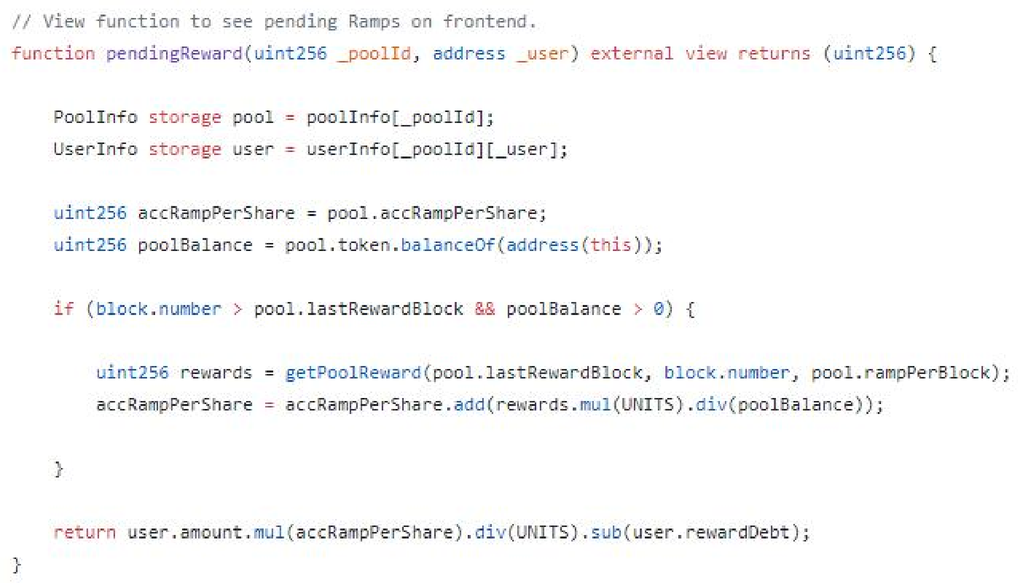
```
248     // Withdraw without caring about rewards. EMERGENCY ONLY.
249     function emergencyWithdraw(uint256 _poolId) public {
250         PoolInfo storage pool = poolInfo[_poolId];
251         UserInfo storage user = userInfo[_poolId][msg.sender];
252
253         pool.lpToken.safeTransfer(address(msg.sender), user.amount);
254
255         user.amount = 0;
256         user.rewardDebt = 0;
257
258         emit EmergencyWithdraw(msg.sender, _poolId, user.amount);
259     }
```

Figure 13 The origin source code of function *emergencyWithdraw*

- Suggestion for modification: It is recommended to trigger the event before line 255.
- Fix Result: Fixed.

4.3 pendingReward

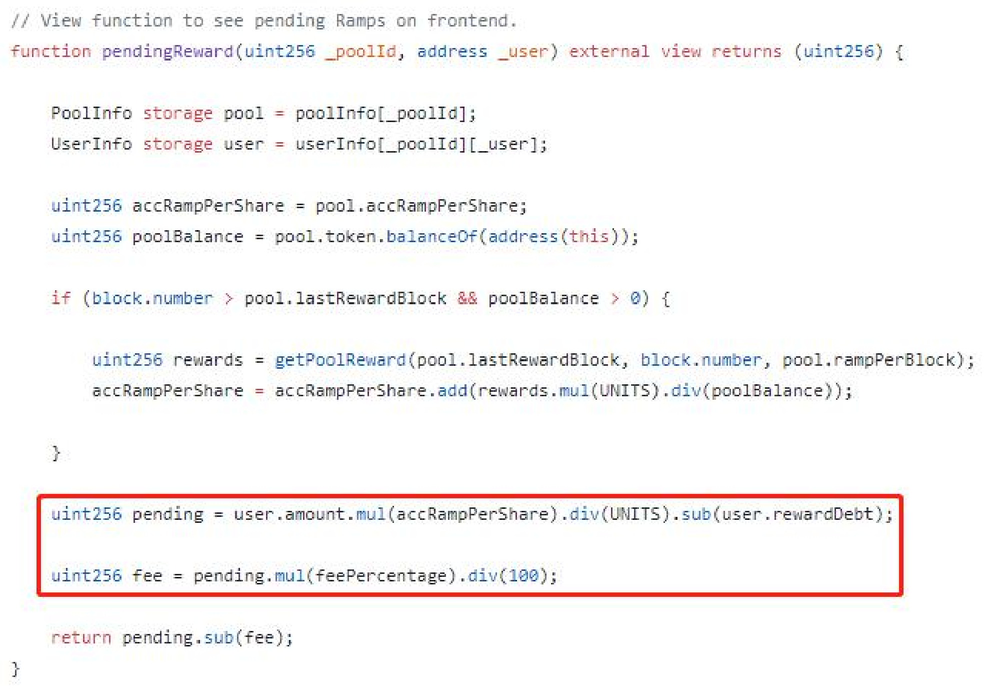
- Description: The return value of the *pendingReward* function is not deducted from the fee, which will cause the number of rewards that the user can query through this function to exceed the actual amount of RAMP rewards that can be claimed.



```
// View function to see pending Ramps on frontend.
function pendingReward(uint256 _poolId, address _user) external view returns (uint256) {

    PoolInfo storage pool = poolInfo[_poolId];
    UserInfo storage user = userInfo[_poolId][_user];

    uint256 accRampPerShare = pool.accRampPerShare;
    uint256 poolBalance = pool.token.balanceOf(address(this));

    if (block.number > pool.lastRewardBlock && poolBalance > 0) {

        uint256 rewards = getPoolReward(pool.lastRewardBlock, block.number, pool.rampPerBlock);
        accRampPerShare = accRampPerShare.add(rewards.mul(UNITS).div(poolBalance));

    }

    return user.amount.mul(accRampPerShare).div(UNITS).sub(user.rewardDebt);
}
```

Figure 14 The origin source code of function *pendingReward*

- Suggestion for modification: It is recommended that the return value of the *pendingReward* function deducted the fee.
- Fix Result: Fixed. The final code is shown below.



```
// View function to see pending Ramps on frontend.
function pendingReward(uint256 _poolId, address _user) external view returns (uint256) {

    PoolInfo storage pool = poolInfo[_poolId];
    UserInfo storage user = userInfo[_poolId][_user];

    uint256 accRampPerShare = pool.accRampPerShare;
    uint256 poolBalance = pool.token.balanceOf(address(this));

    if (block.number > pool.lastRewardBlock && poolBalance > 0) {

        uint256 rewards = getPoolReward(pool.lastRewardBlock, block.number, pool.rampPerBlock);
        accRampPerShare = accRampPerShare.add(rewards.mul(UNITS).div(poolBalance));

    }

    uint256 pending = user.amount.mul(accRampPerShare).div(UNITS).sub(user.rewardDebt);

    uint256 fee = pending.mul(feePercentage).div(100);

    return pending.sub(fee);
}
```

Figure 15 The final source code of function *pendingReward*

5. Conclusion

Beosin(ChengduLianAn) conducted a detailed audit on the design and code implementation of the project RAMP_STAKING. All the problems found in the audit process were notified to the project party, and got quick feedback and repair from the project party. Beosin (Chengdu LianAn) confirms that all the problems found have been properly fixed or have reached an agreement with the project party has on how to deal with it. **The overall audit result of the project RAMP_STAKING is Pass.**

BEOSIN

Blockchain Security

Official Website

https://lianantech.com

E-mail

vaas@lianantech.com

Twitter

https://twitter.com/Beosin_com