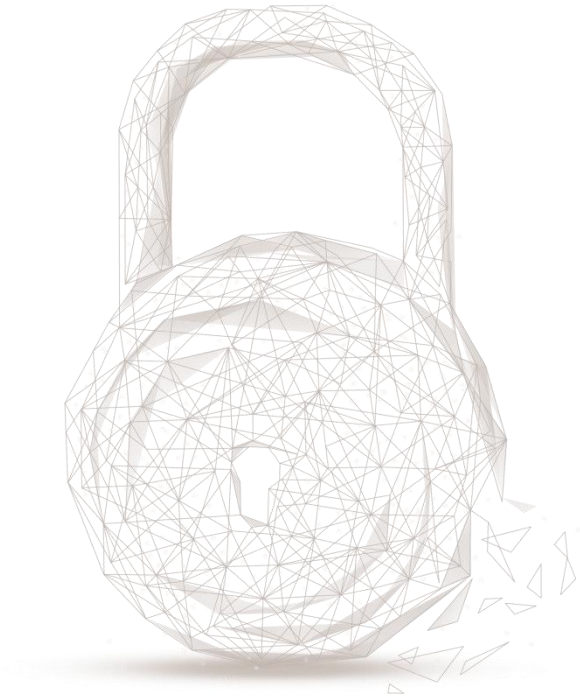
BEOSIN
Blockchain Security

Smart contract security audit report



Audit Number：202104221503

Report Query Name: steakbank

Audit Link:

https://github.com/steakbankfinance/steakbank-contract/tree/master/contracts

Audit contract:

BlindFarmingCenter.sol

FarmRewardLock.sol

FarmingCenter.sol

Commit hash:

330dd5924e2b2ac6218fae96afc2cc3e4885e59b

Start Date：2021.04.19

Completion Date：2021.04.22

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

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 project steakbank, including Coding Standards, Security, and Business Logic. **The steakbank project passed all audit items. The overall result is Pass. The smart contract is able to function properly.**

Audit Contents:

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 whether 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 BlindFarmingCenter

(1) initialize

- Description: The contract implements *initialize* for the contract to initialize the sbf address and owner, there is no permission requirement, it can only be called once and should be called immediately after the contract is deployed.

```
48      function initialize(
49          address _owner,
50          IBEP20 _sbf
51      ) public
52      {
53          require(!initialized, "already initialized");
54          initialized = true;
55
56          super.initializeOwner(_owner);
57          sbf = _sbf;
58          sbfPerBlock = 0;
59          startBlock = 0;
60          endBlock = 0;
61          releaseHeight = uint256(-1);
62      }
```

Figure 1 source code of *initialize*

- Related functions: *initialize*
- Result: Pass

(2) Set farm

- Description: The contract implements *startBindFarming* for the owner to set the reward cycle, requiring the current time to be less than the cycle start time and the cycle end time to be less than the release time to calculate and send the reward ahead to this contract.

```
64      function startBindFarming(uint256 sbfRewardPerBlock, uint256 startHeight, uint256 farmingPeriod) public onlyOwner {
65          require(block.number < startHeight, "startHeight must be larger than current block height");
66          require(startHeight.add(farmingPeriod) < releaseHeight, "farming endHeight must be less than releaseHeight");
67          massUpdatePools();
68
69          uint256 sbfAmount = sbfRewardPerBlock.mul(farmingPeriod);
70          sbf.safeTransferFrom(msg.sender, address(this), sbfAmount);
71          sbfPerBlock = sbfRewardPerBlock;
72          startBlock = startHeight;
73          endBlock = startHeight.add(farmingPeriod);
74
75          for (uint256 pid = 0; pid < poolInfo.length; ++pid) {
76              PoolInfo storage pool = poolInfo[pid];
77              pool.lastRewardBlock = startHeight;
78          }
79      }
```

Figure 2 source code of *startBindFarming*

- Related functions: *startBindFarming, massUpdatePools*
- Safety recommendation: If a reward cycle has been set up and is called before its end, it may cause confusion in the reward calculation, resulting in the last remaining part of the sbf in the contract not being taken out. It is suggested to add a judgment statement that requires the current time to be greater than the endBlock.
- Repair result: Ignore, the project declares that the function will be called only once.
- Result: Pass

(3) Increase reward

- Description: The contract implements the *increaseBlindFarmingReward* function for increasing the reward per block for the reward cycle; the *increaseBlindFarmingPeriod* function is used to round up the duration of the reward cycle. Both functions can be called only by the owner and will calculate and send the increased reward to the contract.

```
81      function increaseBlindFarmingReward(uint256 increasedRewardPerBlock) public onlyOwner {
82          require(block.number < endBlock, "Previous farming is already completed");
83          massUpdatePools();
84
85          uint256 sbfAmount = increasedRewardPerBlock.mul(endBlock.sub(block.number));
86          sbf.safeTransferFrom(msg.sender, address(this), sbfAmount);
87          sbfPerBlock = sbfPerBlock.add(increasedRewardPerBlock);
88      }
89
90      function increaseBlindFarmingPeriod(uint256 increasedBlockNumber) public onlyOwner {
91          require(block.number < endBlock, "Previous farming is already completed");
92          massUpdatePools();
93
94          uint256 sbfAmount = sbfPerBlock.mul(increasedBlockNumber);
95          sbf.safeTransferFrom(msg.sender, address(this), sbfAmount);
96          endBlock = endBlock.add(increasedBlockNumber);
97      }
```

Figure 3 source code of *increaseBlindFarmingReward* and *increaseBlindFarmingPeriod*

- Related functions: *increaseBlindFarmingReward, increaseBlindFarmingPeriod*
- Result: Pass

(4) Add new pool

- Description: The contract implements the *add* function for the owner to add new lp tokens to the stake pool, not to add sbf as the stake pool for lp tokens, and to choose whether to update the stake pool when adding. To avoid errors in the reward statistics, it is recommended that *_withUpdate* be set to true. Note: Adding a stake pool of the same lp token will cause an error in the reward calculation.

```
103     function add(uint256 _allocPoint, IBEP20 _lpToken, bool _withUpdate) public onlyOwner {
104         require(_lpToken!=sbf, "can't support SBF pool");
105         if (_withUpdate) {
106             massUpdatePools();
107         }
108         uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
109         totalAllocPoint = totalAllocPoint.add(_allocPoint);
110         poolInfo.push(PoolInfo({
111             lpToken: _lpToken,
112             allocPoint: _allocPoint,
113             lastRewardBlock: lastRewardBlock,
114             accSBFPerShare: 0
115         }));
116     }
```

Figure 4 source code of *add*

- Related functions: *add, massUpdatePools*
- Safety recommendation: Add check for adding duplicate lp tokens.
- Repair result: Ignore, the project states that this will be changed in a later version.
- Result: Pass

(5) Set point

- Description: The contract implements the *set* function for the owner to modify the points of the specified collateral pool, which requires the existence of the specified pool, and can choose whether to execute *massUpdatePools*; after that, it determines whether the totalAllocPoint needs to be updated. To avoid errors in the reward statistics, it is recommended that *_withUpdate* be set to true.

```
118     function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
119         require(_pid < poolInfo.length, "invalid pool id");
120         if (_withUpdate) {
121             massUpdatePools();
122         }
123         uint256 prevAllocPoint = poolInfo[_pid].allocPoint;
124         poolInfo[_pid].allocPoint = _allocPoint;
125         if (prevAllocPoint != _allocPoint) {
126             totalAllocPoint = totalAllocPoint.sub(prevAllocPoint).add(_allocPoint);
127         }
128     }
```

Figure 5 source code of *set*

- Related functions: *set, massUpdatePools*
- Result: Pass

(6) Update all staking pool data function

- Description: The contract implements the *massUpdatePools* function to update all staking pool information by traversing all staking pools and calling the *updatePool* function.

```
154     function massUpdatePools() public {
155         uint256 length = poolInfo.length;
156         for (uint256 pid = 0; pid < length; ++pid) {
157             updatePool(pid);
158         }
159     }
```

Figure 6 source code of *massUpdatePools*

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

(7) Update specified staking pool data

- Description: The contract implements the *updatePool* function to update the specified staking pool information. Calling this function requires that the current block number is greater than the last reward block number, and the lp token balance of the specified staking pool in the contract is greater than 0. Finally update the relevant parameters of the staking pool.

```
161     function updatePool(uint256 _pid) public {
162         require(_pid < poolInfo.length, "invalid pool id");
163         PoolInfo storage pool = poolInfo[_pid];
164         if (block.number <= pool.lastRewardBlock) {
165             return;
166         }
167         uint256 lpSupply = pool.lpToken.balanceOf(address(this));
168         if (lpSupply == 0) {
169             pool.lastRewardBlock = block.number;
170             return;
171         }
172         uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
173         uint256 sbfReward = multiplier.mul(sbfPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
174         pool.accSBFPerShare = pool.accSBFPerShare.add(sbfReward.mul(REWARD_CALCULATE_PRECISION).div(lpSupply));
175         pool.lastRewardBlock = block.number;
176     }
```

Figure 7 source code of *updatePool*

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

(8) Deposit function

- Description: The contract implements the *deposit* function for users to deposit lp tokens for rewards. The user is required to stake the existence of the pool and update the pool information before pledging; if the user has previously staked, the sbf reward will be calculated; if the sbf reward is greater than 0, *rewardSBF* will be called to determine whether to lock the reward, and the relevant parameters of the user will be updated after the token is deposited.

```
178     function deposit(uint256 _pid, uint256 _amount) public {
179         require(_pid < poolInfo.length, "invalid pool id");
180         PoolInfo storage pool = poolInfo[_pid];
181         UserInfo storage user = userInfo[_pid][msg.sender];
182         updatePool(_pid);
183         uint256 pending;
184         if (user.amount > 0) {
185             pending = user.amount.mul(pool.accSBFPerShare).div(REWARD_CALCULATE_PRECISION).sub(user.rewardDebt);
186 
187             if (pending > 0) {
188                 pending = rewardSBF(msg.sender, pending);
189             }
190         }
191         if (_amount > 0) {
192             pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
193             user.amount = user.amount.add(_amount);
194         }
195         user.rewardDebt = user.amount.mul(pool.accSBFPerShare).div(REWARD_CALCULATE_PRECISION);
196         emit Deposit(msg.sender, _pid, _amount, pending);
197     }
```

Figure 8 source code of *deposit*

- Related functions: *deposit*, *updatePool*

- Result: Pass

(9) Withdraw function

- Description: The contract implements the *withdraw* function to enable users to withdraw the staked lp tokens from the specified staking pool. When the user calls this function, the current staking pool information will be updated first, the rewards should be calculated and *rewardSBF* will be called to determine whether to lock the reward, and the relevant parameters of the user will be updated after the token is withdrawn.

```
199     function withdraw(uint256 _pid, uint256 _amount) public {
200         require(_pid < poolInfo.length, "invalid pool id");
201         PoolInfo storage pool = poolInfo[_pid];
202         UserInfo storage user = userInfo[_pid][msg.sender];
203         uint256 reward;
204         require(user.amount >= _amount, "withdraw: not good");
205 
206         updatePool(_pid);
207         uint256 pending = user.amount.mul(pool.accSBFPerShare).div(REWARD_CALCULATE_PRECISION).sub(user.rewardDebt);
208 
209         if (pending > 0) {
210             pending = rewardSBF(msg.sender, pending);
211         }
212 
213         if (_amount > 0) {
214             user.amount = user.amount.sub(_amount);
215             pool.lpToken.safeTransfer(address(msg.sender), _amount);
216         }
217         user.rewardDebt = user.amount.mul(pool.accSBFPerShare).div(REWARD_CALCULATE_PRECISION);
218         emit Withdraw(msg.sender, _pid, _amount, pending);
219     }
```

Figure 9 source code of *withdraw*

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

(10) emergencyWithdraw function

- Description: The contract implements the *emergencyWithdraw* function for emergency withdrawal and exit. Users calling this function will withdraw all lp tokens staked in the specified staking pool without any reward.

```
221     function emergencyWithdraw(uint256 _pid) public {
222         require(_pid < poolInfo.length, "invalid pool id");
223         PoolInfo storage pool = poolInfo[_pid];
224         UserInfo storage user = userInfo[_pid][msg.sender];
225         pool.lpToken.safeTransfer(address(msg.sender), user.amount);
226         emit EmergencyWithdraw(msg.sender, _pid, user.amount);
227         user.amount = 0;
228         user.rewardDebt = 0;
229     }
```

Figure 10 source code of *emergencyWithdraw*

- Related functions: *emergencyWithdraw*
- Result: Pass

(11) Set release time

- Description: The contract implements *setReleaseHeight* for the owner to set the release time, requiring the release time to be greater than the current time.

```
241     function setReleaseHeight(uint256 newReleaseHeight) public onlyOwner {
242         require(newReleaseHeight > block.number, "release height must be larger than current height");
243         releaseHeight = newReleaseHeight;
244     }
```

Figure 11 source code of *setReleaseHeight*

- Related functions: *setReleaseHeight*
- Safety recommendation: ***setReleaseHeight* function has too high administrator privileges, arbitrarily modify the unlocking start time may lead to the user can not receive rewards, it is recommended to delete.**
- Repair result: Ignore, the project states that this will be changed in a later version.
- Result: Pass

(12) Receive referral rewards function

- Description: The contract implements the *claimReward* function for the caller to collect the locked reward, requiring that the current time is not less than the release time.

```
246     function claimReward() public {
247         require(block.number>=releaseHeight, "release height is not reached");
248         uint256 reward = userLockedRewardAmount[msg.sender];
249         require(reward>0, "no reward");
250         userLockedRewardAmount[msg.sender] = 0;
251         uint256 actualReward = safeTransferSBF(address(msg.sender), reward);
252         emit Reward(msg.sender, actualReward);
253     }
```

Figure 12 source code of *claimReward*

- Related functions:*claimReward*
- Result: Pass

3.2 Business analysis of Contract FarmingCenter

(1) initialize

- Description: The contract implements *initialize* for the contract to initialize related parameters and add an sbf stake pool. There is no permission requirement, it can only be called once and should be called immediately after the contract is deployed.

```
47      function initialize(
48          address _owner,
49          IBEP20 _sbf,
50          IFarmRewardLock _farmRewardLock,
51          uint256 _moleculerOfLockRate,
52          uint256 _denominatorOfLockRate11
53      ) public
54      {
55          require(!initialized, "already initialized");
56          initialized = true;
57
58          sbfRewardVault = new SBFRewardVault(_sbf, address(this));
59
60          super.initializeOwner(_owner);
61          farmRewardLock = _farmRewardLock;
62          sbf = _sbf;
63          sbfPerBlock = 0;
64          startBlock = 0;
65          endBlock = 0;
66
67          poolInfo.push(PoolInfo({
68              lpToken: IBEP20(address(_sbf)),
69              allocPoint: 1000,
70              lastRewardBlock: startBlock,
71              accSBFPerShare: 0,
72              moleculerOfLockRate: _moleculerOfLockRate,
73              denominatorOfLockRate: _denominatorOfLockRate
74          }));
75          totalAllocPoint = 1000;
76      }
```

Figure 13 source code of *initialize*

- Related functions: *initialize*
- Result: Pass

(2) addNewFarmingPeriod function

- Description: The contract implements the *addNewFarmingPeriod* function for the owner to set a new reward cycle, requiring the current time to be greater than endBlock and not greater than startHeight, sending the sbf of this cycle in advance to the sbfRewardVault address and updating the relevant parameters.

```
function addNewFarmingPeriod(uint256 farmingPeriod, uint256 startHeight, uint256 sbfRewardPerBlock) public onlyOwner {
    require(block.number > endBlock, "Previous farming is not completed yet");
    require(block.number <= startHeight, "Start height must be in the future");
    require(sbfRewardPerBlock > 0, "sbfRewardPerBlock must be larger than 0");
    require(farmingPeriod > 0, "farmingPeriod must be larger than 0");

    massUpdatePools();

    uint256 totalSBFAmount = farmingPeriod.mul(sbfRewardPerBlock);
    sbf.safeTransferFrom(msg.sender, address(sbfRewardVault), totalSBFAmount);

    sbfPerBlock = sbfRewardPerBlock;
    startBlock = startHeight;
    endBlock = startHeight.add(farmingPeriod);

    for (uint256 pid = 0; pid < poolInfo.length; ++pid) {
        PoolInfo storage pool = poolInfo[pid];
        pool.lastRewardBlock = startHeight;
    }
}
```

Figure 14 source code of *addNewFarmingPeriod*

- Related functions: *addNewFarmingPeriod*
- Result: Pass

(3) Pool manage

- Description: The contract implements the *add* function for the owner to add new lp tokens to the stake pool, and to choose whether to update the stake pool when adding. Note: Adding a stake pool of the same lp token will cause an error in the reward calculation; The *set* function for the owner to modify the points of the specified collateral pool, which requires the existence of the specified pool, and can choose whether to execute massUpdatePools; after that, it determines whether the totalAllocPoint needs to be updated. Both functions will call *updateSBFPool* at the end to ensure that the sbf collateral pool does not fall below 20% of the total number of points. To avoid errors in the reward statistics, it is recommended that *_withUpdate* be set to true.

```
function add(uint256 _allocPoint, IBEP20 _lpToken, bool _withUpdate, uint256 molecularOfLockRate, uint256 denominatorOfLockRate) public
onlyOwner {
    require(denominatorOfLockRate>0&&denominatorOfLockRate>=molecularOfLockRate, "invalid denominatorOfLockRate or molecularOfLockRate");
    if (_withUpdate) {
        massUpdatePools();
    }
    uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolInfo.push(PoolInfo({
        lpToken: _lpToken,
        allocPoint: _allocPoint,
        lastRewardBlock: lastRewardBlock,
        accSBFPerShare: 0,
        molecularOfLockRate: molecularOfLockRate,
        denominatorOfLockRate: denominatorOfLockRate
        }));
    updateSBFPool();
}

function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
    require(_pid < poolInfo.length, "invalid pool id");
    if (_withUpdate) {
        massUpdatePools();
    }
    uint256 prevAllocPoint = poolInfo[_pid].allocPoint;
    poolInfo[_pid].allocPoint = _allocPoint;
    if (prevAllocPoint != _allocPoint) {
        totalAllocPoint = totalAllocPoint.sub(prevAllocPoint).add(_allocPoint);
        updateSBFPool();
    }
}
```

Figure 15 source code of *add* and *set*

- Related functions: *add, set*
- Safety recommendation: Add check for adding duplicate lp tokens.
- Repair result: Ignore, the project states that this will be changed in a later version.

- Result: Pass

(4) Update pool

- Description: The contract implements the *massUpdatePools* function to update all staking pool information by traversing all staking pools and calling the *updatePool* function; *updatePool* function to update the specified staking pool information. Calling this function requires that the current block number is greater than the last reward block number, and the lp token balance of the specified staking pool in the contract is greater than 0. Finally update the relevant parameters of the staking pool.

```
181     function massUpdatePools() public {
182         uint256 length = poolInfo.length;
183         for (uint256 pid = 0; pid < length; ++pid) {
184             updatePool(pid);
185         }
186     }
187 
188     function updatePool(uint256 _pid) public {
189         require(_pid < poolInfo.length, "invalid pool id");
190         PoolInfo storage pool = poolInfo[_pid];
191         if (block.number <= pool.lastRewardBlock) {
192             return;
193         }
194         uint256 lpSupply = pool.lpToken.balanceOf(address(this));
195         if (lpSupply == 0) {
196             pool.lastRewardBlock = block.number;
197             return;
198         }
199         uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
200         uint256 sbfReward = multiplier.mul(sbfPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
201         pool.accSBFPerShare = pool.accSBFPerShare.add(sbfReward.mul(REWARD_CALCULATE_PRECISION).div(lpSupply));
202         pool.lastRewardBlock = block.number;
203     }
```

Figure 16 source code of *massUpdatePools* and *updatePool*

- Related functions: *massUpdatePools, updatePool, getMultiplier*
- Result: Pass

(5) Deposit and Withdrawal

- Description: The contract implements the *deposit* function for users to deposit lp tokens for rewards. The user is required to stake the existence of the pool and update the pool information before pledging; if the user has previously staked, the sbf reward will be calculated; if the sbf reward is greater than 0, *rewardSBF* will be called to determine if a locked reward is needed and send the locked reward to the farmRewardLock contract after calculation, and the relevant parameters of the user will be updated after the token is deposited; *withdraw* function to enable users to withdraw the staked lp tokens from the specified staking pool. When the user calls this function, the current staking pool information will be updated first, the rewards should be calculated and *rewardSBF* will be called to determine if a locked reward is needed and send the locked reward to the farmRewardLock contract after calculation, and the relevant parameters of the user will be updated after the token is withdrawn; *emergencyWithdraw* function for emergency withdrawal and exit. Users calling this function will withdraw all lp tokens staked in the specified staking pool without any reward.

```
205     function deposit(uint256 _pid, uint256 _amount) public {
206         require(_pid < poolInfo.length, "invalid pool id");
207         PoolInfo storage pool = poolInfo[_pid];
208         UserInfo storage user = userInfo[_pid][msg.sender];
209         updatePool(_pid);
210         uint256 reward;
211         uint256 lockedReward;
212         if (user.amount > 0) {
213             uint256 pending = user.amount.mul(pool.accSBFPerShare).div(REWARD_CALCULATE_PRECISION).sub(user.rewardDebt);
214
215             if (pending > 0) {
216                 (reward, lockedReward) = rewardSBF(msg.sender, pending, pool.molecularOfLockRate, pool.denominatorOfLockRate);
217             }
218         }
219         if (_amount > 0) {
220             pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
221             user.amount = user.amount.add(_amount);
222         }
223         user.rewardDebt = user.amount.mul(pool.accSBFPerShare).div(REWARD_CALCULATE_PRECISION);
224         emit Deposit(msg.sender, _pid, _amount, reward, lockedReward);
225     }
226
227     function withdraw(uint256 _pid, uint256 _amount) public {
228         require(_pid < poolInfo.length, "invalid pool id");
229         PoolInfo storage pool = poolInfo[_pid];
230         UserInfo storage user = userInfo[_pid][msg.sender];
231         uint256 reward;
232         uint256 lockedReward;
233         require(user.amount >= _amount, "withdraw: not good");
234
235         updatePool(_pid);
236         uint256 pending = user.amount.mul(pool.accSBFPerShare).div(REWARD_CALCULATE_PRECISION).sub(user.rewardDebt);
237
238         if (pending > 0) {
239             (reward, lockedReward) = rewardSBF(msg.sender, pending, pool.molecularOfLockRate, pool.denominatorOfLockRate);
240         }
241
242         if (_amount > 0) {
243             user.amount = user.amount.sub(_amount);
244             pool.lpToken.safeTransfer(address(msg.sender), _amount);
245         }
246         user.rewardDebt = user.amount.mul(pool.accSBFPerShare).div(REWARD_CALCULATE_PRECISION);
247         emit Withdraw(msg.sender, _pid, _amount, reward, lockedReward);
248     }
249
250     function emergencyWithdraw(uint256 _pid) public {
251         require(_pid < poolInfo.length, "invalid pool id");
252         PoolInfo storage pool = poolInfo[_pid];
253         UserInfo storage user = userInfo[_pid][msg.sender];
254         pool.lpToken.safeTransfer(address(msg.sender), user.amount);
255         emit EmergencyWithdraw(msg.sender, _pid, user.amount);
256         user.amount = 0;
257         user.rewardDebt = 0;
258     }
```

Figure 17 source code of *deposit, withdraw,* and *emergencyWithdraw*

- Related functions: *deposit, withdraw, emergencyWithdraw*
- Result: Pass

3.3 Business analysis of Contract FarmRewardLock

(1) initialize

- Description: The contract implements *initialize* for the contract to initialize related parameters. There is no permission requirement, it can only be called once and should be called immediately after the contract is deployed.

```
41      function initialize(
42          IBEP20 _sbf,
43          uint256 _startReleaseHeight,
44          uint256 _releasePeriod,
45          address _farmingCenter,
46          address _owner
47      ) public
48      {
49          require(!initialized, "FarmRewardLock: already initialized");
50          initialized = true;
51
52          require(_releasePeriod>0, "FarmRewardLock: releasePeriod must be positive");
53
54          sbf = _sbf;
55          startReleaseHeight = _startReleaseHeight;
56          releasePeriod = _releasePeriod;
57          farmingCenter = _farmingCenter;
58          super.initializeOwner(_owner);
59      }
```

Figure 18 source code of *initialize*

- Related functions:
- Result: Pass

(2) Lock function

- Description: The contract implements *notifyDeposit* for FarmingCenter contract to lock the specified user's sbf rewards, requiring the current time is less than the release end time, will update the relevant parameters.

```
65      function notifyDeposit(address user, uint256 amount) onlyFarmingCenter override external returns (bool) {
66          require(block.number<startReleaseHeight.add(releasePeriod), "FarmRewardLock: should not deposit after lockEndHeight");
67
68          UserLockInfo storage lockInfo = userLockInfos[user];
69          if (block.number <= startReleaseHeight) {
70              lockInfo.lockedAmount = lockInfo.lockedAmount.add(amount);
71          } else {
72              uint256 lastUpdateHeight = lockInfo.lastUpdateHeight;
73              if (lastUpdateHeight == 0) {
74                  lastUpdateHeight = startReleaseHeight;
75              }
76              uint256 lastRestLockPeriod = startReleaseHeight.add(releasePeriod).sub(lastUpdateHeight);
77              uint256 newUnlockAmount = lockInfo.lockedAmount.mul(block.number-lastUpdateHeight).div(lastRestLockPeriod);
78              lockInfo.unlockedAmount = lockInfo.unlockedAmount.add(newUnlockAmount);
79              lockInfo.lockedAmount = lockInfo.lockedAmount.sub(newUnlockAmount).add(amount);
80              lockInfo.lastUpdateHeight = block.number;
81          }
82
83          emit DepositSBF(user, amount);
84          return true;
85      }
```

Figure 19 source code of *notifyDeposit*

- Related functions: *notifyDeposit*
- Result: Pass

(3) Claim function

- Description: The contract implements the *claim* function for the user to claim the locked rewards, first calling *unlockedAmount* to calculate the rewards that can be claimed, then updating the relevant parameters and the contract will send the rewards.

```
107     function claim() external returns (bool) {
108         (uint256 alreadyUnlockAmount, uint256 newUnlockAmount) = unlockedAmount(_msgSender());
109         uint256 claimAmount = alreadyUnlockAmount.add(newUnlockAmount);
110         require(claimAmount > 0, "FarmRewardLock: no locked reward");
111         UserLockInfo storage lockInfo = userLockInfos[_msgSender()];
112         lockInfo.lockedAmount = lockInfo.lockedAmount.sub(newUnlockAmount);
113         lockInfo.unlockedAmount = 0;
114         lockInfo.lastUpdateHeight = block.number;
115 
116         sbf.safeTransfer(_msgSender(), claimAmount);
117         return true;
118     }
```

Figure 20 source code of *claim*

- Related functions: *claim, unlockedAmount*
- Result: Pass

(4) Owner Functions

- Description: The contract implements the following functions for the owner to modify the relevant parameters: *setFarmingCenter* to modify the farmingCenter contract address; *setStartReleaseHeight* to modify the unlocking release start time; *setReleasePeriod* to modify the unlocking release duration.

```
120     function setFarmingCenter(address newFarmingCenter) onlyOwner external {
121         farmingCenter = newFarmingCenter;
122     }
123 
124     function setStartReleaseHeight(uint256 newStartReleaseHeight) onlyOwner external {
125         startReleaseHeight = newStartReleaseHeight;
126     }
127 
128     function setReleasePeriod(uint256 newReleasePeriod) onlyOwner external {
129         releasePeriod = newReleasePeriod;
130     }
```

Figure 21 source code of *setFarmingCenter, setStartReleaseHeight, setReleasePeriod*

- Related functions: *setFarmingCenter, setStartReleaseHeight, setReleasePeriod*
- Safety recommendation: ***setStartReleaseHeight* function has too high administrator privileges, arbitrarily modify the unlocking start time may lead to the user can not receive rewards, it is recommended to delete.**
- Repair result: Ignore, the project states that this will be changed in a later version.
- Result: Pass

(5) Query functions

- Description: The contract implements the *getLockEndHeight* function for querying the locking cycle end time and *unlockedAmount* for querying the rewards available for unlocking at the specified address.

```
61      function getLockEndHeight() override external view returns (uint256) {
62          return startReleaseHeight.add(releasePeriod);
63      }
```

Figure 22 source code of *getLockEndHeight*

```
87      function unlockedAmount(address userAddr) public view returns (uint256, uint256) {
88          if (block.number <= startReleaseHeight) {
89              return (0, 0);
90          } else if (block.number >= startReleaseHeight.add(releasePeriod)) {
91              UserLockInfo memory lockInfo = userLockInfos[userAddr];
92              return (lockInfo.unlockedAmount, lockInfo.lockedAmount);
93          }
94          UserLockInfo memory lockInfo = userLockInfos[userAddr];
95
96          uint256 lastUpdateHeight = lockInfo.lastUpdateHeight;
97          if (lastUpdateHeight == 0) {
98              lastUpdateHeight = startReleaseHeight;
99          }
100
101         uint256 lastRestLockPeriod = startReleaseHeight.add(releasePeriod).sub(lastUpdateHeight);
102         uint256 newUnlockAmount = lockInfo.lockedAmount.mul(block.number-lastUpdateHeight).div(lastRestLockPeriod);
103
104         return (lockInfo.unlockedAmount, newUnlockAmount);
105     }
```

Figure 23 source code of *unlockedAmount*

- Related functions: *getLockEndHeight*, *unlockedAmount*
- Result: Pass

4. Conclusion

Beosin(ChengduLianAn) conducted a detailed audit on the design and code implementation of the smart contracts project steakbank. The problems found by the audit team during the audit process have been notified to the project party and fixed, the overall audit result of the steakbank project's smart contract is **Pass**.

Official Website

https://lianantech.com

E-mail

vaas@lianantech.com

Twitter

https://twitter.com/Beosin_com