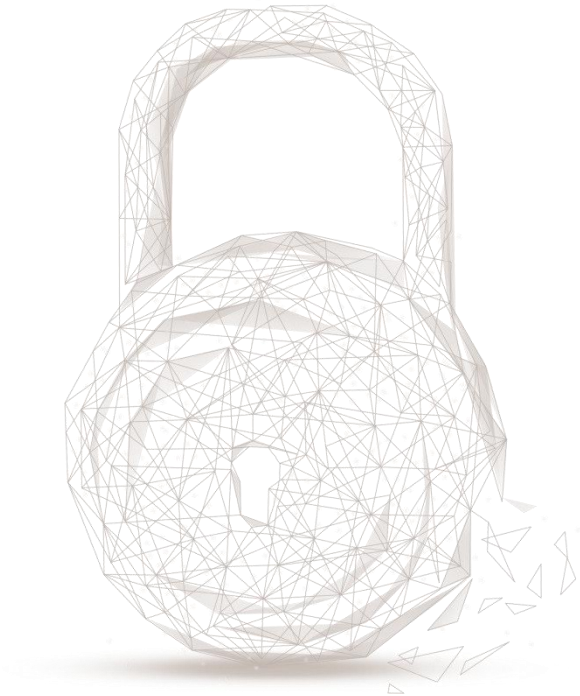
BEOSIN
Blockchain Security

Smart contract security audit report



Audit Number: 202108231030

Report Query Name: ElfinKingdom

Audit link: https://github.com/jiana860908/ElfinKingdom

Audit hash (final version): c984ce02bdefadc9b29590c2cfc42808e0fd08787071e99fb823b91f2c011dbd

Start Date：2021.05.24

Completion Date：2021.06.24

Report Update Date：2021.08.23

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

Disclaimer: This report is made in response to the project code. No description, expression or wording in this report shall be construed as an endorsement, affirmation or confirmation of the project. 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 project ElfinKingdom, including Coding Standards, Security, and Business Logic. **The ElfinKingdom project passed all audit items. The overall result is Pass. The smart contracts are 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 BNB.
- 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 token ELFIN

(1) Basic Token Information

Token name	Happy Farm Token
Token symbol	ELFIN
decimals	18
totalSupply	Mintable without cap, initial is 10 million, burnable
Token type	BEP-20

Table 1 Basic Token Information

(2) BEP-20 Token Standard Functions

- Description: The token contract implements a token which conforms to the BEP-20 Standards. It should be noted that the user can directly call the *approve* function to set the approval value for the specified address, but in order to avoid multiple authorizations, it is recommended to use the *increaseAllowance* and *decreaseAllowance* functions when modifying the approval value instead of using the *approve* function directly.
- Related functions: *name, symbol, decimals, totalSupply, balanceOf, allowance, transfer, transferFrom, approve, increaseAllowance, decreaseAllowance, burn, mint*
- Result: Pass

3.2 Business analysis of contract token EKD

(1) Basic Token Information

Token name	Happy Farm Dollar
Token symbol	EKD
decimals	18
totalSupply	Mintable without cap, initial is 0, burnable
Token type	BEP-20

Table 2 Basic Token Information

(2) BEP-20 Token Standard Functions

- Description: The token contract implements a token which conforms to the BEP-20 Standards. It should be noted that the user can directly call the *approve* function to set the approval value for the specified address, but in order to avoid multiple authorizations, it is recommended to use the

increaseAllowance and *decreaseAllowance* functions when modifying the approval value instead of using the *approve* function directly.

- Related functions: *name, symbol, decimals, totalSupply, balanceOf, allowance, transfer, transferFrom, approve, increaseAllowance, decreaseAllowance, burn, mint*

- Result: Pass

(3) setminter function

- Description: The contract implements the *setminter* function for the owner to grant minter privileges to the specified address, and only the minter of the contract can call the *mint* function to mint tokens. NOTE: The minter of the contract has the privileges to mint unlimited tokens and burn any user's EKD tokens, and the owner needs to hand over the minter rights to the BANK contract immediately after the contract is deployed.

```
376    function setminter(address newMinter) public  onlyOwner {
377      minter = newMinter;
378    }
```

Figure 1 source code of *setminter*

- Related functions: *setminter*
- Result: Pass

3.3 Business analysis of contract BANK

(1) deposit function

- Description: The contract implements a *deposit* function for the user to pledge a SupportToken of the specified name and 1:1 to obtain EKD tokens (the EKD minter needs to be set to the BANK contract), the user sends the SupportToken to this contract by pre-authorization, and then calls the *mint* function in the EKD contract to mint the token to the caller. If the reward count is not used up and meet the award delivery requirements, the reward is sent to the caller.

```
function deposit(string memory tokenname,uint256 amount) public updateRewardsLimit{

    //_totalSupply = _totalSupply.add(amount);

   // _balances[msg.sender] = _balances[msg.sender].add(amount);

    userAmount[msg.sender][tokenname] += amount;


    //1:1
    uint256 mintAmount    = amount;

    require(supportToken[tokenname] != address(0),"not supportToken");

    depositTokenaddr = IERC20(supportToken[tokenname]);

    //require(supportToken[tokenname] != address(0));

    depositTokenaddr.transferFrom(msg.sender, address(this), amount);

    //minter
   (bool success, bytes memory data) = HFDAddr.call(abi.encodeWithSelector(0x40c10f19,msg.sender,amount));
   require(success && (data.length == 0 || abi.decode(data, (bool))));

    //
    if((rewardsUsed < rewardsLimit)&&(amount >= 1e18)){
        rewardToken.transfer(msg.sender,rewardsPerTrans);
        rewardsUsed += 1;
    }

    emit Deposited(msg.sender,tokenname, amount, mintAmount,rewardsPerTrans);
```

Figure 2 source code of *deposit*

- Related functions: *deposit*
- Result: Pass

(2) withdraw function

- Description: The contract implements the *withdraw* function for the owner to extract the user's pledged supportToken to the user's address and destroy the corresponding number of EKD's. Different supportToken pledges are counted independently and do not affect other.

```
function withdraw(address account,string memory tokenname , uint256 amount) public onlyOwner {

    //require(_balances[account]            >=  amount, "not enough balance");
    //require(_totalSupply                  >= amount, "not enough totalSupply");
    require(userAmount[account][tokenname] >= amount);

    //_balances[account] -= amount;

    userAmount[account][tokenname] -= amount  ;

    depositTokenaddr = IERC20(supportToken[tokenname]);

    depositTokenaddr.transfer(account,amount);

    //minter
    (bool success, bytes memory data) = HFDAddr.call(abi.encodeWithSelector(0x9dc29fac, account,amount));
    require(success && (data.length == 0 || abi.decode(data, (bool))));
    //_totalSupply -= amount;

    emit Withdrawaled(account, tokenname ,amount);

  }
```

Figure 3 source code of *withdraw* (past)

- Safety recommendation: It is recommended that the *onlyOwner* modifier be removed and that users be free to withdraw their pledged tokens.
- Repair result: Fixed. Users are now free to withdraw their pledged tokens or help others to do so.

```
function withdraw(address account,string memory tokenname , uint256 amount) public {

    //require(_balances[account]            >=  amount, "not enough balance");
    //require(_totalSupply                  >= amount, "not enough totalSupply");
    require(userAmount[account][tokenname] >= amount);

    //_balances[account] -= amount;

    userAmount[account][tokenname] -= amount  ;

    depositTokenaddr = IERC20(supportToken[tokenname]);

    depositTokenaddr.transfer(account,amount);

    //minter
    (bool success, bytes memory data) = HFDAddr.call(abi.encodeWithSelector(0x9dc29fac, account,amount));
    require(success && (data.length == 0 || abi.decode(data, (bool))));
    //_totalSupply -= amount;

    emit Withdrawaled(account, tokenname ,amount);

  }
```

Figure 4 source code of *withdraw*(fixed)

- Related functions: *withdraw*
- Result: Pass

3.4 Business analysis of contract factory

(1) Basic Token Information

Token name	Set at deployment
Token symbol	Set at deployment
Token type	BEP-721

Table 3 Basic Token Information

(2) Mint function

- Description:The contract implements the *mint* function for minter to mint an BEP-721 token with a specified id. The *MintWithValue* function is used to pledge a supportToken to the contract address while the user is minting the token, if the reward count is not used up and meet the award delivery requirements, the reward is sent to the caller. NOTE: The id of the coin minted by the *MintWithValue* function is automatically generated. When Minter calls the *mint* function to specify the id of the coin to be minted, care should be taken to prevent conflicts from causing the *MintWithValue* function to be unavailable.

```
function MintWithValue(address[] memory erc20Address, uint256[] memory erc20Value) public   updateRewardsLimit{

    require(erc20Address.length == erc20Value.length);
    tokenIdIdex += 1;
    _mint(msg.sender, tokenIdIdex);

    bool rewardornot = false;

    for (uint i = 0; i < erc20Address.length; i++){

        string memory tokenSymbol = IERC20(erc20Address[i]).symbol();

        require(supportToken[tokenSymbol] == erc20Address[i],"Not in supportToken" );

        userAmount[tokenIdIdex][tokenSymbol] += erc20Value[i];

        IERC20(erc20Address[i]).transferFrom(msg.sender, address(this), erc20Value[i]);

        //values[tokenIdIdex][erc20Address[i]] += erc20Value[i];

        if(erc20Value[i] >= 1e18){
            rewardornot    = true;
        }

    }
    //payToken.transferFrom(msg.sender, owner(), amount);
    if((rewardsUsed < rewardsLimit) && rewardornot ){
        rewardToken.transfer(msg.sender,rewardsPerTrans);
        rewardsUsed += 1;
    }

    emit minted( msg.sender,tokenIdIdex , erc20Address ,erc20Value  ,rewardsPerTrans);
}
```

Figure 5 source code of *MintWithValue*

- Related functions: *mint*, *MintWithValue*
- Result: Pass

(3) Approve function

- Description: The contract implements the *approve* function for the user to authorize the token with the specified id to the specified address, requiring the caller to be the token owner or have full authorization; the *setApprovalForAll* function is used for the user to fully authorize the specified address.
- Related functions: *approve*, *setApprovalForAll*
- Result: Pass

(4) Transfer function

- Description: The contract implements the *transfer* function for the user to send tokens with the specified id to the specified address, the *transferFrom* and *safeTransferFrom* functions for proxy transfers, and the *safeTransferFrom* function can enter data and execute it. *transferAll* function is used for the user to send multiple tokens with different tokens with different ids to the specified address. The *transfer* and *transferAll* functions are only available when the contract state is start (the default is true and cannot be changed). Functions other than *transferFrom* check whether the target contract supports receiving BEP-721 tokens if the target address is a contract.
- Related functions: *transfer*, *transferFrom*, *safeTransferFrom*, *transferAll*

- Result: Pass

(5) Withdraw function

- Description: The contract implements the *withdraw* function for the administrator to extract the additional collateral BEP-20 tokens of the specified id tokens to its owner address by calling internal function *withDrawValue* .

```
537     function withdraw(address _to, uint256 _tokenId, address[] memory erc20Address, uint256[] memory erc20Value) public onlyOwner {
538
539         require(ownerOf(_tokenId) == _to);
540
541
542
543         withDrawValue(_to,_tokenId,erc20Address,erc20Value);
544
545     }
546
547     function withDrawValue(address _to, uint256 _tokenId, address[] memory erc20Address, uint256[] memory erc20Value) internal  {
548
549         require(ownerOf(_tokenId) == _to);
550         require(erc20Address.length == erc20Value.length);
551
552         for (uint i = 0; i < erc20Address.length; i++){
553             if (values[_tokenId][erc20Address[i]] > 0){
554
555                 string memory tokenSymbol = IERC20(erc20Address[i]).symbol();
556
557                 require(supportToken[tokenSymbol] == erc20Address[i],"Not in supportToken" );
558
559                 require(userAmount[tokenIdIdex][tokenSymbol] == erc20Value[i] );
560
561                 values[_tokenId][erc20Address[i]] = 0;
562                 IERC20(erc20Address[i]).transfer(_to, erc20Value[i]);
563                 //values[_tokenId][erc20Address[i]] = 0;
564             }
565         }
566     }
```

Figure 6 source code of *withdraw* and *withDrawValue*

- Safety recommendation: It is recommended that the *onlyOwner* modifier be removed and that users be free to withdraw the pledged tokens.
- Repair result: Fixed. Users are now free to withdraw the pledged tokens or help others to do so.

```
    function withdraw(address _to, uint256 _tokenId, address[] memory erc20Address, uint256[] memory erc20Value) public {

        require(ownerOf(_tokenId) == _to);



        withDrawValue(_to,_tokenId,erc20Address,erc20Value);

    }

    function withDrawValue(address _to, uint256 _tokenId, address[] memory erc20Address, uint256[] memory erc20Value) internal  {

        require(ownerOf(_tokenId) == _to);
        require(erc20Address.length == erc20Value.length);

        for (uint i = 0; i < erc20Address.length; i++){

            string memory tokenSymbol = IERC20(erc20Address[i]).symbol();

            require(supportToken[tokenSymbol] == erc20Address[i],"Not in supportToken" );

            require(userAmount[tokenIdIdex][tokenSymbol] == erc20Value[i] );

            if (userAmount[tokenIdIdex][tokenSymbol] > 0){


                userAmount[tokenIdIdex][tokenSymbol] = 0;
                IERC20(erc20Address[i]).transfer(_to, erc20Value[i]);
                //values[_tokenId][erc20Address[i]] = 0;
            }
        }
    }
```

Figure 7 source code of *withdraw* and *withDrawValue*

- Related functions: *withdraw*, *withDrawValue*

- Result: Pass

(6) Other function

- Description: The contract implements the *addMinter* function for the minter to grant minter privileges to other addresses; the *renounceMinter* function to relinquish the caller's own minter privileges; and *setUri* for the minter to modify the contract's uri
- Related functions: *addMinter*, *renounceMinter*, *setUri*
- Result: Pass

3.5 Business analysis of contract HfAuction

(1) nftOnList function

- Description: The contract implements the *nftOnList* function for the user to send tokens with the specified id to the contract address for sale(Pre-authorisation of this contract is required), the user needs to set the price and the default sale time is 7 days.

```
223     function nftOnList(uint256 tokenId, uint256 price ) public returns(uint256 ) {
224
225         require(erc721token.ownerOf( tokenId ) == msg.sender );
226         require(price > 0);
227
228         Product memory newProduct;
229         newProduct.owner     = msg.sender;
230         newProduct.tokenId   = tokenId;
231         newProduct.price     = price;
232         newProduct.starttime = block.timestamp;
233         newProduct.duration  = 604800;
234         newProduct.onSale    = true;
235
236         uint256 index = productCnt + 1;
237
238         products[index]      = newProduct;
239
240         productCnt = index;
241
242         erc721token.safeTransferFrom(msg.sender,address(this),tokenId);
243
244         emit nftOnListed(index,msg.sender,tokenId,price,newProduct.starttime,newProduct.duration,newProduct.onSale);
245
246     }
```

Figure 8 source code of *nftOnList*

- Related functions: *nftOnList*
- Result: Pass

(2) buyNFT function

- Description: The contract implements the *buyNFT* function for users to purchase tokens that are already on the shelf, execute *checkOnSale* before calling to update the information, requiring that the specified token be available for purchase and that the payment be equal to the pricing. The token used is EKD and the constants are not modifiable. After sending the EKD to the contract, the contract sends the specified BEP-721 tokens to the user and updates the relevant information. If the reward count is not used up, the reward is sent to the caller.

```
260     function buyNFT(uint256 index ,uint256 amount) public checkOnSale(index) updateRewardsLimit {
261         require( products[index].onSale == true );
262         //require( products[index].owner  != msg.sender );
263         require( products[index].price  == amount );
264 
265         //products[index].onSale = false;
266         //products[index].owner  = msg.sender;
267 
268         //pay price
269         hfdToken.transferFrom(msg.sender,products[index].owner,amount);
270 
271         //change owner
272         erc721token.safeTransferFrom(address(this),msg.sender,products[index].tokenId);
273         products[index].owner  = msg.sender;
274 
275         //hfdToken.Transfer();
276 
277         products[index].onSale == false;
278 
279         //pay rewards
280         if( rewardsUsed < rewardsLimit ){
281             rewardToken.transfer(msg.sender,rewardsPerTrans);
282             rewardsUsed += 1;
283         }
284 
285         emit buyNFTed(index ,msg.sender,amount,rewardsPerTrans);
286 
287     }
```

Figure 9 source code of *buyNFT*

- Related functions: *buyNFT, checkOnSale*
- Result: Pass

(3) getProductByIndex function

- Description: The contract implements the *getProductByIndex* function to obtain information about the specified product (selling price, id, owner, time of sale, whether sold or not).

```
256     function getProductByIndex(uint256 index) public view returns(address,uint256,uint256,uint256,uint256,bool){
257         return (products[index].owner,products[index].tokenId,products[index].price,products[index].starttime,products[index].duration,products
        [index].onSale);
258     }
```

Figure 10 source code of *getProductByIndex*

- Related functions: *getProductByIndex*,
- Result: Pass

(4) Reward related function

- Description: The three contracts BANK, factory and HfAuction have unique reward mechanisms. The *setRewardsLimit* function is implemented to modify the daily reward limit; *setRewardsPerTrans* is used to modify the number of tokens per reward; and the *updateRewardsLimit* modifier is used to reset the number of rewards to 0 each day.
- Related functions: *setRewardsLimit, setRewardsPerTrans, updateRewardsLimit*
- Result: Pass

3.6 Business analysis of contract hfcontrol

(1) Init function

- Description: The contract implements the *initFarmer* function for the CMO initialization of the contract to specify farmlandAll. Each numbered farmlandAll contains a different number of farmland

addresses, size=4 for sFarmer, 8 for mFarmer, 10 for lFarmer. num must not exceed the limit on the number of corresponding farmer. Each famelandAll can only be initialised once.

- Related functions: *initFarmer*
- Result: Pass
- Result: Pass

(2) beFarmer function

- Description: The contract implements the *beFarmer* function for the user to become the owner of the specified farmlandAll, which has different prices for different sizes of farmlandAll. This is achieved by calling the *beFarmer* function in each of the fameland contracts in famelandAll..
- Related functions: *beFarmer*
- Result: Pass

(3) Set function

- Description: The contract implements *setSFarmerLimit*, *setMFarmerLimit*, and *setLFarmerLimit* functions for the owner of the contract to modify the farmlandAll quantity limit of the specified size, which must be less than 100.
- Related functions: *setSFarmerLimit*, *setMFarmerLimit*, *setLFarmerLimit*
- Result: Pass

(4) Query function

- Description: The contract implements the *getFarmer* function for querying the farmer of the specified farmlandAll; the *getFarmerLand* function for querying the specified farmland of the specified farmlandAll; and *checkFarmerInit* for querying whether the farmlandAll is initialized.
- Related functions: *getFarmer*, *getFarmerLand*, *checkFarmerInit*
- Result: Pass

3.7 Business analysis of contract stake

(1) notifyRewardAmount function

- Description: The contract implements the *notifyRewardAmount* function to initialize the reward-related parameters of the contract, which can only be called by rewardDistribution with an interval of at least 15 minutes.

```
function notifyRewardAmount(uint256 reward,uint256 new_DURATION,uint256 new_settlement_DURATION)
external
onlyRewardDistribution
updateReward(address(0))
 {
    require(block.timestamp.sub(lastUpadteNotifyTime) > 900, "cannot trigger twice in 15 min");
    require(new_DURATION > 0);
    require(new_DURATION > new_settlement_DURATION);
    lastUpadteNotifyTime = block.timestamp;
    rewardNow = reward;
    rewardNext = reward;
    DURATION = new_DURATION;
    DURATION_NEXT = new_DURATION;
    settlement_DURATION = new_settlement_DURATION;
    settlement_DURATION_NEXT = new_settlement_DURATION;

    if (block.timestamp > starttime) {
        if (block.timestamp >= periodFinish) {
            rewardRate = reward.div(DURATION - settlement_DURATION);
        } else {
            uint256 remaining = (periodFinish- settlement_DURATION).sub(block.timestamp);
            uint256 leftover = remaining.mul(rewardRate);
            rewardRate = reward.add(leftover).div(DURATION - settlement_DURATION);
        }
        lastUpdateTime = block.timestamp;
        periodFinish = block.timestamp.add(DURATION);
        emit RewardAdded(reward);
    } else {
        rewardRate = reward.div(DURATION - settlement_DURATION);
        lastUpdateTime = starttime;
        periodFinish = starttime.add(DURATION);
        emit RewardAdded(reward);
    }
    emit notifyRewardAmounted( reward, new_DURATION, new_settlement_DURATION);
```

Figure 11 source code of *notifyRewardAmount*

- Related functions: *notifyRewardAmount*
- Result: Pass

(2) beFarmer function

- Description: The contract implements the *beFarmer* function for the controller of the contract to change the address of the farmer.
- Related functions: *beFarmer*
- Result: Pass

(3) bePlayer function

- Description: The contract implements the *bePlayer* function for the user to acquire player rights for a specified token by sending a certain number of supportToken to the farmer, each token has a limited number of player rights and multiple player rights can be purchased at once. Player rights are related to the maximum amount of pledges in the pledge pool.

```
1007    function bePlayer(uint8 buySize , uint256 amount) public canFarmering canBuy checkStart  {
1008        string memory tokenName = IERC20(tokenAddr).symbol();
1009
1010        require(farmerSize >= (farmerUsed.add(buySize)));
1011
1012        super.bePlayer(tokenName ,   buySize ,  amount);
1013
1014        farmerUsed += buySize;
1015
1016        supportToken.transferFrom(msg.sender,farmer,amount);
1017
1018        emit bePlayered(tokenName,msg.sender,buySize,amount,farmerUsed,super.totalSupply());
1019    }
```

Figure 12 source code of *bePlayer*

- Related functions: *bePlayer*
- Result: Pass

(4) stake function

- Description: The contract implements the *stake* function for Player to pledge specified tokens and receive rewards over the pledge time. The reward-related data is updated via the *updateReward* modifier before pledging, and the *playerStake* function updates Player's used limits.

```
1029    function stake(uint256 amount) public updateReward(msg.sender) onlyPlayer(msg.sender)  checkStart {
1030        require(amount > 0, "Cannot stake 0");
1031        super.playerStake(msg.sender,amount);
1032        super.stake(amount);
1033        emit Staked(msg.sender, amount ,super.totalSupply());
1034        emit SnapShot(msg.sender, balanceOf(msg.sender));
1035    }
```

Figure 13 source code of *stake*

- Related functions: *stake*, *updateReward*, *playerStake*
- Result: Pass

(5) withdraw function

- Description: The contract implements the *withdraw* function for the Player to withdraw his pledged tokens, which is updated with the *updateReward* function before withdrawal, and the *playerWithdraw* function to update the Player's used limit.

```
1037    function withdraw(uint256 amount) public updateReward(msg.sender) onlyPlayer(msg.sender) checkStart {
1038        require(amount > 0, "Cannot withdraw 0");
1039        super.playerWithdraw(msg.sender,amount);
1040        super.withdraw(amount);
1041        emit Withdrawn(msg.sender, amount);
1042        emit SnapShot(msg.sender, balanceOf(msg.sender));
1043    }
```

Figure 14 source code of *withdraw*

- Related functions: *withdraw*, *updateReward*, *playerWithdraw*
- Result: Pass

(6) playerWithdrawAndGetReward function

- Description: The contract implements the *playerWithdrawAndGetReward* function for the user to withdraw the specified number of collateral tokens and receive the collateral reward, update the reward-

related information via the *updateReward* function, and then call the *withdraw* and *playerGetReward* functions to complete the operation.

```
1053    function playerWithdrawAndGetReward(uint256 amount) public updateReward(msg.sender) checkStart {
1054        require(amount <= balanceOf(msg.sender), "Cannot withdraw exceed the balance");
1055        withdraw(amount);
1056        playerGetReward();
1057    }
```

Figure 15 source code of *playerWithdrawAndGetReward*

- Related functions: *playerWithdrawAndGetReward*, *updateReward*, *withdraw*, *playerGetReward*
- Result: Pass

(7) playerExit function

- Description: The contract implements the *playerExit* function for the user to withdraw all collateral tokens and collect the collateral reward, which is done by calling *withdraw* and *playerGetReward*.

```
1059    function playerExit() external {
1060        withdraw(balanceOf(msg.sender));
1061        playerGetReward();
1062    }
```

Figure 16 source code of *playerExit*

- Related functions: *playerExit*, *withdraw*, *playerGetReward*
- Result: Pass

(8) GetReward function

- Description: The contract implements the *farmerGetReward* function for Farmer to collect all farmerRewards; the *playerGetReward* function is used for Player to collect all collateral rewards, and the rewards related data is updated by the *updateReward* function before the call.

```
1064    function farmerGetReward() public updateReward(msg.sender) onlyFarmer() checkStart {
1065        uint256 trueReward = farmerRewards;
1066        if (trueReward > 0) {
1067            //rewards[msg.sender] = 0;
1068            farmerRewards     = 0;
1069            supportToken.safeTransfer(msg.sender, trueReward);
1070            emit farmerGetRewarded(msg.sender, trueReward);
1071        }
1072    }
1073
1074    function playerGetReward() public updateReward(msg.sender) onlyPlayer(msg.sender) checkStart {
1075        uint256 trueReward = allRewards[msg.sender];
1076        if (trueReward > 0) {
1077            //rewards[msg.sender] = 0;
1078            allRewards[msg.sender] = 0;
1079            supportToken.safeTransfer(msg.sender, trueReward);
1080            emit playerGetRewarded(msg.sender, trueReward);
1081        }
1082    }
```

Figure 17 source code of *farmerGetReward* and *playerGetReward*

- Related functions: *farmerGetReward*, *playerGetReward*, *updateReward*
- Result: Pass

(9) doSettlement function

- Description: The contract implements the *doSettlement* function for the contract's oper to update the rewards available to the specified address. 5% of the reward is sent to the watering address (if it is a 0 address, it is kept in this contract), 75% is the player reward and 20% is the farmer reward. The call is preceded by an *updateReward* to update the information about the account address reward. Note: This function must be called to update the corresponding player rewards before the player can receive the latest rewards. The project declares that this design meets its design requirements.

```
968     function doSettlement(address account,address watering) onlyOper checkDoSettlement updateReward(account) public{
969
970         require(earned(account) > 0);
971
972         //rewardPerTokenStored = rewardPerToken();
973
974         uint256 farmerEarn   = earned(account).mul(20).div(100);
975         uint256 playerEarn   = earned(account).mul(75).div(100);
976         uint256 wateringEarn = earned(account).mul(5).div(100);
977
978         allRewards[account] += playerEarn;
979         farmerRewards       += farmerEarn;
980
981         //userRewardPerTokenPaid[account] = rewardPerTokenStored;
982
983         if(watering != address(0)){
984             supportToken.transfer(watering,wateringEarn);
985         }
986         else{
987             noOneWateringRewards += wateringEarn;
988         }
989
990         rewards[account] = 0;
991
992         emit doSettlemented(account,watering,farmerEarn,playerEarn,wateringEarn);
993
994     }
```

Figure 18 source code of *doSettlement*

- Safety recommendation: The *doSettlement* function, if called multiple times at the same time, will double-count and accumulate the relevant reward data. Updating a specified user reward will also cause the reward to be accrued if it is updated again before the user claims it. It is recommended that userRewardPerTokenPaid[account] be updated after the *doSettlement* function is executed.
- Repair result: Fixed, execute *updateReward* to update the information about the account address reward.
- Related functions: *doSettlement*
- Result: Pass

(10) setNextRewardInfo function

- Description: The contract implements the *setNextRewardInfo* function to modify the parameters associated with the next award cycle, and the caller needs to have RewardDistribution access.
- Related functions: *setNextRewardInfo*
- Result: Pass

(11) rescue function

- Description: The contract implements *rescue* function for the owner to withdraw other tokens from the contract to a specified address, but not the reward tokens or collateral tokens in the contract.

```
1154    function rescue(address to_, IERC20 token_, uint256 amount_)
1155    external
1156    onlyOwner
1157    {
1158        require(to_ != address(0), "must not 0");
1159        require(amount_ > 0, "must gt 0");
1160        require(token_ != supportToken, "must not hfToken");
1161        require(token_ != tokenAddr, "must not this stakeToken");
1162
1163        token_.transfer(to_, amount_);
1164        emit RescueToken(to_, address(token_), amount_);
1165    }
```

Figure 19 source code of *rescue*

- Related functions: *rescue*
- Result: Pass

(12) setRewardRate function

- Description: The contract implements the *setRewardRate* function to modify the award rate, which is called before the *updateReward* function is executed to update the relevant data, and the caller needs to have RewardDistribution access.

```
function setRewardRate(uint256 newRewardRate) public onlyRewardDistribution  updateReward(address(0)){
    require(newRewardRate < rewardRate);
    rewardRate = newRewardRate;
}
```

- Safety recommendation: The administrator has too much authority and the loss of the private key may lead to a miscalculation of the reward, it is recommending to delete.
- Rpair result: Deleted.
- Related functions: *setRewardRate*, *updateReward*
- Result: Pass

(13) emergencyWithdraw function

- Description: The contract implements the *emergencyWithdraw* function for player emergency withdrawals of pledged tokens, and does not call *updateReward* to update reward-related information.

```
1050    function emergencyWithdraw() public onlyPlayer(msg.sender){
1051        //require(amount > 0, "Cannot withdraw 0");
1052
1053        uint256 amount = super.getUsedLimit(msg.sender);
1054
1055        super.playerWithdraw(msg.sender,amount);
1056        super.withdraw(amount);
1057        emit Withdrawn(msg.sender, amount);
1058        emit SnapShot(msg.sender, balanceOf(msg.sender));
1059    }
```

Figure 20 source code of *emergencyWithdraw*

- Related functions: *emergencyWithdraw*
- Result: Pass

4. Conclusion

Beosin(Chengdu LianAn) conducted a detailed audit on the design and code implementation of the smart contracts project ElfinKingdom. The problems found by the audit team during the audit process have been notified to the project party and and agree on the outcome of the restoration, the overall audit result of the ElfinKingdom project's smart contracts is **Pass**.

BEOSIN
Blockchain Security

Official Website

https://lianantech.com

E-mail

vaas@lianantech.com

Twitter

https://twitter.com/Beosin_com