
Armors Labs
BIB META NFT
Smart Contract Audit

BIB META NFT Audit Summary

Project name : BIB META NFT Contract

Project address: None

Code URL : https://github.com/bibvip-github/bibnft/tree/audit-revised

Commit : 9a651c2ab895446b0b9c8ba89bf5cdf95e19f024

Project target : BIB META NFT Contract Audit

Blockchain : Binance Smart Chain（BSC）

Test result : PASSED

Audit Info

Audit NO : 0X202209220009

Audit Team : Armors Labs

Audit Proofreading: https://armors.io/#project-cases

BIB META NFT Audit

The BIB META NFT team asked us to review and audit their BIB META NFT contract. We looked at the code and now publish our results.

Here is our assessment and recommendations, in order of importance.

Document information

Name	Auditor	Version	Date
BIB META NFT Audit	Rock, Sophia, Rushairer, Rico, David, Alice	1.0.0	2022-09-22

Audit results

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the BIB META NFT contract. The above should not be construed as investment advice.

Based on the widely recognized security status of the current underlying blockchain and smart contract, this audit report is valid for 3 months from the date of output.

Disclaimer

Armors Labs Reports is not and should not be regarded as an "approval" or "disapproval" of any particular project or team. These reports are not and should not be regarded as indicators of the economy or value of any "product" or "asset" created by any team. Armors do not cover testing or auditing the integration with external contract or services (such as Unicrypt, Uniswap, PancakeSwap etc'…)

Armors Labs Reports represent an extensive auditing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology. Armors does not guarantee the safety or functionality of the technology agreed to be analyzed.

Armors Labs postulates that the information provided is not missing, tampered, deleted or hidden. If the information provided is missing, tampered, deleted, hidden or reflected in a way that is not consistent with the actual situation, Armors Labs shall not be responsible for the losses and adverse effects caused. Armors Labs Audits should not be used in any way to make decisions around investment or involvement with any particular project. These reports in no way provide investment advice, nor should be leveraged as investment advice of any sort.

Audited target file

file	md5
./misc/FeeCollector.sol	b1247ce328fd8309aca67f1e466f5190
./nft/SoccerStarNft.sol	4af66d7a3a865b75ede6588e0e39eacd
./staked/StakedSoccerStarNft.sol	eb055e1e5a42132a909615b5e01bf369
./libs/SafeMath.sol	3cb6d6c39f2053165534979d30dadf84
./libs/SafeCast.sol	01c9361b3616b67d2f85ca11bb7ab4c5
./market/SoccerStarNftMarket.sol	4673b4e17866298f907183ccbd8af8d3
./composed/ComposedSoccerStarNft.sol	2a6184fb0cf965d9662c8f9d8bdd4a2b
./interfaces/IComposedSoccerStarNft.sol	154aba4bd086ce6d89b2b4eaf8c89983
./interfaces/IRewardDistributor.sol	805f459db33358b6b51b50887252b19e
./interfaces/ISoccerStarNftMarket.sol	d9fbdef876a75f63eb21e25fc66c818e
./interfaces/IFeeCollector.sol	dd06ac197a17a0fa54439aac27ff6ef3
./interfaces/IStakedSoccerStarNft.sol	c7a2ef82236c5d1a25dcc223d7118897
./interfaces/ISoccerStarNft.sol	97e80b588dde4c0a1e86a4c02128af01

Vulnerability analysis

Vulnerability distribution

vulnerability level	number
Critical severity	0
High severity	0
Medium severity	0
Low severity	0

Summary of audit results

Vulnerability	status
Re-Entrancy	safe
Arithmetic Over/Under Flows	safe
Unexpected Blockchain Currency	safe
Delegatecall	safe
Default Visibilities	safe
Entropy Illusion	safe
External Contract Referencing	safe
Short Address/Parameter Attack	safe
Unchecked CALL Return Values	safe
Race Conditions / Front Running	safe
Denial Of Service (DOS)	safe
Block Timestamp Manipulation	safe
Constructors with Care	safe
Unintialised Storage Pointers	safe
Floating Points and Numerical Precision	safe
tx.origin Authentication	safe
Permission restrictions	safe

Contract file

```
//SPDX-License-Identifier: MIT

pragma solidity >=0.8.0;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol";
import "@openzeppelin/contracts/token/ERC721/IERC721.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/Address.sol";
import '@uniswap/v2-periphery/contracts/interfaces/IUniswapV2Router02.sol';
import {SafeMath} from "../libs/SafeMath.sol";
import {SafeCast} from "../libs/SafeCast.sol";
import {IComposedSoccerStarNft} from "../interfaces/IComposedSoccerStarNft.sol";
import {ISoccerStarNft} from "../interfaces/ISoccerStarNft.sol";

contract ComposedSoccerStarNft is IComposedSoccerStarNft, Ownable {
    using SafeMath for uint;

    address constant public BLOCK_HOLE = address(0x0000000000000000000000000000000000000001);

    ISoccerStarNft public tokenContract;
    IERC20 public bibContract;
    IERC20 public busdContract;
    IUniswapV2Router02 public router;
```

```
    // fill with default
    uint[12] public feeRates = [360000,  730000,    1200000, 2200000,
                      1800000, 3650000,   6000000, 11000000,
                      9000000, 18250000,  30000000, 55000000];

    address public treasury;

    uint constant public MAX_STARLEVEL = 4;
    uint constant public STARLEVEL_RANGE = 4;
    uint constant public ORACLE_PRECISION = 1e18;

    event TokenContractChanged(address sender, address oldValue, address newValue);
    event BIBContractChanged(address sender, address oldValue, address newValue);
    event BUSDContractChanged(address sender, address oldValue, address newValue);
    event TreasuryChanged(address sender, address oldValue, address newValue);
    event SwapRouterChanged(address sender, address oldValue, address newValue);
    event FeeRateChanged(address sender, uint[12] oldValue, uint[12] newValue);

    constructor(
    address _tokenContract,
    address _bibContract,
    address _busdContract,
    address _treasury,
    address _router
    ){
        tokenContract = ISoccerStarNft(_tokenContract);
        bibContract = IERC20(_bibContract);
        busdContract = IERC20(_busdContract);
        treasury = _treasury;
        router = IUniswapV2Router02(_router);
    }

    function setTokenContract(address _tokenContract) public onlyOwner{
        require(address(0) != _tokenContract, "INVLID_ADDRESS");
        emit TokenContractChanged(msg.sender, address(tokenContract), _tokenContract);
        tokenContract = ISoccerStarNft(_tokenContract);
    }

    function setBIBContract(address _bibContract) public onlyOwner{
        require(address(0) != _bibContract, "INVLID_ADDRESS");
        emit BIBContractChanged(msg.sender, address(bibContract), _bibContract);
        bibContract = IERC20(_bibContract);
    }

    function setTreasury(address _treasury) public onlyOwner{
        require(address(0) != _treasury, "INVLID_ADDRESS");
        emit TreasuryChanged(msg.sender, treasury, _treasury);
        treasury = _treasury;
    }

    function setSwapRouter(address _router) public onlyOwner{
        require(address(0) != _router, "INVLID_ADDRESS");
        emit SwapRouterChanged(msg.sender, address(router), _router);
        router = IUniswapV2Router02(_router);
    }

    function setBUSDContract(address _busdContract) public onlyOwner{
        require(address(0) != _busdContract, "INVLID_ADDRESS");
        emit BUSDContractChanged(msg.sender, address(busdContract), _busdContract);
        busdContract = IERC20(_busdContract);
    }

    function configFeeRate(uint[12] memory _feeRates) public onlyOwner{
        require(_feeRates.length == feeRates.length, "INVLID_FEERATES");
        emit FeeRateChanged(msg.sender, feeRates, _feeRates);
```

```
        for(uint i = 0; i < _feeRates.length; i++){
            feeRates[i] = _feeRates[i];
        }
    }

    function compose(
    uint[] memory tokenIds,
    ComposeMode mode,
    uint extralToken,
    PayMethod payMethod
    ) public override{
        require(4 == tokenIds.length, "NEED_FOUR_TOKENS");
        require(validToken(tokenIds[0], tokenIds), "NEED_SAME_TOKEN_PROPER");
        require(validStarLevel(tokenIds[0]), "NEED_LOWER_STARLEVEL");

        // valid owner ship
        validOwnership(tokenIds);

        // burn all
        for(uint i = 0; i < tokenIds.length; i++){
            IERC721(address(tokenContract)).transferFrom(msg.sender, BLOCK_HOLE, tokenIds[i]);
        }

        // compose new
        ISoccerStarNft.SoccerStar memory soccerStar = tokenContract.getCardProperty(tokenIds[0]);

        uint payAmount = 0;
        if(ComposeMode.COMPOSE_NORMAL == mode) {
            require(msg.sender == IERC721(address(tokenContract)).ownerOf(extralToken), "TOKEN_NOT_BE
            // burn the extral
            IERC721(address(tokenContract)).transferFrom(msg.sender, BLOCK_HOLE, extralToken);
        } else {
            payAmount = caculateBurnAmount(soccerStar.starLevel, soccerStar.gradient);
            if(PayMethod.PAY_BIB == payMethod){
                bibContract.transferFrom(msg.sender, BLOCK_HOLE, payAmount);
            } else {
                payAmount = caculateBUSDAmount(payAmount);
                busdContract.transferFrom(msg.sender, treasury, payAmount);
            }
        }

        uint newToken = tokenContract.protocolMint();
        tokenContract.protocolBind(newToken, soccerStar);
        IERC721(address(tokenContract)).transferFrom(address(this), msg.sender, newToken);
        emit Composed(msg.sender, tokenIds, extralToken,newToken, mode, payMethod, payAmount);
    }

    function caculateBurnAmount(uint starLevel, uint gradient) public view returns(uint){
        uint decimals = IERC20Metadata(address(bibContract)).decimals();
        return feeRates[(starLevel - 1) * STARLEVEL_RANGE + (gradient - 1)].exp(decimals);
    }

    function caculateBUSDAmount(uint bibAmount) public view returns(uint){
        // the price has ORACLE_PRECISION
        address[] memory path = new address[](2);
        path[0] = address(bibContract);
        path[1] = address(busdContract);
        return router.getAmountsOut(bibAmount, path)[1];
    }

    function validOwnership(uint[] memory tokensToValid) internal view {
        for(uint i = 0; i < tokensToValid.length; i++){
            require(msg.sender == IERC721(address(tokenContract)).ownerOf(tokensToValid[i]), "TOKEN_N
        }
    }
```

```
    function validStarLevel(uint tokenId) internal view returns(bool){
        return tokenContract.getCardProperty(tokenId).starLevel < MAX_STARLEVEL;
    }

    function validToken(uint base, uint[] memory tokensToValid) internal view returns(bool){
        if(0 == tokensToValid.length){
            return false;
        }

        ISoccerStarNft.SoccerStar memory baseProperty = tokenContract.getCardProperty(base);
        for(uint i = 0; i < tokensToValid.length; i++){
            if(!cmpProperty(baseProperty, tokenContract.getCardProperty(tokensToValid[i]))){
                return false;
            }
        }
        return true;
    }

    function cmpProperty (
    ISoccerStarNft.SoccerStar memory a,
    ISoccerStarNft.SoccerStar memory b) internal pure returns(bool){
        return keccak256(bytes(a.name)) == keccak256(bytes(b.name))
        && keccak256(bytes(a.country)) == keccak256(bytes(b.country))
        && keccak256(bytes(a.position)) == keccak256(bytes(b.position))
        && a.gradient == b.gradient;
    }
}



//SPDX-License-Identifier: MIT

pragma solidity >=0.8.0;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC721/IERC721.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/Address.sol";
import {SafeMath} from "../libs/SafeMath.sol";
import {SafeCast} from "../libs/SafeCast.sol";
import {ISoccerStarNft} from "../interfaces/ISoccerStarNft.sol";
import {ISoccerStarNftMarket} from "../interfaces/ISoccerStarNftMarket.sol";
import {IFeeCollector} from "../interfaces/IFeeCollector.sol";

contract SoccerStarNftMarket is ISoccerStarNftMarket, Ownable{
    using SafeMath for uint;

    address public treasury;

    IERC20 public bibContract;
    IERC20 public busdContract;
    ISoccerStarNft public tokenContract;
    IFeeCollector feeCollector;

    event TokenContractChanged(address sender, address oldValue, address newValue);
    event BIBContractChanged(address sender, address oldValue, address newValue);
    event BUSDContractChanged(address sender, address oldValue, address newValue);
    event FeeRatioChanged(address sender, uint oldValue, uint newValue);
    event RoyaltyRatioChanged(address sender, uint oldValue, uint newValue);
    event FeeCollectorChanged(address sender, address oldValue, address newValue);

    uint public nextOrderIndex;
    uint public nextOfferIndex;

    uint public feeRatio = 25;
    uint public royaltyRatio = 75;
```

```
    uint public constant FEE_RATIO_DIV = 1000;

    // orders in the market
    uint[] public orders;
    // fast index from order id to order
    mapping(uint=>Order) public orderTb;
    // orders belong to the specfic owner
    mapping(address=>uint[]) public userOrderTb;
    //  offers of the order
    mapping(uint=>uint[]) public orderOfferTb;
    // fast index from offer id to offer
    mapping(uint=>Offer) public offerTb;

    constructor(
    address _treasury,
    address _tokenContract,
    address _bibContract,
    address _busdContract
    ){
        treasury = _treasury;
        tokenContract = ISoccerStarNft(_tokenContract);
        bibContract = IERC20(_bibContract);
        busdContract = IERC20(_busdContract);
    }

    function getBlockTime() public override view returns(uint){
        return block.timestamp;
    }

    function setTokenContract(address _tokenContract) public onlyOwner{
        require(address(0) != _tokenContract, "INVALID_ADDRESS");
        emit TokenContractChanged(msg.sender, address(tokenContract), _tokenContract);
        tokenContract = ISoccerStarNft(_tokenContract);
    }

    function setBIBContract(address _bibContract) public onlyOwner{
        require(address(0) != _bibContract, "INVALID_ADDRESS");
        emit BIBContractChanged(msg.sender, address(bibContract), _bibContract);
        bibContract = IERC20(_bibContract);
    }

    function setBUSDContract(address _busdContract) public onlyOwner{
        require(address(0) != _busdContract, "INVALID_ADDRESS");
        emit BUSDContractChanged(msg.sender, address(busdContract), _busdContract);
        busdContract = IERC20(_busdContract);
    }

    function setFeeCollector(address _feeCollector) public onlyOwner{
        require(address(0) != _feeCollector, "INVALID_ADDRESS");
        emit FeeCollectorChanged(msg.sender, address(feeCollector), _feeCollector);
        feeCollector = IFeeCollector(_feeCollector);
    }

    function setFeeRatio(uint _feeRatio) public override onlyOwner{
        require(_feeRatio <= FEE_RATIO_DIV, "INVALID_RATIO");
        emit FeeRatioChanged(msg.sender,feeRatio, _feeRatio);
        feeRatio = _feeRatio;
    }

   function setRoyaltyRatio(uint _royaltyRatio) override public onlyOwner {
       require(_royaltyRatio <= FEE_RATIO_DIV, "INVALID_ROYALTY_RATIO");
       emit RoyaltyRatioChanged(msg.sender, royaltyRatio, _royaltyRatio);
       royaltyRatio = _royaltyRatio;
   }

    // user create a order
```

```
    function openOrder(address issuer, uint tokenId, PayMethod payMethod, uint price, uint expiration
        require(address(0) != issuer, "INVALID_ISSURE");
        require(expiration > block.timestamp, "EXPIRATION_TOO_SMALL");
        require(price > 0, "PRICE_NOT_BE_ZEROR");
        require(msg.sender == IERC721(address(issuer)).ownerOf(tokenId),
        "TOKEN_NOT_BELLONG_TO_SENDER");

        // delegate token to protocol
        IERC721(address(issuer)).transferFrom(msg.sender, address(this), tokenId);

        // record order
        Order memory order = Order({
            issuer: issuer,
            orderId: nextOrderIndex++,
            tokenId: tokenId,
            owner: msg.sender,
            payMethod: payMethod,
            price: price,
            expiration: expiration
        });

        orders.push(order.orderId);
        userOrderTb[msg.sender].push(order.orderId);
        orderTb[order.orderId] = order;

        emit OpenOrder(issuer, msg.sender, order.orderId, tokenId, payMethod, price, expiration);
    }

    // get orders by page
    function getUserOrdersByPage(address user, uint pageSt, uint pageSz)
    public view override returns(Order[] memory){
        uint[] storage _orders= userOrderTb[user];
        Order[] memory ret;

        if(pageSt < _orders.length){
            uint end = pageSt + pageSz;
            end = end > _orders.length ? _orders.length : end;
            ret =  new Order[](end - pageSt);
            for(uint i = 0;pageSt < end; i++){
                ret[i] = orderTb[_orders[pageSt]];
                pageSt++;
            }
        }

        return ret;
    }

    function getOrdersByPage(uint pageSt, uint pageSz)
    public view override returns(Order[] memory){
        Order[] memory ret;

        if(pageSt < orders.length){
            uint end = pageSt + pageSz;
            end = end > orders.length ? orders.length : end;
            ret =  new Order[](end - pageSt);
            for(uint i = 0;pageSt < end; i++){
                ret[i] = orderTb[orders[pageSt]];
                pageSt++;
            }
        }

        return ret;
    }

    function getOrderOffersByPage(uint orderId, uint pageSt, uint pageSz)
    public view  override returns(Offer[] memory){
```

```
        uint[] storage offers = orderOfferTb[orderId];
        Offer[] memory ret;

        if(pageSt < offers.length){
            uint end = pageSt + pageSz;
            end = end > offers.length ? offers.length : end;
            ret =  new Offer[](end - pageSt);
            for(uint i = 0;pageSt < end; i++){
                ret[i] = offerTb[offers[pageSt]];
                pageSt++;
            }
        }

        return ret;
    }

    function caculateFees(uint amount) view public returns(uint, uint ){
        // caculate owner fee + taker fee
        return (amount.mul(feeRatio).div(FEE_RATIO_DIV), amount.mul(royaltyRatio).div(FEE_RATIO_DIV))
    }

    // Owner accept the price
    function collectFeeWhenBuyerAsMaker(PayMethod payMethod, uint fees) internal {
        if(payMethod == PayMethod.PAY_BNB) {
            if(address(0) != address(feeCollector)) {
                payable(address(feeCollector)).transfer(fees);
                feeCollector.handleCollectBNB(fees);
            } else {
                payable(address(treasury)).transfer(fees);
            }
        } else if(payMethod == PayMethod.PAY_BUSD) {
            if(address(0) != address(feeCollector)) {
                busdContract.transfer(address(feeCollector), fees);
                feeCollector.handleCollectBUSD(fees);
            } else {
                busdContract.transfer(treasury, fees);
            }
        } else {
            if(address(0) != address(feeCollector)) {
                bibContract.transfer(address(feeCollector), fees);
                feeCollector.handleCollectBIB(fees);
            } else {
                bibContract.transfer(treasury, fees);
            }
        }
    }

    // Buyer accept the price
    function collectFeeWhenSellerAsMaker(PayMethod payMethod, uint fees) internal {
        if(payMethod == PayMethod.PAY_BNB) {
            if(address(0) != address(feeCollector)) {
                payable(address(feeCollector)).transfer(fees);
                feeCollector.handleCollectBNB(fees);
            } else {
                payable(address(treasury)).transfer(fees);
            }
        } else if(payMethod == PayMethod.PAY_BUSD) {
            if(address(0) != address(feeCollector)) {
                busdContract.transferFrom(msg.sender, address(feeCollector), fees);
                feeCollector.handleCollectBUSD(fees);
            } else {
                busdContract.transferFrom(msg.sender, treasury, fees);
            }
        } else {
            if(address(0) != address(feeCollector)) {
                bibContract.transferFrom(msg.sender, address(feeCollector), fees);
```

```
                feeCollector.handleCollectBIB(fees);
            } else {
                bibContract.transferFrom(msg.sender, treasury, fees);
            }
        }
    }

    // Buyer accept the price and makes a deal with the sepcific order
    function acceptOffer(uint orderId) public  override payable {
        Order storage order = orderTb[orderId];
        require(address(0) != order.issuer,"INVALID_ORDER");
        require(msg.sender != order.owner, "SHOULD_NOT_BE_ORDER_OWNER");
        require(order.expiration > block.timestamp, "ORDER_EXPIRED");

        // caculate fees
        (uint txFee, uint royaltyFee )= caculateFees(order.price);
        uint fees = txFee.add(royaltyFee);
        uint amount = order.price.sub(txFee).sub(fees);

        // fee + royalty goese to BIB treasury
        if(order.payMethod == PayMethod.PAY_BNB){
            require(msg.value >= order.price, "INSUFFICIENT_FUNDS");
            payable(address(order.owner)).transfer(amount);

            collectFeeWhenSellerAsMaker(PayMethod.PAY_BNB, fees);

            // refunds
            if(msg.value > order.price){
                payable(address(msg.sender)).transfer(msg.value.sub(order.price));
            }
        } else if(order.payMethod == PayMethod.PAY_BUSD){
            busdContract.transferFrom(msg.sender, order.owner, amount);

            collectFeeWhenSellerAsMaker(PayMethod.PAY_BUSD, fees);
        } else {
            bibContract.transferFrom(msg.sender, order.owner, amount);

            collectFeeWhenSellerAsMaker(PayMethod.PAY_BIB, fees);
        }

        // send token
        IERC721(address(order.issuer)).transferFrom(address(this), msg.sender, order.tokenId);

        emit AcceptOffer(
                msg.sender,
                order.owner,
                msg.sender,
                fees,
                orderId,
                0,
                order.payMethod,
                order.price);

        // refund commodity and currency
        acceptAndRefundOffer(order, orderOfferTb[orderId], orderOfferTb[orderId].length);

        // close order
        _closeOrder(orderId);
    }

    // Owner accept the offer and make a deal
    function acceptOffer(uint orderId, uint offerId) public  override payable{
        Order storage order = orderTb[orderId];
        require(address(0) != order.issuer,"INVALID_ORDER");
        require(msg.sender == order.owner, "SHOULD_BE_ORDER_OWNER");
        require(order.expiration > block.timestamp, "ORDER_EXPIRED");
```

```
        Offer storage offer = offerTb[offerId];
        require(address(0) != offer.buyer, "INVALID_OFFER_ID");
        require(offer.expiration > block.timestamp, "OFFER_EXPIRED");

        // caculate sales
       (uint txFee, uint royaltyFee )= caculateFees(offer.bid);
        uint fees = txFee.add(royaltyFee);
        uint amount = offer.bid.sub(txFee).sub(royaltyFee);

        // fee + royalty goese to BIB treasury
        if(order.payMethod == PayMethod.PAY_BNB){
            payable(address(order.owner)).transfer(amount);
            collectFeeWhenBuyerAsMaker(PayMethod.PAY_BNB, fees);
        } else if(order.payMethod == PayMethod.PAY_BUSD){
            busdContract.transfer(order.owner, amount);
            collectFeeWhenBuyerAsMaker(PayMethod.PAY_BUSD, fees);
        } else {
            bibContract.transfer(order.owner, amount);
            collectFeeWhenBuyerAsMaker(PayMethod.PAY_BIB, fees);
        }

        // send token
        IERC721(address(order.issuer)).transferFrom(address(this), offer.buyer, order.tokenId);

        emit AcceptOffer(
                msg.sender,
                offer.buyer,
                order.owner,
                fees,
                orderId,
                offerId,
                order.payMethod,
                offer.bid);

        // refund commodity and currency
        acceptAndRefundOffer(order, orderOfferTb[orderId], offerId);

        // close order
        _closeOrder(orderId);
    }

    // Owner updates order price
    function updateOrderPrice(uint orderId, uint price) public override payable{
        Order storage order = orderTb[orderId];
        require(address(0) != order.issuer,"INVALID_ORDER");
        require(msg.sender == order.owner, "SHOULD_BE_ORDER_OWNER");
        require(order.expiration > block.timestamp, "ORDER_EXPIRED");
        require(price > 0, "PRICE_LTE_ZERO");

        emit UpdateOrderPrice(msg.sender, orderId, order.price, price);
        order.price = price;
    }

    function _closeOrder(uint orderId) internal {
        Order storage order = orderTb[orderId];
        require(address(0) != order.issuer,"INVALID_ORDER");

        uint indexToRm = orders.length;
        for(uint i = 0; i < orders.length; i++){
           if(orderTb[orders[i]].orderId == orderId){
                indexToRm = i;
           }
        }
        require(indexToRm < orders.length, "ORDER_NOT_EXIST");
        for(uint i = indexToRm; i < orders.length - 1; i++){
```

```
            orders[i] = orders[i+1];
        }
        orders.pop();

        uint[] storage userOrders = userOrderTb[order.owner];
        indexToRm = userOrders.length;
        for(uint i = 0; i < userOrders.length; i++){
           if(orderTb[userOrders[i]].orderId == orderId){
                indexToRm = i;
           }
        }
        require(indexToRm < userOrders.length, "ORDER_NOT_EXIST");
        for(uint i = indexToRm; i < userOrders.length - 1; i++){
            userOrders[i] = userOrders[i+1];
        }
        userOrders.pop();

        uint[] memory offerIds = orderOfferTb[orderId];
        for(uint i = 0; i < offerIds.length; i++){
            delete offerTb[offerIds[i]];
        }
        delete orderOfferTb[orderId];

        delete orderTb[orderId];

        emit CloseOrder(msg.sender, orderId);
    }

    function acceptAndRefundOffer(Order storage order, uint[] storage offers, uint acceptOfferId) int
        for(uint i = 0; i < offers.length; i++){
            Offer storage offer = offerTb[offers[i]];

            if(acceptOfferId == offer.offerId) {
                continue;
            }

            if(order.payMethod == PayMethod.PAY_BNB){
                payable(address(offer.buyer)).transfer(offer.bid);
            } else if(order.payMethod == PayMethod.PAY_BUSD){
                busdContract.transfer(offer.buyer, offer.bid);
            } else {
                bibContract.transfer(offer.buyer, offer.bid);
            }
        }
    }

    // Owner close the specific order if not dealed
    function closeOrder(uint orderId) public override{
        Order storage order = orderTb[orderId];
        require(address(0) != order.issuer,"INVALID_ORDER");
        require(msg.sender == order.owner, "SHOULD_BE_ORDER_OWNER");

        // refund commodity and currency
        acceptAndRefundOffer(order, orderOfferTb[orderId], orderOfferTb[orderId].length);

        IERC721(address(tokenContract)).transferFrom(address(this), order.owner, order.tokenId);

        _closeOrder(orderId);
    }

    // Buyer make a offer to the specific order
    function makeOffer(uint orderId, uint price, uint expiration) public override payable{
        Order storage order = orderTb[orderId];
        require(address(0) != order.issuer,"INVALID_ORDER");
        require(msg.sender != order.owner, "CANT_MAKE_OFFER_WITH_SELF");
        require(expiration > block.timestamp, "EXPIRATION_TOOL_SMALL");
```

```
        require(price > 0, "PRICE_NOT_BE_ZERO");

        // check if has made offer before
        uint[] storage offers = orderOfferTb[orderId];
        for(uint i = 0; i < offers.length; i++){
            if(offerTb[offers[i]].buyer == msg.sender){
                revert("HAS_MADE_OFFER");
            }
        }

        if(order.payMethod == PayMethod.PAY_BNB){
            require(msg.value >= price, "INSUFFICIENT_FUNDS");
            // refunds
            if(msg.value > order.price){
                payable(address(msg.sender)).transfer(msg.value.sub(price));
            }
        } else if(order.payMethod == PayMethod.PAY_BUSD){
            busdContract.transferFrom(msg.sender, address(this), price);
        } else {
            bibContract.transferFrom(msg.sender, address(this), price);
        }

        Offer memory offer = Offer({
            offerId: nextOfferIndex,
            buyer: msg.sender,
            bid: price,
            expiration: expiration
        });

        orderOfferTb[orderId].push(offer.offerId);
        offerTb[nextOfferIndex] = offer;

        emit MakeOffer(msg.sender,  order.owner, orderId, nextOfferIndex++, price, expiration);
    }

    // Buyer udpate offer bid price
    function updateOffer(uint orderId, uint offerId, uint price) public override payable{
        Order storage order = orderTb[orderId];
        require(address(0) != order.issuer,"INVALID_ORDER");
        require(order.expiration > block.timestamp, "ORDER_EXPIRED");

        Offer storage offer = offerTb[offerId];
        require(msg.sender == offer.buyer, "INVALID_OFFER_ID");
        require(offer.expiration > block.timestamp, "OFFER_EXPIRED");
        require(price > 0, "PRICE_NOT_BE_ZERO");

        uint delt  = 0;
        if(offer.bid > price){
            delt = offer.bid.sub(price);
            if(order.payMethod == PayMethod.PAY_BNB){
                payable(address(offer.buyer)).transfer(delt);
            } else if(order.payMethod == PayMethod.PAY_BUSD){
                busdContract.transfer(offer.buyer, delt);
            } else {
                bibContract.transfer(offer.buyer, delt);
            }
        } else {
            delt = price.sub(offer.bid);
            if(order.payMethod == PayMethod.PAY_BNB){
                require(msg.value >= delt, "INSUFFICIENT_FUNDS");
                // refunds
                if(msg.value > delt){
                    payable(address(msg.sender)).transfer(msg.value.sub(delt));
                }
            } else if(order.payMethod == PayMethod.PAY_BUSD){
                busdContract.transferFrom(msg.sender, address(this), delt);
```

```
            } else {
                bibContract.transferFrom(msg.sender, address(this), delt);
            }
        }

        emit UpdateOfferPrice(msg.sender, orderId, offerId, offer.bid, price);

        offer.bid = price;
    }

    // Buyer cancle the specific order
    function cancelOffer(uint orderId, uint offerId) public override{
        Order storage order = orderTb[orderId];
        require(address(0) != order.issuer,"INVALID_ORDER");

        Offer storage offer = offerTb[offerId];
        require(msg.sender == offer.buyer, "SHOULD_BE_BUYER");

        if(order.payMethod == PayMethod.PAY_BNB){
            payable(address(offer.buyer)).transfer(offer.bid);
        } else if(order.payMethod == PayMethod.PAY_BUSD){
            busdContract.transfer(offer.buyer, offer.bid);
        } else {
            bibContract.transfer(offer.buyer, offer.bid);
        }

        uint[] storage offers = orderOfferTb[orderId];
        uint indexToRm = offers.length;
        for(uint i = 0; i < offers.length; i++){
           if(offerTb[offers[i]].offerId == offerId){
                indexToRm = i;
           }
        }
        require(indexToRm < offers.length, "OFFER_NOT_EXIST");
        for(uint i = indexToRm; i < offers.length - 1; i++){
            offers[i] = offers[i+1];
        }
        offers.pop();
        delete offerTb[offerId];

        emit CancelOffer(msg.sender, orderId, offerId);
    }
}


// SPDX-License-Identifier: agpl-3.0
pragma solidity >=0.8.0;

import {SafeMath} from "../libs/SafeMath.sol";
import {SafeCast} from "../libs/SafeCast.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {SafeERC20} from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import {Ownable} from  "@openzeppelin/contracts/access/Ownable.sol";
import {IRewardDistributor} from "../interfaces/IRewardDistributor.sol";

contract FeeCollector is  Ownable {
    using SafeMath for uint;
    using SafeCast for uint;

    uint totalBNBRecieved;
    uint totalBUSDRecieved;
    uint totalBIBRecieved;

    mapping(address=>bool) protocolAddress;

    uint public vaultRatio;
```

```
    uint public constant FEE_RATIO_DIV = 1000;

    IRewardDistributor rewardDistributor;
    address vault;

    IERC20 bibToken;
    IERC20 busdToken;

    enum TokenType{
        TOKEN_TYPE_ETH,
        TOKEN_TYPE_BIB,
        TOKEN_TYPE_BUSD
    }

    event VaultChanged(address sender, address oldValue, address newValue);
    event BIBContractChanged(address sender, address oldValue, address newValue);
    event BUSDContractChanged(address sender, address oldValue, address newValue);
    event VaultRatioChanged(address sender, uint oldValue, uint newValue);
    event RewardDistributorChanged(address sender, address oldValue, address newValue);
    event HandleCollect(address sender, uint vault, uint reward, TokenType tokenType);

    constructor(
        address _vault,
        address _bibToken,
        address _busdToken,
        address _rewardDistributor
        ){
        vault = _vault;
        bibToken = IERC20(_bibToken);
        busdToken = IERC20(_busdToken);
        rewardDistributor = IRewardDistributor(_rewardDistributor);
    }

    function setBIBContract(address _bibToken) public onlyOwner{
        require(address(0) != _bibToken, "INVALID_ADDRESS");
        emit BIBContractChanged(msg.sender, address(bibToken), _bibToken);
        bibToken = IERC20(_bibToken);
    }

    function setRewardDistributor(address _rewardDistributor) public onlyOwner{
        require(address(0) != _rewardDistributor, "INVALID_ADDRESS");
        emit RewardDistributorChanged(msg.sender, address(rewardDistributor), _rewardDistributor);
        rewardDistributor = IRewardDistributor(_rewardDistributor);
    }

    function setBUSDContract(address _busdToken) public onlyOwner{
        require(address(0) != _busdToken, "INVALID_ADDRESS");
        emit BUSDContractChanged(msg.sender, address(busdToken), _busdToken);
        busdToken = IERC20(_busdToken);
    }

    function setVault(address _vault) public onlyOwner{
        require(address(0) != _vault, "INVALID_ADDRESS");
        emit VaultChanged(msg.sender, address(vault), _vault);
        vault = _vault;
    }

    function setVaultRatio(uint _vaultRatio) public onlyOwner{
        require(_vaultRatio <= FEE_RATIO_DIV, "INVALID_RATIO");
        emit VaultRatioChanged(msg.sender, vaultRatio, _vaultRatio);
        vaultRatio = _vaultRatio;
    }

    function addProtocolAdress(address protocolAddr) public onlyOwner{
        protocolAddress[protocolAddr] = true;
    }
```

```
    function removeProtocolAddress(address protocolAddr) public onlyOwner{
        delete protocolAddress[protocolAddr];
    }

    function isProtocolAddress(address protocolAddr) public view returns(bool){
        return protocolAddress[protocolAddr];
    }

    modifier onlyProtocolAddress(){
        require(protocolAddress[msg.sender], "ONLY_PROTOCOL_ADDRESS_CAN_CALL");
        _;
    }

    function caculateFees(uint amount) public pure  returns(uint, uint){
        uint vaultPart =  amount.mul(FEE_RATIO_DIV).div(FEE_RATIO_DIV);
        return (vaultPart, amount.sub(vaultPart));
    }

    function distributeFees() public onlyOwner(){
        handleCollectBIB(bibToken.balanceOf(address(this)));
        handleCollectBUSD(busdToken.balanceOf(address(this)));
        handleCollectBNB(address(this).balance);
    }

    function handleCollectBIB(uint amount) public onlyProtocolAddress{
        if(address(0) != address(rewardDistributor) && address(0) != vault){
            (uint vaultPart, uint rewardPart) = caculateFees(amount);
            bibToken.transfer(vault, vaultPart);
            bibToken.transfer(address(rewardDistributor), rewardPart);
            rewardDistributor.distributeBIBReward(amount);
            emit HandleCollect(msg.sender, vaultPart, rewardPart, TokenType.TOKEN_TYPE_BIB);
        }
    }

    function handleCollectBUSD(uint amount) public onlyProtocolAddress{
        if(address(0) != address(rewardDistributor) && address(0) != vault){
            (uint vaultPart, uint rewardPart) = caculateFees(amount);
            busdToken.transfer(vault, vaultPart);
            busdToken.transfer(address(rewardDistributor), rewardPart);
            rewardDistributor.distributeBUSDReward(amount);
            emit HandleCollect(msg.sender, vaultPart, rewardPart, TokenType.TOKEN_TYPE_BUSD);
        }
    }

    function handleCollectBNB(uint amount) public onlyProtocolAddress{
        if(address(0) != address(rewardDistributor) && address(0) != vault){
            (uint vaultPart, uint rewardPart) = caculateFees(amount);
            payable(vault).transfer(vaultPart);
            rewardDistributor.distributeETHReward{value:rewardPart}(amount);
            emit HandleCollect(msg.sender, vaultPart, rewardPart, TokenType.TOKEN_TYPE_BUSD);
        }
    }
}


// SPDX-License-Identifier: MIT

pragma solidity >=0.8.0;

import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/utils/Strings.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import '@uniswap/v2-periphery/contracts/interfaces/IUniswapV2Router02.sol';
import "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";
```

```
import "@openzeppelin/contracts-upgradeable/security/PausableUpgradeable.sol";
import "../interfaces/ISoccerStarNft.sol";
import "erc721a/contracts/ERC721A.sol";
import {SafeMath} from "../libs/SafeMath.sol";

contract SoccerStarNft is
ISoccerStarNft,
ERC721A,
Ownable {
    using Strings for uint;
    using SafeMath for uint;

    IERC20 public bibContract;
    IERC20 public busdContract;
    IUniswapV2Router02 public router;

    //URI of the NFTs when revealed
    string public baseURI;
    //URI of the NFTs when not revealed
    string public notRevealedURI;
    //The extension of the file containing the Metadatas of the NFTs
    string public constant BASE_EXTENSION = ".json";

    //Are the NFTs revealed yet ?
    bool public revealed = false;

    uint constant public ORACLE_PRECISION = 1e18;

    address constant public BLACK_HOLE = 0x0000000000000000000000000000000000000001;

    uint public maxMintSupply;

    uint constant public PRE_SELL_ROUND  = 0;
    uint constant public PUB_SELL_ROUND1 = 1;
    uint constant public PUB_SELL_ROUND2 = 2;
    uint constant public PUB_SELL_ROUND3 = 3;
    uint constant public PUB_SELL_ROUND4 = 4;
    uint constant public PUB_SELL_ROUND5 = 5;
    uint constant public MAX_ROUND = PUB_SELL_ROUND5;

    event BIBContractChanged(address sender, address oldValue, address newValue);
    event BUSDContractChanged(address sender, address oldValue, address newValue);
    event TreasuryChanged(address sender, address oldValue, address newValue);
    event SwapRouterChanged(address sender, address oldValue, address newValue);
    event ComposerChanged(address sender, address oldValue, address newValue);
    event SellTimeChanged(address sender, uint oldValue, uint newValue);
    event Changed(address sender, uint oldValue, uint newValue);
    event UpdateStarLevel(address sender, uint oldValue, uint newValue);

    address public treasury;

    uint256 public maxPubicsaleUserMintAmount;

    uint constant public MAX_PROPERTY_VALUE = 4;

    mapping(uint256 => bool) public isOwnerMint; // if the NFT was freely minted by owner
    mapping(uint256 => SoccerStar) public cardProperty;

    // round->boxType->price
    mapping(uint=>mapping(BlindBoxesType=>uint)) public mintPriceTb;
    // round->boxType->amount
    mapping(uint=>mapping(BlindBoxesType=>uint)) public maxAmountTb;
    // round->boxType->maxAmount
    mapping(uint=>mapping(BlindBoxesType=>uint)) public mintAmountTb;
    mapping(address=>mapping(uint=>uint))        public mintAmountPerAddrTb;
    mapping(uint=>TimeInfo) public timeInfoTb;
```

```
mapping(address=>bool) public allowProtocolToCallTb;
mapping(address=>bool) public allowToCallTb;

struct QuotaTracker {
    uint quota;
    uint used;
}

// track busd quota per public round
mapping(uint=>QuotaTracker) public busdQuotaPerPubRoundTb;

// track user quota at pre-round
mapping(address=>QuotaTracker) public userQutaPreRoundTb;

constructor(uint _maxMintSupply,
address _bibContract,
address _busdContract,
address _treasury,
address _router)ERC721A("BIBMetaSuperstar", "BMSTAR"){
    maxMintSupply = _maxMintSupply;
    bibContract = IERC20(_bibContract);
    busdContract = IERC20(_busdContract);
    treasury = _treasury;
    router = IUniswapV2Router02(_router);

    maxPubicsaleUserMintAmount = 10;
    revealed = false;
}

function setAllowProtocolToCall(address _protAddr, bool value)
public onlyOwner{
    allowProtocolToCallTb[_protAddr] = value;
}

modifier onlyAllowProtocolToCall() {
    require(allowProtocolToCallTb[msg.sender], "ONLY_PROTOCOL_CALL");
    _;
}

function setAllowToCall(address _caller, bool value) public onlyOwner{
    allowToCallTb[_caller] = value;
}

modifier onlyAllowToCall(){
      require(allowToCallTb[msg.sender] || msg.sender == owner(), "ONLY_PERMIT_CALLER");
    _;
}

function  getRemainingAmount(uint round, BlindBoxesType boxType)
public view returns(uint){
    return maxAmountTb[round][boxType].sub(mintAmountTb[round][boxType]);
}

function setBIBContract(address _bibContract) public onlyOwner{
    require(address(0) != _bibContract, "INVLID_ADDRESS");
    emit BIBContractChanged(msg.sender, address(bibContract), _bibContract);
    bibContract = IERC20(_bibContract);
}

function setTreasury(address _treasury) public onlyOwner{
    require(address(0) != _treasury, "INVLID_ADDRESS");
    emit TreasuryChanged(msg.sender, treasury, _treasury);
    treasury = _treasury;
}
```

```
    function setSwapRouter(address _router) public onlyOwner{
        require(address(0) != _router, "INVLID_ADDRESS");
        emit SwapRouterChanged(msg.sender, address(router), _router);
        router = IUniswapV2Router02(_router);
    }

    function setBUSDContract(address _busdContract) public onlyOwner{
        require(address(0) != _busdContract, "INVLID_ADDRESS");
        emit BUSDContractChanged(msg.sender, address(busdContract), _busdContract);
        busdContract = IERC20(_busdContract);
    }

    function caculateBUSDAmount(uint bibAmount) public view returns(uint){
        // the price has ORACLE_PRECISION
        address[] memory path = new address[](2);
        path[0] = address(bibContract);
        path[1] = address(busdContract);
        return router.getAmountsOut(bibAmount, path)[1];
    }

   // only allow protocol related contract to mint
    function protocolMint()
    public override onlyAllowProtocolToCall returns(uint tokenId){
        tokenId = _nextTokenId();
        _mint(msg.sender, 1);
        require(msg.sender == ownerOf(tokenId), "PROTOCOL_MINT_FAILED");
    }

    // only allow protocol related contract to mint to burn
    function protocolBurn(uint tokenId)
    public override onlyAllowProtocolToCall {
        require(msg.sender == ownerOf(tokenId), "TOKEN_NOT_BELLONG_TO_CALLER");
        _burn(tokenId);
        require(msg.sender != ownerOf(tokenId), "PROTOCOL_BRUN_FAILED");
    }

    // only allow protocol related contract to bind star property
    function protocolBind(uint tokenId, SoccerStar memory soccerStar)
    public override onlyAllowProtocolToCall {
        require(msg.sender == ownerOf(tokenId), "TOKEN_NOT_BELLONG_TO_CALLER");
        require(cardProperty[tokenId].starLevel == 0, "TOKEN_REVEALED");
        cardProperty[tokenId] = soccerStar;
    }

    function addUserQuotaPreRoundBatch(address[] memory users, uint[] memory quotas)
    public override onlyAllowToCall {
        require(users.length == quotas.length, "SHOULD_BE_SAME_LENGTH");
        for(uint i = 0; i < users.length; i++){
            userQutaPreRoundTb[users[i]].quota = quotas[i];
        }
    }

    function setUserQuotaPreRound(address user, uint quota)
    public override onlyAllowToCall {
        userQutaPreRoundTb[user].quota = quota;
    }

    function getUserQuotaPreRound(address user) public override view returns(uint){
        return userQutaPreRoundTb[user].quota;
    }

    function getUserRemainningQuotaPreRound(address user) public view override returns(uint){
        return userQutaPreRoundTb[user].quota.sub(userQutaPreRoundTb[user].used);
    }

    function setBUSDQuotaPerPubRound(uint round, uint quota)
```

```
    public override onlyAllowToCall {
        require(isPublicRound(round), "NOT_PUBLIC_ROUND");
        busdQuotaPerPubRoundTb[round].quota = quota;
    }

    function getBUSDQuotaPerPubRound(uint round) public view override returns(uint){
        require(isPublicRound(round), "NOT_PUBLIC_ROUND");
        return busdQuotaPerPubRoundTb[round].quota;
    }

    function getBUSDUsedQuotaPerPubRound(uint round) public view override returns(uint){
        require(isPublicRound(round), "NOT_PUBLIC_ROUND");
        return busdQuotaPerPubRoundTb[round].used;
    }

    function getBUSDRemainningQuotaPerPubRound(uint round) public view returns(uint){
        require(isPublicRound(round), "NOT_PUBLIC_ROUND");
        return busdQuotaPerPubRoundTb[round].quota.sub(busdQuotaPerPubRoundTb[round].used);
    }

    function setMaxMintAmount(uint round, BlindBoxesType boxType, uint amount)
    public onlyAllowToCall{
        maxAmountTb[round][boxType] = amount;
    }

    function getMaxMintAmount(uint round, BlindBoxesType boxType) public view returns(uint){
        return maxAmountTb[round][boxType];
    }

    function setSellTime(uint round, uint _startTime, uint _endTime, uint _revealTime)
    public onlyAllowToCall {
        require(round <= MAX_ROUND, "INVLID_ROUND");
        require(_startTime >= block.timestamp, "INVLID_START_TIME");
        require(_endTime >= _startTime, "INVLID_END_TIME");
        require(_revealTime >= _endTime, "INVLID_END_TIME");

        timeInfoTb[round] = TimeInfo({
            startTime: _startTime,
            endTime: _endTime,
            revealTime: _revealTime
        });
    }

    function getSellTime(uint round) public view returns (uint){
        return timeInfoTb[round].startTime;
    }

    function setMaxMintSupply(uint _maxMintSupply) public onlyAllowToCall{
        maxMintSupply = _maxMintSupply;
    }

    function getMaxMintSupply() public view returns(uint){
        return maxMintSupply;
    }

    function setMaxAmountPerAddress(uint _amount)
    public onlyAllowToCall{
        maxPubicsaleUserMintAmount = _amount;
    }

    function getMaxAmountPerAddress() public view returns(uint){
        return maxPubicsaleUserMintAmount;
    }

    function setMintPrice(uint round, uint256 _mintPrice, BlindBoxesType boxType)
    public onlyAllowToCall {
```

```
        require(round<= MAX_ROUND, "INVLID_ROUND");
        mintPriceTb[round][boxType] = _mintPrice;
    }

    function getMintPrice(uint round, BlindBoxesType boxType) public view returns(uint){
        return mintPriceTb[round][boxType];
    }

    function setBaseURI(string memory uri) external onlyAllowToCall {
        baseURI = uri;
    }

     /**
    * @notice Allows to set the revealed variable to true
    **/
    function reveal(bool _revealed)
    external onlyAllowToCall {
        revealed = _revealed;
    }

    function getCardProperty(uint256 tokenId) public view override
    returns(SoccerStar memory){
        return cardProperty[tokenId];
    }

   function updateProperty(uint[] memory tokenIds, SoccerStar[] memory _soccerStars)
        external
        onlyAllowToCall{
        require(tokenIds.length == _soccerStars.length, "NEED_SAME_LENGTH");
        for(uint i = 0; i < _soccerStars.length; i++){
            cardProperty[tokenIds[i]] = _soccerStars[i];
            require(cardProperty[tokenIds[i]].starLevel != 0, "INVALID_PROPERTY");
        }
    }

    function isRoundOpen(uint round) public view returns(bool){
        TimeInfo storage timeInfo = timeInfoTb[round];
        return (currentTime() >= timeInfo.startTime)
        && (currentTime() <= timeInfo.endTime);
    }

    function preSellMint(uint256 quantity)
        external
        payable{
        require(isRoundOpen(PRE_SELL_ROUND), "PRE_SELL_ROUND_NOT_OPENED");
        require(getUserRemainningQuotaPreRound(msg.sender) >= quantity, "USER_HAS_NO_QUOTA");
        require(
           getRemainingAmount(PRE_SELL_ROUND,BlindBoxesType.presale) >=  quantity,
            "EXCEED_MAX_MINT_AMOUNT"
        );

        // burn bib tokens
        uint sales = quantity.mul(getMintPrice(PRE_SELL_ROUND, BlindBoxesType.presale));
        bibContract.transferFrom(msg.sender, BLACK_HOLE, sales);

        _safeMint(msg.sender, quantity);

        mintAmountTb[PRE_SELL_ROUND][BlindBoxesType.presale] = mintAmountTb[PRE_SELL_ROUND][BlindBoxe

        // deducate user presell quota
        userQutaPreRoundTb[msg.sender].used = userQutaPreRoundTb[msg.sender].used.add(quantity);

        emit Mint(msg.sender,
        PRE_SELL_ROUND,
        BlindBoxesType.presale,
        _nextTokenId().sub(quantity),
```

```
        quantity,
        PayMethod.PAY_BIB,
        sales);
    }

    function isPublicRound(uint round) public pure returns(bool){
        return (round >= PUB_SELL_ROUND1) && (round <= PUB_SELL_ROUND5);
    }

    function getPubicRoundMintAmountByUser(address user, uint round) public view returns(uint){
        return mintAmountPerAddrTb[user][round];
    }

    function publicSellMint(
    uint round,
    BlindBoxesType boxType,
    uint256 quantity,
    PayMethod payMethod) public {
        require(isPublicRound(round), "NOT_PUBLIC_ROUND_NUM");
        require(isRoundOpen(round), "ROUND_NOT_OPEN");
        require(boxType != BlindBoxesType.presale, "PRESALE_BOX_NOT_ALLOWED");

        // check constraint per address
        require(getPubicRoundMintAmountByUser(msg.sender, round).add(quantity) <= getMaxAmountPerAddr
                "EXCEED_ADDRESS_MAX_MINT_AMOUNT");
        // check constraint per round && box type
        require(getRemainingAmount(round, boxType) >=  quantity,
                "EXCEED_MAX_MINT_AMOUNT");

        uint sales = quantity.mul(getMintPrice(round, boxType));
        if(payMethod == PayMethod.PAY_BIB){
            // burn out bib
            bibContract.transferFrom(msg.sender, BLACK_HOLE, sales);
        } else {
            //check BUSD quota
            require(getBUSDRemainningQuotaPerPubRound(round) >= quantity, "EXCEED_MAX_BUSD_QUOTA");
            sales = caculateBUSDAmount(sales);

            // transfer to treasury
            busdContract.transferFrom(msg.sender, treasury, sales);

            // update used quota
            busdQuotaPerPubRoundTb[round].used = busdQuotaPerPubRoundTb[round].used.add(quantity);
        }
        _safeMint(msg.sender, quantity);

        mintAmountTb[round][boxType] = mintAmountTb[round][boxType].add(quantity);
        mintAmountPerAddrTb[msg.sender][round] = mintAmountPerAddrTb[msg.sender][round].add(quantity)

        emit Mint(msg.sender,
        round,
        boxType,
        _nextTokenId().sub(quantity),
        quantity,
        payMethod,
        sales);
     }

    function ownerMint(uint256 quantity) external onlyOwner {
        require(
            _totalMinted() + quantity <= getMaxMintSupply(),
            "MAX_SUPPLY_REACHED"
        );

         _safeMint(msg.sender, quantity);
```

```
        for (uint256 i = _nextTokenId() - quantity; i < _nextTokenId(); i++) {
            isOwnerMint[i] = true;
        }

        emit Mint(msg.sender,
        PRE_SELL_ROUND,
        BlindBoxesType.presale,
        _nextTokenId().sub(quantity),
        quantity,
        PayMethod.PAY_BIB,
        0);
    }

    function currentTime() public view returns(uint) {
        return block.timestamp;
    }

    function _startTokenId() internal view virtual override returns (uint256) {
        return 1;
    }

    function tokenURI(uint _nftId) public view override(ERC721A) returns (string memory) {
        require(_exists(_nftId), "This NFT doesn't exist.");
        if(revealed == false) {
            return notRevealedURI;
        }

        string memory currentBaseURI = _baseURI();
        return
            bytes(currentBaseURI).length > 0
            ? string(abi.encodePacked(currentBaseURI, _nftId.toString(), BASE_EXTENSION))
            : "";
    }

    function _baseURI() internal view override returns (string memory) {
        return baseURI;
    }

    function setNotRevealURI(string memory _notRevealedURI) external onlyOwner {
        notRevealedURI = _notRevealedURI;
    }
}


//SPDX-License-Identifier: MIT

pragma solidity >=0.8.0;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC721/IERC721.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/Address.sol";
import {SafeMath} from "../libs/SafeMath.sol";
import {SafeCast} from "../libs/SafeCast.sol";
import {ISoccerStarNft} from "../interfaces/ISoccerStarNft.sol";
import {IStakedSoccerStarNft} from "../interfaces/IStakedSoccerStarNft.sol";

contract StakedSoccerStarNft is IStakedSoccerStarNft, Ownable {
    using SafeMath for uint;
    using SafeCast for uint;

    event FrozenDurationChanged(uint oldValue, uint newValue);
    event RewardPeriodChanged(uint oldValue, uint newValue);
    event RewardStartChanged(uint oldValue, uint newValue);
    event RewardContractChanged(address oldValue, address newValue);
    event NftContractChanged(address oldValue, address newValue);
```

```
    IERC20 public rewardContract;
    IERC721 public nftContract;

    int constant public INVLID_ROUND = -1;
    uint public rewardPeriod = 1 days;
    uint public frozenDuration = 7 days;
    uint public rewardStart;
    uint public totalStaked;
    uint public totalPower;

    DepositInfo[] public depositInfos;

    mapping(address=>UserStakedInfo[]) public stakedInfos;

    // Keep a fast index to avoid too many times of loop
    mapping(uint=>UserStakedInfo) public tokenToUserStakedInfo;

    constructor(
        address _rewardContract,
        address _nftContract,
        uint _rewardPeriod,
        uint _rewardStart
        ){
        rewardContract = IERC20(_rewardContract);
        nftContract = IERC721(_nftContract);
        rewardPeriod = _rewardPeriod;
        rewardStart = _rewardStart;
    }

    modifier onlyContract(address cntr) {
        require(address(0) != cntr, "INVALID_ADDRESS");
        require(Address.isContract(cntr), "NOT_CONTRACT");
        _;
    }

    modifier onlyStarted(){
        require(isStakedStart(), "REWARD_NOT_START");
        _;
    }

    modifier ownToken(uint tokenId){
        require(msg.sender == nftContract.ownerOf(tokenId), "TOKEN_NOT_BELONG_TO_SENDER");
        _;
    }

    // check if staked start
    function isStakedStart() public view returns(bool){
        return rewardStart <= block.timestamp;
    }

   function setFrozenDuration(uint newValue) public onlyOwner {
        emit FrozenDurationChanged(frozenDuration, newValue);
        frozenDuration = newValue;
    }

    function setRewardPeroid(uint newValue) public onlyOwner {
        require(newValue >= 1 days,  "REWARD_PERIOD_TOO_SHORT");
        emit RewardPeriodChanged(rewardPeriod, newValue);
        rewardPeriod = newValue;
    }

    function setRewardStart(uint newValue) public onlyOwner {
        require(newValue >= block.timestamp, "REWARD_START_LESS_THAN_CURRENT");
        emit RewardStartChanged(rewardStart, newValue);
        rewardStart = newValue;
```

```
    }

    function setRewardContract(address newValue) public onlyOwner onlyContract(newValue){
        emit RewardContractChanged(address(rewardContract), newValue);
        rewardContract = IERC20(newValue);
    }

    function setNftContract(address newValue) public onlyOwner onlyContract(newValue){
        emit NftContractChanged(address(nftContract), newValue);
        nftContract = IERC721(newValue);
    }

      // deposit a specified funds to pool
    function deposit(uint amount) public override onlyStarted{
        require(amount > 0, "AMOUNT_TOOL_SMALL");

        (bool hasRound, uint round, DepositInfo memory roundInfo) = checkAndGetRound();
        if(!hasRound || (round > roundInfo.round)){
            // close old
            if(hasRound){
                closeRound();
            }

            DepositInfo memory depositInfo = DepositInfo({
                round:round,
                totalDeposit:amount,
                totalStaked:0,
                totalPower:0,
                totalClaimed:0
            });
            depositInfos.push(depositInfo);
        }else {
            roundInfo.totalDeposit += amount;
            depositInfos[depositInfos.length - 1] = roundInfo;
        }

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

        emit Deposit(msg.sender, round, amount);
    }

    function getCurrentRound() public view override onlyStarted returns(uint round){
        return block.timestamp.sub(rewardStart).div(rewardPeriod);
    }

    function checkAndGetRound() public view  returns(bool, uint, DepositInfo memory){
        DepositInfo memory roundInfo;
        if(depositInfos.length <= 0){
            return (false, 0, roundInfo);
        }
        uint round = getCurrentRound();
        roundInfo = depositInfos[depositInfos.length - 1];
        require(round >= roundInfo.round, "INVALID_ROUND");
        return (true, round, roundInfo);
    }

    // close a reward period
    function closeRound() public onlyStarted override{
        (bool hasRound, uint round, DepositInfo memory roundInfo) = checkAndGetRound();
        if(hasRound && (round > roundInfo.round) ){
            // finalize total amount
            roundInfo.totalStaked = totalStaked;
            roundInfo.totalPower = totalPower;
            depositInfos[depositInfos.length - 1] = roundInfo;

            emit CloseRound(
```

```
            msg.sender,
            roundInfo.round,
            roundInfo.totalDeposit,
            roundInfo.totalStaked,
            roundInfo.totalPower
            );
        }
    }

    // check is the specified token is staked
    function isStaked(uint tokenId) public view override returns(bool){
        // give that the token id start with 1
        return tokenToUserStakedInfo[tokenId].tokenId == tokenId
        && tokenToUserStakedInfo[tokenId].unfrozenTime == 0;
    }

    // Check if is the specified token is staking
    function isStaking(uint tokenId) public view override returns(bool){
        return (tokenToUserStakedInfo[tokenId].tokenId == tokenId);
    }

    // Check if the specified token is unfreezing
    function isUnfreezing(uint tokenId) public view override returns(bool){
        uint unfrozenTime = tokenToUserStakedInfo[tokenId].unfrozenTime;
        return (unfrozenTime > 0
        && block.timestamp <=  unfrozenTime);
    }

    // Check if the specified token is withdrawable
    function isWithdrawAble(uint tokenId) public view override returns(bool){
        uint unfrozenTime = tokenToUserStakedInfo[tokenId].unfrozenTime;
        return (unfrozenTime > 0 && block.timestamp >  unfrozenTime);
    }

    // user staken one or more nft card to safty module
    function stake(uint tokenId) public override onlyStarted ownToken(tokenId){
        require(!isStaking(tokenId), "TOKEN_STAKING");

        // check if need to close the old round
        closeRound();

        totalStaked++;
        totalPower += getTokenPower(tokenId);

        UserStakedInfo memory userStakedInfo = UserStakedInfo({
            tokenId: tokenId,
            round: getCurrentRound(),
            unfrozenTime: 0,
            claimedRound: INVLID_ROUND
        });
        stakedInfos[msg.sender].push(userStakedInfo);
        tokenToUserStakedInfo[tokenId] = userStakedInfo;

        nftContract.transferFrom(msg.sender, address(this), tokenId);

        emit Stake(msg.sender, tokenId);
    }

    function getTokenPower(uint tokenId) public view returns(uint power){
        ISoccerStarNft.SoccerStar memory cardInfo = ISoccerStarNft(address(nftContract)).getCardPrope
        require(cardInfo.starLevel > 0, "CARD_UNREAL");
        // The power equation: power = gradient * 10 ^ (starLevel -1)
        return cardInfo.gradient.exp(cardInfo.starLevel.sub(1));
    }

    // user redeem one or more nft cards
```

```
    function redeem(uint tokenId) public override onlyStarted ownToken(tokenId){
        require(isStaked(tokenId), "TOKEN_NOT_STAKED");

        // check if need to close the old round
        closeRound();

        // claim all rewards
        claimRewards();

        // update global data
        totalPower -= getTokenPower(tokenId);
        totalStaked--;

        // update fast index
        tokenToUserStakedInfo[tokenId].unfrozenTime = block.timestamp + frozenDuration;

        UserStakedInfo[] storage userStakedInfos = stakedInfos[msg.sender];
        uint indexToRm = userStakedInfos.length;
        for(uint i = 0; i < userStakedInfos.length; i++){
           if(userStakedInfos[i].tokenId == tokenId){
                indexToRm = i;
           }
        }
        require(indexToRm < userStakedInfos.length, "TOKEN_NOT_EXIST");
        // delete from index
        for(uint i = indexToRm; i < userStakedInfos.length - 1; i++){
            userStakedInfos[i] = userStakedInfos[i+1];
        }
        userStakedInfos.pop();

        emit Redeem(msg.sender, tokenId);
    }

    // withdraw token after the unfrozen period
    function withdraw(uint tokenId) public override onlyStarted ownToken(tokenId){
        require(isWithdrawAble(tokenId), "TOKEN_NOT_WITHDRAWABLE");

        // check if need to close the old round
        closeRound();

        // delete from the fast index
        delete tokenToUserStakedInfo[tokenId];

        nftContract.transferFrom(address(this), msg.sender, tokenId);

        emit Withdraw(msg.sender, tokenId);
    }

    function getUnClaimedRewardsByToken(uint tokenId) public view override returns(uint){
        if(!isStaked(tokenId)){
            return 0;
        }

        if( 0 == depositInfos.length){
            return 0;
        }

        uint curRound = getCurrentRound();
        uint totalRewards = 0;

        UserStakedInfo storage userStakedInfo = tokenToUserStakedInfo[tokenId];
        if(curRound != userStakedInfo.round){

            // walk through from the last to the frist
            for(uint j = depositInfos.length - 1; j >= 0; ){
```

```
                DepositInfo storage depositInfo = depositInfos[j];

                if(depositInfo.round != curRound){
                    if(depositInfo.round.toInt() == userStakedInfo.claimedRound){
                        break;
                    }

                    if(depositInfo.totalPower > 0
                    && depositInfo.totalDeposit > 0){
                        totalRewards += depositInfo.totalDeposit.mul(getTokenPower(userStakedInfo.tok
                    }
                }

                // avoid overflow
                if(j > 0){
                    j--;
                } else {
                    break;
                }
            }
        }
        return totalRewards;
    }

        // Get unclaimed rewards by a set of the specified tokens
    function getUnClaimedRewardsByTokens(uint[] memory tokenIds) public view override returns(uint[]
        uint[] memory rewards = new uint[](tokenIds.length);
        for(uint i = 0; i < tokenIds.length; i++){
            rewards[i] = getUnClaimedRewardsByToken(tokenIds[i]);
        }
        return rewards;
    }

    function getUnClaimedRewards(address user) public override view returns(uint amount){
        uint totalRewards = 0;

        // go through to accurate the rewards
        UserStakedInfo[] storage userStakedInfos = stakedInfos[user];
        for(uint i = 0; i < userStakedInfos.length; i++){
            totalRewards += getUnClaimedRewardsByToken(userStakedInfos[i].tokenId);
        }

        return totalRewards;
    }

    // Get unclaimed rewards
    function getAndMarkUnClaimedRewards(address user, bool markClaim)
     internal returns(uint amount){
        uint totalRewards = 0;
        uint curRound = getCurrentRound();

        if(0 == depositInfos.length){
            return 0;
        }

        // go through to accurate the rewards
        UserStakedInfo[] storage userStakedInfos = stakedInfos[user];
        for(uint i = 0; i < userStakedInfos.length; i++){

            UserStakedInfo storage userStakedInfo = userStakedInfos[i];

            int tailRound = INVLID_ROUND;

            if(curRound != userStakedInfo.round){

                // walk through from the last to the frist
```

```
                for(uint j = depositInfos.length - 1; j >= 0; ){

                    DepositInfo storage depositInfo = depositInfos[j];

                    if(depositInfo.round != curRound){

                        if(depositInfo.round.toInt() == userStakedInfo.claimedRound){
                            break;
                        }

                        // record the tail round
                        if(tailRound == INVLID_ROUND){
                            tailRound = depositInfo.round.toInt();
                        }

                        if(depositInfo.totalPower > 0
                        && depositInfo.totalDeposit > 0){
                            uint share = depositInfo.totalDeposit
                            .mul(getTokenPower(userStakedInfo.tokenId))
                            .div(depositInfo.totalPower);

                            if(markClaim){
                                depositInfo.totalClaimed = depositInfo.totalClaimed.add(share);
                            }
                            totalRewards += share;
                        }
                    }

                    // avoid overflow
                    if(j > 0){
                        j--;
                    } else {
                        break;
                    }
                }

                if(markClaim){
                    userStakedInfo.claimedRound = tailRound;
                }
            }
        }

        return totalRewards;
    }

    // Claim rewards
    function claimRewards() public onlyStarted override{
        // close round
        closeRound();

        uint unClaimedRewards = getAndMarkUnClaimedRewards(msg.sender, true);
        rewardContract.transfer(msg.sender, unClaimedRewards);

        emit ClaimReward(msg.sender,  unClaimedRewards);
    }

    // Get staked info
    function getDepositInfo() public view override returns(DepositInfo[] memory){
        return depositInfos;
    }

    // Get deposit info by page
    function getDepositInfoByPage(uint pageSt, uint pageSz) public view override
     returns(DepositInfo[] memory ){
        DepositInfo[] memory retDepositInfos;
        if(pageSt < depositInfos.length){
```

```
            uint end = pageSt + pageSz;
            end = end > depositInfos.length?depositInfos.length : end;
            retDepositInfos = new DepositInfo[](end - pageSt);
            for(uint i = 0;pageSt < end; i++){
                retDepositInfos[i] = depositInfos[pageSt];
                pageSt++;
            }
        }
        return retDepositInfos;
    }


    // Get user staked info
    function getUserStakedInfo(address user) public view override
    returns(UserStakedInfo[] memory userStaked){
        return stakedInfos[user];
    }

    function getUserStakedInfoByPage(address user,uint pageSt, uint pageSz)
     public view override returns(UserStakedInfo[] memory userStaked){
        UserStakedInfo[] memory retUserStakedInfos;
        UserStakedInfo[] storage userStakedInfos = stakedInfos[user];

        if(pageSt < userStakedInfos.length){
            uint end = pageSt + pageSz;
            end = end > userStakedInfos.length?userStakedInfos.length : end;
            retUserStakedInfos = new UserStakedInfo[](end - pageSt);
            for(uint i = 0;pageSt < end; i++){
                retUserStakedInfos[i] = userStakedInfos[pageSt];
                pageSt++;
            }
        }

        return retUserStakedInfos;
     }

}
```


Analysis of audit results

Re-Entrancy

- **Description:**
  One of the features of smart contracts is the ability to call and utilise code of other external contracts. Contracts also typically handle Blockchain Currency, and as such often send Blockchain Currency to various external user addresses. The operation of calling external contracts, or sending Blockchain Currency to an address, requires the contract to submit an external call. These external calls can be hijacked by attackers whereby they force the contract to execute further code (i.e. through a fallback function) , including calls back into itself. Thus the code execution "re-enters" the contract. Attacks of this kind were used in the infamous DAO hack.
- **Detection results:**

  ```
  PASSED!
  ```

- **Security suggestion:**
  no.

Arithmetic Over/Under Flows

- **Description:**
  The Virtual Machine (EVM) specifies fixed-size data types for integers. This means that an integer variable, only has a certain range of numbers it can represent. A uint8 for example, can only store numbers in the range [0,255]. Trying to store 256 into a uint8 will result in 0. If care is not taken, variables in Solidity can be exploited if user input is unchecked and calculations are performed which result in numbers that lie outside the range of the data type that stores them.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unexpected Blockchain Currency

- **Description:**
  Typically when Blockchain Currency is sent to a contract, it must execute either the fallback function, or another function described in the contract. There are two exceptions to this, where Blockchain Currency can exist in a contract without having executed any code. Contracts which rely on code execution for every Blockchain Currency sent to the contract can be vulnerable to attacks where Blockchain Currency is forcibly sent to a contract.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

Delegatecall

- **Description:**
  The CALL and DELEGATECALL opcodes are useful in allowing developers to modularise their code. Standard external message calls to contracts are handled by the CALL opcode whereby code is run in the context of the external contract/function. The DELEGATECALL opcode is identical to the standard message call, except that the code executed at the targeted address is run in the context of the calling contract along with the fact that msg.sender and msg.value remain unchanged. This feature enables the implementation of libraries whereby developers can create reusable code for future contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

Default Visibilities

- **Description:**
  Functions in Solidity have visibility specifiers which dictate how functions are allowed to be called. The visibility determines whBlockchain Currency a function can be called externally by users, by other derived contracts, only internally or only externally. There are four visibility specifiers, which are described in detail in the Solidity Docs. Functions default to public allowing users to call them externally. Incorrect use of visibility specifiers can lead to some devestating vulernabilities in smart contracts as will be discussed in this section.

- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Entropy Illusion

- **Description:**
  All transactions on the blockchain are deterministic state transition operations. Meaning that every transaction modifies the global state of the ecosystem and it does so in a calculable way with no uncertainty. This ultimately means that inside the blockchain ecosystem there is no source of entropy or randomness. There is no rand() function in Solidity. Achieving decentralised entropy (randomness) is a well established problem and many ideas have been proposed to address this (see for example, RandDAO or using a chain of Hashes as described by Vitalik in this post).
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

External Contract Referencing

- **Description:**
  One of the benefits of the global computer is the ability to re-use code and interact with contracts already deployed on the network. As a result, a large number of contracts reference external contracts and in general operation use external message calls to interact with these contracts. These external message calls can mask malicious actors intentions in some non-obvious ways, which we will discuss.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unsolved TODO comments

- **Description:**
  Check for Unsolved TODO comments
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Short Address/Parameter Attack

- **Description:**
  This attack is not specifically performed on Solidity contracts themselves but on third party applications that may interact with them. I add this attack for completeness and to be aware of how parameters can be manipulated in contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unchecked CALL Return Values

- **Description:**
  There a number of ways of performing external calls in solidity. Sending Blockchain Currency to external accounts is commonly performed via the transfer() method. However, the send() function can also be used and, for more versatile external calls, the CALL opcode can be directly employed in solidity. The call() and send() functions return a boolean indicating if the call succeeded or failed. Thus these functions have a simple caveat, in that the transaction that executes these functions will not revert if the external call (intialised by call() or send()) fails, rather the call() or send() will simply return false. A common pitfall arises when the return value is not checked, rather the developer expects a revert to occur.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Race Conditions / Front Running

- **Description:**
  The combination of external calls to other contracts and the multi-user nature of the underlying blockchain gives rise to a variety of potential Solidity pitfalls whereby users race code execution to obtain unexpected states. Re-Entrancy is one example of such a race condition. In this section we will talk more generally about different kinds of race conditions that can occur on the blockchain. There is a variety of good posts on this subject, a few are: Wiki - Safety, DASP - Front-Running and the Consensus - Smart Contract Best Practices.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Denial Of Service (DOS)

- **Description:**
  This category is very broad, but fundamentally consists of attacks where users can leave the contract inoperable for a small period of time, or in some cases, permanently. This can trap Blockchain Currency in these contracts forever, as was the case with the Second Parity MultiSig hack
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Block Timestamp Manipulation

- **Description:**
  Block timestamps have historically been used for a variety of applications, such as entropy for random numbers (see the Entropy Illusion section for further details), locking funds for periods of time and various state-changing conditional statements that are time-dependent. Miner's have the ability to adjust timestamps slightly which can prove to be quite dangerous if block timestamps are used incorrectly in smart contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Constructors with Care

- **Description:**
  Constructors are special functions which often perform critical, privileged tasks when initialising contracts. Before solidity v0.4.22 constructors were defined as functions that had the same name as the contract that contained them. Thus, when a contract name gets changed in development, if the constructor name isn't changed, it becomes a normal, callable function. As you can imagine, this can (and has) lead to some interesting contract hacks.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unintialised Storage Pointers

- **Description:**
  The EVM stores data either as storage or as memory. Understanding exactly how this is done and the default types for local variables of functions is highly recommended when developing contracts. This is because it is possible to produce vulnerable contracts by inappropriately intialising variables.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Floating Points and Numerical Precision

- **Description:**
  As of this writing (Solidity v0.4.24), fixed point or floating point numbers are not supported. This means that floating point representations must be made with the integer types in Solidity. This can lead to errors/vulnerabilities if not implemented correctly.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

tx.origin Authentication

- **Description:**
  Solidity has a global variable, tx.origin which traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in smart contracts leaves the contract vulnerable to a phishing-like attack.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Permission restrictions

- **Description:**
  Contract managers who can control liquidity or pledge pools, etc., or impose unreasonable restrictions on other users.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

armors.io

contact@armors.io