CyberCrime Shield
cybercrimeshield.org

Smart Contract Audit Report

BlockRewards

blockrewards.pro

AUDIT TYPE: **PUBLIC**

YOU CAN CHECK THE VALIDITY USING THE QR CODE OR LINK:

https://cybercrimeshield.org/secure/blockrewards

Report ID: A0125989

July 28, 2022

TABLE OF CONTENTS

DESCRIPTION

Up to 40% New Buyer Bonus

Up to 20% Referral Rewards (paid directly in BNB)

Variable Daily Return. Starts at 3%

Variable JumpStartTVL Sell Fee. Starts at 2%

6% Dev & Marketing

24 Hours Rewards Accumulation CutOff

SMART CONTRACT

https://cybercrimeshield.org/secure/uploads/BlockRewards.sol

Compiler Version: v0.8.9+commit.e5eed63a

License: MIT

CRC32: 5A8CEAB9

MD5: E1FEF5CDC2B3FB7DF26540EA61909A3A

SHA-1: CA587443AD780A7F85FFFE2B530C5B2C46B8187C

INTRODUCTION

Blockchain platforms, such as Nakamoto's Bitcoin, enable the trade of crypto currencies between mutually mistrusting parties.

To eliminate the need for trust, Nakamoto designed a peer-to-peer network that enables its peers to agree on the trading transactions.

Smart contracts have shown to be applicable in many domains including financial industry, public sector and cross-industry.

The increased adoption of smart contracts demands strong security guarantees. Unfortunately, it is challenging to create smart contracts that are free of security bugs.

Consequently, critical vulnerabilities in smart contracts are discovered and exploited every few months.

In turn, these exploits have led to losses reaching billions worth of USD in the past few years.

It is apparent that effective security checks for smart contracts are strictly needed.

Our company provides comprehensive, independent smart contract auditing.

We help stakeholders confirm the quality and security of their smart contracts using our standardized audit process.

The scope of this audit was to analyze and document the Block Rewards contract.

This document is not financial advice, you perform all financial actions on your own responsibility.

AUDIT METHODOLOGY

1. Design Patterns

We inspect the structure of the smart contract, including both manual and automated analysis.

2. Static Analysis

The static analysis is performed using a series of automated tools, purposefully designed to test the security of the contract.
All the issues found by tools were manually checked (rejected or confirmed).

3. Manual Analysis

Contract reviewing to identify common vulnerabilities. Comparing of requirements and implementation. Reviewing of a smart contract for compliance with specified customer requirements. Checking for energy optimization and self-documentation. Running tests of the properties of the smart contract in test net.

ISSUES DISCOVERED

Issues are listed from most critical to least critical. Severity is determined by an assessment of the risk of exploitation or otherwise unsafe behavior.

Severity Levels

Critical - Funds may be allocated incorrectly, lost or otherwise result in a significant loss.
Medium - Affects the ability of the contract to operate.
Low - Minimal impact on operational ability.
Informational - No impact on the contract.

AUDIT SUMMARY

The summary result of the audit performed is presented in the table below

Findings list:

LEVEL	AMOUNT
Critical	0
Medium	0
Low	0
Informational	0

CONCLUSION

- The contract has a clear structure and is easy to read
- Gas usage is optimal
- Contract is fully Ethereum & BSC completable
- No backdoors or overflows are present in the contract

Safety guaranteed by CyberCrimeShield Security Company

SOURCE CODE

```
/**
 *Submitted for verification at BscScan.com on 2022-07-26
*/

// SPDX-License-Identifier: MIT

library SafeMath {
    /**
     * @dev Returns the addition of two unsigned integers, with an overflow flag.
     *
     * _Available since v3.4._
     */
    function tryAdd(uint256 a, uint256 b) internal pure returns (bool, uint256) {
        unchecked {
            uint256 c = a + b;
            if (c < a) return (false, 0);
            return (true, c);
        }
    }

    /**
     * @dev Returns the substraction of two unsigned integers, with an overflow flag.
     *
     * _Available since v3.4._
     */
    function trySub(uint256 a, uint256 b) internal pure returns (bool, uint256) {
        unchecked {
            if (b > a) return (false, 0);
            return (true, a - b);
        }
    }

    /**
     * @dev Returns the multiplication of two unsigned integers, with an overflow flag.
     *
     * _Available since v3.4._
     */
    function tryMul(uint256 a, uint256 b) internal pure returns (bool, uint256) {
        unchecked {
            // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
            // benefit is lost if 'b' is also tested.
            // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
            if (a == 0) return (true, 0);
            uint256 c = a * b;
            if (c / a != b) return (false, 0);
            return (true, c);
        }
    }

    /**
     * @dev Returns the division of two unsigned integers, with a division by zero flag.
     *
     * _Available since v3.4._
     */
    function tryDiv(uint256 a, uint256 b) internal pure returns (bool, uint256) {
        unchecked {
            if (b == 0) return (false, 0);
            return (true, a / b);
        }
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers, with a division by zero flag.
     *
     * _Available since v3.4._
     */
    function tryMod(uint256 a, uint256 b) internal pure returns (bool, uint256) {
        unchecked {
            if (b == 0) return (false, 0);
            return (true, a % b);
        }
    }

    /**
     * @dev Returns the addition of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `+` operator.
     *
     * Requirements:
     *
     * - Addition cannot overflow.
     */
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        return a + b;
    }

    /**
```

```
     * @dev Returns the subtraction of two unsigned integers, reverting on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     *
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return a - b;
    }

    /**
     * @dev Returns the multiplication of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `*` operator.
     *
     * Requirements:
     *
     * - Multiplication cannot overflow.
     */
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        return a * b;
    }

    /**
     * @dev Returns the integer division of two unsigned integers, reverting on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator.
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return a / b;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * reverting when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        return a % b;
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
     * overflow (when the result is negative).
     *
     * CAUTION: This function is deprecated because it requires allocating memory for the error
     * message unnecessarily. For custom revert reasons use {trySub}.
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     *
     * - Subtraction cannot overflow.
     */
    function sub(
        uint256 a,
        uint256 b,
        string memory errorMessage
    ) internal pure returns (uint256) {
        unchecked {
            require(b <= a, errorMessage);
            return a - b;
        }
    }

    /**
     * @dev Returns the integer division of two unsigned integers, reverting with custom message on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function div(
        uint256 a,
        uint256 b,
        string memory errorMessage
    ) internal pure returns (uint256) {
        unchecked {
            require(b > 0, errorMessage);
            return a / b;
        }
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * reverting with custom message when dividing by zero.
     *
     * CAUTION: This function is deprecated because it requires allocating memory for the error
     * message unnecessarily. For custom revert reasons use {tryMod}.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function mod(
        uint256 a,
        uint256 b,
        string memory errorMessage
    ) internal pure returns (uint256) {
        unchecked {
            require(b > 0, errorMessage);
            return a % b;
        }
    }
```

```
}

pragma solidity 0.8.9;

/**
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 */
abstract contract Context {
    function _msgSender() internal view virtual returns (address) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes calldata) {
        return msg.data;
    }
}

contract Ownable is Context {
    address private _owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed
newOwner);

    /**
    * @dev Initializes the contract setting the deployer as the initial owner.
    */
    constructor () {
      address msgSender = _msgSender();
      _owner = msgSender;
      emit OwnershipTransferred(address(0), msgSender);
    }

    /**
    * @dev Returns the address of the current owner.
    */
    function owner() public view returns (address) {
      return _owner;
    }

    modifier onlyOwner() {
      require(_owner == _msgSender(), "Ownable: caller is not the owner");
      _;
    }

    function transferOwnership(address newOwner) public onlyOwner {
      _transferOwnership(newOwner);
    }

    function _transferOwnership(address newOwner) internal {
      require(newOwner != address(0), "Ownable: new owner is the zero address");
      emit OwnershipTransferred(_owner, newOwner);
      _owner = newOwner;
    }
}

abstract contract ReentrancyGuard {
    uint256 private constant _NOT_ENTERED = 1;
    uint256 private constant _ENTERED = 2;

    uint256 internal _status;

    constructor() {
        _status = _NOT_ENTERED;
    }

    modifier nonReentrant() {
        _nonReentrantBefore();
        _;
        _nonReentrantAfter();
    }

    function _nonReentrantBefore() private {
        require(_status != _ENTERED, "ReentrancyGuard: reentrant call");
        _status = _ENTERED;
    }

    function _nonReentrantAfter() private {
        _status = _NOT_ENTERED;
    }
}

```

```
contract BlockRewards is Context, Ownable, ReentrancyGuard {
    using SafeMath for uint256;

    uint256 private PSN = 10000;
    uint256 private PSNH = 5000;
    uint256 public MAX_REWARDS_ACCUMULATION_CUTOFF = 86400; //24*60*60. Seconds in 1 day. Rewards will accumulate till max 24 hours until after the user will have to either compound or sell
    uint256 public EGGS_TO_HATCH_1MINERS = 2880000; // 86400/2880000 = 3% APY
    uint256 public DEV_PERCENT = 6;
    uint256 public JumpStartTVL_PERCENT = 2;
    uint256 public REFERRAL_PERCENT = 7;

    bool private initialized = false;
    bool public sellCheck = true;
    address payable private recAdd;
    address payable private JumpStartTVLFund; //separate wallet. Only to be used for jumpstarting TVL

    bool public referralWhitelistActive = false;
    mapping(address => bool) public referralWhitelisted;
    uint256 public referralsNeededToEarnCommission = 3;

    bool public blacklistActive = true;
    mapping(address => bool) public Blacklisted;

    uint256 public ADJUSTED_REWARD_PERCENT_NEWBUYERS = 100;     //100 = No buying bonus. 125 = 25% buying bonus
    uint256 public ADJUSTED_REWARD_PERCENT = 100;
    uint256 public CAPPED_DAILY_REWARD_AMOUNT = 5000 * 1e16;    // 50 BNB/Day == 50 * 1e18 ==  5000 * 1e16 . Disabled By Default by setting a high amount- 50 BNB/day. Please see Docs for more details.

    uint256 public marketEggs;
    uint256 public MARKETEGGS_BUY_INFLATION = 0;                // By Default, no inflation on buy
    uint256 public MARKETEGGS_HATCH_INFLATION = 20;
    uint256 public MARKETEGGS_SELL_INFLATION = 100;

    uint256 public totalStaked;     // Total BNB bought
    uint256 public totalDeposits;   // Total Deposits
    uint256 public totalCompound;   // Total BNB compounded. Not used
    uint256 public totalRefBonus;   // Total BNB paid out for Referrals
    uint256 public totalWithdrawn;  // Total BNB Withdrawn

```

```solidity
    struct User {
            uint256 initialDeposit;      // Initially Deposit
            uint256 userDeposit;         // Total Compounded Deposit including Initial Deposit
            uint256 miners;              // Your miners
            uint256 lastHatch;           // last time you bought or sold or compounded. Seconds passed since last epoch
            address referrer;            // who referred me
            uint256 referralsCount;      // how many people i have referred
            uint256 refRewardsinBNB;     // Total BNB paid to me as Referrals
            uint256 totalWithdrawn;      // TotalWithdrawn
            uint256 farmerCompoundCount; // Added to monitor farmer consecutive compound without cap. Total amount of times i ever compounded.
            uint256 lastWithdrawTime;
    }
    mapping(address => User) public users;

    constructor() {
        recAdd = payable(msg.sender);
        JumpStartTVLFund = payable(address(0x02B6084E1d568d2cF61bb1B1Cf4B3dCb5851492C));
    }

    function hatchEggs() public {
        require(initialized);
        User storage user = users[msg.sender];

        uint256 eggsByExistingMiners = getMyEggs(msg.sender);
        uint256 myEggs = SafeMath.div(SafeMath.mul(eggsByExistingMiners, ADJUSTED_REWARD_PERCENT),100);

        user.lastHatch = block.timestamp;

        uint256 newMiners = SafeMath.div(myEggs, EGGS_TO_HATCH_1MINERS);
        user.miners = SafeMath.add(user.miners, newMiners);

        uint256 eggValueInBNB = calculateTrade(myEggs, marketEggs,address(this).balance);
        user.userDeposit = SafeMath.add(user.userDeposit, eggValueInBNB);
        totalCompound = SafeMath.add(totalCompound, eggValueInBNB);
        user.farmerCompoundCount = SafeMath.add(user.farmerCompoundCount, 1);

        uint256 inflation = SafeMath.div(SafeMath.mul(myEggs, MARKETEGGS_HATCH_INFLATION),100);
        marketEggs=SafeMath.add(marketEggs, inflation);
    }

    function buyEggs(address ref) public payable nonReentrant{
        require(initialized);
        User storage user = users[msg.sender];

        uint256 eggsBought = calculateEggBuy(msg.value,SafeMath.sub(address(this).balance,msg.value));

        user.initialDeposit = SafeMath.add(user.initialDeposit, msg.value);
        user.userDeposit = SafeMath.add(user.userDeposit, msg.value);

        if (user.referrer == address(0)) {
            if (ref != msg.sender) {
                user.referrer = ref;  // set who referred me
            }

            address referrer = user.referrer;
            if (referrer != address(0)) {
                users[referrer].referralsCount = users[referrer].referralsCount.add(1);  // increment referral count for whoever referred me
            }
        }

        if (user.referrer != address(0)) {
            address referrer = user.referrer;
            if (referrer != address(0)) {

                bool payReferral = false;

                if (referralWhitelistActive) { // default disabled. can enable later if needed.
                    if(referralWhitelisted[referrer]) {
                        payReferral = true;
                    }
                } else {
                    payReferral = true;
                }

                if(payReferral && users[referrer].referralsCount > referralsNeededToEarnCommission) {
                    uint256 refRewardsBNB = SafeMath.div(SafeMath.mul(msg.value,REFERRAL_PERCENT),100);
                    payable(address(referrer)).transfer(refRewardsBNB);
```

```

              users[referrer].refRewardsinBNB = users[referrer].refRewardsinBNB.add(refRewardsBNB);
              totalRefBonus = totalRefBonus.add(refRewardsBNB);

              uint256 referralInflation = SafeMath.div(SafeMath.mul(eggsBought, REFERRAL_PERCENT),100);
              marketEggs=SafeMath.add(marketEggs, referralInflation);
            }

          }
        }

        uint256 eggsBoughtWithBonus = applyBuyerBonus(msg.sender, eggsBought, msg.value);

        uint256 bonusEggs = eggsBoughtWithBonus - eggsBought;
        if(bonusEggs > 0) {
          marketEggs=SafeMath.add(marketEggs, bonusEggs);
        }

        eggsBoughtWithBonus = SafeMath.sub(eggsBoughtWithBonus,devFee(eggsBought)); //devFee calculated on eggBought without bonus
        uint256 fee = devFee(msg.value);
        recAdd.transfer(fee);

        totalStaked = SafeMath.add(totalStaked, SafeMath.sub(msg.value, fee));
        totalDeposits = SafeMath.add(totalDeposits, 1);

        uint256 eggsProducedByExistingMiners = getMyEggs(msg.sender);
        if(eggsProducedByExistingMiners > 0){ // for existing Blockholders
          uint256 compoundInflation = SafeMath.div(SafeMath.mul(eggsProducedByExistingMiners, MARKETEGGS_HATCH_INFLATION),100);
          marketEggs=SafeMath.add(marketEggs, compoundInflation);
        }

        uint256 totalEggs = SafeMath.add(eggsBoughtWithBonus, eggsProducedByExistingMiners);

        uint256 newMiners = SafeMath.div(totalEggs,EGGS_TO_HATCH_1MINERS);
        user.miners = SafeMath.add(user.miners, newMiners);

        user.lastHatch = block.timestamp;
```

```solidity

        if(MARKETEGGS_BUY_INFLATION > 0) { // by default. No Buy inflation.
            uint256 inflation = SafeMath.div(SafeMath.mul(totalEggs, MARKETEGGS_BUY_INFLATION),100);
            marketEggs=SafeMath.add(marketEggs, inflation);
        }
    }

    function applyBuyerBonus(address adr, uint256 eggsPurchased, uint256 buyAmount) private view returns(uint256) {
        uint256 eggsWithBonus = eggsPurchased;
        if(ADJUSTED_REWARD_PERCENT_NEWBUYERS > 100) {  //  if applicable, apply Bonus to both New Buyers and Existing Buyers
            if(users[adr].miners == 0) { // for new buyers
                eggsWithBonus =  SafeMath.div(SafeMath.mul(eggsPurchased, ADJUSTED_REWARD_PERCENT_NEWBUYERS),100); // Give Full Bonus to New Buyers
            } else {
                //Logic for existing buyers. Only give bonus on amount above their daily estimated reward so they don't cash out reward and buyback in for the bonus available.
                uint256 estimatedDailyReward = getEstimatedDailyReward(adr);

                if(buyAmount > estimatedDailyReward) {  // existing buyer is buying more than their daily estimated reward
                    uint256 amountEligibleForBonus = SafeMath.sub(buyAmount, estimatedDailyReward);
                    uint256 eggsEligibleForBonus =  SafeMath.div(SafeMath.mul(amountEligibleForBonus, eggsPurchased), buyAmount);
                    uint256 bonusEggs = SafeMath.div(SafeMath.mul(SafeMath.sub(ADJUSTED_REWARD_PERCENT_NEWBUYERS, 100), eggsEligibleForBonus), 100);
                    eggsWithBonus = SafeMath.add(eggsPurchased, bonusEggs);
                }
            }
        }
        return eggsWithBonus;
    }

    function getEstimatedDailyReward(address adr) public view returns(uint256){
        uint256 myEggsInOneDay = SafeMath.mul(MAX_REWARDS_ACCUMULATION_CUTOFF, users[adr].miners);
        myEggsInOneDay = SafeMath.div(SafeMath.mul(myEggsInOneDay, ADJUSTED_REWARD_PERCENT),100);
        uint256 estimatedDailyReward = calculateEggSell(myEggsInOneDay);  //1 day estimated reward in BNB
        return estimatedDailyReward;
    }

    function sellEggs() public nonReentrant{
        require(initialized);

        User storage user = users[msg.sender];

        if(sellCheck){
            require(block.timestamp.sub(user.lastHatch) >= MAX_REWARDS_ACCUMULATION_CUTOFF, "Please wait for 24 hours before trying to sell.");
        }

        if (blacklistActive) {
            require(!Blacklisted[msg.sender], "Address is blacklisted.");
        }

        uint256 eggsByExistingMiners = getMyEggs(msg.sender);

        uint256 myEggs = SafeMath.div(SafeMath.mul(eggsByExistingMiners, ADJUSTED_REWARD_PERCENT),100); // adjust reward if needed

        uint256 eggValue = calculateEggSell(myEggs);   // value in BNB

        if (eggValue == CAPPED_DAILY_REWARD_AMOUNT) {  // exceeds daily withdrawal limit
                // Adjusting myEggs for setting it to CAPPED_DAILY_REWARD_AMOUNT. Adjusted amount needs to be added to marketeggs
                uint256 myEggsAdjusted= SafeMath.div(SafeMath.mul(eggValue,SafeMath.mul(PSN,marketEggs)), SafeMath.sub(SafeMath.mul(PSN,address(this).balance),SafeMath.mul(2,SafeMath.mul(PSNH,eggValue))));
                if(myEggsAdjusted <= myEggs){
                    myEggs = myEggsAdjusted;
                }
        }

        uint256 fee = devFee(eggValue);
        recAdd.transfer(fee);

        uint256 netEggValue= SafeMath.sub(eggValue, fee);

        uint256 fee2 = JumpStartTVLFee(netEggValue);
        JumpStartTVLFund.transfer(fee2);

        netEggValue= SafeMath.sub(netEggValue, fee2);

        if(getBalance() < netEggValue) {
            netEggValue = getBalance();
        }

        user.lastWithdrawTime = block.timestamp;
        user.lastHatch = block.timestamp;

        payable(address(msg.sender)).transfer(netEggValue);

        user.totalWithdrawn = SafeMath.add(user.totalWithdrawn, netEggValue);
        totalWithdrawn =  SafeMath.add(totalWithdrawn, netEggValue);

        uint256 inflation = SafeMath.div(SafeMath.mul(myEggs, MARKETEGGS_SELL_INFLATION),100);
        marketEggs=SafeMath.add(marketEggs, inflation);
    }

    function userRewards(address adr) public view returns(uint256) {
        uint256 myEggs = getMyEggs(adr);     // Eggs Produced by ExistingMiners
        myEggs = SafeMath.div(SafeMath.mul(myEggs, ADJUSTED_REWARD_PERCENT),100);

        uint256 eggValue = calculateEggSell(myEggs);
        return eggValue;
    }

    function calculateTrade(uint256 rt,uint256 rs, uint256 bs) private view returns(uint256) {
        return SafeMath.div(SafeMath.mul(PSN,bs),SafeMath.add(PSNH,SafeMath.div(SafeMath.add(SafeMath.mul(PSN,rs),SafeMath.mul(PSNH,rt)),rt)));
    }

    function calculateEggSell(uint256 eggs) public view returns(uint256) {
        uint256 eggValue = calculateTrade(eggs, marketEggs, address(this).balance);   // value in BNB

        if (eggValue > CAPPED_DAILY_REWARD_AMOUNT) {   // exceeds daily withdrawal limit
            eggValue = CAPPED_DAILY_REWARD_AMOUNT;
        }

         return eggValue;
    }

    function calculateEggBuy(uint256 eth,uint256 contractBalance) public view returns(uint256) {
        return calculateTrade(eth, contractBalance, marketEggs);    // value in Eggs
    }

    function calculateMiners(uint256 eth) public view returns(uint256){
        uint256 eggsBought = calculateEggBuy(eth, getBalance());
        uint256 miners = eggsBought.div(EGGS_TO_HATCH_1MINERS);
        return miners;
    }

    function calculateMinersWithBonus(uint256 eth, address adr) public view returns(uint256){
        uint256 eggsBoughtWithoutBonus = calculateEggBuy(eth, getBalance());
        uint256 eggsBoughtWithBonus =applyBuyerBonus(adr, eggsBoughtWithoutBonus, eth);
        uint256 miners = eggsBoughtWithBonus.div(EGGS_TO_HATCH_1MINERS);
        return miners;
    }

    function devFee(uint256 amount) private view returns(uint256) {
        return SafeMath.div(SafeMath.mul(amount, DEV_PERCENT),100);
    }

    function JumpStartTVLFee(uint256 amount) private view returns(uint256) {
        return SafeMath.div(SafeMath.mul(amount, JumpStartTVL_PERCENT),100);
    }

    function seedMarket() public payable onlyOwner {
        require(marketEggs == 0);
        initialized = true;
        marketEggs = 108000000000;
    }

    //fund contract with BNB before launch.
    function fundContract() external payable {}

    function getUserInfo(address adr) public view returns(uint256 _initialDeposit, uint256 _userDeposit, uint256 _miners,
     uint256 _lastHatch, address _referrer, uint256 _referralsCount, uint256 _totalWithdrawn, uint256 _refRewardsinBNB,
     uint256 _farmerCompoundCount, uint256 _lastWithdrawTime) {
         _initialDeposit = users[adr].initialDeposit;
         _userDeposit = users[adr].userDeposit;
         _miners = users[adr].miners;
         _lastHatch = users[adr].lastHatch;
         _referrer = users[adr].referrer;
         _referralsCount = users[adr].referralsCount;
         _totalWithdrawn = users[adr].totalWithdrawn;
         _refRewardsinBNB = users[adr].refRewardsinBNB;
         _farmerCompoundCount = users[adr].farmerCompoundCount;  // total # of times person has ever compounded
         _lastWithdrawTime = users[adr].lastWithdrawTime;
    }

    function getBalance() public view returns(uint256) {
        return address(this).balance;
    }

    function getTimeStamp() public view returns (uint256) {
        return block.timestamp;
    }

    function getSiteInfo() public view returns (uint256 _totalStaked, uint256 _totalDeposits, uint256 _totalCompound, uint256 _totalRefBonus) {
        return (totalStaked, totalDeposits, totalCompound, totalRefBonus);
    }

    function getMyMiners(address adr) public view returns(uint256) {
        return users[adr].miners;
    }

    function getMyEggs(address adr) public view returns(uint256) {
        uint256 secondsPassed= min(MAX_REWARDS_ACCUMULATION_CUTOFF, SafeMath.sub(block.timestamp, users[adr].lastHatch));  // time passed since last buy/sell/compound
        return SafeMath.mul(secondsPassed, users[adr].miners);
    }

    function min(uint256 a, uint256 b) private pure returns (uint256) {
        return a < b ? a : b;
    }

    /** Admin functions */

    function changeRecAdd(address payable value) external onlyOwner{
        recAdd = value;
    }

    function changeJumpStartTVLFund(address payable value) external onlyOwner{
        JumpStartTVLFund = value;
    }

    function Set_RewardRate_NewBuyers(uint256 value) external onlyOwner {
        // 100 => No Bonus. 125 => 25% New Buyer Bonus oR 1.25x more Eggs
        require(value >= 100 && value <= 500);
        ADJUSTED_REWARD_PERCENT_NEWBUYERS = value;
    }

    function Set_RewardRate_ExistingBuyers(uint256 value) external onlyOwner {
        // Base: 3%. if ADJUSTED_REWARD_PERCENT = 50 => 3% * 50/100 = 1.5% Net Apy. if ADJUSTED_REWARD_PERCENT = 100 =>  3% * 100/100 = 3% Net Apy
        require(value >= 5 && value <= 50000);
        ADJUSTED_REWARD_PERCENT = value;
    }

    function Set_Capped_DailyRewardAmount(uint256 capped_daily_reward_amount) external onlyOwner {
        // 50 BNB/Day == 50 * 1e18  ==  5000 * 1e16.
        require(capped_daily_reward_amount >= 0);
        CAPPED_DAILY_REWARD_AMOUNT = capped_daily_reward_amount * 1e16;
    }

    function Set_Referral_Percent(uint256 value) external onlyOwner {
        require(value >= 0 && value <= 100);
        REFERRAL_PERCENT = value;
    }

    function Set_Dev_Percent(uint256 value) external onlyOwner {
        require(value >= 0 && value <= 15);
        DEV_PERCENT = value;
    }

    function Set_JumpStartTVL_PERCENT(uint256 value) external onlyOwner {
        require(value >= 0);
        JumpStartTVL_PERCENT = value;
    }

    function SetSellCheckActive(bool isSellCheckActive) external onlyOwner{
        sellCheck = isSellCheckActive;
    }

    function Set_MarketEggs_Buy_Inflation(uint256 value) external onlyOwner {
        require(value >= 0);
        MARKETEGGS_BUY_INFLATION = value;
    }

    function Set_MarketEggs_Hatch_Inflation(uint256 value) external onlyOwner {
        require(value >= 0);
        MARKETEGGS_HATCH_INFLATION = value;
    }

    function Set_MarketEggs_Sell_Inflation(uint256 value) external onlyOwner {
        require(value >= 0);
        MARKETEGGS_SELL_INFLATION = value;
    }

    function SetBlacklistActive(bool isActive) external onlyOwner{
        blacklistActive = isActive;
    }

    function blackListWallet(address Wallet, bool isBlacklisted) external
onlyOwner{
        Blacklisted[Wallet] = isBlacklisted;
    }

    function blackMultipleWallets(address[] calldata Wallet, bool
isBlacklisted) external onlyOwner{
        for(uint256 i = 0; i < Wallet.length; i++) {
            Blacklisted[Wallet[i]] = isBlacklisted;
        }
    }

    function checkIfBlacklisted(address Wallet) external onlyOwner view
returns(bool blacklisted){
        blacklisted = Blacklisted[Wallet];
    }

    function setReferralWhitelistActive(bool isActive) external onlyOwner{
        referralWhitelistActive = isActive;
    }

    function whitelistReferralWallet(address Wallet, bool isWhitelisted)
external onlyOwner{
```

```
            referralWhitelisted[Wallet] = isWhitelisted;
        }

        function whitelistMultipleReferralWallets(address[] calldata Wallet, bool
    isWhitelisted) external onlyOwner{
            for(uint256 i = 0; i < Wallet.length; i++) {
                referralWhitelisted[Wallet[i]] = isWhitelisted;
            }
        }

        function checkIfReferralWhitelisted(address Wallet) external onlyOwner view
    returns(bool whitelisted){
            whitelisted = referralWhitelisted[Wallet];
        }

        function Set_ReferralsNeededToEarnCommission(uint256 value) external
    onlyOwner {
            require(value >= 0);
            referralsNeededToEarnCommission = value;
        }

        function Override_Reentrancy_Status() external onlyOwner {
            // Set to 1 (_NOT_ENTERED) to override if ever need to
            _status = 1;
        }
    }
```