Armors Labs

Mart Audit Report

Smart Contract Audit

Mart Audit Report Audit Summary

Project name : Mart Audit Report Contract

Project address: None

Code URL : https://hecoinfo.com/address/0xF6c823d5BdbD97F245aE045cD711892E05348690#code, https://hecoinfo.com/address/0xc62caB4B1283c89Eab89Fcb2D38a64Aa3543cC6e#code

Commit : None

Project target : Mart Audit Report Contract Audit

Blockchain : Huobi ECO Chain （HECO）

Test result : PASSED

Audit Info

Audit NO : 0X202110120026

Audit Team : Armors Labs

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

Mart Audit Report Audit

The Mart Audit Report team asked us to review and audit their Mart Audit Report 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
Mart Audit Report Audit	Rock, Sophia, Rushairer, Rico, David, Alice	1.0.0	2021-10-12

Audit results

Warning:

```
There are no restrictions on the transaction fee settings, which can be modified by the owner at
any time
```

Notice:

```
1. Pledge logic code has no boundary limit judgment on user input variables, which may cause
   unpredictable data errors. The marttrademining contract sets global transaction limits
   for redemption operations.

2. Note that as of the date of publishing, the above review reflects the current understanding
```

```
of known security patterns as they relate to the Mart Audit Report 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
MarToken.sol	23891417e9aa3e050b88f11e625a50ea
MartTradeMining.sol	816d93e7c6736bee81ce5a1fe6c24d82

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

Vulnerability	status
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

MarToken.sol

```
/**
 *Submitted for verification at hecoinfo.com on 2021-10-11
*/

pragma solidity 0.8.0;
//SPDX-License-Identifier: SimPL-2.0
abstract contract Context {
    function _msgSender() internal view virtual returns (address) {
        return msg.sender;
    }

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

/**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * By default, the owner account will be the one that deploys the contract. This
 * can later be changed with {transferOwnership}.
 *
 * This module is used through inheritance. It will make available the modifier
 * `onlyOwner`, which can be applied to your functions to restrict their use to
 * the owner.
 */
```

```
abstract 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 virtual returns (address) {
        return _owner;
    }

    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(owner() == _msgSender(), "Ownable: caller is not the owner");
        _;
    }

    /**
     * @dev Leaves the contract without owner. It will not be possible to call
     * `onlyOwner` functions anymore. Can only be called by the current owner.
     *
     * NOTE: Renouncing ownership will leave the contract without an owner,
     * thereby removing any functionality that is only available to the owner.
     */
    function renounceOwnership() public virtual onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}
library Address {
    /**
     * @dev Returns true if `account` is a contract.
     *
     * [IMPORTANT]
     * ====
     * It is unsafe to assume that an address for which this function returns
     * false is an externally-owned account (EOA) and not a contract.
     *
     * Among others, `isContract` will return false for the following
     * types of addresses:
     *
     *  - an externally-owned account
     *  - a contract in construction
     *  - an address where a contract will be created
     *  - an address where a contract lived, but was destroyed
     * ====
     */
    function isContract(address account) internal view returns (bool) {
        // This method relies on extcodesize, which returns 0 for contracts in
        // construction, since the code is only stored at the end of the
        // constructor execution.

        uint256 size;
        assembly {
            size := extcodesize(account)
        }
        return size > 0;
    }

    /**
     * @dev Replacement for Solidity's `transfer`: sends `amount` wei to
     * `recipient`, forwarding all available gas and reverting on errors.
     *
     * https://eips.ethereum.org/EIPS/eip-1884[EIP1884] increases the gas cost
     * of certain opcodes, possibly making contracts go over the 2300 gas limit
     * imposed by `transfer`, making them unable to receive funds via
     * `transfer`. {sendValue} removes this limitation.
     *
     * https://diligence.consensys.net/posts/2019/09/stop-using-soliditys-transfer-now/[Learn more].
     *
     * IMPORTANT: because control is transferred to `recipient`, care must be
     * taken to not create reentrancy vulnerabilities. Consider using
     * {ReentrancyGuard} or the
     * https://solidity.readthedocs.io/en/v0.5.11/security-considerations.html#use-the-checks-effects
     */
    function sendValue(address payable recipient, uint256 amount) internal {
        require(address(this).balance >= amount, "Address: insufficient balance");

        (bool success, ) = recipient.call{value: amount}("");
        require(success, "Address: unable to send value, recipient may have reverted");
    }

    /**
     * @dev Performs a Solidity function call using a low level `call`. A
     * plain `call` is an unsafe replacement for a function call: use this
     * function instead.
     *
     * If `target` reverts with a revert reason, it is bubbled up by this
     * function (like regular Solidity function calls).
     *
     * Returns the raw returned data. To convert to the expected return value,
     * use https://solidity.readthedocs.io/en/latest/units-and-global-variables.html?highlight=abi.de
     *
     * Requirements:
     *
     * -  `target` must be a contract.
     * - calling `target` with `data` must not revert.
     *
     * _Available since v3.1._
     */
    function functionCall(address target, bytes memory data) internal returns (bytes memory) {
        return functionCall(target, data, "Address: low-level call failed");
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`], but with
     * `errorMessage` as a fallback revert reason when `target` reverts.
     *
     * _Available since v3.1._
     */
    function functionCall(
        address target,
        bytes memory data,
        string memory errorMessage
    ) internal returns (bytes memory) {
        return functionCallWithValue(target, data, 0, errorMessage);
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],
     * but also transferring `value` wei to `target`.
     *
     * Requirements:
     *
     * - the calling contract must have an ETH balance of at least `value`.
     * - the called Solidity function must be `payable`.
     *
     * _Available since v3.1._
     */
    function functionCallWithValue(
        address target,
        bytes memory data,
        uint256 value
    ) internal returns (bytes memory) {
        return functionCallWithValue(target, data, value, "Address: low-level call with value failed"
    }

    /**
     * @dev Same as {xref-Address-functionCallWithValue-address-bytes-uint256-}[`functionCallWithValu
     * with `errorMessage` as a fallback revert reason when `target` reverts.
     *
     * _Available since v3.1._
     */
    function functionCallWithValue(
        address target,
        bytes memory data,
        uint256 value,
        string memory errorMessage
    ) internal returns (bytes memory) {
        require(address(this).balance >= value, "Address: insufficient balance for call");
        require(isContract(target), "Address: call to non-contract");

        (bool success, bytes memory returndata) = target.call{value: value}(data);
        return _verifyCallResult(success, returndata, errorMessage);
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],
     * but performing a static call.
     *
     * _Available since v3.3._
     */
    function functionStaticCall(address target, bytes memory data) internal view returns (bytes memor
        return functionStaticCall(target, data, "Address: low-level static call failed");
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-string-}[`functionCall`],
     * but performing a static call.
     *
     * _Available since v3.3._
     */
    function functionStaticCall(
        address target,
        bytes memory data,
        string memory errorMessage
    ) internal view returns (bytes memory) {
        require(isContract(target), "Address: static call to non-contract");

        (bool success, bytes memory returndata) = target.staticcall(data);
        return _verifyCallResult(success, returndata, errorMessage);
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],
     * but performing a delegate call.
     *
     * _Available since v3.4._
     */
    function functionDelegateCall(address target, bytes memory data) internal returns (bytes memory)
        return functionDelegateCall(target, data, "Address: low-level delegate call failed");
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-string-}[`functionCall`],
     * but performing a delegate call.
     *
     * _Available since v3.4._
     */
    function functionDelegateCall(
        address target,
        bytes memory data,
        string memory errorMessage
    ) internal returns (bytes memory) {
        require(isContract(target), "Address: delegate call to non-contract");

        (bool success, bytes memory returndata) = target.delegatecall(data);
        return _verifyCallResult(success, returndata, errorMessage);
    }

    function _verifyCallResult(
        bool success,
        bytes memory returndata,
        string memory errorMessage
    ) private pure returns (bytes memory) {
        if (success) {
            return returndata;
        } else {
            // Look for revert reason and bubble it up if present
            if (returndata.length > 0) {
                // The easiest way to bubble the revert reason is using memory via assembly

                assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata), returndata_size)
                }
            } else {
                revert(errorMessage);
            }
        }
    }
}
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;
        }
    }
}
interface IERC20 {
    /**
     * @dev Returns the amount of tokens in existence.
     */
    function totalSupply() external view returns (uint256);

    /**
     * @dev Returns the amount of tokens owned by `account`.
     */
    function balanceOf(address account) external view returns (uint256);

    /**
     * @dev Moves `amount` tokens from the caller's account to `recipient`.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transfer(address recipient, uint256 amount) external returns (bool);

    /**
     * @dev Returns the remaining number of tokens that `spender` will be
     * allowed to spend on behalf of `owner` through {transferFrom}. This is
     * zero by default.
     *
     * This value changes when {approve} or {transferFrom} are called.
     */
    function allowance(address owner, address spender) external view returns (uint256);

    /**
     * @dev Sets `amount` as the allowance of `spender` over the caller's tokens.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * IMPORTANT: Beware that changing an allowance with this method brings the risk
     * that someone may use both the old and the new allowance by unfortunate
     * transaction ordering. One possible solution to mitigate this race
     * condition is to first reduce the spender's allowance to 0 and set the
     * desired value afterwards:
     * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
     *
     * Emits an {Approval} event.
     */
    function approve(address spender, uint256 amount) external returns (bool);

    /**
     * @dev Moves `amount` tokens from `sender` to `recipient` using the
     * allowance mechanism. `amount` is then deducted from the caller's
     * allowance.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transferFrom(
        address sender,
        address recipient,
        uint256 amount
    ) external returns (bool);

    /**
     * @dev Emitted when `value` tokens are moved from one account (`from`) to
     * another (`to`).
     *
     * Note that `value` may be zero.
     */
    event Transfer(address indexed from, address indexed to, uint256 value);

    /**
     * @dev Emitted when the allowance of a `spender` for an `owner` is set by
     * a call to {approve}. `value` is the new allowance.
     */
    event Approval(address indexed owner, address indexed spender, uint256 value);
}
interface  Mar  {
    function balanceOf(address account) external view  returns (uint256);
    function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);
}
contract MarToken is Context, IERC20, Ownable{

    using SafeMath for uint256;
    using Address for address;

    mapping (address => uint256) private _lockU;
    mapping (address => uint256) private _lockR;

    mapping (address => mapping (address => uint256)) private _allowances;

    mapping (address => bool) private _isNotFromFee;
    mapping (address => bool) private _isNotToFee;
    mapping (address => bool) public _isLocked;
    address[] private _locked;

    uint256 private _supply = 4.5*10**8*10**9;
    uint256 private _investAmount = 1*10**8*10**9;
    uint256 private _teamAmount   = 1*10**8*10**9;
    uint256 private rate    = _supply;
    uint256 public _tTotal;
    uint256 public _rTotal;
    uint256 public _tFeeTotal;

    string private _name      = "MART Token";
    string private _symbol    = "MART";
    uint8  private _decimals = 9;
    uint256 private _taxFee  = 1;
    uint256 private _burnRate = 80;
    uint256 private _previousTaxFee = _taxFee;
    uint256 public _maxTxAmount = 50 * 10**6 * 10**9;

    address public transContract;
    address public airdropAddr;
    address public investAddr ;
    address public teamAddr ;

    uint256 private _lockTotal;
    struct LockUBalance {
        uint256 lockOne;
        uint256 lockTwo;
        uint256 lockThr;
        uint256 lockFou;
    }
    struct LockUser {
        uint256 amount;
        uint256 unlockBlocknum;
        bool    isLock;
        bool    isUsed;
        uint256 posRate;
    }
    struct LockConf {
        uint256 rate;
        uint256 lockBlockNum;
        uint256 posRate;
        bool    isPos;
        bool    isUsed;
    }
    struct InvetTeam{
        uint256 start;
        uint256 total;
        uint256 hadReleased;
        uint256 releaseTime;
        uint256 timeinterval;
        uint256 everyReleaseAmt;
    }
    uint256 public  blockNum;
    uint256 public  startBlock;

    mapping(uint8=>mapping(address=>LockUser)) public lockUserMap;
    mapping (uint8 => LockConf)  public lockConfMap;
    mapping (address => LockUBalance) public LockR;
    mapping (address => uint256) private sumRateLockUser;
    mapping (uint8  => InvetTeam) public invetTeamConf;

    Mar mar;
    uint256 public _lockAllWeight;
    uint256 public _releaseAll;

    uint256 public _releaseLoop;
    uint256 public _releaseInvestLoop;
    uint256 public _releaseTeamLoop;

    uint8[] public  lockList =[1,2,3,4];
    uint256[] private releaseBlock;
    uint256[] private releaseAmount;
    uint256[] private releaseAirAmount;
    bool isinitVest = false;
    bool isinitTeam = false;
    event LockCoin(string,address,uint256,uint8);
    event ReleaseCoin(string,uint256,uint8,address);
    event ReleaseAward(string,uint256,uint8);
    event Reaward(string,address,uint256);
    event PrintLog(string,uint256);
    event BurnLog(string,uint256);
    event ReleaseAirLog(string,address,uint256,uint256);
    event ExchangeLog(string,address,uint256);
    constructor(
        uint256 blocknums,
        uint256 Liquidity,
        uint256 startblock,
        address LiquidAddr,
        uint256[] memory releaselist,
        uint256[] memory amountlist,
        uint256[] memory airAmountList,
        address  OldmartAddress
    ){
        mar = Mar(OldmartAddress);
        blockNum = blocknums;
        startBlock = startblock;
        releaseBlock = releaselist;
        releaseAmount = amountlist;
        releaseAirAmount    = airAmountList;
        _lockR[LiquidAddr]  = Liquidity*rate;
        _rTotal             = Liquidity*rate;
        _tTotal             = Liquidity;
        LockConf memory conf1 = LockConf(10000,blockNum*7,0,false,true);
        lockConfMap[1] = conf1;
        LockConf memory conf2 = LockConf(20000,blockNum*30,0,false,true);
        lockConfMap[2] = conf2;
        LockConf memory conf3 = LockConf(40000,blockNum*90,10,false,true);
        lockConfMap[3] = conf3;
        LockConf memory conf4 = LockConf(80000,blockNum*180,12,false,true);
        lockConfMap[4] = conf4;
        _isNotFromFee[owner()] = true;
        _isNotToFee[owner()]   = true;
        _isNotFromFee[LiquidAddr] = true;
        _isNotToFee[LiquidAddr]   = true;
        _isNotFromFee[address(this)] = true;
        _isNotToFee[address(this)] = true;
        emit Transfer(address(0), LiquidAddr, Liquidity);
    }

    function releaseAirdrop() public  returns(bool){
        require(airdropAddr!=address(0),"mat:address is error");
        (uint256 currloop,) = getAirDaysupply();
        if(currloop <= _releaseLoop) return false;
        uint256 releaseAmt  = 0;
        if(currloop.sub(_releaseLoop)>50) currloop = _releaseLoop.add(50);
        for(uint256 i = _releaseLoop.add(1);i<currloop;i++){
            (,uint256 amounts) = getAirDaysupply(i);
            releaseAmt          = releaseAmt.add(amounts);
            emit ReleaseAirLog("airdrop",airdropAddr,amounts,i);
            emit Transfer(address(0),airdropAddr,amounts);
        }
        _lockR[airdropAddr]    = _lockR[airdropAddr].add(releaseAmt.mul(_getRate()));
        _rTotal                = _rTotal.add(releaseAmt.mul(_getRate()));
        _tTotal                = _tTotal.add(releaseAmt);
        _releaseLoop           = currloop.sub(1);
        return true;
    }

    function releaseInvest() public  returns(bool){
        require(investAddr!=address(0),"mat:address is error");
        if(block.timestamp < invetTeamConf[1].start) return false;
        uint256 loop = (block.timestamp.sub(invetTeamConf[1].start)).div(invetTeamConf[1].timeinterva
        if(loop<=_releaseInvestLoop) return false;
        if(_releaseInvestLoop>=invetTeamConf[1].releaseTime) return false;
        if(loop>invetTeamConf[1].releaseTime) loop = invetTeamConf[1].releaseTime.add(1);
        uint256 releaseAmt = 0;
        for(uint256 i = _releaseInvestLoop.add(1);i<loop;i++){
            uint256 amounts     = invetTeamConf[1].everyReleaseAmt;
            releaseAmt          = releaseAmt.add(amounts);
            emit ReleaseAirLog("investrelease",investAddr,amounts,i);
            emit Transfer(address(0),investAddr,amounts);
        }
        invetTeamConf[1].hadReleased = invetTeamConf[1].hadReleased.add(releaseAmt);
        assert(invetTeamConf[1].total>invetTeamConf[1].hadReleased);
        _lockR[investAddr]    = _lockR[investAddr].add(releaseAmt.mul(_getRate()));
        _rTotal               = _rTotal.add(releaseAmt.mul(_getRate()));
        _tTotal               = _tTotal.add(releaseAmt);
        _releaseInvestLoop    = loop.sub(1);
        return true;
    }

    function releaseTeam() public  returns(bool){
        require(teamAddr!=address(0),"mat:address is error");
        if(block.timestamp < invetTeamConf[2].start) return false;
        uint256 loop = block.timestamp.sub(invetTeamConf[2].start).div(invetTeamConf[2].timeinterval)
        if(loop<=_releaseTeamLoop) return false;
        if(_releaseTeamLoop>invetTeamConf[2].releaseTime) return false;
        if(loop>invetTeamConf[2].releaseTime) loop = invetTeamConf[2].releaseTime.add(1);
        uint256 releaseAmt = 0;
        for(uint256 i = _releaseTeamLoop.add(1);i<loop;i++){
            uint256 amounts    = invetTeamConf[2].everyReleaseAmt;
            releaseAmt         = releaseAmt.add(amounts);
            emit ReleaseAirLog("teamrelease",teamAddr,amounts,i);
            emit Transfer(address(0),teamAddr,amounts);
        }
        invetTeamConf[2].hadReleased = invetTeamConf[2].hadReleased.add(releaseAmt);
        assert(invetTeamConf[2].total>invetTeamConf[2].hadReleased);
        _lockR[teamAddr]    = _lockR[teamAddr].add(releaseAmt.mul(_getRate()));
        _rTotal             = _rTotal.add(releaseAmt.mul(_getRate()));
        _tTotal             = _tTotal.add(releaseAmt);
        _releaseTeamLoop    = loop.sub(1);
        return true;
    }

    function setContract(address transaddress)  public  onlyOwner returns(bool){
        require(transaddress != address(0),"mat:address is zero");
        transContract = transaddress;
        return true;
    }

    function setAirdrop(address airdrop) public  onlyOwner returns(bool){
        require(airdrop != address(0),"mat:address is zero");
        airdropAddr = airdrop;
        return true;
    }
    function setInvestTeamConf(
        uint8   types,
        uint256 startTime,
        uint256 hadreleased,
        uint256 timeinterval,
        uint256 releaseTime,
        uint256 everyReleaseAmt
    ) public onlyOwner returns(bool){
        require(types>0,"mat:param is error");
        require(startTime>block.timestamp,"mat:param is error");
        require(hadreleased>=0,"mat:param is error");
        require(timeinterval>0,"mat:param is error");
        require(releaseTime>0,"mat:param is error");
        if(types ==1){
            if(isinitVest) return false;
            isinitVest = true;
            InvetTeam memory invetTeamconf = InvetTeam(startTime,_investAmount,hadreleased,releaseTim
            invetTeamConf[types] = invetTeamconf;
        }
        else{
            if(isinitTeam) return false;
            isinitTeam = true;
            InvetTeam memory invetTeamconf = InvetTeam(startTime,_teamAmount,hadreleased,releaseTime,
            invetTeamConf[types] = invetTeamconf;
        }
        return true;
    }
    function setInvestAddr(address investaddr) public  onlyOwner returns(bool){
        require(investaddr != address(0),"mat:address is zero");
        investAddr = investaddr;
        return true;
    }
    function setTeamAddr(address teamaddr) public  onlyOwner returns(bool){
        require(teamaddr != address(0),"mat:address is zero");
        teamAddr = teamaddr;
        return true;
    }

    //set lockconf data
    function setLockConf(uint8 level,uint256 rates,uint256 lockBlocks,uint256 posrate,bool ispos,bool
        require(level>0&&level<20,"mar:level is err");
        require(rates>1,"mar:rate is err");
        require(lockBlocks>0,"mar:lockblock is err");
        lockConfMap[level].rate = rates;
        lockConfMap[level].lockBlockNum = lockBlocks;
        lockConfMap[level].posRate = posrate;
        lockConfMap[level].isPos   = ispos;
        lockConfMap[level].isUsed  = isuse;
        return true;
    }

    function getReleaseAll() public view returns (uint256){
        return _releaseAll;
    }
    function getCurrentAir() public view returns(uint256){
        return _releaseLoop;
    }
    function name() public view returns (string memory) {
        return _name;
    }
    function symbol() public view returns (string memory) {
        return _symbol;
    }
    function decimals() public view returns (uint8) {
        return _decimals;
    }

    function totalSupply() public view override returns (uint256) {
        return _supply;
    }

    function balanceOf(address account) public view override returns (uint256) {
         return _lockR[account].div(_getRate());
    }

    function getbalance(address account) public view  returns (uint256) {
        uint256 ubalance = 0;
        uint256 lrewardOne = 0;
        uint256 lrewardTwo = 0;
        uint256 lrewardThr = 0;
        uint256 lrewardFou = 0;
        if(LockR[account].lockThr>0){
            lrewardOne = LockR[account].lockThr.div(_getRate());
            ubalance = ubalance.add(lrewardOne.sub(lockUserMap[3][account].amount.mul(lockConfMap[3]
        }
        if(LockR[account].lockTwo>0){
            lrewardTwo = LockR[account].lockTwo.div(_getRate());
            ubalance = ubalance.add(lrewardTwo.sub(lockUserMap[2][account].amount.mul(lockConfMap[2]
        }
        if(LockR[account].lockOne>0){
            lrewardThr = LockR[account].lockOne.div(_getRate());
            ubalance = ubalance.add(lrewardThr.sub(lockUserMap[1][account].amount.mul(lockConfMap[1]
        }
        if(LockR[account].lockFou>0){
            lrewardFou = LockR[account].lockFou.div(_getRate());
            ubalance = ubalance.add(lrewardFou.sub(lockUserMap[4][account].amount.mul(lockConfMap[4]
        }
        return _lockR[account].div(_getRate()).add(ubalance);
    }
    function getRate() public view returns(uint256,uint256){
        return (_rTotal,_tTotal);
    }
    function getLockLog(address lockAddress) public view returns(LockUser[] memory) {
        require(_isLocked[lockAddress],"mat:address not lockdata");
        LockUser[] memory lockdata  = new LockUser[](lockList.length);
        for(uint256 i =0;i<lockList.length;i++){
            LockUser memory u = lockUserMap[lockList[i]][lockAddress];
            lockdata[i] = u;
        }
        return lockdata;
    }
    function getlevel() public view returns(LockConf[] memory){
        LockConf[] memory lock  = new LockConf[](lockList.length);
        for(uint8 i =0;i<lockList.length;i++){
            LockConf memory u = lockConfMap[lockList[i]];
            lock[i] = u;
        }
        return lock;
    }
    function getLockAdrr() public view returns(address[] memory){
        return _locked;
    }
    function getMarAdrr() public view returns(address){
        return address(mar);
    }
    function transfer(address recipient, uint256 amount) public override returns (bool) {
        _transfer(_msgSender(), recipient, amount);
        return true;
    }

    function allowance(address owner, address spender) public view override returns (uint256) {
        return _allowances[owner][spender];
    }

    function approve(address spender, uint256 amount) public override returns (bool) {
        _approve(_msgSender(), spender, amount);
        return true;
    }

    function transferFrom(address sender, address recipient, uint256 amount) public override returns
        _transfer(sender, recipient, amount);
        _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount, "ERC20: transfer
        return true;
    }

    function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool) {
        _approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(addedValue));
        return true;
    }

    function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool
        _approve(_msgSender(), spender, _allowances[_msgSender()][spender].sub(subtractedValue, "ERC2
        return true;
    }

    function isLockedFromReward(address account) public view returns (bool) {
        return _isLocked[account];
    }

    function totalFees() public view returns (uint256) {
        return _tFeeTotal;
    }

    function setUseFee(address account,uint8 types) public onlyOwner {
        if(types==1){
            _isNotFromFee[account] = false;
        }
        else{
            _isNotToFee[account] = false;
        }
    }
    function setNotUseFee(address account,uint8 types) public onlyOwner {
        if(types ==1){
            _isNotFromFee[account] = true;
        }
        else{
             _isNotToFee[account]  = true;
        }
    }

    function setTaxFeePercent(uint256 taxFee) external onlyOwner() {
        _taxFee = taxFee;
    }
    function setTaxRfeePercent(uint256 taxRfee) external onlyOwner() {
        _burnRate = taxRfee;
    }

    function _sumFee( uint256 rFee ,uint256 tFee) private {
        _rTotal    = _rTotal.sub(rFee);
        _tFeeTotal = _tFeeTotal.add(tFee);
        emit BurnLog("burn",tFee.sub(calculateBurnRfee(tFee)));
    }

    function _getValues(uint256 tAmount) private view returns (uint256, uint256, uint256, uint256, ui
        (uint256 tTransferAmount, uint256 tFee) = _getTValues(tAmount);
        (uint256 rAmount, uint256 rTransferAmount, uint256 rFee) = _getRValues(tAmount, tFee, _getRat
        return (rAmount, rTransferAmount, rFee, tTransferAmount, tFee);
    }

    function _getRValues(uint256 tAmount, uint256 tFee, uint256 currentRate) private pure returns (ui
        uint256 rAmount = tAmount.mul(currentRate);
        uint256 rFee = tFee.mul(currentRate);
        uint256 rTransferAmount = rAmount.sub(rFee);
        return (rAmount, rTransferAmount, rFee);
    }

    function _getTValues(uint256 tAmount) private view returns (uint256, uint256) {
        uint256 tFee = calculateTaxFee(tAmount);
        uint256 tTransferAmount = tAmount.sub(tFee);
        return (tTransferAmount, tFee);
    }
    function _getRate() private view returns(uint256) {
        return _rTotal.div(_tTotal);
    }

    function calculateTaxFee(uint256 _amount) private view returns (uint256) {
        return _amount.mul(_taxFee).div(
            10**3
        );
    }
    function calculateBurnRfee(uint256 _amount) private view returns (uint256) {
        return _amount.mul(_burnRate).div(
            10**2
        );
    }

    function removeAllFee() private {
        if(_taxFee == 0) return;
        _previousTaxFee = _taxFee;
        _taxFee = 0;
    }

    function restoreAllFee() private {
        _taxFee = _previousTaxFee;
    }

    function _approve(address owner, address spender, uint256 amount) private {
        require(owner != address(0), "mat:ERC20: approve from the zero address");
        require(spender != address(0), "mat:ERC20: approve to the zero address");
        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }

    function _transfer(
        address from,
        address to,
        uint256 amount
    ) private {
        require(from != address(0), "mat:zero address");
        require(to != address(0), "mat:zero address");
        require(amount > 0, "mat:amount zero");
        if(from != owner() && to != owner())
            require(amount <= _maxTxAmount, "mat:Transfer limit maxTxAmount.");
        bool takeFee = true;

        if(_isNotFromFee[from] || _isNotToFee[to]){
            takeFee = false;
        }
        _tokenTransfer(from,to,amount,takeFee);
    }


    function _tokenTransfer(address sender, address recipient, uint256 amount,bool takeFee) private {
        if(!takeFee)
            removeAllFee();
        if(recipient == investAddr||recipient == airdropAddr||recipient == teamAddr){
            require(false,"recipient do not receive coin");
        }
        _transferStandard(sender, recipient, amount);
        if(!takeFee)
            restoreAllFee();
    }

    function _transferStandard(address sender, address recipient, uint256 tAmount) private {
        (uint256 rAmount, uint256 rTransferAmount,uint256 rFee,uint256 tTransferAmount, uint256 tFee)
        _lockR[sender]    = _lockR[sender].sub(rAmount);
        _lockR[recipient] = _lockR[recipient].add(rTransferAmount);
        _sumFee(calculateBurnRfee(rFee),tFee);
        emit Transfer(sender, recipient, tTransferAmount);
    }


    function addLock(uint256 lockAmount,uint8 lockLevel) public  returns(bool){
        require(_msgSender()!=address(0),"mat:address is zero");
        require(lockAmount > 0, "mat:lockAmount err");
        require(lockLevel >0,"mat:locklevel err");
        require(block.number >startBlock,"mat:lock is not start");
        require(lockConfMap[lockLevel].isUsed,"mat:locklevel is error");
        if(lockUserMap[lockLevel][_msgSender()].isUsed&&lockUserMap[lockLevel][_msgSender()].isLock){
            lockUserMap[lockLevel][_msgSender()].amount = lockUserMap[lockLevel][_msgSender()].amount
            lockUserMap[lockLevel][_msgSender()].unlockBlocknum = block.number.add(lockConfMap[lockLe
            bool islevelPos = lockConfMap[lockLevel].isPos;
            if(islevelPos){
                lockUserMap[lockLevel][_msgSender()].posRate = lockConfMap[lockLevel].posRate;
            }else{
                lockUserMap[lockLevel][_msgSender()].posRate = 0;
            }
            lockUse(_msgSender(),lockAmount,lockLevel);
        }
        else{
            _isLocked[_msgSender()]  = true;
            _locked.push(_msgSender());
            bool islevelPos = lockConfMap[lockLevel].isPos;
            if(islevelPos){
                LockUser memory u = LockUser(lockAmount,block.number.add(lockConfMap[lockLevel].lockB
                lockUserMap[lockLevel][_msgSender()] = u;
            }else{
                LockUser memory u = LockUser(lockAmount,block.number.add(lockConfMap[lockLevel].lockB
                lockUserMap[lockLevel][_msgSender()] = u;
            }
            lockUse(_msgSender(),lockAmount,lockLevel);
        }
        return true;
    }
    function lockUse(address senders ,uint256 lockAmount,uint8 level) private {
        _lockTotal = _lockTotal.add(lockAmount);
        uint256  lockWeight   = lockAmount.mul(lockConfMap[level].rate);

        sumRateLockUser[senders] = sumRateLockUser[senders].add(lockWeight);
        _lockAllWeight      = _lockAllWeight.add(lockWeight);

        (uint256 ruAmount,,,,) = _getValues(lockAmount);
        _lockR[senders]     = _lockR[senders].sub(ruAmount);
        _tTotal             = _tTotal.sub(lockAmount);
        _rTotal             = _rTotal.sub(ruAmount);

        (uint256 rAmount,,,,) = _getValues(lockWeight);
        if(level == 4){
            LockR[senders].lockFou = LockR[senders].lockFou.add(rAmount);
        }else if(level==3){
            LockR[senders].lockThr = LockR[senders].lockThr.add(rAmount);
        }else if(level ==2){
            LockR[senders].lockTwo = LockR[senders].lockTwo.add(rAmount);
        }
        else{
            LockR[senders].lockOne = LockR[senders].lockOne.add(rAmount);
        }
        _tTotal = _tTotal.add(lockWeight);
        _rTotal = _rTotal.add(rAmount);
        emit LockCoin("lock",senders,lockAmount,level);
    }

    function unlockUsercoinAll() external  returns(bool){
        require(_isLocked[_msgSender()],"mat:notlockdata");
        uint8 ishad = 0;
        for(uint8 i = 1;i<4;i++){
            if(!lockUserMap[i][_msgSender()].isUsed){
                continue;
            }
            if(lockUserMap[i][_msgSender()].isLock&&lockUserMap[i][_msgSender()].unlockBlocknum>block
                ishad = 1;
            }
            unlockHander(i,_msgSender());
        }
        if(ishad==0){
            _isLocked[_msgSender()] = false;
            for(uint256 i =0;i<_locked.length;i++){
                if(_locked[i]==_msgSender()){
                    _locked[i] = _locked[_locked.length-1];
                    _locked.pop();
                }
            }
        }
        return true;
    }

    function unlockUsercoin(uint8 level) public  returns(bool){
        require(_isLocked[_msgSender()],"mat:notlockdata");
        require(level>0,"mat:paramerr");
        if(!lockUserMap[level][_msgSender()].isUsed){
            return false;
        }
        unlockHander(level,_msgSender());
        return true;
    }
    function unlockHander(uint8 level,address lockaddress) private {
        if(lockUserMap[level][lockaddress].isLock&&lockUserMap[level][lockaddress].unlockBlocknum<blo
            uint256 amounts = lockUserMap[level][lockaddress].amount;
            uint256 amountPos = 0;
            if(lockUserMap[level][lockaddress].posRate>0){
                amountPos = amounts.mul(lockConfMap[level].posRate).div(100);
            }

            uint256  lockWeight = amounts.mul(lockConfMap[level].rate);
            sumRateLockUser[lockaddress] = sumRateLockUser[lockaddress].sub(lockWeight);

            _lockAllWeight  = _lockAllWeight.sub(lockWeight);
            uint256 lockuR  = 0;
            if(level==4){
                lockuR = LockR[lockaddress].lockFou;
                LockR[lockaddress].lockFou = 0;
            }else if(level==3){
                lockuR = LockR[lockaddress].lockThr;
                LockR[lockaddress].lockThr = 0;
            }else if(level ==2){
                lockuR = LockR[lockaddress].lockTwo;
                LockR[lockaddress].lockTwo = 0;
            }
            else{
                lockuR = LockR[lockaddress].lockOne;
                LockR[lockaddress].lockOne = 0;
            }

            uint256 reaward        = lockuR.div(_getRate());
            _tTotal                = _tTotal.sub(reaward);
            _rTotal                = _rTotal.sub(lockuR);
            assert(reaward>=lockWeight);
            amounts = amounts.add(reaward-lockWeight);
            amounts = amounts.add(amountPos);
            (uint256 rAmount,,,,) = _getValues(amounts);
            _lockR[lockaddress] = _lockR[lockaddress].add(rAmount);
            _tTotal             = _tTotal.add(amounts);
            _rTotal             = _rTotal.add(rAmount);
            delete lockUserMap[level][lockaddress];
            emit ReleaseCoin("releaseCoin",amounts,level,lockaddress);
            emit ReleaseCoin("releasePos",amountPos,level,lockaddress);
        }
    }

    function getUsereward(address addr,uint256 amount) public  returns(bool){
        require(addr!=address(0),"mat:address is zero");
        require(amount>0&&amount<_supply,"mat:amount must be greater 0");
        require(_msgSender()== transContract,"mat:address is not permition");
        (uint256 rAmount,,,,) = _getValues(amount);
        _tTotal              = _tTotal.add(amount);
        _rTotal              = _rTotal.add(rAmount);
        _lockR[addr]    = _lockR[addr].add(rAmount);
        _releaseAll     = _releaseAll.add(amount);
        emit Transfer(address(0),addr,amount);
        return true;
    }

    function getDaysupply() public view returns(uint256,uint256){
        if(block.number<=startBlock){
            return(0,0);
        }
        else{
            uint256 loop = (block.number.sub(startBlock)).div(blockNum).add(1);
            for(uint256 i=0;i<releaseBlock.length;i++){
                 if(i==releaseBlock.length-1){
                    return(loop,releaseAmount[i]*10**9);
                }
                if(block.number>releaseBlock[i]&&block.number<=releaseBlock[i+1]){
                     return(loop,releaseAmount[i]*10**9);
                }
            }
            return(0,0);
        }
    }

    function getPreDaysupply(uint256 loop) public view returns(uint256,uint256){
        require(loop>0,"mat:param err");
        if(block.number<=startBlock){
            return(0,0);
        }
        else{
            uint256 preBlocknum = startBlock.add(blockNum.mul(loop));
            for(uint256 i=0;i<releaseBlock.length;i++){
                if(i==releaseBlock.length-1){
                    return(loop,releaseAmount[i]*10**9);
                }
                if((preBlocknum>releaseBlock[i])&&(preBlocknum<=releaseBlock[i+1])){
                    return(loop,releaseAmount[i]*10**9);
                }
            }
            return(0,0);
        }
    }
    function getAirDaysupply(uint256 loop) public view returns(uint256,uint256){
        require(loop>0,"mat:param err");
        if(block.number<=startBlock){
            return(0,0);
        }
        else{
            uint256 preBlocknum = startBlock.add(blockNum.mul(loop));
            for(uint256 i=0;i<releaseBlock.length;i++){
                if(i==releaseBlock.length-1){
                    return(loop,releaseAirAmount[i]*10**9);
                }
                if(preBlocknum>releaseBlock[i]&&(preBlocknum)<=releaseBlock[i+1]){
                    return(loop,releaseAirAmount[i]*10**9);
                }
            }
            return(0,0);
        }
    }
    function getAirDaysupply() public view returns(uint256,uint256){
        if(block.number<=startBlock){
            return(0,0);
        }
        else{
            uint256 loop = (block.number.sub(startBlock)).div(blockNum).add(1);
            for(uint256 i=0;i<releaseBlock.length-1;i++){
                if(i==releaseBlock.length-1){
                    return(loop,releaseAirAmount[i]*10**9);
                }
                if(block.number>releaseBlock[i]&&block.number<=releaseBlock[i+1]){
                    return(loop,releaseAirAmount[i]*10**9);
                }
            }
            return(0,0);
        }
    }
    //burn coin from old contract and mint to new contract
    function exchangeCoin(uint256 amt) public returns(bool){
        require(amt>0,"amt is error");
        require(_msgSender()!=address(mar),"address is error");
        bool ret = mar.transferFrom(_msgSender(),address(mar),amt);
        if(ret){
           (uint256 ruAmount,,,,) = _getValues(amt);
           _lockR[_msgSender()]   = _lockR[_msgSender()].add(ruAmount);
           _rTotal                = _rTotal.add(ruAmount);
           _tTotal                = _tTotal.add(amt);
           emit Transfer(address(0),_msgSender(),amt);
           emit ExchangeLog("exchangecoin",_msgSender(),amt);
           return true;
        }
        return false;
    }
}
```

MartTradeMining.sol

```
/**
 *Submitted for verification at hecoinfo.com on 2021-10-11
*/

// SPDX-License-Identifier: MIT

pragma solidity >=0.6.0 <0.8.0;

/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when an
 * operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
 * class of bugs, so it's recommended to use it always.
 */
library SafeMathUpgradeable {
    /**
     * @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) {
        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) {
        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) {
        // 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) {
        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) {
        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) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");
        return c;
    }

    /**
     * @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) {
        require(b <= a, "SafeMath: subtraction overflow");
        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) {
        if (a == 0) return 0;
        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");
        return c;
    }

    /**
     * @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. 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) internal pure returns (uint256) {
        require(b > 0, "SafeMath: division by zero");
        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) {
        require(b > 0, "SafeMath: modulo by zero");
        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) {
        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.
     *
     * CAUTION: This function is deprecated because it requires allocating memory for the error
     * message unnecessarily. For custom revert reasons use {tryDiv}.
     *
     * 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) {
        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) {
        require(b > 0, errorMessage);
        return a % b;
    }
}

// File: @openzeppelin/contracts-upgradeable/utils/AddressUpgradeable.sol


pragma solidity >=0.6.2 <0.8.0;

/**
 * @dev Collection of functions related to the address type
 */
library AddressUpgradeable {
    /**
     * @dev Returns true if `account` is a contract.
     *
     * [IMPORTANT]
     * ====
     * It is unsafe to assume that an address for which this function returns
     * false is an externally-owned account (EOA) and not a contract.
     *
     * Among others, `isContract` will return false for the following
     * types of addresses:
     *
     *  - an externally-owned account
     *  - a contract in construction
     *  - an address where a contract will be created
     *  - an address where a contract lived, but was destroyed
     * ====
     */
    function isContract(address account) internal view returns (bool) {
        // This method relies on extcodesize, which returns 0 for contracts in
        // construction, since the code is only stored at the end of the
        // constructor execution.

        uint256 size;
        // solhint-disable-next-line no-inline-assembly
        assembly { size := extcodesize(account) }
        return size > 0;
    }

    /**
     * @dev Replacement for Solidity's `transfer`: sends `amount` wei to
     * `recipient`, forwarding all available gas and reverting on errors.
     *
     * https://eips.ethereum.org/EIPS/eip-1884[EIP1884] increases the gas cost
     * of certain opcodes, possibly making contracts go over the 2300 gas limit
     * imposed by `transfer`, making them unable to receive funds via
     * `transfer`. {sendValue} removes this limitation.
     *
     * https://diligence.consensys.net/posts/2019/09/stop-using-soliditys-transfer-now/[Learn more].
     *
     * IMPORTANT: because control is transferred to `recipient`, care must be
     * taken to not create reentrancy vulnerabilities. Consider using
     * {ReentrancyGuard} or the
     * https://solidity.readthedocs.io/en/v0.5.11/security-considerations.html#use-the-checks-effects
     */
    function sendValue(address payable recipient, uint256 amount) internal {
        require(address(this).balance >= amount, "Address: insufficient balance");

        // solhint-disable-next-line avoid-low-level-calls, avoid-call-value
        (bool success, ) = recipient.call{ value: amount }("");
        require(success, "Address: unable to send value, recipient may have reverted");
    }

    /**
     * @dev Performs a Solidity function call using a low level `call`. A
     * plain`call` is an unsafe replacement for a function call: use this
     * function instead.
     *
     * If `target` reverts with a revert reason, it is bubbled up by this
     * function (like regular Solidity function calls).
     *
     * Returns the raw returned data. To convert to the expected return value,
     * use https://solidity.readthedocs.io/en/latest/units-and-global-variables.html?highlight=abi.de
     *
     * Requirements:
     *
     * - `target` must be a contract.
     * - calling `target` with `data` must not revert.
     *
     * _Available since v3.1._
     */
    function functionCall(address target, bytes memory data) internal returns (bytes memory) {
      return functionCall(target, data, "Address: low-level call failed");
    }

    /**
```

```
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`], but with
     * `errorMessage` as a fallback revert reason when `target` reverts.
     *
     * _Available since v3.1._
     */
    function functionCall(address target, bytes memory data, string memory errorMessage) internal ret
        return functionCallWithValue(target, data, 0, errorMessage);
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],
     * but also transferring `value` wei to `target`.
     *
     * Requirements:
     *
     * - the calling contract must have an ETH balance of at least `value`.
     * - the called Solidity function must be `payable`.
     *
     * _Available since v3.1._
     */
    function functionCallWithValue(address target, bytes memory data, uint256 value) internal returns
        return functionCallWithValue(target, data, value, "Address: low-level call with value failed"
    }

    /**
     * @dev Same as {xref-Address-functionCallWithValue-address-bytes-uint256-}[`functionCallWithValu
     * with `errorMessage` as a fallback revert reason when `target` reverts.
     *
     * _Available since v3.1._
     */
    function functionCallWithValue(address target, bytes memory data, uint256 value, string memory er
        require(address(this).balance >= value, "Address: insufficient balance for call");
        require(isContract(target), "Address: call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.call{ value: value }(data);
        return _verifyCallResult(success, returndata, errorMessage);
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],
     * but performing a static call.
     *
     * _Available since v3.3._
     */
    function functionStaticCall(address target, bytes memory data) internal view returns (bytes memor
        return functionStaticCall(target, data, "Address: low-level static call failed");
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-string-}[`functionCall`],
     * but performing a static call.
     *
     * _Available since v3.3._
     */
    function functionStaticCall(address target, bytes memory data, string memory errorMessage) intern
        require(isContract(target), "Address: static call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.staticcall(data);
        return _verifyCallResult(success, returndata, errorMessage);
    }

    function _verifyCallResult(bool success, bytes memory returndata, string memory errorMessage) pri
        if (success) {
            return returndata;
        } else {
            // Look for revert reason and bubble it up if present
            if (returndata.length > 0) {
                // The easiest way to bubble the revert reason is using memory via assembly

                // solhint-disable-next-line no-inline-assembly
                assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata), returndata_size)
                }
            } else {
                revert(errorMessage);
            }
        }
    }
}

// File: @openzeppelin/contracts-upgradeable/proxy/Initializable.sol



// solhint-disable-next-line compiler-version
pragma solidity >=0.4.24 <0.8.0;


/**
 * @dev This is a base contract to aid in writing upgradeable contracts, or any kind of contract that
 * behind a proxy. Since a proxied contract can't have a constructor, it's common to move constructor
 * external initializer function, usually called `initialize`. It then becomes necessary to protect t
 * function so it can only be called once. The {initializer} modifier provided by this contract will
 *
 * TIP: To avoid leaving the proxy in an uninitialized state, the initializer function should be call
 * possible by providing the encoded function call as the `_data` argument to {UpgradeableProxy-const
 *
 * CAUTION: When used with inheritance, manual care must be taken to not invoke a parent initializer
 * that all initializers are idempotent. This is not verified automatically as constructors are by So
 */
abstract contract Initializable {

    /**
     * @dev Indicates that the contract has been initialized.
     */
    bool private _initialized;

    /**
     * @dev Indicates that the contract is in the process of being initialized.
     */
    bool private _initializing;

    /**
     * @dev Modifier to protect an initializer function from being invoked twice.
     */
    modifier initializer() {
        require(_initializing || _isConstructor() || !_initialized, "Initializable: contract is alrea

        bool isTopLevelCall = !_initializing;
        if (isTopLevelCall) {
            _initializing = true;
            _initialized = true;
        }

        _;

        if (isTopLevelCall) {
            _initializing = false;
        }
    }

    /// @dev Returns true if and only if the function is running in the constructor
    function _isConstructor() private view returns (bool) {
        return !AddressUpgradeable.isContract(address(this));
    }
}

// File: @openzeppelin/contracts-upgradeable/utils/ReentrancyGuardUpgradeable.sol



pragma solidity >=0.6.0 <0.8.0;


/**
 * @dev Contract module that helps prevent reentrant calls to a function.
 *
 * Inheriting from `ReentrancyGuard` will make the {nonReentrant} modifier
 * available, which can be applied to functions to make sure there are no nested
 * (reentrant) calls to them.
 *
 * Note that because there is a single `nonReentrant` guard, functions marked as
 * `nonReentrant` may not call one another. This can be worked around by making
 * those functions `private`, and then adding `external` `nonReentrant` entry
 * points to them.
 *
 * TIP: If you would like to learn more about reentrancy and alternative ways
 * to protect against it, check out our blog post
 * https://blog.openzeppelin.com/reentrancy-after-istanbul/[Reentrancy After Istanbul].
 */
abstract contract ReentrancyGuardUpgradeable is Initializable {
    // Booleans are more expensive than uint256 or any type that takes up a full
    // word because each write operation emits an extra SLOAD to first read the
    // slot's contents, replace the bits taken up by the boolean, and then write
    // back. This is the compiler's defense against contract upgrades and
    // pointer aliasing, and it cannot be disabled.

    // The values being non-zero value makes deployment a bit more expensive,
    // but in exchange the refund on every call to nonReentrant will be lower in
    // amount. Since refunds are capped to a percentage of the total
    // transaction's gas, it is best to keep them low in cases like this one, to
    // increase the likelihood of the full refund coming into effect.
    uint256 private constant _NOT_ENTERED = 1;
    uint256 private constant _ENTERED = 2;

    uint256 private _status;

    function __ReentrancyGuard_init() internal initializer {
        __ReentrancyGuard_init_unchained();
    }

    function __ReentrancyGuard_init_unchained() internal initializer {
        _status = _NOT_ENTERED;
    }

    /**
     * @dev Prevents a contract from calling itself, directly or indirectly.
     * Calling a `nonReentrant` function from another `nonReentrant`
     * function is not supported. It is possible to prevent this from happening
     * by making the `nonReentrant` function external, and make it call a
     * `private` function that does the actual work.
     */
    modifier nonReentrant() {
        // On the first call to nonReentrant, _notEntered will be true
        require(_status != _ENTERED, "ReentrancyGuard: reentrant call");
        // Any calls to nonReentrant after this point will fail
        _status = _ENTERED;

        _;

        // By storing the original value once again, a refund is triggered (see
        // https://eips.ethereum.org/EIPS/eip-2200)
        _status = _NOT_ENTERED;
    }
    uint256[49] private __gap;
}

// File: contracts/interface/IInvit.sol


pragma solidity 0.7.0;

interface IInvit {
    function isTrade(address userAddress) external view returns (bool);
    function getInvit(address userAddress) external view returns (address[] memory);
    function appendInvit (address beInvit ,address invit) external returns(bool);
}

// File: contracts/interface/IMartToken.sol


pragma solidity >=0.4.22 <0.9.0;
pragma experimental ABIEncoderV2;

interface IMartToken {
    function startBlock() external view returns (uint256);
    function getDaysupply() external view returns (uint256, uint256);
    function getPreDaysupply(uint256 _day) external view returns (uint256, uint256);
    function getUsereward(address addr, uint256 amount) external returns (bool);
    function addLockFee() external returns (bool);
}

// File: @openzeppelin/contracts-upgradeable/token/ERC20/IERC20Upgradeable.sol



/**
 * @dev Interface of the ERC20 standard as defined in the EIP.
 */
interface IERC20Upgradeable {
    /**
     * @dev Returns the amount of tokens in existence.
     */
    function totalSupply() external view returns (uint256);

    /**
     * @dev Returns the amount of tokens owned by `account`.
     */
    function balanceOf(address account) external view returns (uint256);

    /**
     * @dev Moves `amount` tokens from the caller's account to `recipient`.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transfer(address recipient, uint256 amount) external returns (bool);

    /**
```

```
     * @dev Returns the remaining number of tokens that `spender` will be
     * allowed to spend on behalf of `owner` through {transferFrom}. This is
     * zero by default.
     *
     * This value changes when {approve} or {transferFrom} are called.
     */
    function allowance(address owner, address spender) external view returns (uint256);

    /**
     * @dev Sets `amount` as the allowance of `spender` over the caller's tokens.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * IMPORTANT: Beware that changing an allowance with this method brings the risk
     * that someone may use both the old and the new allowance by unfortunate
     * transaction ordering. One possible solution to mitigate this race
     * condition is to first reduce the spender's allowance to 0 and set the
     * desired value afterwards:
     * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
     *
     * Emits an {Approval} event.
     */
    function approve(address spender, uint256 amount) external returns (bool);

    /**
     * @dev Moves `amount` tokens from `sender` to `recipient` using the
     * allowance mechanism. `amount` is then deducted from the caller's
     * allowance.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);

    /**
     * @dev Emitted when `value` tokens are moved from one account (`from`) to
     * another (`to`).
     *
     * Note that `value` may be zero.
     */
    event Transfer(address indexed from, address indexed to, uint256 value);

    /**
     * @dev Emitted when the allowance of a `spender` for an `owner` is set by
     * a call to {approve}. `value` is the new allowance.
     */
    event Approval(address indexed owner, address indexed spender, uint256 value);
}

// File: contracts/interface/IERC20.sol

pragma solidity 0.7.0;



interface IERC20 is IERC20Upgradeable {
    function decimals() external view returns (uint8);
}

// File: @openzeppelin/contracts-upgradeable/utils/ContextUpgradeable.sol



pragma solidity >=0.6.0 <0.8.0;
```

```
/*
 * @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 GSN 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 ContextUpgradeable is Initializable {
    function __Context_init() internal initializer {
        __Context_init_unchained();
    }

    function __Context_init_unchained() internal initializer {
    }
    function _msgSender() internal view virtual returns (address payable) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes memory) {
        this; // silence state mutability warning without generating bytecode - see https://github.co
        return msg.data;
    }
    uint256[50] private __gap;
}

// File: @openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol



pragma solidity >=0.6.0 <0.8.0;


/**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * By default, the owner account will be the one that deploys the contract. This
 * can later be changed with {transferOwnership}.
 *
 * This module is used through inheritance. It will make available the modifier
 * `onlyOwner`, which can be applied to your functions to restrict their use to
 * the owner.
 */
abstract contract OwnableUpgradeable is Initializable, ContextUpgradeable {
    address private _owner;

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

    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    function __Ownable_init() internal initializer {
        __Context_init_unchained();
        __Ownable_init_unchained();
    }

    function __Ownable_init_unchained() internal initializer {
        address msgSender = _msgSender();
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }
    /**
     * @dev Returns the address of the current owner.
     */
    function owner() public view virtual returns (address) {
        return _owner;
    }

    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(owner() == _msgSender(), "Ownable: caller is not the owner");
        _;
    }

    /**
     * @dev Leaves the contract without owner. It will not be possible to call
     * `onlyOwner` functions anymore. Can only be called by the current owner.
     *
     * NOTE: Renouncing ownership will leave the contract without an owner,
     * thereby removing any functionality that is only available to the owner.
     */
    function renounceOwnership() public virtual onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
    uint256[49] private __gap;
}

// File: contracts/interface/IAggregatorV3Interface.sol


pragma solidity 0.7.0;

interface IAggregatorV3Interface {

  function decimals()
    external
    view
    returns (
      uint8
    );

  function description()
    external
    view
    returns (
      string memory
    );

  function version()
    external
    view
    returns (
      uint256
    );

  // getRoundData and latestRoundData should both raise "No data present"
  // if they do not have data to report, instead of returning unset values
  // which could be misinterpreted as actual reported values.
  function getRoundData(
    uint80 _roundId
  )
    external
    view
    returns (
      uint80 roundId,
      int256 answer,
      uint256 startedAt,
      uint256 updatedAt,
      uint80 answeredInRound
    );

  function latestRoundData()
    external
    view
    returns (
      uint80 roundId,
      int256 answer,
      uint256 startedAt,
      uint256 updatedAt,
      uint80 answeredInRound
    );

}

// File: contracts/Oracle/Oracle.sol

pragma solidity 0.7.0;




contract Oracle is OwnableUpgradeable {
    mapping(address => IAggregatorV3Interface) public getContractAddress;
    mapping(address => uint256) public getPrice;
    function _Oracle_INIT_ () public initializer {
            __Ownable_init();
    }
    /// @dev Set price oracle contract addresses in batches
    function setOracleBatch (address[] memory _tokens, address[] memory _aggregatorV3Interface) publi
        require(_tokens.length < 100, 'Oracle: Too many settings at once');
        require(_tokens.length == _aggregatorV3Interface.length, 'Oracle: Unequal length');
        for (uint8 i = 0; i < _tokens.length; i++) {
            getContractAddress[_tokens[i]] = IAggregatorV3Interface(_aggregatorV3Interface[i]);
        }
    }
    /// @dev Update the price of a certain token from the price oracle machine
    function _updatePrice(address _token) internal {
        (,int price,,,) = getContractAddress[_token].latestRoundData();
        require(price >= 0, 'Oracle: price verify error');
        getPrice[_token] = uint256(price);
    }
}

// File: contracts/MartTradeMining.sol


pragma solidity >=0.4.22 <0.9.0;
```

```
contract MartTradeMining is Oracle, ReentrancyGuardUpgradeable {
    using SafeMathUpgradeable for uint;
    // User data structure
    struct PersonData {
        // cycle
        uint256 _days;
        // The value of the fee for the current period => USD
        uint256 toDayFeeAmt;
        // Number of transaction mining rewards
        uint256 toBeReleased;
        // Number of referral rewards
        uint256 toBeReleasedByInvit;
    }
    // Total fee per cycle
    mapping(uint256 => uint256) private totalFeeAmts;
    // Release amount of MART released per cycle
    mapping(uint256 => uint256) public martSupply;
    // User statistics mapping
    mapping(address => PersonData) private userReward;
    // Abandoned
    mapping(address => uint256) public getInvitReward;
    // Allow to write the address of the user fee
    mapping(address => bool) public Authorization;
    // Abandoned
    address public exchange;
    // MART token contract address
    address public martToken;
    // Referrer contract address
    address public invit;
    // Minimum handling fee decimal
    uint32 public feeDecimals;
    // The last statistical period in the contract
    uint256 public lastDay;
    // Whether to allow extraction
    bool public allowExtraction;

    // Linear release
    struct LockUserData {
        uint256 totalAmount;
        uint256 left;
        uint256 preReleaseBlock;
        uint16  releaseLoop;
        uint256 releasedAmt;
        uint256 singleQuantity;
        uint256 preReleaseLoop;
    }
    mapping(address => LockUserData) public LockMap;
    // Release cycle
    uint16 public releaseTime;
    // Number of locked blocks per withdrawal
    uint16 public lockBlockNum;
    // The number of users who should reduce production in the corresponding period
    mapping(address => mapping(uint256 => uint256)) public reduceLoopAmount;
    /// @dev Transaction mining withdrawal event
    event WithDrawReward(address indexed user_address, address indexed  _to, uint256 _amount);
    // Receive referral reward
    event ReceiveReferralReward(address indexed from, address indexed to, uint256 amount);
    // Receive transaction mining rewards
    event ReceiveMiningRewards(address indexed _address, uint256 _loop, uint256 amount);
    /// @dev Initialization method
    function __martTradeMiningInit__ (address _invit, address _mart, uint32 _feeDecimals, bool _allow
        _Oracle_INIT_();
        __ReentrancyGuard_init();
        feeDecimals = _feeDecimals;
        invit = _invit;
        martToken = _mart;
        allowExtraction = _allowExtraction;
    }
    /// @dev Statistic fee authority
    modifier isAuth () {
        require(Authorization[_msgSender()], 'MartTradeMining: auth error');
        _;
    }
    /// @dev Whether to allow extraction
    modifier isAllow () {
        require(allowExtraction, 'MartTradeMining: allowExtraction false');
        _;
    }
    /// @dev Set release period
    function setReleaseLoop (uint16 _loop) public onlyOwner {
        releaseTime = _loop;
    }
    /// @dev Set a locked block
    function setLockBlock(uint16 _block) public onlyOwner {
        lockBlockNum = _block;
    }
    /// @dev Configure statistics fee permissions in batches
    function setAuthorizationBatch(address[] memory _addrs, bool[] memory isOpen) public onlyOwner {
        require(_addrs.length < 100, 'MartTradeMining: Too many settings at once');
        require(_addrs.length == isOpen.length, 'MartTradeMining: Unequal length');
        for (uint8 i = 0; i < _addrs.length; i++) {
            Authorization[_addrs[i]] = isOpen[i];
        }
    }
    /// @dev Set the token contract address
    function setMartAddress (address _mart) external onlyOwner {
        require(_mart != address(0), 'MartTradeMining:address is zero address');
        martToken = _mart;
    }
    /// @dev Set the mining switch for withdrawal transactions
    function setAllowExtraction (bool _isAllow) external onlyOwner {
        allowExtraction = _isAllow;
    }
    /// @dev Set recommender contract address
    function setInvit(address _invit) external onlyOwner {
        require(_invit != address(0), 'MartTradeMining: _invit is zero address');
        invit = _invit;
    }
    /// @dev Statistics account transaction fees
    function countUserExchangeFee(address user_address, address tokenAddress, uint256 _days, uint256
        require(user_address != address(0), 'MartTradeMining: user_address is zero address');
        require(martSupply[_days] == 0, 'MartTradeMining: Reward has been released');
        if (_days == 0) {
            return true;
        }
        require(_days >= lastDay, 'MartTradeMining: Outdated trading cycle');
        // If the cycle changes, the rewards of the previous cycle will be settled and the token pric
        if (_days > lastDay) {
            if (lastDay > 0 && martSupply[lastDay] == 0) {
                (, uint256 _supply) = IMartToken(martToken).getPreDaysupply(lastDay);
                martSupply[lastDay] = _supply;
            }
            lastDay = _days;
            _updatePrice(tokenAddress);
        }
        // If the user’s statistics are out of date, reset
        if (lastDay > userReward[user_address]._days) {
            clearUserFeeAmt(user_address);
            userReward[user_address]._days = lastDay;
        }
        uint8 tokenDesc = 18;
        if (tokenAddress != address(0)) {
            tokenDesc = IERC20(tokenAddress).decimals();
        }
        uint feeRate = 10 ** tokenDesc;
        // tokenprice decimals 8
        uint256 tokenPrice = getPrice[tokenAddress];
        uint256 marketValueUsdBig = SafeMathUpgradeable.mul(tokenPrice, _fee);
        uint256 marketValueUsd = SafeMathUpgradeable.div(marketValueUsdBig, feeRate);
        totalFeeAmts[_days] = SafeMathUpgradeable.add(totalFeeAmts[_days], marketValueUsd);
        userReward[user_address].toDayFeeAmt = SafeMathUpgradeable.add(userReward[user_address].toDay
        return true;
    }
    /// @dev Reset last period statistics
    function clearUserFeeAmt(address user_address) internal {
        (uint256 userToBeReleased,,uint256 invitUserFeeAmt) = calculateRewardAmt(user_address, 2);
        // Give rewards to recommenders
        if (invitUserFeeAmt > 0) {
            changeInvitUserFeeAmt(user_address, invitUserFeeAmt);
        }
        emit ReceiveMiningRewards(user_address, userReward[user_address]._days, SafeMathUpgradeable.s
        userReward[user_address].toBeReleased = userToBeReleased;
        userReward[user_address].toDayFeeAmt = 0;
        updateLockMap(user_address, true);
        userReward[user_address].toBeReleased = 0;
    }
    /// @dev Get user statistics
    function getUserData (address user_address) external view returns (PersonData memory) {
        return userReward[user_address];
    }
    /// @dev Settlement referral fee
    function changeInvitUserFeeAmt (address user_address, uint256 feeAmt) internal returns (bool) {
        address[] memory invitAddressList = IInvit(invit).getInvit(user_address);
        // Two-level recommender, 7% at the first level, 3% at the second level
        uint8 level = invitAddressList.length >= 2 ? 2 : uint8(invitAddressList.length);
        for (uint8 _i = 0; _i < level; _i++) {
            uint256 invitReward;
            if (_i == 0) {
                invitReward = SafeMathUpgradeable.div(SafeMathUpgradeable.mul(feeAmt, 7), 10);
            } else if (_i == 1) {
                invitReward = SafeMathUpgradeable.div(SafeMathUpgradeable.mul(feeAmt, 3), 10);
            }
            userReward[invitAddressList[_i]].toBeReleasedByInvit = SafeMathUpgradeable.add(userReward
            addInvitRewardLinearRelease(invitAddressList[_i], userReward[invitAddressList[_i]].toBeRe
            userReward[invitAddressList[_i]].toBeReleasedByInvit = 0;
            emit ReceiveReferralReward(user_address, invitAddressList[_i], invitReward);
        }
        return true;
    }
    /// @dev Calculate the number of transaction mining and referrer rewards
    /// @param user_address userAddress
    /// @param _type 1 Referrer rewards 2 Reward payable to recommender
    function calculateRewardAmt (address user_address, uint256 _type) internal view returns (uint256
        uint256 martReleaseAmt = martSupply[userReward[user_address]._days];
        uint256 userToDayFee = SafeMathUpgradeable.mul(userReward[user_address].toDayFeeAmt, feeDecim
        uint256 todayTotalFee = totalFeeAmts[userReward[user_address]._days];
        rewardAmt = userReward[user_address].toBeReleased;
        invitReward = userReward[user_address].toBeReleasedByInvit;
        // According to the proportion of the handling fee in the user cycle, the reward that should
        if (todayTotalFee > 0 && martReleaseAmt > 0 && userToDayFee > 0) {
            uint256 _rate = SafeMathUpgradeable.div(userToDayFee, todayTotalFee);
            uint256 yestDayRewardAmtBig = SafeMathUpgradeable.mul(martReleaseAmt, _rate);
            uint256 yestDayRewardAmt = SafeMathUpgradeable.div(yestDayRewardAmtBig, feeDecimals);
            uint256 payInvitFee = SafeMathUpgradeable.div(SafeMathUpgradeable.mul(yestDayRewardAmt, 1
            if (_type == 2) {
                invitReward = payInvitFee;
            }
            invitFee = SafeMathUpgradeable.div(SafeMathUpgradeable.mul(yestDayRewardAmt, 10), 100);
            rewardAmt = SafeMathUpgradeable.add(rewardAmt, SafeMathUpgradeable.sub(yestDayRewardAmt,
        }
    }
    /// @dev Get the number of transaction mining and referrer rewards
    function rewardOf (address user_address) public view returns (uint256 userRewardAmt, uint256 invi
        (userRewardAmt, invitReward,) = calculateRewardAmt(user_address, 1);
    }
    /// @dev Get the number of rewards to be released
    function getReleaseLeft(address user_address) external view returns(uint256 left) {
        (uint256 userToBeReleased, uint256 invitReward,) = calculateRewardAmt(user_address, 1);
        uint256 total = SafeMathUpgradeable.add(userToBeReleased, invitReward);
        total = SafeMathUpgradeable.add(total, LockMap[user_address].releasedAmt);
        total = SafeMathUpgradeable.add(total, LockMap[user_address].left);
        uint256 released = extractableAmt(user_address);
        left = SafeMathUpgradeable.sub(total, released);
    }

    /// @dev Get the total transaction fee amount in a certain period
    function getTotalFeeAmt(uint256 _days) external view returns (uint256) {
        return totalFeeAmts[_days];
    }
    /// @dev Reward method for extracting transaction mining, Prevent filling
    function withdrawReward (address to) external nonReentrant isAllow returns (uint256) {
        require(to != address(0), 'MartTradeMining: to is zero address');
        // Get the total amount that can be withdrawn and the last release period
        if (martSupply[userReward[_msgSender()]._days] > 0) {
            clearUserFeeAmt(_msgSender());
        } else {
            updateLockMap(_msgSender(), true);
        }
        uint256 amount = LockMap[_msgSender()].releasedAmt;
        _releaseReward(_msgSender(), to, amount);
    }
    /// @dev Release and send rewards
    function _releaseReward (address addr, address _to, uint256 amt) internal {
        require(amt <= LockMap[addr].releasedAmt, 'MartTradeMining: Insufficient withdrawable balance
        LockMap[addr].releasedAmt = SafeMathUpgradeable.sub(LockMap[addr].releasedAmt, amt);
        // Send Reward
        require(IMartToken(martToken).getUsereward(_to, amt), 'MartTradeMining:get reward error');
        emit WithDrawReward(_msgSender(), _to, amt);
    }
    /// @dev  Get the amount of rewards that can be withdrawn
    function extractableAmt(address _addr) public view  returns (uint256 amount) {
        (uint256 amountBefore,,) = releaseAmountBefore(_addr);
        (uint256 amountCurrent,) = releaseAmountCurrent(_addr);
        amount = amount.add(amountBefore).add(amountCurrent).add(LockMap[_addr].releasedAmt);
    }
    /// @dev The received referral reward is added to the linear release
    function addInvitRewardLinearRelease(address _addr, uint256 _amount) internal {
        if (LockMap[_addr].left > 0) {
            updateLockMap(_addr, false);
        }
        uint256 singleQuantity = _amount.div(releaseTime);
        reduceLoopAmount[_addr][lastDay.add(releaseTime)] = singleQuantity;
        LockMap[_addr].left = LockMap[_addr].left.add(_amount);
        LockMap[_addr].singleQuantity = LockMap[_addr].singleQuantity.add(singleQuantity);
        LockMap[_addr].preReleaseLoop = lastDay;
    }
    /// @dev Calculate the number of rewards released through automatic settlement
    function releaseAmountBefore(address _addr) internal view returns (uint256, uint256, uint256) {
        require(_addr != address(0), 'MartTradeMining: _addr is zero address');
        (uint256 newReward,) = rewardOf(_addr);
        // Transaction mining reward must be greater than 0
        if (newReward == 0) return (0, 0, 0);
        uint256 totalAmount = newReward;
        // Allowed release cycle
        uint256 _loop = lastDay.sub(userReward[_addr]._days);
        // Release per cycle
        uint256 singleQuantity = totalAmount.div(releaseTime);
        // If the latest period is more than the period to be released from the trading period, then
        if (_loop >= releaseTime) return (totalAmount, newReward, singleQuantity);
        return (singleQuantity.mul(_loop), newReward, singleQuantity);
    }
    /// @dev Obtain the period to which the specified block belongs
    function getLoopFromBlock(uint256 _blockNumber) public view returns (uint256) {
        if (_blockNumber == 0) {
            return 0;
        }
        uint256 startBlock = IMartToken(martToken).startBlock();
        return _blockNumber.sub(startBlock).div(lockBlockNum).add(1);
    }
    /// @dev Calculate the number of releases in the LockMap
    function releaseAmountCurrent(address _addr) internal view returns (uint256 amount, uint256 singl
        require(_addr != address(0), 'MartTradeMining: _addr is zero address');
        LockUserData memory lockData = LockMap[_addr];
        if (lockData.left == 0) return (0, 0);
        // Allowed release cycle
        uint256 loop = lastDay.sub(lockData.preReleaseLoop);
        uint256 startLoop = lockData.preReleaseLoop;
        // Release per cycle
        singleQuantity = lockData.singleQuantity;
        // Compatible with the old linear release format
        if (lockData.preReleaseLoop == 0 && lockData.totalAmount > 0 && lockData.left > 0) {
            startLoop = getLoopFromBlock(lockData.preReleaseBlock);
            loop = lastDay.sub(startLoop);
            singleQuantity = lockData.totalAmount.div(releaseTime);
        }
        // If the latest period is more than the period to be released from the trading period, then
        if (loop >= releaseTime) return (lockData.left, singleQuantity);
        if (loop == 0) return (0, singleQuantity);
        // The maximum number of cycles is the release cycle-1
        for (uint256 i = startLoop.add(1); i <= lastDay; i++) {
            amount = amount.add(singleQuantity);
            // Cycle acquisition cycle number reduction
            singleQuantity = singleQuantity.sub(reduceLoopAmount[_addr][i]);
        }
        amount = amount > lockData.left ? lockData.left : amount;
    }
    /// @dev Update LockMap
    /**
    *@param _addr user address
    *@param isReset Whether to clear the automatically settled transaction mining rewards
    */
    function updateLockMap(address _addr, bool isReset) internal {
        (uint256 releasedBefore, uint256 newReward, uint256 sigleQuantityBefore)= releaseAmountBefore
        uint256 rReward = newReward.sub(releasedBefore);
        (uint256 releasedAmt, uint256 singleQuantityLast) = releaseAmountCurrent(_addr);

        if (isReset) {
            LockMap[_addr].releasedAmt = LockMap[_addr].releasedAmt.add(releasedBefore);
        }
        // If the user is the old linear release data format, replace it with the new one at the firs
        if (LockMap[_addr].totalAmount > 0 && LockMap[_addr].left > 0 && LockMap[_addr].preReleaseBlo
            LockUserData memory lockData = LockMap[_addr];
            if (lockData.left.sub(releasedAmt) == 0) {
                singleQuantityLast = 0;
            } else {
                uint256 rLoop = lockData.left.div(singleQuantityLast);
                if (rLoop.mul(singleQuantityLast) < lockData.left) {
                    rLoop = rLoop.add(1);
                }
                uint256 startLoop = getLoopFromBlock(lockData.preReleaseBlock);
                uint256 endLoop = startLoop.add(rLoop);
                reduceLoopAmount[_addr][endLoop] = singleQuantityLast;
                LockMap[_addr].totalAmount = 0;
                LockMap[_addr].preReleaseBlock = 0;
                LockMap[_addr].releaseLoop = 0;
            }
        }
        LockMap[_addr].releasedAmt = LockMap[_addr].releasedAmt.add(releasedAmt);
        LockMap[_addr].left = LockMap[_addr].left.sub(releasedAmt);
        LockMap[_addr].preReleaseLoop = lastDay;
        LockMap[_addr].singleQuantity =  singleQuantityLast;
        // reset
        if (rReward > 0 && isReset) {
            uint256 endLoop = userReward[_addr]._days.add(releaseTime);
            reduceLoopAmount[_addr][endLoop] = sigleQuantityBefore;
            LockMap[_addr].left = LockMap[_addr].left.add(rReward);
            LockMap[_addr].singleQuantity = LockMap[_addr].singleQuantity.add(sigleQuantityBefore);
        }
        // clear
        if (LockMap[_addr].left == 0) {
            LockMap[_addr].preReleaseLoop = 0;
            LockMap[_addr].singleQuantity = 0;
            // Empty old fields
            if (LockMap[_addr].totalAmount > 0) {
                LockMap[_addr].totalAmount = 0;
            }
            if (LockMap[_addr].preReleaseBlock > 0) {
                LockMap[_addr].preReleaseBlock = 0;
            }
        }
    }
}
```

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 vulnerabilities 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