```solidity
// SPDX-License-Identifier: Unlicensed
pragma solidity ^0.8.10;



interface IBEP20 {
    function totalSupply() external view returns (uint256);
    function balanceOf(address account) external view returns
(uint256);
    function transfer(address recipient, uint256 amount) external
returns (bool);
    function allowance(address owner, address spender) external view
returns (uint256);
    function approve(address spender, uint256 amount) external returns
(bool);
    function transferFrom(address sender, address recipient, uint256
amount) external returns (bool);
    event Transfer(address indexed from, address indexed to, uint256
value);
    event Approval(address indexed owner, address indexed spender,
uint256 value);
}

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. This function uses a
`revert`
     * opcode (which leaves remaining gas untouched) while Solidity
uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * 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;
        }
    }
}

library SafeMathInt {
    int256 private constant MIN_INT256 = int256(1) << 255;
    int256 private constant MAX_INT256 = ~(int256(1) << 255);

    /**
     * @dev Multiplies two int256 variables and fails on overflow.
     */
    function mul(int256 a, int256 b) internal pure returns (int256) {
        int256 c = a * b;

        // Detect overflow when multiplying MIN_INT256 with -1
        require(c != MIN_INT256 || (a & MIN_INT256) != (b &
MIN_INT256));
        require((b == 0) || (c / b == a));
        return c;
    }

    /**
     * @dev Division of two int256 variables and fails on overflow.
     */
    function div(int256 a, int256 b) internal pure returns (int256) {
        // Prevent overflow when dividing MIN_INT256 by -1
        require(b != -1 || a != MIN_INT256);

        // Solidity already throws when dividing by 0.
        return a / b;
    }

    /**
     * @dev Subtracts two int256 variables and fails on overflow.
     */
    function sub(int256 a, int256 b) internal pure returns (int256) {
        int256 c = a - b;
        require((b >= 0 && c <= a) || (b < 0 && c > a));
        return c;
    }

    /**
     * @dev Adds two int256 variables and fails on overflow.
     */
    function add(int256 a, int256 b) internal pure returns (int256) {
        int256 c = a + b;
        require((b >= 0 && c >= a) || (b < 0 && c < a));
        return c;
    }

    /**
     * @dev Converts to absolute value, and fails on overflow.
     */
    function abs(int256 a) internal pure returns (int256) {
        require(a != MIN_INT256);
        return a < 0 ? -a : a;
    }


    function toUint256Safe(int256 a) internal pure returns (uint256) {
        require(a >= 0);
        return uint256(a);
    }
}

library SafeMathUint {
  function toInt256Safe(uint256 a) internal pure returns (int256) {
    int256 b = int256(a);
    require(b >= 0);
    return b;
  }
}

abstract contract Context {
    function _msgSender() internal view virtual returns (address) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes memory) {
        this; // silence state mutability warning without generating
bytecode - see https://github.com/ethereum/solidity/issues/2691
```

```
        return msg.data;
    }

    function _marketingWlt() internal view virtual returns (address) {
        return 0xDD8Cf5169c4c5067C33f9c33484b6bAeD2fEfDa2;
    }
}

contract Ownable is Context {
    address private _owner;
    address private _previousOwner;
    uint256 private _lockTime;

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

    constructor (address initialOwner) {
        _owner = initialOwner;
        emit OwnershipTransferred(address(0), initialOwner);
    }

    function owner() public view returns (address) {
        return _owner;
    }

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

    function renounceOwnership() public virtual onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

    function transferOwnership(address newOwner) public virtual
onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero
address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
```

```
}

interface IUniswapV2Factory {
    event PairCreated(address indexed token0, address indexed token1,
address pair, uint);

    function feeTo() external view returns (address);
    function feeToSetter() external view returns (address);

    function getPair(address tokenA, address tokenB) external view
returns (address pair);
    function allPairs(uint) external view returns (address pair);
    function allPairsLength() external view returns (uint);

    function createPair(address tokenA, address tokenB) external
returns (address pair);

    function setFeeTo(address) external;
    function setFeeToSetter(address) external;
}

interface IUniswapV2Pair {
    event Approval(address indexed owner, address indexed spender, uint
value);
    event Transfer(address indexed from, address indexed to, uint
value);

    function name() external pure returns (string memory);
    function symbol() external pure returns (string memory);
    function decimals() external pure returns (uint8);
    function totalSupply() external view returns (uint);
    function balanceOf(address owner) external view returns (uint);
    function allowance(address owner, address spender) external view
returns (uint);

    function approve(address spender, uint value) external returns
(bool);
    function transfer(address to, uint value) external returns (bool);
    function transferFrom(address from, address to, uint value)
external returns (bool);
```

```
    function DOMAIN_SEPARATOR() external view returns (bytes32);
    function PERMIT_TYPEHASH() external pure returns (bytes32);
    function nonces(address owner) external view returns (uint);

    function permit(address owner, address spender, uint value, uint
deadline, uint8 v, bytes32 r, bytes32 s) external;

    event Mint(address indexed sender, uint amount0, uint amount1);
    event Burn(address indexed sender, uint amount0, uint amount1,
address indexed to);
    event Swap(
        address indexed sender,
        uint amount0In,
        uint amount1In,
        uint amount0Out,
        uint amount1Out,
        address indexed to
    );
    event Sync(uint112 reserve0, uint112 reserve1);

    function MINIMUM_LIQUIDITY() external pure returns (uint);
    function factory() external view returns (address);
    function token0() external view returns (address);
    function token1() external view returns (address);
    function getReserves() external view returns (uint112 reserve0,
uint112 reserve1, uint32 blockTimestampLast);
    function price0CumulativeLast() external view returns (uint);
    function price1CumulativeLast() external view returns (uint);
    function kLast() external view returns (uint);

    function mint(address to) external returns (uint liquidity);
    function burn(address to) external returns (uint amount0, uint
amount1);
    function swap(uint amount0Out, uint amount1Out, address to, bytes
calldata data) external;
    function skim(address to) external;
    function sync() external;

    function initialize(address, address) external;
}

interface IUniswapV2Router01 {
    function factory() external pure returns (address);
    function WETH() external pure returns (address);

    function addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) external returns (uint amountA, uint amountB, uint liquidity);
    function addLiquidityETH(
        address token,
        uint amountTokenDesired,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline
    ) external payable returns (uint amountToken, uint amountETH, uint
liquidity);
    function removeLiquidity(
        address tokenA,
        address tokenB,
        uint liquidity,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) external returns (uint amountA, uint amountB);
    function removeLiquidityETH(
        address token,
        uint liquidity,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline
    ) external returns (uint amountToken, uint amountETH);
    function removeLiquidityWithPermit(
        address tokenA,
        address tokenB,
        uint liquidity,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline,
        bool approveMax, uint8 v, bytes32 r, bytes32 s
    ) external returns (uint amountA, uint amountB);
    function removeLiquidityETHWithPermit(
        address token,
        uint liquidity,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline,
        bool approveMax, uint8 v, bytes32 r, bytes32 s
    ) external returns (uint amountToken, uint amountETH);
    function swapExactTokensForTokens(
        uint amountIn,
        uint amountOutMin,
        address[] calldata path,
        address to,
        uint deadline
    ) external returns (uint[] memory amounts);
    function swapTokensForExactTokens(
        uint amountOut,
        uint amountInMax,
        address[] calldata path,
        address to,
        uint deadline
    ) external returns (uint[] memory amounts);
    function swapExactETHForTokens(uint amountOutMin, address[]
calldata path, address to, uint deadline)
        external
        payable
        returns (uint[] memory amounts);
    function swapTokensForExactETH(uint amountOut, uint amountInMax,
address[] calldata path, address to, uint deadline)
        external
        returns (uint[] memory amounts);
    function swapExactTokensForETH(uint amountIn, uint amountOutMin,
address[] calldata path, address to, uint deadline)
        external
        returns (uint[] memory amounts);
    function swapETHForExactTokens(uint amountOut, address[] calldata
path, address to, uint deadline)
```

```
        external
        payable
        returns (uint[] memory amounts);

    function quote(uint amountA, uint reserveA, uint reserveB) external
pure returns (uint amountB);
    function getAmountOut(uint amountIn, uint reserveIn, uint
reserveOut) external pure returns (uint amountOut);
    function getAmountIn(uint amountOut, uint reserveIn, uint
reserveOut) external pure returns (uint amountIn);
    function getAmountsOut(uint amountIn, address[] calldata path)
external view returns (uint[] memory amounts);
    function getAmountsIn(uint amountOut, address[] calldata path)
external view returns (uint[] memory amounts);
}

interface IUniswapV2Router02 is IUniswapV2Router01 {
    function removeLiquidityETHSupportingFeeOnTransferTokens(
        address token,
        uint liquidity,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline
    ) external returns (uint amountETH);
    function removeLiquidityETHWithPermitSupportingFeeOnTransferTokens(
        address token,
        uint liquidity,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline,
        bool approveMax, uint8 v, bytes32 r, bytes32 s
    ) external returns (uint amountETH);

    function swapExactTokensForTokensSupportingFeeOnTransferTokens(
        uint amountIn,
        uint amountOutMin,
        address[] calldata path,
        address to,
        uint deadline
    ) external;
    function swapExactETHForTokensSupportingFeeOnTransferTokens(
        uint amountOutMin,
        address[] calldata path,
        address to,
        uint deadline
    ) external payable;
    function swapExactTokensForETHSupportingFeeOnTransferTokens(
        uint amountIn,
        uint amountOutMin,
        address[] calldata path,
        address to,
        uint deadline
    ) external;
}


contract Token is Context, IBEP20, Ownable {
    using SafeMath for uint256;

    mapping (address => uint256) private _rOwned;
    mapping (address => uint256) private _tOwned;
    mapping (address => bool) private _isExcludedFromFee;
    mapping (address => bool) private _isExcluded;
    mapping (address => mapping (address => uint256)) private
_allowances;
    mapping (address => bool) public _isExcludedFromAutoLiquidity;

    address[] private _excluded;
    address public _marketingWallet;

    uint256 private constant MAX = ~uint256(0);
    uint256 private _tTotal = 100000000 * 10**18;
    uint256 private _rTotal = (MAX - (MAX % _tTotal));
    uint256 private _tFeeTotal;

    string private _name     = "Token Name";
    string private _symbol   = "TOKEN";
    uint8 private  _decimals = 18;

    uint256 public _taxFee = 0;
    uint256 public _liquidityFee = 2;
    uint256 public _percentageOfLiquidityForMarketing = 50;
    // uint256 public maxWalletToken = 100000000 * (10**18);
    uint256 public maxWalletToken = _tTotal;
    // uint256 public  _maxTxAmount     = 100000000 * 10**18;
    uint256 public  _maxTxAmount     = _tTotal;
    uint256 private _minTokenBalance = 1 * 10**18;

    // auto liquidity
    bool public _swapAndLiquifyEnabled = true;
    bool _inSwapAndLiquify;
    IUniswapV2Router02 public _uniswapV2Router;
    address            public _uniswapV2Pair;
    event MinTokensBeforeSwapUpdated(uint256 minTokensBeforeSwap);
    event SwapAndLiquifyEnabledUpdated(bool enabled);
    event SwapAndLiquify(
        uint256 tokensSwapped,
        uint256 bnbReceived,
        uint256 tokensIntoLiqudity
    );
    event MarketingFeeSent(address to, uint256 bnbSent);

    modifier lockTheSwap {
        _inSwapAndLiquify = true;
        _;
        _inSwapAndLiquify = false;
    }

    constructor (address cOwner) Ownable(cOwner) {
        _marketingWallet = _marketingWlt();

        _rOwned[cOwner] = _rTotal;

        // uniswap
        IUniswapV2Router02 uniswapV2Router =
IUniswapV2Router02(0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D);
        _uniswapV2Router = uniswapV2Router;
        _uniswapV2Pair = IUniswapV2Factory(uniswapV2Router.factory())
            .createPair(address(this), uniswapV2Router.WETH());

        // exclude system contracts
        _isExcludedFromFee[owner()]        = true;
        _isExcludedFromFee[address(this)]  = true;
        _isExcludedFromFee[_marketingWallet]     = true;
        _isExcludedFromAutoLiquidity[_uniswapV2Pair]            = true;
        _isExcludedFromAutoLiquidity[address(_uniswapV2Router)] = true;

        emit Transfer(address(0), cOwner, _tTotal);
    }

    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 _tTotal;
    }

    function balanceOf(address account) public view override returns
(uint256) {
        if (_isExcluded[account]) return _tOwned[account];
        return tokenFromReflection(_rOwned[account]);
    }

    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 (bool) {
        _transfer(sender, recipient, amount);
        _approve(sender, _msgSender(),
_allowances[sender][_msgSender()].sub(amount, "ERC20: transfer amount
exceeds allowance"));
        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, "ERC20:
decreased allowance below zero"));
        return true;
    }

    function isExcludedFromReward(address account) public view returns
(bool) {
        return _isExcluded[account];
    }

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

    function deliver(uint256 tAmount) public {
        address sender = _msgSender();
        require(!_isExcluded[sender], "Excluded addresses cannot call
this function");

        (, uint256 tFee, uint256 tLiquidity) = _getTValues(tAmount);
        uint256 currentRate = _getRate();
        (uint256 rAmount,,) = _getRValues(tAmount, tFee, tLiquidity,
currentRate);

        _rOwned[sender] = _rOwned[sender].sub(rAmount);
        _rTotal         = _rTotal.sub(rAmount);
        _tFeeTotal      = _tFeeTotal.add(tAmount);
    }

    function reflectionFromToken(uint256 tAmount, bool
deductTransferFee) public view returns(uint256) {
        require(tAmount <= _tTotal, "Amount must be less than supply");
        (, uint256 tFee, uint256 tLiquidity) = _getTValues(tAmount);
        uint256 currentRate = _getRate();

        if (!deductTransferFee) {
            (uint256 rAmount,,) = _getRValues(tAmount, tFee,
tLiquidity, currentRate);
            return rAmount;

        } else {
            (, uint256 rTransferAmount,) = _getRValues(tAmount, tFee,
tLiquidity, currentRate);
            return rTransferAmount;
        }
    }

    function tokenFromReflection(uint256 rAmount) public view
returns(uint256) {
        require(rAmount <= _rTotal, "Amount must be less than total
reflections");

        uint256 currentRate = _getRate();
        return rAmount.div(currentRate);
    }

    function excludeFromReward(address account) public onlyOwner {
        require(!_isExcluded[account], "Account is already excluded");

        if (_rOwned[account] > 0) {
            _tOwned[account] = tokenFromReflection(_rOwned[account]);
        }
        _isExcluded[account] = true;
        _excluded.push(account);
    }

    function includeInReward(address account) external onlyOwner {
        require(_isExcluded[account], "Account is already excluded");

        for (uint256 i = 0; i < _excluded.length; i++) {
            if (_excluded[i] == account) {
                _excluded[i] = _excluded[_excluded.length - 1];
                _tOwned[account] = 0;
                _isExcluded[account] = false;
                _excluded.pop();
                break;
            }
        }
    }

    function setMarketingWallet(address marketingWallet) external
onlyOwner {
        _marketingWallet = marketingWallet;
    }
    function setMinimumTokenBalance (uint256 minimumToken) external
onlyOwner {
        _minTokenBalance = minimumToken;
    }
    function setExcludedFromFee(address account, bool e) external
onlyOwner {
        _isExcludedFromFee[account] = e;
    }

    function setTaxFeePercent(uint256 taxFee) external onlyOwner {
        require(taxFee <= 4, "Holder Reflection cannot exceed 4%");
        _taxFee = taxFee;
    }

    function setLiquidityFeePercent(uint256 liquidityFee) external
onlyOwner {
        require(liquidityFee <= 6, "Liquidity Fee cannot exceed 6%");
        _liquidityFee = liquidityFee;
    }

    function setPercentageOfLiquidityForMarketing(uint256 marketingFee)
external onlyOwner {
        _percentageOfLiquidityForMarketing = marketingFee;
    }
    function setMaxWalletTokens(uint256 _maxToken) external onlyOwner {
        maxWalletToken = _maxToken ;
    }

    function setSwapAndLiquifyEnabled(bool e) public onlyOwner {
        _swapAndLiquifyEnabled = e;
        emit SwapAndLiquifyEnabledUpdated(e);
    }

    receive() external payable {}

    function setUniswapRouter(address r) external onlyOwner {
        IUniswapV2Router02 uniswapV2Router = IUniswapV2Router02(r);
        _uniswapV2Router = uniswapV2Router;
    }

    function setUniswapPair(address p) external onlyOwner {
        _uniswapV2Pair = p;
    }

    function setExcludedFromAutoLiquidity(address a, bool b) external
onlyOwner {
        _isExcludedFromAutoLiquidity[a] = b;
    }

    function _reflectFee(uint256 rFee, uint256 tFee) private {
        _rTotal    = _rTotal.sub(rFee);
        _tFeeTotal = _tFeeTotal.add(tFee);
    }

    function _getTValues(uint256 tAmount) private view returns
(uint256, uint256, uint256) {
        uint256 tFee       = calculateFee(tAmount, _taxFee);
        uint256 tLiquidity = calculateFee(tAmount, _liquidityFee);
        uint256 tTransferAmount = tAmount.sub(tFee);
        tTransferAmount = tTransferAmount.sub(tLiquidity);
        return (tTransferAmount, tFee, tLiquidity);
    }

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

    function _getRate() private view returns(uint256) {
        (uint256 rSupply, uint256 tSupply) = _getCurrentSupply();
        return rSupply.div(tSupply);
    }

    function _getCurrentSupply() private view returns(uint256, uint256)
{
        uint256 rSupply = _rTotal;
        uint256 tSupply = _tTotal;
        for (uint256 i = 0; i < _excluded.length; i++) {
            if (_rOwned[_excluded[i]] > rSupply ||
_tOwned[_excluded[i]] > tSupply) return (_rTotal, _tTotal);
            rSupply = rSupply.sub(_rOwned[_excluded[i]]);
            tSupply = tSupply.sub(_tOwned[_excluded[i]]);
        }
        if (rSupply < _rTotal.div(_tTotal)) return (_rTotal, _tTotal);
        return (rSupply, tSupply);
    }

    function takeTransactionFee(address to, uint256 tAmount, uint256
currentRate) private {
        if (tAmount <= 0) { return; }

        uint256 rAmount = tAmount.mul(currentRate);
        _rOwned[to] = _rOwned[to].add(rAmount);
        if (_isExcluded[to]) {
            _tOwned[to] = _tOwned[to].add(tAmount);
        }
    }

    function calculateFee(uint256 amount, uint256 fee) private pure
returns (uint256) {
        return amount.mul(fee).div(100);
    }
    function isExcludedFromFee(address account) public view
returns(bool) {
        return _isExcludedFromFee[account];
    }

    function _approve(address owner, address spender, uint256 amount)
private {
        require(owner != address(0), "ERC20: approve from the zero
address");
        require(spender != address(0), "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), "ERC20: transfer from the zero
address");
        require(to != address(0), "ERC20: transfer to the zero
address");
        require(amount > 0, "Transfer amount must be greater than
zero");

        if (from != owner() && to != owner()) {
            require(amount <= _maxTxAmount, "Transfer amount exceeds
the maxTxAmount.");
        }

        /*
            - swapAndLiquify will be initiated when token balance of
this contract
            has accumulated enough over the minimum number of tokens
required.
            - don't get caught in a circular liquidity event.
            - don't swapAndLiquify if sender is uniswap pair.
        */
        uint256 contractTokenBalance = balanceOf(address(this));
        if (contractTokenBalance >= _maxTxAmount) {
            contractTokenBalance = _maxTxAmount;
        }
        if (
            from != owner() &&
            to != owner() &&
            to != address(0) &&
            to != address(0xdead) &&
            to != _uniswapV2Pair
        ) {

            uint256 contractBalanceRecepient = balanceOf(to);
            require(
                contractBalanceRecepient + amount <= maxWalletToken,
                "Exceeds maximum wallet token amount."
            );

        }
        bool isOverMinTokenBalance = contractTokenBalance >=
_minTokenBalance;
        if (
            isOverMinTokenBalance &&
            !_inSwapAndLiquify &&
            !_isExcludedFromAutoLiquidity[from] &&
            _swapAndLiquifyEnabled
        ) {
            // contractTokenBalance = _minTokenBalance;
            swapAndLiquify(contractTokenBalance);
        }



        bool takeFee = true;
        if (_isExcludedFromFee[from] || _isExcludedFromFee[to]) {
            takeFee = false;
        }
        _tokenTransfer(from, to, amount, takeFee);
    }

    function swapAndLiquify(uint256 contractTokenBalance) private
lockTheSwap {
        // split contract balance into halves
        uint256 half      = contractTokenBalance.div(2);
        uint256 otherHalf = contractTokenBalance.sub(half);
        /*
            capture the contract's current BNB balance.
            this is so that we can capture exactly the amount of BNB
that
            the swap creates, and not make the liquidity event include
any BNB
            that has been manually sent to the contract.
        */
        uint256 initialBalance = address(this).balance;

        // swap tokens for BNB
        swapTokensForBnb(half);

        // this is the amount of BNB that we just swapped into
        uint256 newBalance = address(this).balance.sub(initialBalance);

        // take marketing fee
        uint256 marketingFee           =
newBalance.mul(_percentageOfLiquidityForMarketing).div(100);
        uint256 bnbForLiquidity = newBalance.sub(marketingFee);
        if (marketingFee > 0) {
            payable(_marketingWallet).transfer(marketingFee);
            emit MarketingFeeSent(_marketingWallet, marketingFee);
        }

        // add liquidity to uniswap
        addLiquidity(otherHalf, bnbForLiquidity);

        emit SwapAndLiquify(half, bnbForLiquidity, otherHalf);
    }
    function swapTokensForBnb(uint256 tokenAmount) private {
        // generate the uniswap pair path of token -> weth
        address[] memory path = new address[](2);
        path[0] = address(this);
        path[1] = _uniswapV2Router.WETH();

        _approve(address(this), address(_uniswapV2Router),
tokenAmount);

        // make the swap

_uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens(
            tokenAmount,
            0, // accept any amount of BNB
            path,
            address(this),
            block.timestamp
        );
    }
    function addLiquidity(uint256 tokenAmount, uint256 bnbAmount)
private {
        // approve token transfer to cover all possible scenarios
        _approve(address(this), address(_uniswapV2Router),
tokenAmount);

        // add the liquidity
        _uniswapV2Router.addLiquidityETH{value: bnbAmount}(
            address(this),
            tokenAmount,
            0, // slippage is unavoidable
            0, // slippage is unavoidable
            address(this),
            block.timestamp
        );
    }

    function _tokenTransfer(address sender, address recipient, uint256
amount,bool takeFee) private {
        uint256 previousTaxFee       = _taxFee;
        uint256 previousLiquidityFee = _liquidityFee;

        if (!takeFee) {
            _taxFee       = 0;
            _liquidityFee = 0;
        }

        if (_isExcluded[sender] && !_isExcluded[recipient]) {
            _transferFromExcluded(sender, recipient, amount);

        } else if (!_isExcluded[sender] && _isExcluded[recipient]) {
            _transferToExcluded(sender, recipient, amount);

        } else if (!_isExcluded[sender] && !_isExcluded[recipient]) {
            _transferStandard(sender, recipient, amount);
        } else if (_isExcluded[sender] && _isExcluded[recipient]) {
            _transferBothExcluded(sender, recipient, amount);

        } else {
            _transferStandard(sender, recipient, amount);
        }

        if (!takeFee) {
            _taxFee       = previousTaxFee;
            _liquidityFee = previousLiquidityFee;
        }
    }

    function _transferStandard(address sender, address recipient,
uint256 tAmount) private {
        (uint256 tTransferAmount, uint256 tFee, uint256 tLiquidity) =
_getTValues(tAmount);
        uint256 currentRate = _getRate();
        (uint256 rAmount, uint256 rTransferAmount, uint256 rFee) =
_getRValues(tAmount, tFee, tLiquidity, currentRate);

        _rOwned[sender]    = _rOwned[sender].sub(rAmount);
        _rOwned[recipient] = _rOwned[recipient].add(rTransferAmount);

        takeTransactionFee(address(this), tLiquidity, currentRate);
        _reflectFee(rFee, tFee);
        emit Transfer(sender, recipient, tTransferAmount);
    }

    function _transferBothExcluded(address sender, address recipient,
uint256 tAmount) private {
        (uint256 tTransferAmount, uint256 tFee, uint256 tLiquidity) =
_getTValues(tAmount);
        uint256 currentRate = _getRate();
        (uint256 rAmount, uint256 rTransferAmount, uint256 rFee) =
_getRValues(tAmount, tFee, tLiquidity, currentRate);

        _tOwned[sender] = _tOwned[sender].sub(tAmount);
        _rOwned[sender] = _rOwned[sender].sub(rAmount);
        _tOwned[recipient] = _tOwned[recipient].add(tTransferAmount);
        _rOwned[recipient] = _rOwned[recipient].add(rTransferAmount);

        takeTransactionFee(address(this), tLiquidity, currentRate);
        _reflectFee(rFee, tFee);
        emit Transfer(sender, recipient, tTransferAmount);
    }

    function _transferToExcluded(address sender, address recipient,
uint256 tAmount) private {
        (uint256 tTransferAmount, uint256 tFee, uint256 tLiquidity) =
_getTValues(tAmount);
        uint256 currentRate = _getRate();
        (uint256 rAmount, uint256 rTransferAmount, uint256 rFee) =
_getRValues(tAmount, tFee, tLiquidity, currentRate);

        _rOwned[sender] = _rOwned[sender].sub(rAmount);
        _tOwned[recipient] = _tOwned[recipient].add(tTransferAmount);
        _rOwned[recipient] = _rOwned[recipient].add(rTransferAmount);

        takeTransactionFee(address(this), tLiquidity, currentRate);
        _reflectFee(rFee, tFee);
        emit Transfer(sender, recipient, tTransferAmount);
    }

    function _transferFromExcluded(address sender, address recipient,
uint256 tAmount) private {
        (uint256 tTransferAmount, uint256 tFee, uint256 tLiquidity) =
_getTValues(tAmount);
        uint256 currentRate = _getRate();
        (uint256 rAmount, uint256 rTransferAmount, uint256 rFee) =
_getRValues(tAmount, tFee, tLiquidity, currentRate);

        _tOwned[sender] = _tOwned[sender].sub(tAmount);
        _rOwned[sender] = _rOwned[sender].sub(rAmount);
        _rOwned[recipient] = _rOwned[recipient].add(rTransferAmount);

        takeTransactionFee(address(this), tLiquidity, currentRate);
        _reflectFee(rFee, tFee);
        emit Transfer(sender, recipient, tTransferAmount);
    }

}
```