BEOSIN

Blockchain Security

Smart contract security audit report

Audit Number: 202107051521

Smart Contract Name:

POLKER (PKR)

Smart Contract Address:

BSC: 0xc49DDe62B4A0810074721fAcA54Aab52369f486a

ETH: 0x001A8Ffcb0f03e99141652eBCdecDb0384E3bd6c

TRON: TQDNhvRhqKYLLPLqD6S8mjx7DratAK3nq6

Smart Contract Address Link:

BSC: https://bscscan.com/address/0xc49DDe62B4A0810074721fAcA54Aab52369f486a#code

ETH: https://etherscan.io/address/0x001A8Ffcb0f03e99141652eBCdecDb0384E3bd6c#code

TRON: https://tronscan.org/#/contract/TQDNhvRhqKYLLPLqD6S8mjx7DratAK3nq6/code

Start Date: 2021.07.01

Completion Date: 2021.07.05

Overall Result: Pass (Distinction)

Audit Team: Beosin (Chengdu LianAn) Technology Co. Ltd.

Audit Categories and Results:

No.	Categories	Subitems	Results
1	Coding Conventions	BEP-20 Token Standards	Pass
		Compiler Version Security	Pass
		Visibility Specifiers	Pass
		Gas Consumption	Pass
		SafeMath Features	Pass
		Fallback Usage	Pass
		tx.origin Usage	Pass
		Deprecated Items	Pass
		Redundant Code	Pass
		Overriding Variables	Pass
2	Function Call Audit	Authorization of Function Call	Pass
		Low-level Function (call/delegatecall) Security	Pass

		Returned Value Security	Pass
		selfdestruct Function Security	Pass
3	Business Security	Access Control of Owner	Pass
		Business Logics	Pass
		Business Implementations	Pass
4	Integer Overflow/Underflow	-	Pass
5	Reentrancy	-	Pass
6	Exceptional Reachable State	-	Pass
7	Transaction-Ordering Dependence	-	Pass
8	Block Properties Dependence	-	Pass
9	Pseudo-random Number Generator (PRNG)	-	Pass
10	DoS (Denial of Service)	-	Pass
11	Token Vesting Implementation	-	N/A
12	Fake Deposit	-	Pass
13	event security	-	Pass

Disclaimer: This report is made in response to the project code. No description, expression or wording in this report shall be construed as an endorsement, affirmation or confirmation of the project. This audit is only applied to the type of auditing specified in this report and the scope of given in the results table. Other unknown security vulnerabilities are beyond auditing responsibility. Beosin (Chengdu LianAn) Technology only issues this report based on the attacks or vulnerabilities that already existed or occurred before the issuance of this report. For the emergence of new attacks or vulnerabilities that exist or occur in the future, Beosin (Chengdu LianAn) Technology lacks the capability to judge its possible impact on the security status of smart contracts, thus taking no responsibility for them. The security audit analysis and other contents of this report are based solely on the documents and materials that the contract provider has provided to Beosin (Chengdu LianAn) Technology before the issuance of this report, and the contract provider warrants that there are no missing, tampered, deleted; if the document ts and materials provided by the contract provider are missing, tampered, deleted, concealed or reflected in a situation that is inconsistent with the actual situation, or if the documents and materials provided are changed after the issuance of this report, Beosin (Chengdu LianAn) Technology assumes no responsibility for the resulting loss or adverse effects. The audit report issued by Beosin (Chengdu LianAn) Technology is based on the documents and materials provided by the contract provider, and relies on the technology currently possessed by Beosin (Chengdu LianAn). Due to the technical limitations of any organization, this report conducted by Beosin (Chengdu LianAn) still has the possibility that the entire risk cannot be completely detected. Beosin (Chengdu LianAn) disclaims any liability for the resulting losses.

The final interpretation of this statement belongs to Beosin (Chengdu LianAn).

Audit Results Explained:

Beosin (Chengdu LianAn) Technology has used several methods including Formal Verification, Static Analysis, Typical Case Testing and Manual Review to audit three major aspects of smart contract PKR, including Coding Standards, Security, and Business Logic. **PKR contract passed all audit items. The overall result is Pass (Distinction). The smart contract is able to function properly.** Please find below the basic information of the smart contract:

1. Basic Token Information

Token name	POLKER
Token symbol	PKR
Token decimals	18
Token total supply	1 billion
Token type	BEP-20;TRC20

Table 1 – Basic Token Information in BSC/TRON

Token name	POLKER
Token symbol	PKR
Token decimals	18
Token total supply	Initial 0, mintable
Token type	ERC20

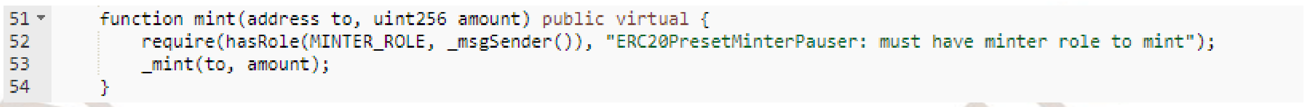
Table 2 – Basic Token Information in ETH

2. Token Vesting Information

N/A

3. Other Functions Description

On ETH, the PKR token uses a different contract from TRON and BSC. The main difference is that the LGEWhitelisted function has been removed, and the suspension and minting functions have been added.



```
function mint(address to, uint256 amount) public virtual {
    require(hasRole(MINTER_ROLE, _msgSender()), "ERC20PresetMinterPauser: must have minter role to mint");
    _mint(to, amount);
}
```

Figure 1 source code of function *mint* in ETH

Audited Source Code with Comments:

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

pragma solidity 0.5.16;

// Beosin (Chengdu LianAn) // Define TRC20 standard interface.
interface ITRC20 {
    /**
      * @dev Returns the amount of tokens in existence.
      */
    function totalSupply() external view returns (uint256);

    /**
      * @dev Returns the token decimals.
      */
    function decimals() external view returns (uint8);

    /**
      * @dev Returns the token symbol.
      */
    function symbol() external view returns (string memory);

    /**
    * @dev Returns the token name.
    */
    function name() external view returns (string memory);

    /**
      * @dev Returns the bep token owner.
      */
    function getOwner() external view returns (address);

    /**
      * @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);
}

/*
  * @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.
  */
contract Context {
    // Empty internal constructor, to prevent people from mistakenly deploying
    // an instance of this contract, which should be used via inheritance.
    constructor () internal { }

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

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

/**
  * @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.
  */
// Beosin (Chengdu LianAn) // The SafeMath library declares these functions for safe mathematical
```

operations.

```
library SafeMath {
    /**
     * @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) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
     * overflow (when the result is negative).
     *
     * 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);
        uint256 c = a - b;

        return c;
    }
```

```
/**
 * @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) {
    // 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 0;
    }

    uint256 c = a * b;
    require(c / a == b, "SafeMath: multiplication overflow");

    return c;
}

/**
 * @dev Returns the integer division of two unsigned integers. Reverts 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) {
    return div(a, b, "SafeMath: division by zero");
}

/**
 * @dev Returns the integer division of two unsigned integers. Reverts 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) {
        // Solidity only automatically asserts when dividing by 0
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts 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 mod(a, b, "SafeMath: modulo by zero");
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts with custom message 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, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }
}

/**
```

```
 * @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.
 */
contract Ownable is Context {
    address private _owner; // Beosin (Chengdu LianAn) // Declare variable '_owner' for storing the
contract owner.

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner); // Beosin
(Chengdu LianAn) // Declare the event 'OwnershipTransferred'.

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

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

    /**
     * @dev Throws if called by any account other than the owner.
     */
    // Beosin (Chengdu LianAn) // Modifier, require that the caller of the modified function must be
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() external 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) external onlyOwner {
    _transferOwnership(newOwner);
  }

  /**
    * @dev Transfers ownership of the contract to a new account (`newOwner`).
    */
  function _transferOwnership(address newOwner) internal {
    require(newOwner != address(0), "Ownable: new owner is the zero address"); // Beosin (Chengdu LianAn) // The non-zero address check for 'newOwner'.
    emit OwnershipTransferred(_owner, newOwner); // Beosin (Chengdu LianAn) // Trigger the event 'OwnershipTransferred'.
    _owner = newOwner;
  }
}

contract LGEWhitelisted is Context {

    using SafeMath for uint256; // Beosin (Chengdu LianAn) // Using the SafeMath library for mathematical operation. Avoid integer overflow.

    struct WhitelistRound { // Beosin (Chengdu LianAn) // Declare the WhitelistRound structure, which is used to store information about a stage whitelist.
        uint256 duration; // Beosin (Chengdu LianAn) // Declare the duration variable to store the duration of the phase.
        uint256 amountMax; // Beosin (Chengdu LianAn) // Declare the amountMax variable, which is used to store the maximum amount of tokens purchased by each address in this phase.
        mapping(address => bool) addresses; // Beosin (Chengdu LianAn) // Declare addresses mapping, used to store whether the address belongs to the whitelist.
        mapping(address => uint256) purchased; // Beosin (Chengdu LianAn) // Declare purchased mapping, used to store the number of tokens purchased by the address.
    }
    WhitelistRound[] public _lgeWhitelistRounds; // Beosin (Chengdu LianAn) // Declare the
_lgeWhitelistRounds array to store all WhitelistRound information.

    uint256 public _lgeTimestamp; // Beosin (Chengdu LianAn) // Declare the _lgeTimestamp variable
to store the start time of the purchase.
    address public _lgePairAddress; // Beosin (Chengdu LianAn) // Declare the _lgePairAddress
variable, which is used to store the transaction pair address.

    address public _whitelister; // Beosin (Chengdu LianAn) // Declare the _whitelister variable, which is
used to store the administrator authority address for setting the whitelist.

    event WhitelisterTransferred(address indexed previousWhitelister, address indexed newWhitelister); //
Beosin (Chengdu LianAn) // Declare the WhitelisterTransferred event.

    constructor () internal {
        _whitelister = _msgSender(); // Beosin (Chengdu LianAn) // Constructor, initially set
msg.sender as whitelist management authority.
    }

    modifier onlyWhitelister() { // Beosin (Chengdu LianAn) // Modifier, only _whitelister address can
call.
        require(_whitelister == _msgSender(), "Caller is not the whitelister");
        _;
    }
    // Beosin (Chengdu LianAn) // Renounce whitelist management authority.
    function renounceWhitelister() external onlyWhitelister {
        emit WhitelisterTransferred(_whitelister, address(0));
        _whitelister = address(0);
    }

    function transferWhitelister(address newWhitelister) external onlyWhitelister {
        _transferWhitelister(newWhitelister); // Beosin (Chengdu LianAn) // Call internal function to
transfer whitelist management authority.
    }

    function _transferWhitelister(address newWhitelister) internal {
        require(newWhitelister != address(0), "New whitelister is the zero address"); // Beosin (Chengdu
LianAn) // Non-zero address check.
        emit WhitelisterTransferred(_whitelister, newWhitelister); // Beosin (Chengdu LianAn) // Trigger
the WhitelisterTransferred event.
        _whitelister = newWhitelister; // Beosin (Chengdu LianAn) // Update the whitelist management
authority address.
    }

    /*
```

```
         * createLGEWhitelist - Call this after initial Token Generation Event (TGE)
         *
         * pairAddress - address generated from createPair() event on DEX
         * durations - array of durations (seconds) for each whitelist rounds
         * amountsMax - array of max amounts (TOKEN decimals) for each whitelist round
         *
         */
        // Beosin (Chengdu LianAn) // Create LGE whitelist, only _whitelister address can call.
        function createLGEWhitelist(address pairAddress, uint256[] calldata durations, uint256[] calldata
amountsMax) external onlyWhitelister() {
                require(durations.length == amountsMax.length, "Invalid whitelist(s)"); // Beosin (Chengdu
LianAn) // Array parameter length consistency check.

                _lgePairAddress = pairAddress; // Beosin (Chengdu LianAn) // Update _lgePairAddress address.

                if(durations.length > 0) {

                        delete _lgeWhitelistRounds; // Beosin (Chengdu LianAn) // Delete the original LGE
whitelist data.

                        for (uint256 i = 0; i < durations.length; i++) {
                                _lgeWhitelistRounds.push(WhitelistRound(durations[i], amountsMax[i])); // Beosin
(Chengdu LianAn) // Add a whitelist stage.
                        }

                }
        }

        /*
         * modifyLGEWhitelistAddresses - Define what addresses are included/excluded from a whitelist round
         *
         * index - 0-based index of round to modify whitelist
         * duration - period in seconds from LGE event or previous whitelist round
         * amountMax - max amount (TOKEN decimals) for each whitelist round
         *
         */
        // Beosin (Chengdu LianAn) // Modify the information about the LGE whitelist. Only _whitelister
address can be called.
        function modifyLGEWhitelist(uint256 index, uint256 duration, uint256 amountMax, address[] calldata
addresses, bool enabled) external onlyWhitelister() {
                require(index < _lgeWhitelistRounds.length, "Invalid index"); // Beosin (Chengdu LianAn) //
Check whether the corresponding stage of the index exists.
                require(amountMax > 0, "Invalid amountMax"); // Beosin (Chengdu LianAn) // amountMax non-
zero check.

                if(duration != _lgeWhitelistRounds[index].duration)
                _lgeWhitelistRounds[index].duration = duration; // Beosin (Chengdu LianAn) // Update the
duration variable of the corresponding stage of index.

            if(amountMax != _lgeWhitelistRounds[index].amountMax)
                _lgeWhitelistRounds[index].amountMax = amountMax; // Beosin (Chengdu LianAn) //
Update the amountMax variable of the corresponding stage of index.

            for (uint256 i = 0; i < addresses.length; i++) {
                _lgeWhitelistRounds[index].addresses[addresses[i]] = enabled; // Beosin (Chengdu LianAn)
// Update the addresses variable of the corresponding stage of index.
            }
        }

        /*
         *    getLGEWhitelistRound
         *
         *    returns:
         *
         *    1. whitelist round number ( 0 = no active round now )
         *    2. duration, in seconds, current whitelist round is active for
         *    3. timestamp current whitelist round closes at
         *    4. maximum amount a whitelister can purchase in this round
         *    5. is caller whitelisted
         *    6. how much caller has purchased in current whitelist round
         *
         */
        // Beosin (Chengdu LianAn) // Return current stage information and caller's status information.
        function getLGEWhitelistRound() public view returns (uint256, uint256, uint256, uint256, bool, uint256)
{

            if(_lgeTimestamp > 0) {

                uint256 wlCloseTimestampLast = _lgeTimestamp;

                for (uint256 i = 0; i < _lgeWhitelistRounds.length; i++) {

                    WhitelistRound storage wlRound = _lgeWhitelistRounds[i];

                    wlCloseTimestampLast = wlCloseTimestampLast.add(wlRound.duration);
                    if(now <= wlCloseTimestampLast)
                        return (i.add(1), wlRound.duration, wlCloseTimestampLast, wlRound.amountMax,
wlRound.addresses[_msgSender()], wlRound.purchased[_msgSender()]);
                }

            }
            return (0, 0, 0, 0, false, 0);
        }

        /*
          * _applyLGEWhitelist - internal function to be called initially before any transfers
          *
          */
        // Beosin (Chengdu LianAn) // Internal function, used to check whether the transaction transferred from the transaction pair address meets the whitelist rules.
        function _applyLGEWhitelist(address sender, address recipient, uint256 amount) internal {

            if(_lgePairAddress == address(0) || _lgeWhitelistRounds.length == 0) // Beosin (Chengdu LianAn) // Check whether the transaction pair address and LGE whitelist are set.
                return;

            if(_lgeTimestamp == 0 && sender != _lgePairAddress && recipient == _lgePairAddress && amount > 0) // Beosin (Chengdu LianAn) // When transferring token to the transaction pair address for the first time, update the value of _lgeTimestamp and start the LGE whitelist rule.
                _lgeTimestamp = now;

            if(sender == _lgePairAddress && recipient != _lgePairAddress) { // Beosin (Chengdu LianAn) // If it is transferred out from the transaction pair address.
                //buying

                (uint256 wlRoundNumber,,,,,) = getLGEWhitelistRound(); // Beosin (Chengdu LianAn) // Read the current LGE whitelist stage.

                if(wlRoundNumber > 0) {

                    WhitelistRound storage wlRound = _lgeWhitelistRounds[wlRoundNumber.sub(1)]; // Beosin (Chengdu LianAn) // Read the structure related information of the corresponding stage.

                    require(wlRound.addresses[recipient], "LGE - Buyer is not whitelisted"); // Beosin (Chengdu LianAn) // Check whether the recipient is a whitelist address.

                    uint256 amountRemaining = 0;

                    if(wlRound.purchased[recipient] < wlRound.amountMax)
                        amountRemaining = wlRound.amountMax.sub(wlRound.purchased[recipient]);

                    require(amount <= amountRemaining, "LGE - Amount exceeds whitelist maximum"); // Beosin (Chengdu LianAn) // Check whether the total purchase amount of recipient recipients exceeds amountMax.
                    wlRound.purchased[recipient] = wlRound.purchased[recipient].add(amount); // Beosin (Chengdu LianAn) // Update the purchase amount of the recipient address.
                    }

                }

            }

}

contract Polker is Context, ITRC20, Ownable, LGEWhitelisted {

        using SafeMath for uint256; // Beosin (Chengdu LianAn) // Using the SafeMath library for
mathematical operation. Avoid integer overflow.

        mapping (address => uint256) private _balances; // Beosin (Chengdu LianAn) // Mapping for storing
token balance of corresponding address.

        mapping (address => mapping (address => uint256)) private _allowances; // Beosin (Chengdu LianAn)
// Mapping for storing the allowance between two addresses.

        uint256 private _totalSupply; // Beosin (Chengdu LianAn) // Declare the private string '_totalSupply'
to store the token total supply.
        uint8 private _decimals; // Beosin (Chengdu LianAn) // Declare private string '_decimals' to store
the token decimals.
        string private _symbol; // Beosin (Chengdu LianAn) // Declare private string '_symbol' to store the
token symbol.
        string private _name; // Beosin (Chengdu LianAn) // Declare private string '_name' to store the
token name.

        constructor() public {
                _name = "POLKER";
                _symbol = "PKR";
                _decimals = 18;
                _totalSupply = 1000000000 * 10 ** 18;
                _balances[_msgSender()] = _totalSupply;

                emit Transfer(address(0), _msgSender(), _totalSupply);
        }

        /**
        * @dev Returns the bep token owner.
        */
        function getOwner() external view returns (address) {
                return owner();
        }

        /**
        * @dev Returns the token decimals.
        */
        function decimals() external view returns (uint8) {
                return _decimals;
        }

        /**
        * @dev Returns the token symbol.
        */
        function symbol() external view returns (string memory) {
                return _symbol;
        }

        /**
        * @dev Returns the token name.
        */
        function name() external view returns (string memory) {
                return _name;
        }

        /**
        * @dev See {TRC20-totalSupply}.
        */
        function totalSupply() external view returns (uint256) {
                return _totalSupply;
        }

        /**
        * @dev See {TRC20-balanceOf}.
        */
        function balanceOf(address account) external view returns (uint256) {
                return _balances[account]; // Beosin (Chengdu LianAn) // Return the balance of the specified
address.
        }

        /**
        * @dev See {TRC20-transfer}.
        *
        * Requirements:
        *
        * - `recipient` cannot be the zero address.
        * - the caller must have a balance of at least `amount`.
        */
        function transfer(address recipient, uint256 amount) external returns (bool) {
                _transfer(_msgSender(), recipient, amount); // Beosin (Chengdu LianAn) // Call the internal
function '_transfer' for token transfer.
```

```
        return true;
    }

    /**
    * @dev See {TRC20-allowance}.
    */
    function allowance(address owner, address spender) external view returns (uint256) {
        return _allowances[owner][spender]; // Beosin (Chengdu LianAn) // Return the approval value
'owner' allowed to 'spender'.
    }

    /**
    * @dev See {TRC20-approve}.
    *
    * Requirements:
    *
    * - `spender` cannot be the zero address.
    */
    // Beosin (Chengdu LianAn) // 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.
    // Beosin (Chengdu LianAn) // Using function 'increaseAllowance' and 'decreaseAllowance' to alter
allowance is recommended.
    function approve(address spender, uint256 amount) external returns (bool) {
        _approve(_msgSender(), spender, amount);    // Beosin (Chengdu LianAn) // Call the internal
function '_approve' to set the caller's approval value for the spender.
        return true;
    }

    /**
    * @dev See {TRC20-transferFrom}.
    *
    * Emits an {Approval} event indicating the updated allowance. This is not
    * required by the EIP. See the note at the beginning of {TRC20};
    *
    * Requirements:
    * - `sender` and `recipient` cannot be the zero address.
    * - `sender` must have a balance of at least `amount`.
    * - the caller must have allowance for `sender`'s tokens of at least
    * `amount`.
    */
    function transferFrom(address sender, address recipient, uint256 amount) external returns (bool) {
        _transfer(sender, recipient, amount); // Beosin (Chengdu LianAn) // Call the internal function
'_transfer' for token transfer.
        _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount, "TRC20: transfer
amount exceeds allowance")); // Beosin (Chengdu LianAn) // Update the allowance which 'sender'
allowed to caller.
```

```
        return true;
    }

    /**
    * @dev Atomically increases the allowance granted to `spender` by the caller.
    *
    * This is an alternative to {approve} that can be used as a mitigation for
    * problems described in {TRC20-approve}.
    *
    * Emits an {Approval} event indicating the updated allowance.
    *
    * Requirements:
    *
    * - `spender` cannot be the zero address.
    */
    function increaseAllowance(address spender, uint256 addedValue) external returns (bool) {
        _approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(addedValue)); //
Beosin (Chengdu LianAn) // Update the allowance which caller allowed to 'spender'.
        return true;
    }

    /**
    * @dev Atomically decreases the allowance granted to `spender` by the caller.
    *
    * This is an alternative to {approve} that can be used as a mitigation for
    * problems described in {TRC20-approve}.
    *
    * Emits an {Approval} event indicating the updated allowance.
    *
    * Requirements:
    *
    * - `spender` cannot be the zero address.
    * - `spender` must have allowance for the caller of at least
    * `subtractedValue`.
    */
    function decreaseAllowance(address spender, uint256 subtractedValue) external returns (bool) {
        _approve(_msgSender(), spender, _allowances[_msgSender()][spender].sub(subtractedValue,
"TRC20: decreased allowance below zero")); // Beosin (Chengdu LianAn) // Update the allowance which
caller allowed to 'spender'.
        return true;
    }

    /**
    * @dev Moves tokens `amount` from `sender` to `recipient`.
    *
    * This is internal function is equivalent to {transfer}, and can be used to
    * e.g. implement automatic token fees, slashing mechanisms, etc.
    *
    * Emits a {Transfer} event.
    *
    * Requirements:
    *
    * - `sender` cannot be the zero address.
    * - `recipient` cannot be the zero address.
    * - `sender` must have a balance of at least `amount`.
    */
    function _transfer(address sender, address recipient, uint256 amount) internal {
        require(sender != address(0), "TRC20: transfer from the zero address"); // Beosin (Chengdu
LianAn) // The non-zero address check for 'sender'.
        require(recipient != address(0), "TRC20: transfer to the zero address"); // Beosin (Chengdu
LianAn) // The non-zero address check for 'recipient'.

        _applyLGEWhitelist(sender, recipient, amount); // Beosin (Chengdu LianAn) // Call the internal
function '_applyLGEWhitelist' for check transfer.

        _balances[sender] = _balances[sender].sub(amount, "TRC20: transfer amount exceeds balance"); //
Beosin (Chengdu LianAn) // Update the sender address token balance.
        _balances[recipient] = _balances[recipient].add(amount); // Beosin (Chengdu LianAn) // Update
the recipient address token balance.
        emit Transfer(sender, recipient, amount); // Beosin (Chengdu LianAn) // Trigger the event
'Transfer'.
    }

    /**
    * @dev Sets `amount` as the allowance of `spender` over the `owner`s tokens.
    *
    * This is internal function is equivalent to `approve`, and can be used to
    * e.g. set automatic allowances for certain subsystems, etc.
    *
    * Emits an {Approval} event.
    *
    * Requirements:
    *
    * - `owner` cannot be the zero address.
    * - `spender` cannot be the zero address.
    */
    function _approve(address owner, address spender, uint256 amount) internal {
        require(owner != address(0), "TRC20: approve from the zero address"); // Beosin (Chengdu
LianAn) // The non-zero address check for 'owner'.
        require(spender != address(0), "TRC20: approve to the zero address"); // Beosin (Chengdu
LianAn) // The non-zero address check for 'spender'.
```

```
                _allowances[owner][spender] = amount; // Beosin (Chengdu LianAn) // The allowance which
'owner' allowed to 'spender' is set to 'amount'.
                emit Approval(owner, spender, amount); // Beosin (Chengdu LianAn) // Trigger the event
'Approval'.
        }

}

// Beosin (Chengdu LianAn) // Recommend the main contract to inherit 'Pausable' module to grant
owner the authority of pausing all transactions when serious issue occurred.
```

BEOSIN

Blockchain Security

Official Website

https://lianantech.com

E-mail

vaas@lianantech.com

Twitter

https://twitter.com/Beosin_com