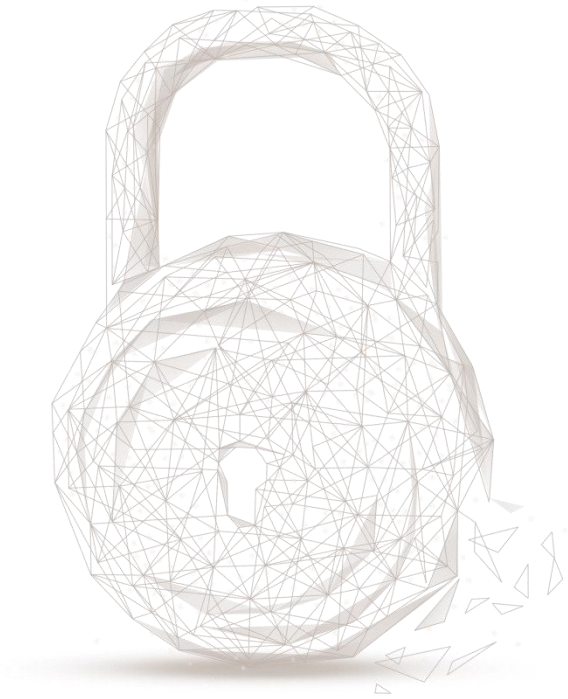
BEOSIN
Blockchain Security

Smart contract security audit report



Audit Number：202104261547

Smart Contract Name：

AppleToken (Apple)

Smart Contract Address：

0x28F1Ec92Dd1BF76e2Bc2eE9290fb841A4547a325

Start Date：2021.04.20

Completion Date：2021.04.26

Overall Result：Pass（Distinction）

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

Audit Categories and Results:

No.	Categories	Subitems	Results
1	Coding Conventions	HRC-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	-	Pass
12	Fake Deposit	-	Pass
13	event security	-	Pass

Note: Audit results and suggestions in code comments

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 Apple, including Coding Standards, Security, and Business Logic. **Apple 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	apple token
Token symbol	apt
decimals	18
totalSupply	Initial 200 million (Mintable, maximum limit is 2 billion)
Token type	HRC-20

Table 1 – Basic Token Information

2、Token Vesting Information

As shown in the figure below, the contract owner can call the *lock* function to transfer and lock tokens for a specified account. When locking, caller need to specify the lock-up address, lock-up amount, and lock-up type. Different types of lock-ups correspond to different daily releases and can be controlled by the contract owner. The minimum lock release time unit is "one day", and it is not calculated if it is less than one day. Therefore, the total lock release time may be longer than the expected time, but it does not affect the total release.

```
389     function lock(address _account, uint256 _amount, uint256 _type) public onlyOwner {
390         require(_account != address(0), "Cannot transfer to the zero address");
391         require(lockedUser[_account].lockedAmount == 0, "exist locked token");
392         require(_account != swapGovContract, "equal to swapGovContract");
393         lockedUser[_account].initLock = _amount;
394         lockedUser[_account].lockedAmount = _amount;
395         lockedUser[_account].lastUnlockTs = block.timestamp >= lockreleasetime ? block.timestamp : lockreleasetime;
396         lockedUser[_account].releaseType = _type;
397         _balances[_msgSender()] = _balances[_msgSender()].sub(_amount);
398         _balances[_account] = _balances[_account].add(_amount);
399         emit Lock(_account, block.timestamp, _amount, _type);
400         emit Transfer(_msgSender(), _account, _amount);
401     }
```

Figure 1 lock function source code

As shown in the figure below, the token lock balance requires the user to manually call the *unlock* function to unlock it, and it will not be automatically unlocked.

```
403     function unlock() public {
404         uint256 amount = getAvailablelockAmount(_msgSender(), lockedUser[_msgSender()].releaseType);
405         require(amount > 0, "amount equal 0");
406         lockedUser[_msgSender()].lockedAmount = lockedUser[_msgSender()].lockedAmount.sub(amount);
407         lockedUser[_msgSender()].lastUnlockTs = block.timestamp;
408         emit UnLock(_msgSender(), block.timestamp, amount);
409     }
```

Figure 2 unlock function source code

- Safety suggestion: When calling *unlock* to unlock, change the *lastUnlockTs* variable to the timestamp of the number of days currently unlocked to avoid errors.
- Fix result: ignore

3、Other function description

- Mint function

The contract implements the coin minting function. The contract owner can add Minter permissions, and addresses with Minter permissions can call the *mint* function to mint coins, and the minting limit is 2 billion.

- Blacklist function

The contract implements the blacklist function, and the contract owner will have the right to add an address into blacklist. The addresses added to the blacklist cannot participate in *transfer* or *transferFrom* operation.

Audited Source Code with Comments:

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

pragma solidity 0.6.12;

/**
 * @title SafeMath
 * @dev Unsigned math operations with safety checks that revert on error.
 */
// Beosin (Chengdu LianAn) // The SafeMath library declares functions for safe mathematical operation.
library SafeMath {
    /**
     * @dev Multiplie two unsigned integers, revert on 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-solidity/pull/522
        if (a == 0) {
            return 0;
        }

        uint256 c = a * b;
        require(c / a == b);

        return c;
    }

    /**
     * @dev Integer division of two unsigned integers truncating the quotient, revert on division by zero.
     */
```

```
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        // Solidity only automatically asserts when dividing by 0
        require(b > 0);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Subtract two unsigned integers, revert on underflow (i.e. if subtrahend is greater than minuend).
     */
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b <= a);
        uint256 c = a - b;

        return c;
    }

    /**
     * @dev Add two unsigned integers, revert on overflow.
     */
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a);

        return c;
    }
}


/**
 * @title Roles
 * @dev Library for managing addresses assigned to a Role.
 */
library Roles {
    struct Role {
        mapping (address => bool) bearer;
    }

    /**
     * @dev Give an account access to this role.
     */
    function add(Role storage role, address account) internal {
        require(!has(role, account), "Roles: account already has role");
```

```
        role.bearer[account] = true;
    }

    /**
     * @dev Remove an account's access to this role.
     */
    function remove(Role storage role, address account) internal {
        require(has(role, account), "Roles: account does not have role");
        role.bearer[account] = false;
    }

    /**
     * @dev Check if an account has this role.
     * @return bool
     */
    function has(Role storage role, address account) internal view returns (bool) {
        require(account != address(0), "Roles: account is the zero address");
        return role.bearer[account];
    }
}


/*
 * @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 {

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

/**
 * @title ERC20 interface
 * @dev See https://eips.ethereum.org/EIPS/eip-20
 */
// Beosin (Chengdu LianAn) // Define ERC20 standard interface.
interface IERC20 {
    function transfer(address to, uint256 value) external returns (bool);
```

```
    function approve(address spender, uint256 value) external returns (bool);

    function transferFrom(address from, address to, uint256 value) external returns (bool);

    function totalSupply() external view returns (uint256);

    function balanceOf(address who) external view returns (uint256);

    function allowance(address owner, address spender) external view returns (uint256);

    event Transfer(address indexed from, address indexed to, uint256 value);

    event Approval(address indexed owner, address indexed spender, uint256 value);
}


/**
 * @title Standard ERC20 token
 * @dev Implementation of the basic standard token.
 */
contract StandardToken is IERC20, Context {
    using SafeMath for uint256; // Beosin (Chengdu LianAn) // Use the SafeMath library for mathematical operation. Avoid integer overflow/underflow.

    mapping (address => uint256) internal _balances; // Beosin (Chengdu LianAn) // Declare the mapping variable '_balances' for storing the token balances of corresponding address.
    mapping (address => mapping (address => uint256)) internal _allowed; // Beosin (Chengdu LianAn) // Declare the mapping variable '_allowed' for storing the allowance tokens between two addresses.

    uint256 internal _totalSupply; // Beosin (Chengdu LianAn) // Declare the variable '_totalSupply' for storing the total tokens.

    /**
     * @dev Total number of tokens in existence.
     */
    function totalSupply() public override view returns (uint256) {
        return _totalSupply;
    }

    /**
     * @dev Get the balance of the specified address.
     * @param owner The address to query the balance of.
     * @return A uint256 representing the amount owned by the passed address.
     */
```

```
    function balanceOf(address owner) public override view    returns (uint256) {
        return _balances[owner];
    }

    /**
     * @dev Function to check the amount of tokens that an owner allowed to a spender.
     * @param owner The address which owns the funds.
     * @param spender The address which will spend the funds.
     * @return A uint256 specifying the amount of tokens still available for the spender.
     */
    function allowance(address owner, address spender) public override view returns (uint256) {
        return _allowed[owner][spender];
    }

    /**
     * @dev Transfer tokens to a specified address.
     * @param to The address to transfer to.
     * @param value The amount to be transferred.
     */
    function transfer(address to, uint256 value) public virtual override returns (bool) {
        _transfer(_msgSender(), to, value); // Beosin (Chengdu LianAn) // Call the internal function '_transfer' for token transfer.
        return true;
    }

    /**
     * @dev Approve the passed address to spend the specified amount of tokens on behalf of msg.sender.
     * 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
     * @param spender The address which will spend the funds.
     * @param value The amount of tokens to be spent.
     */
    // 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(ChengduLianAn) // Using function 'increaseAllowance' and 'decreaseAllowance' to alter Allowance is recommended.
    function approve(address spender, uint256 value) public override returns (bool) {
        _approve(_msgSender(), spender, value); // Beosin (Chengdu LianAn) // Call the internal function '_approve' to set the caller's approval value for the sender.
        return true;
```

```
    }

    /**
     * @dev Transfer tokens from one address to another.
     * Note that while this function emits an Approval event, this is not required as per the specification,
     * and other compliant implementations may not emit the event.
     * @param from The address which you want to send tokens from.
     * @param to The address which you want to transfer to.
     * @param value The amount of tokens to be transferred.
     */
    function transferFrom(address from, address to, uint256 value) public virtual override returns (bool)
{
        _transfer(from, to, value); // Beosin (Chengdu LianAn) // Call the internal function '_transfer' for token transfer.
        _approve(from, _msgSender(), _allowed[from][_msgSender()].sub(value)); // Beosin (Chengdu LianAn) // Call the internal function '_approve' to alter the allowance between two addresses.
        return true;
    }

    /**
     * @dev Increase the amount of tokens that an owner allowed to a spender.
     * approve should be called when _allowed[msg.sender][spender] == 0. To increment
     * allowed value is better to use this function to avoid 2 calls (and wait until
     * the first transaction is mined)
     * From MonolithDAO Token.sol
     * Emits an Approval event.
     * @param spender The address which will spend the funds.
     * @param addedValue The amount of tokens to increase the allowance by.
     */
    function increaseAllowance(address spender, uint256 addedValue) public returns (bool) {
        _approve(_msgSender(), spender, _allowed[_msgSender()][spender].add(addedValue)); // Beosin (Chengdu LianAn) // Call the internal function '_approve' to increase the allowance between two addresses.
        return true;
    }

    /**
     * @dev Decrease the amount of tokens that an owner allowed to a spender.
     * approve should be called when _allowed[msg.sender][spender] == 0. To decrement
     * allowed value is better to use this function to avoid 2 calls (and wait until
     * the first transaction is mined)
     * From MonolithDAO Token.sol
     * Emits an Approval event.
     * @param spender The address which will spend the funds.
     * @param subtractedValue The amount of tokens to decrease the allowance by.
```

```
     */
    function decreaseAllowance(address spender, uint256 subtractedValue) public returns (bool) {
        _approve(_msgSender(), spender, _allowed[_msgSender()][spender].sub(subtractedValue));
// Beosin (Chengdu LianAn) // Call the internal function '_approve' to decrease the allowance between two addresses.
        return true;
    }

    /**
     * @dev Transfer tokens for a specified address.
     * @param from The address to transfer from.
     * @param to The address to transfer to.
     * @param value The amount to be transferred.
     */
    function _transfer(address from, address to, uint256 value) internal {
        require(to != address(0), "Cannot transfer to the zero address"); // Beosin (Chengdu LianAn) // The non-zero address check for 'to'.
        _balances[from] = _balances[from].sub(value); // Beosin (Chengdu LianAn) // Alter the token balance of 'from'.
        _balances[to] = _balances[to].add(value); // Beosin (Chengdu LianAn) // Alter the token balance of 'to'.
        emit Transfer(from, to, value); // Beosin (Chengdu LianAn) // Trigger the event 'Transfer'.
    }

    /**
     * @dev Approve an address to spend another addresses' tokens.
     * @param owner The address that owns the tokens.
     * @param spender The address that will spend the tokens.
     * @param value The number of tokens that can be spent.
     */
    function _approve(address owner, address spender, uint256 value) internal {
        require(spender != address(0), "Cannot approve to the zero address"); // Beosin (Chengdu LianAn) // The non-zero address check for 'spender'.
        require(owner != address(0), "Setter cannot be the zero address"); // Beosin (Chengdu LianAn) // The non-zero address check for 'owner'.
        _allowed[owner][spender] = value; // Beosin (Chengdu LianAn) // The allowance which 'owner' allowed to 'spender' is set to 'amount'.
        emit Approval(owner, spender, value); // Beosin (Chengdu LianAn) // Trigger the event 'Approval'.
    }

    /** @dev Creates `amount` tokens and assigns them to `account`, increasing
     * the total supply.
     *
     * Emits a `Transfer` event with `from` set to the zero address.
     *
```

```
     * Requirements
     *
     * - `to` cannot be the zero address.
     */
    function _mint(address account, uint256 amount) internal {
        require(account != address(0), "ERC20: mint to the zero address");

        _totalSupply = _totalSupply.add(amount);
        _balances[account] = _balances[account].add(amount);
        emit Transfer(address(0), account, amount);
    }

    /**
     * @dev Destroys `amount` tokens from `account`, reducing the
     * total supply.
     *
     * Emits a {Transfer} event with `to` set to the zero address.
     *
     * Requirements
     *
     * - `account` cannot be the zero address.
     * - `account` must have at least `amount` tokens.
     */
    function _burn(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: burn from the zero address"); // Beosin (Chengdu LianAn) // The non-zero address check for 'account'.
        _balances[account] = _balances[account].sub(amount); // Beosin (Chengdu LianAn) // Alter the token balance of 'account'.
        _totalSupply = _totalSupply.sub(amount); // Beosin (Chengdu LianAn) // Alter the total token supply of 'amount'.
        emit Transfer(account, address(0), amount); // Beosin (Chengdu LianAn) // Trigger the event 'Transfer'.
    }

    /**
     * @dev Destroys `amount` tokens from `account`.`amount` is then deducted
     * from the caller's allowance.
     *
     * See {_burn} and {_approve}.
     */
    function _burnFrom(address account, uint256 amount) internal virtual {
        _burn(account, amount); // Beosin (Chengdu LianAn) // Call the internal function '_burn' for token burn.
        _approve(account, _msgSender(), _allowed[account][_msgSender()].sub(amount)); // Beosin (Chengdu LianAn) // Call the internal function '_approve' to alter the allowance between two addresses.
```

```
    }

}


contract MinterRole {
    using Roles for Roles.Role;

    event MinterAdded(address indexed account);
    event MinterRemoved(address indexed account);

    Roles.Role private _minters;

    constructor () internal {
        _addMinter(msg.sender);
    }

    modifier onlyMinter() {
        require(isMinter(msg.sender), "MinterRole: caller does not have the Minter role");
        _;
    }

    function isMinter(address account) public view returns (bool) {
        return _minters.has(account);
    }

    function addMinter(address account) public onlyMinter {
        _addMinter(account);
    }

    function renounceMinter() public {
        _removeMinter(msg.sender);
    }

    function _addMinter(address account) internal {
        _minters.add(account);
        emit MinterAdded(account);
    }

    function _removeMinter(address account) internal {
        _minters.remove(account);
        emit MinterRemoved(account);
    }
}
```

```
/**
 * @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.
 *
 * 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 internal _owner; // Beosin (Chengdu LianAn) // Declare the variable '_owner' to store the address of the contract owner.

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

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

    /**
     * @dev Throws if called by any account other than the owner.
     */
    // Beosin (Chengdu LianAn) // Modifier, check that the caller is '_owner'.
    modifier onlyOwner() {
        require(isOwner(), "Ownable: caller is not the owner");
        _;
    }

    /**
     * @dev Returns true if the caller is the current owner.
     */
    function isOwner() public view returns (bool) {
        return _msgSender() == _owner;
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public onlyOwner {
        _transferOwnership(newOwner); // Beosin (Chengdu LianAn) // Call the internal function '_transferOwnership' for change the '_owner' address.
```

```
    }

    /**
     * @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; // Beosin (Chengdu LianAn) // Change the '_owner' address.
    }
}


/**
 * @dev Extension of `ERC20` that adds a set of accounts with the `MinterRole`,
 * which have permission to mint (create) new tokens as they see fit.
 *
 * At construction, the deployer of the contract is the only minter.
 */
contract ERC20Mintable is StandardToken, MinterRole {
    uint256 public constant cap = 2000000000 * (10**18); //20
    /**
     * @dev See `ERC20._mint`.
     *
     * Requirements:
     *
     * - the caller must have the `MinterRole`.
     */
    function mint(address account, uint256 amount) public onlyMinter returns (bool) {
        require(totalSupply().add(amount)<= cap, "more than token limit");
        _mint(account, amount);
        return true;
    }
}
contract AppleToken is ERC20Mintable, Ownable{

//----------------Token Info----------------//
    string public constant name = "apple token"; // Beosin (Chengdu LianAn) // Declare the constant 'name' for storing the token name.
    string public constant symbol = "apt"; // Beosin (Chengdu LianAn) // Declare the constant 'symbol' for storing the token symbol.
    uint8 public constant decimals = 18; // Beosin (Chengdu LianAn) // Declare the constant 'decimals' for storing the token decimals.
```

```
    uint256 public constant INITIAL_SUPPLY = 200000000 * 10 ** 18; // Beosin (Chengdu LianAn) // Declare the constant 'INITIAL_SUPPLY' for storing the token initial supply.
    address public swapGovContract; // Beosin (Chengdu LianAn) // Declare the variable 'swapGovContract' for storing the contract of GraSwapGov address.
    // bool setGovFlag = false;

//----------Lock Info----------//
    uint256 public techReleaseByDay = 6040 * 10 ** 18; // Beosin (Chengdu LianAn) // Declare the variable 'techReleaseByDay' to store the number of unlocked tokens per day for "technical party".
    uint256 public capitalReleaseByDay =64 * 10 ** 18; // Beosin (Chengdu LianAn) // Declare the variable 'capitalReleaseByDay' to store the number of unlocked tokens per day for "capital party".
    uint256 public nodeReleaseByDay = 44 * 10 ** 18; // Beosin (Chengdu LianAn) // Declare the variable 'nodeReleaseByDay' to store the number of unlocked tokens per day for "super node party".
    uint256 public lockreleasetime =1620576000; //05.10    // Beosin (Chengdu LianAn) // Declare the variable 'lockreleasetime' to store the start time of unlocked.
    // uint256 public lockedAmount;

    // Beosin (Chengdu LianAn) // Declare the struct 'LockInfo' for storing the lock information.
    struct LockInfo {
        uint256 initLock; // Beosin (Chengdu LianAn) // The init lock amount.
        uint256 lockedAmount; // Beosin (Chengdu LianAn) // The remaining lock.
        uint256 lastUnlockTs; // Beosin (Chengdu LianAn) // The last unlock time.
        uint256 releaseType; // Beosin (Chengdu LianAn) // The type of 'LockInfo'.
    }

    mapping(address => LockInfo) public lockedUser; // Beosin (Chengdu LianAn) // Declare the variable 'lockedUser' for storing the corresponding user lock information.

    event Lock(address account, uint256 startTime, uint256 amount, uint256 releaseType); // Beosin (Chengdu LianAn) // Declare the event 'Lock'.
    event UnLock(address account, uint256 unlockTime, uint256 amount); // Beosin (Chengdu LianAn) // Declare the event 'UnLock'.
//----------Blacklist module----------//
    mapping(address => bool) private _isBlackListed; // Beosin (Chengdu LianAn) // Declare the variable '_isBlackListed' for storing the blacklist address.
    event AddedBlackLists(address[]); // Beosin (Chengdu LianAn) // Declare the event 'AddedBlackLists'.
    event RemovedBlackLists(address[]); // Beosin (Chengdu LianAn) // Declare the event 'RemovedBlackLists'.

    constructor() public {
        _totalSupply = INITIAL_SUPPLY; // Beosin (Chengdu LianAn) // Set '_totalSupply' to 'INITIAL_SUPPLY'.
```

```
        _balances[msg.sender] = _totalSupply; // Beosin (Chengdu LianAn) // Send all tokens to the msg.sender address.
        emit Transfer(address(0), msg.sender, INITIAL_SUPPLY); // Beosin (Chengdu LianAn) // Trigger the event 'Transfer'.
        _owner = msg.sender; // Beosin (Chengdu LianAn) // Set '_owner' to 'msg.sender'.
        emit OwnershipTransferred(address(0), msg.sender); // Beosin (Chengdu LianAn) // Trigger the event 'OwnershipTransferred'.


    }



    function isBlackListed(address user) public view returns (bool) {
        return _isBlackListed[user];
    }

    // Beosin (Chengdu LianAn) // Add blacklists in batches, only contract owners can call.
    function addBlackLists(address[] calldata _evilUser) public onlyOwner {
        for (uint i = 0; i < _evilUser.length; i++) {
            _isBlackListed[_evilUser[i]] = true;
        }
        emit AddedBlackLists(_evilUser);
    }

    // Beosin (Chengdu LianAn) // Remove blacklists in batches, only contract owners can call.
    function removeBlackLists(address[] calldata _clearedUser) public onlyOwner {
        for (uint i = 0; i < _clearedUser.length; i++) {
            delete _isBlackListed[_clearedUser[i]];
        }
        emit RemovedBlackLists(_clearedUser);
    }



    // function lockToGov() public onlyOwner {
    //         _transfer(_owner, swapGovContract, MINEREWARD); // transfer/freeze to swapGovContract
    //     lockedAmount = lockedAmount.add(MINEREWARD);
    // }
    function lock(address _account, uint256 _amount, uint256 _type) public onlyOwner {
    require(_type > 0 && _type < 4); // Beosin (Chengdu LianAn) // The check for '_type'.
        require(_account != address(0), "Cannot transfer to the zero address"); // Beosin (Chengdu LianAn) // The non-zero address check for '_account'.
        require(lockedUser[_account].lockedAmount == 0, "exist locked token"); // Beosin (Chengdu LianAn) // Check that the '_account' releases the existing lock, if it exists, owner cannot add a new lock to it.
```

```
        require(_account != swapGovContract, "equal to swapGovContract"); // Beosin (Chengdu LianAn) // The lock-up address cannot be the address of the 'swapGovContract'.
        lockedUser[_account].initLock = _amount; // Beosin (Chengdu LianAn) // Set lock information 'initLock'.
        lockedUser[_account].lockedAmount = _amount; // Beosin (Chengdu LianAn) // Set lock information 'lockedAmount'.
        lockedUser[_account].lastUnlockTs = block.timestamp >= lockreleasetime ? block.timestamp : lockreleasetime; // Beosin (Chengdu LianAn) // Set lock information 'lastUnlockTs'.
        lockedUser[_account].releaseType = _type; // Beosin (Chengdu LianAn) // Set lock information 'releaseType'.
        _balances[_msgSender()] = _balances[_msgSender()].sub(_amount); // Beosin (Chengdu LianAn) // Reduce the caller's token balance.
        _balances[_account] = _balances[_account].add(_amount); // Beosin (Chengdu LianAn) // Increase the token balance of the lock-up address.
        emit Lock(_account, block.timestamp, _amount, _type); // Beosin (Chengdu LianAn) // Trigger the event 'Lock'.
        emit Transfer(_msgSender(), _account, _amount); // Beosin (Chengdu LianAn) // Trigger the event 'Transfer'.

    }

    function unlock() public {
        uint256 amount = getAvailablelockAmount(_msgSender(), lockedUser[_msgSender()].releaseType); // Beosin (Chengdu LianAn) // Call the function 'getAvailablelockAmount' for calculate the number of newly unlocked tokens.
        require(amount > 0, "amount equal 0"); // Beosin (Chengdu LianAn) // The non-zero check for 'amount'.
        lockedUser[_msgSender()].lockedAmount = lockedUser[_msgSender()].lockedAmount.sub(amount); // Beosin (Chengdu LianAn) // Update lock information 'lockedAmount'.
        lockedUser[_msgSender()].lastUnlockTs = block.timestamp; // Beosin (Chengdu LianAn) // Update lock information 'lastUnlockTs'.
        emit UnLock(_msgSender(), block.timestamp, amount); // Beosin (Chengdu LianAn) // Trigger the event 'Unlock'.

    }

    function getAvailablelockAmount(address account, uint256 releaseType) public view returns (uint256) {
        if(lockedUser[account].lockedAmount == 0) { // Beosin (Chengdu LianAn) // The non-zero check for 'lockedAmount'.
            return 0;
        }
```

```
        if(block.timestamp <= lockedUser[account].lastUnlockTs) { // Beosin (Chengdu LianAn) // Check that the current time is greater than the 'lastUnlockTs'.
            return 0;
        }

        uint256 _days = block.timestamp.sub(lockedUser[account].lastUnlockTs).div(86400); // Beosin (Chengdu LianAn) // Calculate the time interval for the current data to be unlocked last time, in "days" as the unit.
        if(_days > 0 && releaseType == 1) {
            uint256 _releaseAmount = _days.mul(techReleaseByDay); // Beosin (Chengdu LianAn) // Calculate the unlock amount according to the category.
            return lockedUser[account].lockedAmount > _releaseAmount ? _releaseAmount : lockedUser[account].lockedAmount; // Beosin (Chengdu LianAn) // Returns the final unlockable value.
        }

        if(_days > 0 && releaseType == 2) {
            uint256 _releaseAmount = _days.mul(capitalReleaseByDay); // Beosin (Chengdu LianAn) // Calculate the unlock amount according to the category.
            return lockedUser[account].lockedAmount > _releaseAmount ? _releaseAmount : lockedUser[account].lockedAmount; // Beosin (Chengdu LianAn) // Returns the final unlockable value.
        }

        if(_days > 0 && releaseType == 3) {
            uint256 _releaseAmount = _days.mul(nodeReleaseByDay); // Beosin (Chengdu LianAn) // Calculate the unlock amount according to the category.
            return lockedUser[account].lockedAmount > _releaseAmount ? _releaseAmount : lockedUser[account].lockedAmount; // Beosin (Chengdu LianAn) // Returns the final unlockable value.
        }
        return 0;
    }

    function transfer(address _to, uint256 _value) public override returns (bool) {
        require(!isBlackListed(_msgSender())); // Beosin (Chengdu LianAn) // The blacklist check for caller.
        require(!isBlackListed(_to)); // Beosin (Chengdu LianAn) // The blacklist check for '_to'.
        require(_balances[_msgSender()].sub(lockedUser[_msgSender()].lockedAmount) >= _value); // Beosin (Chengdu LianAn) // Available balance check for caller.
        return super.transfer(_to, _value); // Beosin (Chengdu LianAn) // Call the 'transfer' function in the parent contract to transfer.
    }

    function transferFrom(address _from, address _to, uint256 _value) public override returns (bool) {
```

```
        require(!isBlackListed(_msgSender())); // Beosin (Chengdu LianAn) // The blacklist check for caller.
        require(!isBlackListed(_from)); // Beosin (Chengdu LianAn) // The blacklist check for '_from'.
        require(!isBlackListed(_to)); // The blacklist check for '_to'.
        require(_balances[_from].sub(lockedUser[_from].lockedAmount) >= _value); // Beosin (Chengdu LianAn) // Available balance check for '_from'.
        return super.transferFrom(_from, _to, _value); // Beosin (Chengdu LianAn) // Call the 'transferFrom' function in the parent contract to transfer.
    }

    /**
     * @dev Transfer tokens to multiple addresses.
     */
    function batchTransfer(address[] memory addressList, uint256[] memory amountList) public onlyOwner returns (bool) {
    uint256 length = addressList.length; // Beosin (Chengdu LianAn) // Declare the variable 'length' for storing the addressList length.
        require(addressList.length == amountList.length, "Inconsistent array length"); // Beosin (Chengdu LianAn) // Check that the length of 'addressList' is equal to 'amountList'.
        require(length > 0 && length <= 150, "Invalid number of transfer objects"); // Beosin (Chengdu LianAn) // Check that the length of 'addressList' is non-zero and does not exceed 150.
        uint256 amount; // Beosin (Chengdu LianAn) // Declare the temporary variable 'amount' to store the total amount.
        for (uint256 i = 0; i < length; i++) {
            require(amountList[i] > 0, "The transfer amount cannot be 0"); // Beosin (Chengdu LianAn) // The non-zero check for 'amountList[i]'.
            require(addressList[i] != address(0), "Cannot transfer to the zero address"); // Beosin (Chengdu LianAn) // The non-zero address check for 'addressList[i]'.
            require(!isBlackListed(addressList[i])); // Beosin (Chengdu LianAn) // The blacklist check for 'addressList[i]'.
            amount = amount.add(amountList[i]); // Beosin (Chengdu LianAn) // Update the temporary variable 'amount'.
            _balances[addressList[i]] = _balances[addressList[i]].add(amountList[i]); // Beosin (Chengdu LianAn) // Increase the token balance of 'addressList[i]'.
            emit Transfer(_msgSender(), addressList[i], amountList[i]); // Beosin (Chengdu LianAn) // Trigger the event 'Transfer'.
        }
        require(_balances[_msgSender()].sub(lockedUser[_msgSender()].lockedAmount) >= amount, "Not enough tokens to transfer"); // Beosin (Chengdu LianAn) // Check that the caller has enough available tokens.
        _balances[_msgSender()] = _balances[_msgSender()].sub(amount); // Beosin (Chengdu LianAn) // Update the caller's token balance.
        return true;

    }
```

```
    function burn(uint256 amount) public virtual {
        _checkBeforeBurn(_msgSender(), amount);
        _burn(_msgSender(), amount); // Beosin (Chengdu LianAn) // Call the internal function '_burn' for token burn.
    }

    function burnFrom(address account, uint256 amount) public virtual {
    _checkBeforeBurn(account, amount);
    _burnFrom(account, amount); // Beosin (Chengdu LianAn) // Call the internal function '_burnFrom' for token burn.
    }

    function _checkBeforeBurn(address account, uint256 amount) internal {
        uint256 amt = lockedUser[account].lockedAmount;
        if (amt > 0) {
            require(balanceOf(account).sub(amt) >= amount, "token balance no enought burn");
            }

    }



    function setGovAddr(address _swapGovContract) public onlyOwner {
        // require(!setGovFlag); // only once
        swapGovContract = _swapGovContract; // Beosin (Chengdu LianAn) // Alter 'swapGovContract' for '_swapGovContract'.
        // setGovFlag = true;
    }
    function setReleaseByDay(uint256 _techReleaseByDay, uint256 _capitalReleaseByDay, uint256 _nodeReleaseByDay) public onlyOwner {
        require(_techReleaseByDay > 0, " must be great 0"); // Beosin (Chengdu LianAn) // The non-zero check for '_techReleaseByDay'.
        require(_capitalReleaseByDay > 0, " must be great 0"); // Beosin (Chengdu LianAn) // The non-zero check for '_capitalReleaseByDay'.
        require(_nodeReleaseByDay > 0, " must be great 0"); // Beosin (Chengdu LianAn) // The non-zero check for '_nodeReleaseByDay'.
    techReleaseByDay = _techReleaseByDay; // Beosin (Chengdu LianAn) // Alter 'techReleaseByDay' for '_techReleaseByDay'.
        capitalReleaseByDay = _capitalReleaseByDay; // Beosin (Chengdu LianAn) // Alter 'capitalReleaseByDay' for '_capitalReleaseByDay'.
        nodeReleaseByDay = _nodeReleaseByDay; // Beosin (Chengdu LianAn) // Alter 'nodeReleaseByDay' for '_nodeReleaseByDay'.

    }
}
```

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

Official Website

https://lianantech.com

E-mail

vaas@lianantech.com

Twitter

https://twitter.com/Beosin_com