Smart Contract
Security Audit Report

The SlowMist Security Team received the team's application for smart contract security audit of the CUSD on 2022.02.16. The following are the details and results of this smart contract security audit:

Token Name :

CUSD

File name and hash (SHA256) :

CUSDToken.sol: 9e2b398e3c173d2b79ddc9359063d7e2982f027649254cdac2339d851d50654e

The audit items and results :

(Other unknown security vulnerabilities are not included in the audit responsibility scope)

NO.	Audit Items	Result
1	Replay Vulnerability	Passed
2	Denial of Service Vulnerability	Passed
3	Race Conditions Vulnerability	Passed
4	Authority Control Vulnerability	Some Risks
5	Integer Overflow and Underflow Vulnerability	Passed
6	Gas Optimization Audit	Passed
7	Design Logic Audit	Passed
8	Uninitialized Storage Pointers Vulnerability	Passed
9	Arithmetic Accuracy Deviation Vulnerability	Passed
10	"False top-up" Vulnerability	Passed
11	Malicious Event Log Audit	Passed
12	Scoping and Declarations Audit	Passed

NO.	Audit Items	Result
13	Safety Design Audit	Passed

Audit Result : Low Risk

Audit Number : 0X002202210003

Audit Date : 2022.02.16 - 2022.02.21

Audit Team : SlowMist Security Team

Summary conclusion : This is a token contract that does not contain the tokenVault section. The total amount of contract tokens can be changed, users can burn their own tokens through the burn function. SafeMath security module is used, which is a recommended approach. The contract does not have the Overflow and the Race Conditions issue.

During the audit, we found the following issue:

1. The owner role can add a new minter and the minter role can mint tokens arbitrarily through the mint function and there is no upper limit on the amount of tokens that can be minted that will lead to token inflation. After communication with the project team, the project team expresses that the mint function is required in their contract because they need it to work in other contract. The other contract will lock the C98 token from the user then mint new CUSD based on the current C98 price. Besides, the owner will be controlled by multisig.
2. The owner role can withdraw any tokens from this contract to a specific address.
3. There is no upper limit of the gas to execute the low-level call to withdraw the native tokens.

The source code:

```
// SPDX-License-Identifier: MIT
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.8.0;

/**
```

```
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when an
 * operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
 * class of bugs, so it's recommended to use it always.
 */
// CAUTION
// This version of SafeMath should only be used with Solidity 0.8 or later,
// because it relies on the compiler's built in overflow checks.

/**
 * @dev Wrappers over Solidity's arithmetic operations.
 *
 * NOTE: `SafeMath` is generally not needed starting with Solidity 0.8, since the
compiler
 * now has built in overflow checking.
 */
//SlowMist// OpenZeppelin's SafeMath security module is used, which is a recommended
approach
library SafeMath {
  /**
   * @dev Returns the addition of two unsigned integers, with an overflow flag.
   *
   * _Available since v3.4._
   */
  function tryAdd(uint256 a, uint256 b) internal pure returns (bool, uint256) {
      unchecked {
        uint256 c = a + b;
        if (c < a) return (false, 0);
        return (true, c);
      }
  }

  /**
   * @dev Returns the substraction of two unsigned integers, with an overflow flag.
   *
   * _Available since v3.4._
   */
```

```
  function trySub(uint256 a, uint256 b) internal pure returns (bool, uint256) {
      unchecked {
        if (b > a) return (false, 0);
        return (true, a - b);
      }
  }

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

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

  /**
   * @dev Returns the remainder of dividing two unsigned integers, with a division by
zero flag.
   *
   * _Available since v3.4._
   */
```

```
function tryMod(uint256 a, uint256 b) internal pure returns (bool, uint256) {
    unchecked {
      if (b == 0) return (false, 0);
      return (true, a % b);
    }
}

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

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

/**
 * @dev Returns the multiplication of two unsigned integers, reverting on
 * overflow.
 *
 * Counterpart to Solidity's `*` operator.
 *
 * Requirements:
 *
 * - Multiplication cannot overflow.
```

```
     */
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        return a * b;
    }

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

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

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom
message on
     * overflow (when the result is negative).
     *
     * CAUTION: This function is deprecated because it requires allocating memory for
the error
     * message unnecessarily. For custom revert reasons use {trySub}.
```

```
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     *
     * - Subtraction cannot overflow.
     */
    function sub(
        uint256 a,
        uint256 b,
        string memory errorMessage
    ) internal pure returns (uint256) {
        unchecked {
          require(b <= a, errorMessage);
          return a - b;
        }
    }

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

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer
```

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

/**
 * @dev Interface of the ERC20 standard as defined in the EIP.
 */
interface IERC20 {
  /**
   * @dev Returns the amount of tokens in existence.
   */
  function totalSupply() external view returns (uint256);

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

  /**
   * @dev Moves `amount` tokens from the caller's account to `recipient`.
   *
```

```
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transfer(address recipient, uint256 amount) external returns (bool);

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

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

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

    /**
```

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

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

/**
 * @dev Required interface of an ERC721 compliant contract.
 */
interface IERC721 {

  /**
     * @dev Transfers `tokenId` token from `from` to `to`.
     *
     * WARNING: Usage of this method is discouraged, use {safeTransferFrom} whenever
possible.
     *
     * Requirements:
     *
     * - `from` cannot be the zero address.
     * - `to` cannot be the zero address.
     * - `tokenId` token must be owned by `from`.
     * - If the caller is not `from`, it must be approved to move this token by either
{approve} or {setApprovalForAll}.
     *
     * Emits a {Transfer} event.
     */
  function transferFrom(
    address from,
    address to,
    uint256 tokenId
  ) external;
}

/*
 * @dev Provides information about the current execution context, including the
```

```
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with GSN meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 */
abstract contract Context {
  function _msgSender() internal view returns (address payable) {
    return payable(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 Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * By default, the owner account will be the one that deploys the contract. This
 * can later be changed with {transferOwnership}.
 *
 * This module is used through inheritance. It will make available the modifier
 * `onlyOwner`, which can be applied to your functions to restrict their use to
 * the owner.
 */
abstract contract Ownable is Context {
  address private _owner;
  address private _newOwner;

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

  /**
   * @dev Initializes the contract setting the deployer as the initial owner.
   */
  constructor () {
    address msgSender = _msgSender();
```

```
    _owner = msgSender;
    emit OwnershipTransferred(address(0), msgSender);
  }

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

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

  /**
   * @dev Accept the ownership transfer. This is to make sure that the contract is
   * transferred to a working address
   *
   * Can only be called by the newly transfered owner.
   */
  function acceptOwnership() public {
    require(_msgSender() == _newOwner, "Ownable: only new owner can accept
ownership");
    address oldOwner = _owner;
    _owner = _newOwner;
    _newOwner = address(0);
    emit OwnershipTransferred(oldOwner, _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 {
    //SlowMist// This check is quite good in avoiding losing control of the contract
caused by user mistakes
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    _newOwner = newOwner;
```

```
  }
}

/**
 * @dev Contract module which allows children to implement an emergency stop
 * mechanism that can be triggered by an authorized account.
 *
 * This module is used through inheritance. It will make available the
 * modifiers `whenNotFrozen` and `whenFrozen`, which can be applied to
 * the functions of your contract. Note that they will not be pausable by
 * simply including this module, only once the modifiers are put in place.
 */
abstract contract Pausable is Context, Ownable {
  /**
   * @dev Emitted when the freeze is triggered by `account`.
   */
  event Frozen(address account);

  /**
   * @dev Emitted when the freeze is lifted by `account`.
   */
  event Unfrozen(address account);

  bool private _frozen;

  /**
   * @dev Initializes the contract in unfrozen state.
   */
  constructor () {
    _frozen = false;
  }

  /**
   * @dev Returns true if the contract is frozen, and false otherwise.
   */
  function frozen() public view returns (bool) {
    return _frozen;
  }

  /**
   * @dev Modifier to make a function callable only when the contract is not frozen.
   *
   * Requirements:
   *
```

```
     * - The contract must not be frozen.
     */
    modifier whenNotFrozen() {
        require(!frozen(), "Freezable: frozen");
        _;
    }

    /**
     * @dev Modifier to make a function callable only when the contract is frozen.
     *
     * Requirements:
     *
     * - The contract must be frozen.
     */
    modifier whenFrozen() {
        require(frozen(), "Freezable: frozen");
        _;
    }

    /**
     * @dev Triggers stopped state.
     *
     * Requirements:
     *
     * - The contract must not be frozen.
     */
    function _freeze() internal whenNotFrozen {
        _frozen = true;
        emit Frozen(_msgSender());
    }

    /**
     * @dev Returns to normal state.
     *
     * Requirements:
     *
     * - Can only be called by the current owner.
     * - The contract must be frozen.
     */
    function _unfreeze() internal whenFrozen {
        _frozen = false;
        emit Unfrozen(_msgSender());
    }
}
```

```
/**
 * @title Blacklistable Token
 * @dev Allows accounts to be blacklisted by a "blacklister" role
 */
contract Blacklistable is Context, Ownable {

  mapping(address => bool) internal _blacklisted;

  event Blacklisted(address indexed account_);
  event UnBlacklisted(address indexed account_);
  event BlacklisterChanged(address indexed newBlacklister_);

  /**
    * @dev Throws if argument account is blacklisted
    * @param account_ The address to check
    */
  modifier notBlacklisted(address account_) {
    require(!_blacklisted[account_], "Blacklistable: account is blacklisted");
    _;
  }

  /**
    * @dev Checks if account is blacklisted
    * @param account_ The address to check
    */
  function isBlacklisted(address account_) external view returns (bool) {
    return _blacklisted[account_];
  }

  /**
    * @dev Adds account to blacklist
    * @param account_ The address to blacklist
    */
  function blacklist(address account_) external onlyOwner {
    _blacklisted[account_] = true;
    emit Blacklisted(account_);
  }

  /**
    * @dev Removes account from blacklist
    * @param account_ The address to remove from the blacklist
    */
  function unBlacklist(address account_) external onlyOwner {
```

```
    _blacklisted[account_] = false;
    emit UnBlacklisted(account_);
  }
}

/**
 * @dev Enable owner to withdraw tokens that wrongly sent to the contract
 */
contract Withdrawable is Ownable {

  /// @dev Emit `Withdrawn` when owner withdraw fund from the contract
  event Withdrawn(address indexed owner, address indexed recipient, address indexed
token, uint256 value);

  /// @dev withdraw token from contract
  /// @param token_ address of the token, use address(0) to withdraw gas token
  /// @param destination_ recipient address to receive the fund
  /// @param amount_ amount of fund to withdaw
  //Slowmist// The owner role can withdraw the any tokens from this contract to a
specific address
  function withdraw(address token_, address destination_, uint256 amount_) external
onlyOwner {
    require(destination_ != address(0), "Withdrawable: Destination is zero address");

    uint256 availableAmount;
    if(token_ == address(0)) {
      availableAmount = address(this).balance;
    } else {
      availableAmount = IERC20(token_).balanceOf(address(this));
    }

    require(amount_ <= availableAmount, "Withdrawable: Not enough balance");

    if(token_ == address(0)) {
      //SlowMist// There is no upper limit of the gas to execute the low-level call
to withdraw the native tokens
      destination_.call{value:amount_}("");
    } else {
      IERC20(token_).transfer(destination_, amount_);
    }

    emit Withdrawn(_msgSender(), destination_, token_, amount_);
  }
```

```
  /// @dev withdraw NFT from contract
  /// @param token_ address of the token, use address(0) to withdraw gas token
  /// @param destination_ recipient address to receive the fund
  /// @param tokenId_ ID of NFT to withdraw
  //SlowMist// The owner role can withdraw the any NFT tokens from this contract to a specific address
  function withdrawNft(address token_, address destination_, uint256 tokenId_) external onlyOwner {
    require(destination_ != address(0), "Withdrawable: destination is zero address");

    IERC721(token_).transferFrom(address(this), destination_, tokenId_);

    emit Withdrawn(_msgSender(), destination_, token_, 1);
  }
}

/**
 * @dev Implementation of the {IERC20} interface.
 *
 * This implementation is agnostic to the way tokens are created. This means
 * that a supply mechanism has to be added in a derived contract using {_mint}.
 * For a generic mechanism see {ERC20PresetMinterPauser}.
 *
 * TIP: For a detailed writeup see our guide
 * https://forum.zeppelin.solutions/t/how-to-implement-erc20-supply-mechanisms/226[How
 * to implement supply mechanisms].
 *
 * We have followed general OpenZeppelin guidelines: functions revert instead
 * of returning `false` on failure. This behavior is nonetheless conventional
 * and does not conflict with the expectations of ERC20 applications.
 *
 * Additionally, an {Approval} event is emitted on calls to {transferFrom}.
 * This allows applications to reconstruct the allowance for all accounts just
 * by listening to said events. Other implementations of the EIP may not emit
 * these events, as it isn't required by the specification.
 *
 * Finally, the non-standard {decreaseAllowance} and {increaseAllowance}
 * functions have been added to mitigate the well-known issues around setting
 * allowances. See {IERC20-approve}.
 */
contract CUSD is Context, Ownable, Pausable, Blacklistable, Withdrawable, IERC20 {
  using SafeMath for uint256;
```

```
uint256 private _totalSupply;

string private _name;
string private _symbol;
uint8 private _decimals;

mapping (address => uint256) private _balances;

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

address private _minter;

/**
 * @dev Sets the values for {name} and {symbol}, initializes {decimals} with
 * a default value of 18.
 *
 * To select a different value for {decimals}, use {_setupDecimals}.
 *
 * All three of these values are immutable: they can only be set once during
 * construction.
 */
constructor() {
  _name = "Coin98 Ecosystem Currency";
  _symbol = "CUSD";
  _decimals = 18;
  _minter = _msgSender();
}

/**
  @dev Emit when new minter is assigned
 */
event MinterUpdated(address indexed previousMinter, address indexed newMinter);

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

/**
 * @dev Returns the symbol of the token, usually a shorter version of the
 * name.
 */
```

```
function symbol() public view returns (string memory) {
  return _symbol;
}

/**
 * @dev Returns the number of decimals used to get its user representation.
 * For example, if `decimals` equals `2`, a balance of `505` tokens should
 * be displayed to a user as `5,05` (`505 / 10 ** 2`).
 *
 * Tokens usually opt for a value of 18, imitating the relationship between
 * Ether and Wei. This is the value {ERC20} uses, unless {_setupDecimals} is
 * called.
 *
 * NOTE: This information is only used for _display_ purposes: it in
 * no way affects any of the arithmetic of the contract, including
 * {IERC20-balanceOf} and {IERC20-transfer}.
 */
function decimals() public view returns (uint8) {
  return _decimals;
}

/**
 * @dev See {IERC20-totalSupply}.
 */
function totalSupply() public view override returns (uint256) {
  return _totalSupply;
}

/**
 * @dev See {IERC20-balanceOf}.
 */
function balanceOf(address account) public view override returns (uint256) {
  return _balances[account];
}

/**
  @dev Extra address which has priviledge to mint more token
 */
function minter() public view returns (address) {
  return _minter;
}

/**
 * @dev Throws if called by any account other than the owner or minter.
```

```
    */
  modifier onlyMinter() {
    require(owner() == _msgSender() || _minter == _msgSender(), "ERC20: caller is not
minter");
    _;
  }

  /**
   * @dev See {IERC20-transfer}.
   *
   * Requirements:
   *
   * - `recipient` cannot be the zero address.
   * - the caller must have a balance of at least `amount`.
   */
  function transfer(address recipient, uint256 amount) public override
    notBlacklisted(_msgSender())
    notBlacklisted(recipient)
    returns (bool)
  {
    _transfer(_msgSender(), recipient, amount);
    //SlowMist// The return value conforms to the EIP20 specification
    return true;
  }

  /**
   * @dev See {IERC20-allowance}.
   */
  function allowance(address owner, address spender) public view override returns
(uint256) {
    return _allowances[owner][spender];
  }

  /**
   * @dev See {IERC20-approve}.
   *
   * Requirements:
   *
   * - `spender` cannot be the zero address.
   */
  function approve(address spender, uint256 amount) public override
    notBlacklisted(_msgSender())
    notBlacklisted(spender)
    returns (bool)
```

```
  {
    _approve(_msgSender(), spender, amount);
    //SlowMist// The return value conforms to the EIP20 specification
    return true;
  }

  /**
   * @dev See {IERC20-transferFrom}.
   *
   * Emits an {Approval} event indicating the updated allowance. This is not
   * required by the EIP. See the note at the beginning of {ERC20}.
   *
   * 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) public
override
    notBlacklisted(_msgSender())
    notBlacklisted(sender)
    notBlacklisted(recipient)
    returns (bool)
  {
    _transfer(sender, recipient, amount);
    _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount,
"ERC20: transfer amount exceeds allowance"));
    //SlowMist// The return value conforms to the EIP20 specification
    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 {IERC20-approve}.
   *
   * Emits an {Approval} event indicating the updated allowance.
   *
   * Requirements:
   *
   * - `spender` cannot be the zero address.
```

```
   */
  function increaseAllowance(address spender, uint256 addedValue) public
    notBlacklisted(_msgSender())
    notBlacklisted(spender)
    returns (bool)
  {
    _approve(_msgSender(), spender, _allowances[_msgSender()]
[spender].add(addedValue));
    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 {IERC20-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) public
    notBlacklisted(_msgSender())
    notBlacklisted(spender)
    returns (bool)
  {
    _approve(_msgSender(), spender, _allowances[_msgSender()]
[spender].sub(subtractedValue, "ERC20: decreased allowance below zero"));
    return true;
  }

  /**
   * @dev Issues `amount` tokens to the designated `address`.
   *
   * Can only be called by the current owner.
   * See {ERC20-_mint}.
   */
  //SlowMist// The Minter role can mint tokens arbitrarily through the mint function
and there is no upper limit on the amount of tokens that can be minted
  function mint(address account, uint256 amount) public
```

```
    onlyMinter
    notBlacklisted(_msgSender())
    notBlacklisted(account)
{
    _mint(account, amount);
}

/**
 * @dev Destroys `amount` tokens from the caller.
 *
 * See {ERC20-_burn}.
 */
function burn(uint256 amount) public
    notBlacklisted(_msgSender())
{
    _burn(_msgSender(), amount);
}

/**
 * @dev Disable the {transfer}, {mint} and {burn} functions of contract.
 *
 * Can only be called by the current owner.
 * The contract must not be frozen.
 */
function freeze() public onlyOwner {
    _freeze();
}

/**
 * @dev Enable the {transfer}, {mint} and {burn} functions of contract.
 *
 * Can only be called by the current owner.
 * The contract must be frozen.
 */
function unfreeze() public onlyOwner {
    _unfreeze();
}

/**
 * @dev Set new minter
 */
function setMinter(address newMinter) public onlyOwner {
    address oldMinter = _minter;
    _minter = newMinter;
```

```
    emit MinterUpdated(oldMinter, newMinter);
  }

  /**
   * @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), "ERC20: transfer from the zero address");
    //SlowMist// This kind of check is very good, avoiding user mistake leading to
the loss of token during transfer
    require(recipient != address(0), "ERC20: transfer to the zero address");

    _beforeTokenTransfer(sender, recipient, amount);

    _balances[sender] = _balances[sender].sub(amount, "ERC20: transfer amount exceeds
balance");
    _balances[recipient] = _balances[recipient].add(amount);
    emit Transfer(sender, recipient, amount);
  }

  /** @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");

    _beforeTokenTransfer(address(0), account, amount);
```

```
    _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 {
    require(account != address(0), "ERC20: burn from the zero address");

    _beforeTokenTransfer(account, address(0), amount);

    _balances[account] = _balances[account].sub(amount, "ERC20: burn amount exceeds
balance");
    _totalSupply = _totalSupply.sub(amount);
    emit Transfer(account, address(0), amount);
  }

  /**
   * @dev Sets `amount` as the allowance of `spender` over the `owner` s tokens.
   *
   * This 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), "ERC20: approve from the zero address");
    //SlowMist// This kind of check is very good, avoiding user mistake leading to
```

approve errors

```
    require(spender != address(0), "ERC20: approve to the zero address");

    _allowances[owner][spender] = amount;
    emit Approval(owner, spender, amount);
  }

  /**
   * @dev Hook that is called before any transfer of tokens. This includes
   * minting and burning.
   *
   * Calling conditions:
   *
   * - when `from` and `to` are both non-zero, `amount` of ``from``'s tokens
   * will be to transferred to `to`.
   * - when `from` is zero, `amount` tokens will be minted for `to`.
   * - when `to` is zero, `amount` of ``from``'s tokens will be burned.
   * - `from` and `to` are never both zero.
   *
   * To learn more about hooks, head to xref:ROOT:extending-contracts.adoc#using-
hooks[Using Hooks].
   */
  function _beforeTokenTransfer(address from, address to, uint256 amount) internal {
    require(!frozen(), "ERC20: token transfer while frozen");
  }

  function _afterTokenTransfer(
    address from,
    address to,
    uint256 amount
  ) internal {}
}
```

Statement

SlowMist issues this report with reference to the facts that have occurred or existed before the issuance of this report, and only assumes corresponding responsibility based on these.

For the facts that occurred or existed after the issuance, SlowMist is not able to judge the security status of this project, and is not responsible for them. The security audit analysis and other contents of this report are based on the documents and materials provided to SlowMist by the information provider till the date of the insurance report (referred to as "provided information"). SlowMist assumes: The information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed, or inconsistent with the actual situation, the SlowMist shall not be liable for any loss or adverse effect resulting therefrom. SlowMist only conducts the agreed security audit on the security situation of the project and issues this report. SlowMist is not responsible for the background and other conditions of the project.

Official Website

www.slowmist.com

E-mail

team@slowmist.com

Twitter

@SlowMist_Team

Github

https://github.com/slowmist