Smart Contract
Security Audit Report

The SlowMist Security Team received the team's application for smart contract security audit of the METANOA on 2022.04.12. The following are the details and results of this smart contract security audit:

Token Name :

METANOA

The contract address :

https://github.com/metanoa2/metanoa_token/blob/main/contracts/NOA.sol

commit: 80c955646df7ab6fdd2292dc7e4963f6552bfd48

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	Passed
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

NO.	Audit Items	Result
12	Scoping and Declarations Audit	Passed
13	Safety Design Audit	Passed
14	Non-privacy/Non-dark Coin Audit	Passed

Audit Result : Passed

Audit Number : 0X002204150001

Audit Date : 2022.04.12 - 2022.04.15

Audit Team : SlowMist Security Team

Summary conclusion : This is a token contract that contains the tokenVault section. The total amount of contract tokens can be changed, burners can burn their tokens through the burn function. The contract does not have the Overflow and the Race Conditions issue.

During the audit, we found the following information:

1. The owner role can transfer other ERC20 tokens that are wrongly transferred to this contract to his address.
2. The supervisor can transfer the ownership to a new supervisor and the supervisor role can unlock the time lock to release the locked tokens before the lock expires.
3. The owner can add a new locker and the locker role can lock users' tokens at a specified period of time, but the lock release time is before the lock expires a count of period.
4. The supervisor role can unlock the VestingLock to release the locked tokens before the lock expires.

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 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 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 virtual returns (address) {
        return msg.sender;
    }
}

/**
 * @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 `whenNotPaused` and `whenPaused`, 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 {
    /**
     * @dev Emitted when the pause is triggered by `account`.
     */
    event Paused(address account);

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

    bool private _paused;

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

    /**
```

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

    /**
     * @dev Modifier to make a function callable only when the contract is not
paused.
     *
     * Requirements:
     *
     * - The contract must not be paused.
     */
    modifier whenNotPaused() {
        require(!paused(), "Pausable: paused");
        _;
    }

    /**
     * @dev Modifier to make a function callable only when the contract is paused.
     *
     * Requirements:
     *
     * - The contract must be paused.
     */
    modifier whenPaused() {
        require(paused(), "Pausable: not paused");
        _;
    }

    /**
     * @dev Triggers stopped state.
     *
     * Requirements:
     *
     * - The contract must not be paused.
     */
    //SlowMist// Suspending all transactions upon major abnormalities is a
recommended approach
    function _pause() internal virtual whenNotPaused {
        _paused = true;
        emit Paused(_msgSender());
    }
```

```
    /**
     * @dev Returns to normal state.
     *
     * Requirements:
     *
     * - The contract must be paused.
     */
    function _unpause() internal virtual whenPaused {
        _paused = false;
        emit Unpaused(_msgSender());
    }
}

/**
 * @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 `to`.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transfer(address to, 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 `from` to `to` 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 from,
        address to,
        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 Interface for the optional metadata functions from the ERC20 standard.
 *
 * _Available since v4.1._
 */
interface IERC20Metadata is IERC20 {
    /**
     * @dev Returns the name of the token.
     */
    function name() external view returns (string memory);

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

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

/**
 * @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 Contracts guidelines: functions revert
 * instead 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 ERC20 is Context, IERC20, IERC20Metadata {
    mapping(address => uint256) private _balances;

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

    uint256 private _totalSupply;

    string private _name;
    string private _symbol;

    /**
     * @dev Sets the values for {name} and {symbol}.
     *
     * The default value of {decimals} is 18. To select a different value for
     * {decimals} you should overload it.
     *
     * All two of these values are immutable: they can only be set once during
     * construction.
     */
    constructor(string memory name_, string memory symbol_) {
        _name = name_;
        _symbol = symbol_;
    }

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

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

```
    function symbol() public view virtual override 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 this function is
     * overridden;
     *
     * 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 virtual override returns (uint8) {
        return 18;
    }

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

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

    /**
     * @dev See {IERC20-transfer}.
     *
     * Requirements:
     *
     * - `to` cannot be the zero address.
     * - the caller must have a balance of at least `amount`.
     */
```

```
    function transfer(address to, uint256 amount) public virtual override returns
(bool) {
        address owner = _msgSender();
        _transfer(owner, to, amount);
        //SlowMist// The return value conforms to the EIP20 specification
        return true;
    }

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

    /**
     * @dev See {IERC20-approve}.
     *
     * NOTE: If `amount` is the maximum `uint256`, the allowance is not updated on
     * `transferFrom`. This is semantically equivalent to an infinite approval.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function approve(address spender, uint256 amount) public virtual override returns
(bool) {
        address owner = _msgSender();
        _approve(owner, 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}.
     *
     * NOTE: Does not update the allowance if the current allowance
     * is the maximum `uint256`.
     *
     * Requirements:
```

```
     *
     * - `from` and `to` cannot be the zero address.
     * - `from` must have a balance of at least `amount`.
     * - the caller must have allowance for ``from``'s tokens of at least
     * `amount`.
     */
    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) public virtual override returns (bool) {
        address spender = _msgSender();
        _spendAllowance(from, spender, amount);
        _transfer(from, to, amount);
        //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 virtual
returns (bool) {
        address owner = _msgSender();
        _approve(owner, spender, allowance(owner, spender) + 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
virtual returns (bool) {
        address owner = _msgSender();
        uint256 currentAllowance = allowance(owner, spender);
        require(currentAllowance >= subtractedValue, "ERC20: decreased allowance
below zero");
        unchecked {
            _approve(owner, spender, currentAllowance - subtractedValue);
        }

        return true;
    }

    /**
     * @dev Moves `amount` of tokens from `sender` to `recipient`.
     *
     * This 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:
     *
     * - `from` cannot be the zero address.
     * - `to` cannot be the zero address.
     * - `from` must have a balance of at least `amount`.
     */
    function _transfer(
        address from,
        address to,
        uint256 amount
    ) internal virtual {
        require(from != 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(to != address(0), "ERC20: transfer to the zero address");
```

```
        _beforeTokenTransfer(from, to, amount);

        uint256 fromBalance = _balances[from];
        require(fromBalance >= amount, "ERC20: transfer amount exceeds balance");
        unchecked {
            _balances[from] = fromBalance - amount;
        }
        _balances[to] += amount;

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

        _totalSupply += amount;
        _balances[account] += 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");

        uint256 accountBalance = _balances[account];
```

```
        require(accountBalance >= amount, "ERC20: burn amount exceeds balance");
        unchecked {
            _balances[account] = accountBalance - amount;
        }
        _totalSupply -= 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 virtual {
        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 Updates `owner` s allowance for `spender` based on spent `amount`.
     *
     * Does not update the allowance amount in case of infinite allowance.
     * Revert if not enough allowance is available.
     *
     * Might emit an {Approval} event.
     */
```

```
    function _spendAllowance(
        address owner,
        address spender,
        uint256 amount
    ) internal virtual {
        uint256 currentAllowance = allowance(owner, spender);
        if (currentAllowance != type(uint256).max) {
            require(currentAllowance >= amount, "ERC20: insufficient allowance");
            unchecked {
                _approve(owner, spender, currentAllowance - 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 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 virtual {}
}

/**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * By default, the owner account will be the one that deploys the contract. This
 * can later be changed with {transferOwnership}.
 *
```

```
 * This module is used through inheritance. It will make available the modifier
 * `onlyOwner`, which can be applied to your functions to restrict their use to
 * the owner.
 */
abstract contract Ownable is Context {
    address private _owner;

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

    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    constructor() {
        _transferOwnership(_msgSender());
    }

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

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

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Internal function without access restriction.
     */
    function _transferOwnership(address newOwner) internal virtual {
        address oldOwner = _owner;
        _owner = newOwner;
        emit OwnershipTransferred(oldOwner, newOwner);
    }
}

/**
```

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

    event SupervisorOwnershipTransferred(address indexed previousSupervisor, address
indexed newSupervisor);

    /**
     * @dev Initializes the contract setting the deployer as the initial supervisor.
     */
    constructor() {
        _transferSupervisorOwnership(_msgSender());
    }

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

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

    /**
     * @dev Transfers supervisor ownership of the contract to a new account
(`newSupervisor`).
     * Internal function without access restriction.
```

```
     */
    function _transferSupervisorOwnership(address newSupervisor) internal virtual {
        address oldSupervisor = _supervisor;
        _supervisor = newSupervisor;
        emit SupervisorOwnershipTransferred(oldSupervisor, newSupervisor);
    }
}

/**
 * @dev Extension of {ERC20} that allows token holders to destroy both their own
 * tokens and those that they have an allowance for, in a way that can be
 * recognized off-chain (via event analysis).
 */
abstract contract Burnable is Context {
    mapping(address => bool) private _burners;

    event BurnerAdded(address indexed account);
    event BurnerRemoved(address indexed account);

    /**
     * @dev Returns whether the address is burner.
     */
    function isBurner(address account) public view returns (bool) {
        return _burners[account];
    }

    /**
     * @dev Throws if called by any account other than the burner.
     */
    modifier onlyBurner() {
        require(_burners[_msgSender()], "Burnable: caller is not the burner");
        _;
    }

    /**
     * @dev Add burner, only owner can add burner.
     */
    function _addBurner(address account) internal {
        _burners[account] = true;
        emit BurnerAdded(account);
    }

    /**
     * @dev Remove operator, only owner can remove operator
```

```
     */
    function _removeBurner(address account) internal {
        _burners[account] = false;
        emit BurnerRemoved(account);
    }
}

/**
 * @dev Contract for freezing mechanism.
 * Owner can add freezed account.
 * Supervisor can remove freezed account.
 */
contract Freezable is Context {
    mapping(address => bool) private _freezes;

    event Freezed(address indexed account);
    event Unfreezed(address indexed account);

    /**
     * @dev Freeze account, only owner can freeze
     */
    function _freeze(address account) internal {
        _freezes[account] = true;
        emit Freezed(account);
    }

    /**
     * @dev Unfreeze account, only supervisor can unfreeze
     */
    function _unfreeze(address account) internal {
        _freezes[account] = false;
        emit Unfreezed(account);
    }

    /**
     * @dev Returns whether the address is freezed.
     */
    function isFreezed(address account) public view returns (bool) {
        return _freezes[account];
    }
}

/**
 * @dev Contract for locking mechanism.
```

```
 * Locker can add locked account.
 * Supervisor can remove locked account.
 */
contract Lockable is Context {
    struct TimeLock {
        uint256 amount;
        uint256 expiresAt;
    }

    struct VestingLock {
        uint256 amount;
        uint256 startsAt;
        uint256 period;
        uint256 count;
    }

    mapping(address => bool) private _lockers;
    mapping(address => TimeLock[]) private _timeLocks;
    mapping(address => VestingLock) private _vestingLocks;

    event LockerAdded(address indexed account);
    event LockerRemoved(address indexed account);
    event TimeLocked(address indexed account);
    event TimeUnlocked(address indexed account);
    event VestingLocked(address indexed account);
    event VestingUnlocked(address indexed account);

    /**
     * @dev Throws if called by any account other than the locker.
     */
    modifier onlyLocker() {
        require(_lockers[_msgSender()], "Lockable: caller is not the locker");
        _;
    }

    /**
     * @dev Returns whether the address is locker.
     */
    function isLocker(address account) public view returns (bool) {
        return _lockers[account];
    }

    /**
     * @dev Add locker, only owner can add locker
```

```
     */
    function _addLocker(address account) internal {
        _lockers[account] = true;
        emit LockerAdded(account);
    }

    /**
     * @dev Remove locker, only owner can remove locker
     */
    function _removeLocker(address account) internal {
        _lockers[account] = false;
        emit LockerRemoved(account);
    }

    /**
     * @dev Add time lock, only locker can add
     */
    function _addTimeLock(
        address account,
        uint256 amount,
        uint256 expiresAt
    ) internal {
        require(amount > 0, "Time Lock: lock amount is 0");
        require(expiresAt > block.timestamp, "Time Lock: invalid expire date");
        _timeLocks[account].push(TimeLock(amount, expiresAt));
        emit TimeLocked(account);
    }

    /**
     * @dev Remove time lock, only supervisor can remove
     * @param account The address want to remove time lock
     * @param index Time lock index
     */
    function _removeTimeLock(address account, uint8 index) internal {
        require(_timeLocks[account].length > index && index >= 0, "Time Lock: invalid
index");

        uint256 len = _timeLocks[account].length;
        if (len - 1 != index) {
            // if it is not last item, swap it
            _timeLocks[account][index] = _timeLocks[account][len - 1];
        }
        _timeLocks[account].pop();
        emit TimeUnlocked(account);
```

```
    }

    /**
     * @dev Get time lock array length
     * @param account The address want to know the time lock length.
     * @return time lock length
     */
    function getTimeLockLength(address account) public view returns (uint256) {
        return _timeLocks[account].length;
    }

    /**
     * @dev Get time lock info
     * @param account The address want to know the time lock state.
     * @param index Time lock index
     * @return time lock info
     */
    function getTimeLock(address account, uint8 index) public view returns (uint256,
uint256) {
        require(_timeLocks[account].length > index && index >= 0, "Time Lock: invalid
index");
        return (_timeLocks[account][index].amount, _timeLocks[account]
[index].expiresAt);
    }

    /**
     * @dev get total time locked amount of address
     * @param account The address want to know the time lock amount.
     * @return time locked amount
     */
    function getTimeLockedAmount(address account) public view returns (uint256) {
        uint256 timeLockedAmount = 0;

        uint256 len = _timeLocks[account].length;
        for (uint256 i = 0; i < len; i++) {
            if (block.timestamp < _timeLocks[account][i].expiresAt) {
                timeLockedAmount = timeLockedAmount + _timeLocks[account][i].amount;
            }
        }
        return timeLockedAmount;
    }

    /**
     * @dev Add vesting lock, only locker can add
```

```
     * @param account vesting lock account.
     * @param amount vesting lock amount.
     * @param startsAt vesting lock release start date.
     * @param period vesting lock period. End date is startsAt + (period - 1) * count
     * @param count vesting lock count. If count is 1, it works like a time lock
     */
    function _addVestingLock(
        address account,
        uint256 amount,
        uint256 startsAt,
        uint256 period,
        uint256 count
    ) internal {
        require(account != address(0), "Vesting Lock: lock from the zero address");
        require(startsAt > block.timestamp, "Vesting Lock: must set after now");
        require(amount > 0, "Vesting Lock: amount is 0");
        require(period > 0, "Vesting Lock: period is 0");
        require(count > 0, "Vesting Lock: count is 0");
        _vestingLocks[account] = VestingLock(amount, startsAt, period, count);
        emit VestingLocked(account);
    }

    /**
     * @dev Remove vesting lock, only supervisor can remove
     * @param account The address want to remove the vesting lock
     */
    function _removeVestingLock(address account) internal {
        _vestingLocks[account] = VestingLock(0, 0, 0, 0);
        emit VestingUnlocked(account);
    }

    /**
     * @dev Get vesting lock info
     * @param account The address want to know the vesting lock state.
     * @return vesting lock info
     */
    function getVestingLock(address account)
        public
        view
        returns (
            uint256,
            uint256,
            uint256,
            uint256
```

```
        )
    {
        return (_vestingLocks[account].amount, _vestingLocks[account].startsAt,
_vestingLocks[account].period, _vestingLocks[account].count);
    }

    /**
     * @dev Get total vesting locked amount of address, locked amount will be
released by 100%/months
     * If months is 5, locked amount released 20% per 1 month.
     * @param account The address want to know the vesting lock amount.
     * @return vesting locked amount
     */
    function getVestingLockedAmount(address account) public view returns (uint256) {
        uint256 vestingLockedAmount = 0;
        uint256 amount = _vestingLocks[account].amount;
        if (amount > 0) {
            uint256 startsAt = _vestingLocks[account].startsAt;
            uint256 period = _vestingLocks[account].period;
            uint256 count = _vestingLocks[account].count;
            uint256 expiresAt = startsAt + period * (count - 1);
            uint256 timestamp = block.timestamp;
            if (timestamp < startsAt) {
                vestingLockedAmount = amount;
            } else if (timestamp < expiresAt) {
                vestingLockedAmount = (amount * ((expiresAt - timestamp) / period +
1)) / count;
            }
        }
        return vestingLockedAmount;
    }

    /**
     * @dev Get all locked amount
     * @param account The address want to know the all locked amount
     * @return all locked amount
     */
    function getAllLockedAmount(address account) public view returns (uint256) {
        return getTimeLockedAmount(account) + getVestingLockedAmount(account);
    }
}

/**
 * @dev Contract for METANOA Token
```

```
 */
contract NOA is Pausable, Ownable, Supervisable, Burnable, Freezable, Lockable, ERC20
{
    uint256 private constant _initialSupply = 1_000_000_000e18;

    constructor() ERC20("METANOA", "NOA") {
        _mint(_msgSender(), _initialSupply);
    }

    /**
     * @dev Recover ERC20 coin in contract address.
     * @param tokenAddress The token contract address
     * @param tokenAmount Number of tokens to be sent
     */
    //SlowMist// The owner role can transfer other ERC20 tokens that are wrongly
transferred to the contract to the owner address
    function recoverERC20(address tokenAddress, uint256 tokenAmount) public onlyOwner
{
        IERC20(tokenAddress).transfer(owner(), tokenAmount);
    }

    /**
     * @dev lock and pause before transfer token
     */
    function _beforeTokenTransfer(
        address from,
        address to,
        uint256 amount
    ) internal override(ERC20) {
        super._beforeTokenTransfer(from, to, amount);

        require(!isFreezed(from), "Freezable: token transfer from freezed account");
        require(!isFreezed(to), "Freezable: token transfer to freezed account");
        require(!isFreezed(_msgSender()), "Freezable: token transfer called from
freezed account");
        require(!paused(), "Pausable: token transfer while paused");
        require(balanceOf(from) - getAllLockedAmount(from) >= amount, "Lockable:
insufficient transfer amount");
    }

    /**
     * @dev Leaves the contract without owner. It will not be possible to call
     * `onlyOwner` functions anymore. Can only be called by the current owner.
     *
```

```
     * NOTE: Renouncing ownership will leave the contract without an owner,
     * thereby removing any functionality that is only available to the owner.
     */
    function renounceOwnership() public onlyOwner whenNotPaused {
        _transferOwnership(address(0));
    }

    /**
     * @dev only supervisor can renounce supervisor ownership
     */
    function renounceSupervisorOwnership() public onlySupervisor whenNotPaused {
        _transferSupervisorOwnership(address(0));
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public onlyOwner whenNotPaused {
        //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");
        _transferOwnership(newOwner);
    }

    /**
     * @dev only supervisor can transfer supervisor ownership
     */
    function transferSupervisorOwnership(address newSupervisor) public onlySupervisor
whenNotPaused {
        require(newSupervisor != address(0), "Supervisable: new supervisor is the
zero address");
        _transferSupervisorOwnership(newSupervisor);
    }

    /**
     * @dev pause all coin transfer
     */
    //SlowMist// Suspending all transactions upon major abnormalities is a
recommended approach
    function pause() public onlyOwner whenNotPaused {
        _pause();
    }
```

```
/**
 * @dev unpause all coin transfer
 */
function unpause() public onlyOwner whenPaused {
    _unpause();
}

/**
 * @dev only owner can lock account
 */
function freeze(address account) public onlyOwner whenNotPaused {
    _freeze(account);
}

/**
 * @dev only supervisor can unfreeze account
 */
function unfreeze(address account) public onlySupervisor whenNotPaused {
    _unfreeze(account);
}

/**
 * @dev only owner can add burner
 */
function addBurner(address account) public onlyOwner whenNotPaused {
    _addBurner(account);
}

/**
 * @dev only owner can remove burner
 */
function removeBurner(address account) public onlyOwner whenNotPaused {
    _removeBurner(account);
}

/**
 * @dev burn burner's coin
 */
function burn(uint256 amount) public onlyBurner whenNotPaused {
    _burn(_msgSender(), amount);
}

/**
 * @dev only owner can add locker
```

```
     */
    function addLocker(address account) public onlyOwner whenNotPaused {
        _addLocker(account);
    }

    /**
     * @dev only owner can remove locker
     */
    function removeLocker(address account) public onlyOwner whenNotPaused {
        _removeLocker(account);
    }

    /**
     * @dev only locker can add time lock
     */
    function addTimeLock(
        address account,
        uint256 amount,
        uint256 expiresAt
    ) public onlyLocker whenNotPaused {
        _addTimeLock(account, amount, expiresAt);
    }

    /**
     * @dev only supervisor can remove time lock
     */
    //SlowMist// The supervisor role can unlock the time lock to release the locked tokens before the lock expires
    function removeTimeLock(address account, uint8 index) public onlySupervisor whenNotPaused {
        _removeTimeLock(account, index);
    }

    /**
     * @dev only locker can add vesting lock
     */
    //SlowMist// The locker role can lock users' tokens at a specified period of time, but the lock release time is before the lock expires a count of period
    function addVestingLock(
        address account,
        uint256 startsAt,
        uint256 period,
        uint256 count
    ) public onlyLocker whenNotPaused {
```

```
        _addVestingLock(account, balanceOf(account), startsAt, period, count);
    }

    /**
     * @dev only supervisor can remove vesting lock
     */
    //SlowMist// The supervisor role can unlock the VestingLock to release the locked tokens before the lock expires
    function removeVestingLock(address account) public onlySupervisor whenNotPaused {
        _removeVestingLock(account);
    }
}
```

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