SLOWMIST

Smart Contract Security Audit Report

The SlowMist Security Team received the team's application for smart contract security audit of the Orbiter Token on 2025.01.16. The following are the details and results of this smart contract security audit:

Token Name :

Orbiter Token

The contract address :

https://github.com/Orbiter-Finance/orbiter-token

commit: 38896dee78281c403af40155619b6dfe5fee4381

Proxy Address: https://etherscan.io/address/0x4af322ff4a6f2858f6b51e546b9ec499654493c5

Implementation Address: https://etherscan.io/address/0x44d64583a7bcacaa99537042a7063950663b2bf4

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 Audit	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
12	Scoping and Declarations Audit	Passed

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

Audit Result : Passed

Audit Number : 0X002501200001

Audit Date : 2025.01.16 - 2025.01.20

Audit Team : SlowMist Security Team

Summary conclusion : This is a token contract that contains the locked grant token section and does not contain the the dark coin functions. The total amount of contract tokens can be changed. The contract does not have the Overflow and the Race Conditions issue.

During the audit, we found the following information:

1. The DEFAULT_ADMIN_ROLE can mint tokens through the mint function and there is an upper limit on the amount of tokens.
2. The owner of the LockedTokenGov contract can create a LockedTokenGrant contract through grantLockedTokens, specify the distributed ERC20 tokens for this contract and transfer them to it. At the same time, a Recipient address will be set for this contract, and only the Recipient address can release the locked tokens through LockedTokenGrant.
3. The three contracts are imported into the UUPSUpgradeable contract so that the Proxy admin can arbitrarily upgrade them, which may create a risk of being overprivileged.

After communicating with the project team, they transferred the DEFAULT_ADMIN_ROLE to a multi-sig wallet address which is controlled by three EOA addresses.

The source code:

LockedTokenGrant.sol

```
// SPDX-License-Identifier: MIT
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.8.26;
```

```
import {ReachedGrantAmountLimit, InvalidAmount} from "src/libraries/Error.sol";
import {ERC20Upgradeable} from "@openzeppelin/contracts-
upgradeable/token/ERC20/ERC20Upgradeable.sol";
import {OwnableUpgradeable} from "@openzeppelin/contracts-
upgradeable/access/OwnableUpgradeable.sol";
import {Initializable} from "@openzeppelin/contracts-
upgradeable/proxy/utils/Initializable.sol";
import {UUPSUpgradeable} from "@openzeppelin/contracts-
upgrasodeable/proxy/utils/UUPSUpgradeable.sol";

/**
  This Contract holds a grant of locked tokens and gradually releases the tokens to
its recipient.

  This contract should be deployed through the {LockedTokenCommon} contract,
  The global lock expiration time may be adjusted through the {LockedTokenCommon}
contract.

  The {LockedTokenGrant} is initialized  with the following parameters:
  `address tokenAddress`: The address of Orbiter token ERC20 contract.
  `address _recipient`: The owner of the grant.
  `uint256 _grantAmount`: The amount of tokens granted in this grant.

  Token Release Operation:
  ======================
  - Tokens are owned by the `recipient`. They cannot be revoked.
  - At any given time the recipient can release any amount of tokens
    as long as the specified amount is available for release.
  - The amount of tokens available for release is the following:
```

availableAmount = unlockedTokens - releasedTokens;

```
- Only the recipient is allowed to trigger release of tokens.
- The released tokens can be transferred ONLY to the recipient address.
*/
contract LockedTokenGrant is
  Initializable,
  OwnableUpgradeable,
  UUPSUpgradeable
{
  ERC20Upgradeable public token;
  uint256 public grantAmount; // Total grant amount of token
  uint256 public releasedTokens; // Total released amount of token
  uint256 public unlockedTokens;
  address public recipient;
```

```
    event TokensSentToRecipient(
        address indexed recipient,
        address indexed grantContract,
        uint256 amountSent,
        uint256 aggregateSent
    );

    modifier onlyRecipient() {
        require(msg.sender == recipient, "Only recipient");
        _;
    }

    /// @custom:oz-upgrades-unsafe-allow constructor
    constructor() {
        _disableInitializers();
    }

    function initialize(
        address tokenAddress,
        address _recipient,
        uint256 _grantAmount
    ) public initializer {
        __Ownable_init(msg.sender);
        __UUPSUpgradeable_init();

        token = ERC20Upgradeable(tokenAddress);
        recipient = _recipient;
        grantAmount = _grantAmount;
    }

    /*
      Returns the available tokens for release.
    */
    function availableTokens() public view returns (uint256) {
        return unlockedTokens - releasedTokens;
    }

    function setUnlockedTokensAmount(
        uint256 requestedAmount
    ) external onlyOwner {
        if (unlockedTokens + requestedAmount > grantAmount) {
            revert ReachedGrantAmountLimit();
        }
        unlockedTokens += requestedAmount;
    }

    /*
      Transfers `requestedAmount` tokens (if available) to the `recipient`.
    */
```

```
  function releaseTokens(uint256 requestedAmount) external onlyRecipient {
      if (requestedAmount > availableTokens()) {
          revert InvalidAmount();
      }

      releasedTokens += requestedAmount;
      //SlowMist// It's recommended to add the return vaule check
      token.transfer(recipient, requestedAmount);
      emit TokensSentToRecipient(
          recipient,
          address(this),
          requestedAmount,
          releasedTokens
      );
  }

  function _authorizeUpgrade(
      address newImplementation
  ) internal view override onlyOwner {
      (newImplementation);
  }
}
```

LockedTokenGov.sol

```
// SPDX-License-Identifier: MIT
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.8.26;

import {TransferFailed} from "src/libraries/Error.sol";
import {LockedTokenGrant} from "src/LockedTokenGrant.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {Initializable} from "@openzeppelin/contracts-
upgradeable/proxy/utils/Initializable.sol";
import {UUPSUpgradeable} from "@openzeppelin/contracts-
upgradeable/proxy/utils/UUPSUpgradeable.sol";
import {OwnableUpgradeable} from "@openzeppelin/contracts-
upgradeable/access/OwnableUpgradeable.sol";

/**
  The {LockedTokenCommon} contract serves one purposes:
  1. Allocate locked token grants in {LockedTokenGrant} contracts.

  Roles:
  =====
  1. At initializtion time, admin is defined as owner.
  Grant Locked Tokens:
  ===================
  Locked token grants are granted using the `grantLockedTokens` here.
  The arguments passed are:
  - recipient - The address of the tokens "owner". When the tokens get unlocked, they
can be released
                to the recipient address, and only there.
  - amount    - The number of tokens to be transfered onto the grant contract upon
creation.
  - allocationPool - The {LockedTokenCommon} doesn't hold liquidity from which it can
grant the tokens,
                     but rather uses an external LP for that. The `allocationPool` is
the address of the LP
                     from which the tokens shall be allocated. The {LockedTokenCommon}
must have sufficient allowance
                     on the `allocationPool` so it can transfer funds from it onto the
creatred grant contract.

    Flow: The {LockedTokenCommon} deploys the contract of the new {LockedTokenGrant},
          transfer the grant amount from the allocationPool onto the new grant,
          and register the new grant in a mapping.
*/
contract LockedTokenGov is Initializable, OwnableUpgradeable, UUPSUpgradeable {
    IERC20 public token;
    LockedTokenGrant public grantImplementation;

    // Maps recipient to its locked grant contract.
    mapping(address => address) public grantByRecipient;

    event LockedTokenGranted(
        address indexed recipient,
        address indexed grantContract,
        uint256 grantAmount
    );

    /// @custom:oz-upgrades-unsafe-allow constructor
    constructor() {
        _disableInitializers();
    }

    function initialize(
        address tokenAddress,
        address admin
    ) public initializer {
        __Ownable_init(admin);
        __UUPSUpgradeable_init();

        token = IERC20(tokenAddress);
        grantImplementation = new LockedTokenGrant();
    }

    function setUnlockedTokensAmount(
        address recipient,
        uint256 unlockedTokensAmount
    ) external onlyOwner {
        require(grantByRecipient[recipient] != address(0x0));
        LockedTokenGrant(grantByRecipient[recipient]).setUnlockedTokensAmount(
            unlockedTokensAmount
        );
    }

    /**
      Deploys a LockedTokenGrant and transfers `grantAmount` tokens onto it.
      Returns the address of the LockedTokenGrant contract.

      Tokens owned by the {LockedTokenGrant} are initially locked, and can only be
used for staking.
      The tokens gradually unlocked and can be transferred to the `recipient`.
    */
    function grantLockedTokens(
        address recipient,
        uint256 grantAmount,
        address allocationPool
    ) external onlyOwner returns (address) {
        require(grantByRecipient[recipient] == address(0x0), "ALREADY_GRANTED");

        address grantAddress = address(
            LockedTokenGrant(
                payable(
                    new ERC1967Proxy(
                        address(grantImplementation),
                        abi.encodeCall(
                            LockedTokenGrant.initialize,
                            (address(token), recipient, grantAmount)
                        )
                    )
                )
            )
        );

        if (!token.transferFrom(allocationPool, grantAddress, grantAmount)) {
            revert TransferFailed();
        }

        grantByRecipient[recipient] = grantAddress;
        emit LockedTokenGranted(recipient, grantAddress, grantAmount);
        return grantAddress;
    }

    function _authorizeUpgrade(
        address newImplementation
    ) internal view override onlyOwner {
        (newImplementation);
    }
}
```

OrbiterToken.sol

```
// SPDX-License-Identifier: MIT
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.8.26;

import {InvalidAddr, ReachedMaximumSupplyLimit} from "src/libraries/Error.sol";
import {ERC20PermitUpgradeable} from "@openzeppelin/contracts-
upgradeable/token/ERC20/extensions/ERC20PermitUpgradeable.sol";
import {AccessControlUpgradeable} from "@openzeppelin/contracts-
upgradeable/access/AccessControlUpgradeable.sol";
import {Initializable} from "@openzeppelin/contracts-
upgradeable/proxy/utils/Initializable.sol";
import {UUPSUpgradeable} from "@openzeppelin/contracts-
upgradeable/proxy/utils/UUPSUpgradeable.sol";

contract OrbiterToken is
    Initializable,
    ERC20PermitUpgradeable,
    AccessControlUpgradeable,
    UUPSUpgradeable
{
    // bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");
    // bytes32 public constant BURNER_ROLE = keccak256("BURNER_ROLE");

    uint256 public MAX_SUPPLY_AMOUNT;

    /// @custom:oz-upgrades-unsafe-allow constructor
    constructor() {
        _disableInitializers();
    }

    /**
     * @dev Initialize function (replaces constructor for upgradeable contracts).
     */
    function initialize(
        string memory name,
        string memory symbol,
        uint256 max_supply_amount,
        address admin
    ) public initializer {
        __ERC20_init(name, symbol);
        __EIP712_init(name, "1");
        __ERC20Permit_init(name);
        __AccessControl_init();
        __UUPSUpgradeable_init();

        if (admin == address(0)) {
            revert InvalidAddr();
        }

        MAX_SUPPLY_AMOUNT = max_supply_amount;

        _grantRole(DEFAULT_ADMIN_ROLE, admin);
    }

    /**
     * @notice Mints `_amount` of tokens to `_account`.
     * @param _account The account to mint tokens to.
     * @param _amount The amount of tokens to mint.
     * @dev Reverts if the caller does not have the DEFAULT_ADMIN_ROLE.
     */
    //SlowMist// The DEFAULT_ADMIN_ROLE can call mint token arbitrarily, but there is
an upper limit for the mint amount of tokens called MAX_SUPPLY_AMOUNT
    function mint(
        address _account,
        uint256 _amount
    ) public onlyRole(DEFAULT_ADMIN_ROLE) {
        if (totalSupply() + _amount > MAX_SUPPLY_AMOUNT) {
            revert ReachedMaximumSupplyLimit();
        }
        _mint(_account, _amount);
    }

    // /**
    //  * @notice Burns `_value` of tokens from `_account`.
    //  * @param _account The account to burn tokens from.
    //  * @param _value The amount of tokens to burn.
    //  * @dev Reverts if the caller does not have the BURNER_ROLE.
    //  */
    // function burn(
    //     address _account,
    //     uint256 _value
    // ) public onlyRole(BURNER_ROLE) {
    //     _burn(_account, _value);
    // }
    // Upgrade authorization (only DEFAULT_ADMIN_ROLE)
    function _authorizeUpgrade(
        address newImplementation
    ) internal override onlyRole(DEFAULT_ADMIN_ROLE) {}

    function transferOwnership(
        address newOwner
    ) external onlyRole(DEFAULT_ADMIN_ROLE) {
        grantRole(DEFAULT_ADMIN_ROLE, newOwner);
        renounceRole(DEFAULT_ADMIN_ROLE, msg.sender);
    }
}
```

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