Smart Contract Security Audit Report

The SlowMist Security Team received the Galxe team's application for smart contract security audit of the Galxe G Token on 2024.05.10. The following are the details and results of this smart contract security audit:

Token Name :

Galxe G Token

The contract address :

GravityTokenG.sol:

https://etherscan.io/address/0x9C7BEBa8F6eF6643aBd725e45a4E8387eF260649

https://bscscan.com/address/0x9C7BEBa8F6eF6643aBd725e45a4E8387eF260649

https://basescan.org/address/0x9c7beba8f6ef6643abd725e45a4e8387ef260649

TokenUpgrader.sol:

https://etherscan.io/address/0x249aC00402716b7bf6d6ED24531d7B4C10788942

https://bscscan.com/address/0x249aC00402716b7bf6d6ED24531d7B4C10788942

TokenVesting contract is not yet deployed:

https://github.com/Galxe/erc20-g-token/blob/2eda9abf9ca567f106594101bddf59108a760cc8/contracts/TokenVesting.sol

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

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

Audit Result : Passed

Audit Number : 0X002405130001

Audit Date : 2024.05.10 - 2024.05.13

Audit Team : SlowMist Security Team

Summary conclusion : This is a token contract that contains the vesting, upgrade, and LimitedMinterManager module.The TokenVesting can managing the phased release of a specific ERC20 token.The TokenUpgrader can upgrade from old tokens to new tokens. The total amount of contract tokens can be changed, only minter and owner roles can mint tokens. 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 change the name of the token through the setName function.
2. The owner role can set the minting limit for the minter through the setMinterLimit function.
3. The owner role can remove the minter through the removeMinterByIndexHint function.
4. The owner role can withdraw ERC20 token from the TokenUpgrader contract through the withdrawERC20Token function.
5. The owner role can withdraw ETH from the TokenUpgrader contract through the withdrawETH function.

6. The owner role can initialize the TokenUpgrader contract through the initialize function.
7. The owner role can release the vested tokens through the release function.
8. The owner role can recover other ERC20 tokens from the TokenVesting contract through the recoverOtherERC20 function.
9. The owner role can recover native token from the TokenVesting contract through the recoverNativeToken function.

Permissions for the owner role of tokens and Upgrader contracts have now been transferred to multi-signature addresses.

The source code:

GravityTokenG.sol

```
// SPDX-License-Identifier: MIT
// Compatible with OpenZeppelin Contracts ^5.0.0
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.8.24;

import { ERC20 } from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import { ERC20Burnable } from
"@openzeppelin/contracts/token/ERC20/extensions/ERC20Burnable.sol";
import { ERC20Pausable } from
"@openzeppelin/contracts/token/ERC20/extensions/ERC20Pausable.sol";
import { ERC20Permit } from
"@openzeppelin/contracts/token/ERC20/extensions/ERC20Permit.sol";
import { Ownable2Step, Ownable } from
"@openzeppelin/contracts/access/Ownable2Step.sol";

import { LimitedMinterManager } from "./LimitedMinterManager.sol";

/// @title Gravity G Token (ERC20) Contract
/// @author Galxe Team
/// @notice G token supports:
/// - pausable transfers, minting and burning
/// - ERC20Permit signatures for approvals
/// - native cross-chain ERC20 by supporting limited minter management for bridges.
/// @custom:security-contact security@galxe.com
contract GravityTokenG is ERC20, ERC20Burnable, ERC20Pausable, ERC20Permit,
LimitedMinterManager, Ownable2Step {
    string private _newName;

    constructor(address initialAdmin) ERC20("Gravity", "G") ERC20Permit("Gravity")
```

```
Ownable(initialAdmin) {
        _newName = super.name();
    }

    /// @notice Pauses the contract.
    //SlowMist// Suspending all transactions upon major abnormalities is a recommended
approach.
    function pause() public onlyOwner {
        _pause();
    }

    /// @notice Unpauses the contract.
    function unpause() public onlyOwner {
        _unpause();
    }

    /// @notice Returns the name of the token.
    /// @dev This is a custom function that overrides the OpenZeppelin function.
    function name() public view override returns (string memory) {
        return _newName;
    }

    /// @notice Sets the name of the token.
    /// @dev This gives the owner the ability to change the name of the token.
    //SlowMist// The owner role can change the name of the token
    function setName(string memory newName) public onlyOwner {
        _newName = newName;
    }

    /// ownerMint can only be called by the owner for initial token distribution
    /// @param to token receiver
    /// @param amount amount of tokens to mint
    //SlowMist// The owner role can mint tokens for a given address
    function ownerMint(address to, uint256 amount) public onlyOwner {
        _mint(to, amount);
    }

    // Overrides required by Solidity.
    function _update(address from, address to, uint256 value) internal
override(ERC20, ERC20Pausable) {
        super._update(from, to, value);
    }

    // cross chain bridge minting

    /// @notice Sets the minting limits for a minter
    /// @param _minter the address of the minter
    /// @param _mintingLimit the limited amount of tokens that can be minted in a
period
```

```
    /// @param _duration the duration window for minting limit.
    //SlowMist// The owner role can set the minting limit for the minter
    function setMinterLimit(address _minter, uint256 _mintingLimit, uint256
_duration) public onlyOwner {
        _setMinterLimit(_minter, _mintingLimit, _duration);
    }

    /// @notice Removes a minter
    /// @dev Can only be called by the owner. Since add/remove minters can only be
done by the owner,
    ///      this indexHint is safe from DoS attacks.
    /// @param _minter The address of the minter we are deleting
    /// @param _indexHint The index hint of the minter
    //SlowMist// The owner role can remove the minter
    function removeMinterByIndexHint(address _minter, uint256 _indexHint) public
onlyOwner {
        _removeMinterByIndexHint(_minter, _indexHint);
    }

    /// @notice Mints tokens for a user by minter
    /// @dev Can only be called by a bridge
    /// @param _user The address of the user who needs tokens minted
    /// @param _amount The amount of tokens being minted
    function mint(address _user, uint256 _amount) public {
        // will revert if not enough limits
        _minterMint(msg.sender, _amount);
        _mint(_user, _amount);
    }
}
```

LimitedMinterManager.sol

```
// SPDX-License-Identifier: MIT
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.8.24;

import { ILimitedMinterManager } from "./interfaces/ILimitedMinterManager.sol";

contract LimitedMinterManager is ILimitedMinterManager {
    /// @notice Maps minter address to minter configurations
    mapping(address => MinterConfig) private _minterConfigs;
    /// @notice Array of minters, making minters enumerable.
    address[] private _minters;

    /// @notice Get the total number of minters
    function getMinterCount() public view returns (uint256) {
        return _minters.length;
```

```
    }

    /// @notice Retrieve the address of a minter by index
    /// @param _index The index of the minter
    function getMinterByIndex(uint256 _index) public view returns (address) {
        if (_index >= _minters.length) {
            revert ILimitedMinterManager_InvalidIndex();
        }
        return _minters[_index];
    }

    /// @notice Retrieve the minter configuration
    /// @param _minter The address of the minter
    function getMinterConfig(address _minter) public view returns (MinterConfig
memory) {
        return _minterConfigs[_minter];
    }

    /// @notice Remove the minter using the index hint.
    /// @dev Can only be called by the owner. Allowing deletion of minter gives
    ///      the owner the ability reset the minters status, clearing the currentLimit
and timestamp.
    function _removeMinterByIndexHint(address _minter, uint256 _index) internal {
        if (_index >= _minters.length) {
            revert ILimitedMinterManager_InvalidIndex();
        }
        if (_minters[_index] != _minter) {
            revert ILimitedMinterManager_InvalidIndexHint();
        }
        delete _minterConfigs[_minter];
        if (_index != _minters.length - 1) {
            _minters[_index] = _minters[_minters.length - 1];
        }
        _minters.pop();
        emit MinterRemoved(_minter);
    }

    /// @notice Updates the limits of a minter, minter will NOT be deleted if the
limit is set to 0.
    /// @param _minter The address of the minter we are setting the limits too
    /// @param _mintingLimit The updated minting limit we are setting to the minter
    /// @param _duration The duration window for maxLimit to be replenished
    function _setMinterLimit(address _minter, uint256 _mintingLimit, uint256
_duration) internal {
        if (_mintingLimit > (type(uint256).max / 2)) {
            revert ILimitedMinterManager_LimitsTooHigh();
        }
        if (_duration == 0) {
            revert ILimitedMinterManager_InvalidDuration();
```

```
        }
        // The duration can never be 0 for a minter, so when duration is currently 0
        // this is a new minter being added.
        if (_minterConfigs[_minter].duration == 0) {
            _minters.push(_minter);
            emit MinterNewlyAdded(_minter);
        }
        _changeMinterLimit(_minter, _mintingLimit, _duration);
        emit MinterLimitsSet(_minter, _mintingLimit, _duration);
    }

    /// @notice use minter's limit to mint token, revert if not enough
    /// @dev Can only be called by the minter
    /// @param _minter The minter address
    /// @param _amount The amount of tokens being minted
    function _minterMint(address _minter, uint256 _amount) internal {
        uint256 _currentLimit = mintingCurrentLimitOf(_minter);
        if (_currentLimit < _amount) revert ILimitedMinterManager_NotEnoughLimits();
        _useMinterLimits(_minter, _amount);
        emit MinterMinted(_minter, _minter, _amount);
    }

    /// @notice Returns the max limit of a minter
    /// @param _minter the minter we are viewing the limits of
    /// @return _limit The limit the minter has
    function mintingMaxLimitOf(address _minter) public view returns (uint256 _limit)
{
        _limit = _minterConfigs[_minter].maxLimit;
    }

    /// @notice Returns the current limit of a minter
    /// @param _minter the minter we are viewing the limits of
    /// @return _limit The limit the minter has
    function mintingCurrentLimitOf(address _minter) public view returns (uint256
_limit) {
        // not a minter
        if (_minterConfigs[_minter].duration == 0) {
            return 0;
        }
        _limit = _getCurrentLimit(
            _minterConfigs[_minter].currentLimit,
            _minterConfigs[_minter].maxLimit,
            _minterConfigs[_minter].duration,
            _minterConfigs[_minter].timestamp
        );
    }

    /// @notice Uses the limit of any minter
    /// @param _minter The address of the minter who is being changed
```

```
    /// @param _change The change in the limit
    function _useMinterLimits(address _minter, uint256 _change) private {
        uint256 _currentLimit = mintingCurrentLimitOf(_minter);
        _minterConfigs[_minter].timestamp = block.timestamp;
        _minterConfigs[_minter].currentLimit = _currentLimit - _change;
    }

    /// @notice Updates the limit of any minter
    /// @dev Can only be called by the owner
    /// @param _minter The address of the minter we are setting the limit too
    /// @param _limit The updated limit we are setting to the minter
    /// @param _duration The duration window for maxLimit to be replenished
    function _changeMinterLimit(address _minter, uint256 _limit, uint256 _duration)
private {
        uint256 _oldLimit = _minterConfigs[_minter].maxLimit;
        uint256 _currentLimit = mintingCurrentLimitOf(_minter);
        _minterConfigs[_minter].maxLimit = _limit;

        _minterConfigs[_minter].currentLimit = _calculateNewCurrentLimit(_limit,
_oldLimit, _currentLimit);
        _minterConfigs[_minter].timestamp = block.timestamp;
        _minterConfigs[_minter].duration = _duration;
    }

    /// @notice Updates the current limit
    /// @param _limit The new limit
    /// @param _oldLimit The old limit
    /// @param _currentLimit The current limit
    /// @return _newCurrentLimit The new current limit
    function _calculateNewCurrentLimit(
        uint256 _limit,
        uint256 _oldLimit,
        uint256 _currentLimit
    ) internal pure returns (uint256 _newCurrentLimit) {
        uint256 _difference;

        if (_oldLimit > _limit) {
            _difference = _oldLimit - _limit;
            _newCurrentLimit = _currentLimit > _difference ? _currentLimit -
_difference : 0;
        } else {
            _difference = _limit - _oldLimit;
            _newCurrentLimit = _currentLimit + _difference;
        }
    }

    /// @notice Gets the current limit
    /// @param _currentLimit The current limit
    /// @param _maxLimit The max limit
```

```
    /// @param _duration The duration window for maxLimit
    /// @return _limit The current limit
    function _getCurrentLimit(
        uint256 _currentLimit,
        uint256 _maxLimit,
        uint256 _duration,
        uint256 _timestamp
    ) internal view returns (uint256 _limit) {
        _limit = _currentLimit;
        if (_limit == _maxLimit) {
            return _limit;
        } else if (_timestamp + _duration <= block.timestamp) {
            _limit = _maxLimit;
        } else if (_timestamp + _duration > block.timestamp) {
            uint256 _timePassed = block.timestamp - _timestamp;
            uint256 _calculatedLimit = _limit + ((_timePassed * _maxLimit) /
_duration);
            _limit = _calculatedLimit > _maxLimit ? _maxLimit : _calculatedLimit;
        }
    }
}
```

TokenUpgrader.sol

```
// SPDX-License-Identifier: MIT
// Compatible with OpenZeppelin Contracts ^5.0.0
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.8.24;

import { Ownable2Step, Ownable } from
"@openzeppelin/contracts/access/Ownable2Step.sol";
import { Pausable } from "@openzeppelin/contracts/utils/Pausable.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import { IERC20Permit } from
"@openzeppelin/contracts/token/ERC20/extensions/IERC20Permit.sol";

/// @title TokenUpgrader Contract for upgrading old tokens to new tokens
/// @author Gravity Team
/// @notice Customized for upgrading old tokens to new tokens, compatible with
unburnable and no-permit tokens
contract TokenUpgrader is Ownable2Step, Pausable {
    using SafeERC20 for IERC20;

    error Uninitialized();
    error AlreadyInitialized();
    error FailedToSendEther();
```

```
    error InvalidReceiverAddress();

    /// @dev The address to which the old tokens are sent to burn
    address public constant DEAD_ADDRESS = address(0xdead);
    /// @notice The ratio of new tokens to old tokens
    uint256 public constant SPLIT_RATIO = 60;

    IERC20 public oldToken;
    IERC20 public newToken;
    bool public initialized;

    constructor(address initialAdmin) Ownable(initialAdmin) {}

    //SlowMist// Suspending all transactions upon major abnormalities is a recommended approach.
    function pause() public onlyOwner {
        _pause();
    }

    function unpause() public onlyOwner {
        _unpause();
    }

    /// Withdraw ERC20 token from the contract
    /// @param token the address of the token to withdraw
    /// @param to the receiver of the token
    /// @param amount the amount of token to withdraw
    //SlowMist// The owner role can withdraw ERC20 token from the contract
    function withdrawERC20Token(address token, address to, uint256 amount) external onlyOwner {
        if (to == address(0)) {
            revert InvalidReceiverAddress();
        }
        IERC20(token).safeTransfer(to, amount);
    }

    /// Withdraw ETH from the contract
    /// @param to address to receive the ETH
    /// @param amount amount of ETH to withdraw
    //SlowMist// The owner role can withdraw ETH from the contract
    function withdrawETH(address to, uint256 amount) external onlyOwner {
        if (to == address(0)) {
            revert InvalidReceiverAddress();
        }
        // use call to transfer native token
        (bool sent, ) = to.call{ value: amount }("");
        if (!sent) {
            revert FailedToSendEther();
        }
```

```
    }

    /// initialize the contract with the old and new token addresses.
    /// @param oldTokenAddress The address of the old token contract
    /// @param newTokenAddress The address of the new token contract
    //SlowMist// The owner role can initialize the contract
    function initialize(address oldTokenAddress, address newTokenAddress) public
onlyOwner {
        if (initialized) {
            revert AlreadyInitialized();
        }
        initialized = true;
        oldToken = IERC20(oldTokenAddress);
        newToken = IERC20(newTokenAddress);
    }

    /// @notice upgrade old tokens to new tokens
    /// @dev msg.sender must approve the amount of old tokens to be upgraded before
calling this function.
    /// @param amount The amount of old tokens to upgrade.
    function upgradeToken(uint256 amount) external onlyInitialized whenNotPaused
returns (bool) {
        // compatible with unburnable tokens
        oldToken.safeTransferFrom(msg.sender, DEAD_ADDRESS, amount);
        newToken.safeTransfer(msg.sender, amount * SPLIT_RATIO);
        return true;
    }

    /// @notice upgrade old tokens to new tokens using permit
    /// @param amount The amount of old tokens to upgrade.
    /// @param deadline The deadline timestamp for the permit signature.
    /// @param v secp256k1 signature: v
    /// @param r secp256k1 signature: r
    /// @param s secp256k1 signature: s
    function upgradeTokenByPermit(
        uint256 amount,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external onlyInitialized whenNotPaused returns (bool) {
        // permit signature front-run protection
        /* solhint-disable no-empty-blocks */
        try IERC20Permit(address(oldToken)).permit(msg.sender, address(this), amount,
deadline, v, r, s) {} catch {}
        /* solhint-enable no-empty-blocks */

        // compatible with unburnable tokens
        oldToken.safeTransferFrom(msg.sender, DEAD_ADDRESS, amount);
```

```
        newToken.safeTransfer(msg.sender, amount * SPLIT_RATIO);
        return true;
    }

    modifier onlyInitialized() {
        if (!initialized) {
            revert Uninitialized();
        }
        _;
    }
}
```

TokenVesting.sol

```
// SPDX-License-Identifier: MIT
// Compatible with OpenZeppelin Contracts ^5.0.0
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.8.24;

import { Ownable2Step, Ownable } from
"@openzeppelin/contracts/access/Ownable2Step.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

contract TokenVesting is Ownable2Step {
    using SafeERC20 for IERC20;

    /// @notice Emitted when tokens are released.
    /// @param token the address of the ERC20 token to be vested.
    /// @param amount the amount of token to be vested.
    event TokensReleased(address token, uint256 amount);

    error InvalidToken();
    error InvalidDuration();
    error InvalidNumVestings();
    error NoTokenReleasable();
    error InvalidRecoverTokenAddress();
    error FailedToSendEther();

    /// @notice ERC20 token that is being vested
    IERC20 public immutable token;
    /// @notice Start time of the vesting, UNIX timestamp.
    uint256 public immutable start;
    /// @notice Duration (cliff) between vesting, in seconds.
    uint256 public immutable duration;
    /// @notice The number of vestings.
    uint256 public immutable numVestings;
```

```
    /// @notice The amount of token that has been released.
    uint256 public released;

    /// @dev Creates a vesting contract that vests its balance of ERC20 token to the
    ///      owner. (total/numVesting) Tokens are vested every duration since start.
    /// @param _token address of token ERC20 token.
    /// @param _start the time (as Unix timestamp) at which point vesting starts
    /// @param _duration the time between vesting
    /// @param _numVestings the number of vesting
    constructor(
        address _owner,
        address _token,
        uint256 _start,
        uint256 _duration,
        uint256 _numVestings
    ) Ownable(_owner) {
        if (_token == address(0)) revert InvalidToken();
        if (_duration == 0) revert InvalidDuration();
        if (_numVestings == 0) revert InvalidNumVestings();

        token = IERC20(_token);
        start = _start;
        duration = _duration;
        numVestings = _numVestings;
    }

    /// @notice Transfers vested tokens to owner.
    //SlowMist// The release function can release the vested tokens to the owner.
    function release() external {
        uint256 unreleased = releasableAmount();
        if (unreleased == 0) revert NoTokenReleasable();
        released = released + unreleased;
        token.safeTransfer(owner(), unreleased);
        emit TokensReleased(address(token), unreleased);
    }

    /// @notice Calculates the amount that has already vested but hasn't been released
yet.
    /// @return the amount of vested tokens that can be released to the beneficiary.
    function releasableAmount() public view returns (uint256) {
        return _vestedAmount() - released;
    }

    /// @notice Calculates the amount that has already vested.
    /// @return the amount of vested tokens.
    function _vestedAmount() private view returns (uint256) {
        uint256 currentBalance = token.balanceOf(address(this));
        uint256 totalBalance = currentBalance + released;
        if (block.timestamp < start) {
```

```
            // not start
            return 0;
        } else if (block.timestamp >= start + duration * numVestings) {
            // all vested, transfer out all remaining tokens.
            return totalBalance;
        } else {
            // For every duration passed after start, vest (totalBalance /
numVestings) tokens.
            return ((block.timestamp - start) / duration) * (totalBalance /
numVestings);
        }
    }

    /// @notice Recover other ERC20 token from contract
    /// @param tokenAddress ERC20 token address to recover
    /// @param tokenAmount amount of token to recover
    //SlowMist// The owner role can recover other ERC20 tokens from the contract.
    function recoverOtherERC20(address tokenAddress, uint256 tokenAmount) external
onlyOwner {
        if (tokenAddress == address(token)) revert InvalidRecoverTokenAddress();
        IERC20(tokenAddress).safeTransfer(owner(), tokenAmount);
    }

    /// @notice Recover native token from contract
    //SlowMist// The owner role can recover native token from the contract.
    function recoverNativeToken() external onlyOwner {
        (bool sent, ) = this.owner().call{ value: address(this).balance }("");
        if (!sent) revert FailedToSendEther();
    }
}
```

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