Smart Contract
Security Audit Report

The SlowMist Security Team received the NFTGalaxy team's application for smart contract security audit of the Galaxy gas optimization on 2022.06.27. The following are the details and results of this smart contract security audit:

Token Name :

Galaxy gas optimization

The audit version :

https://github.com/NFTGalaxy/GalaxyContractAudit/blob/main/contracts/StarNFT/StarNFTV4.sol

commit: 1cbe41371e4e04b92e1de38928f5991e74874fb1

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	Low Risk
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 : Low Risk

Audit Number : 0X002206280001

Audit Date : 2022.06.27 - 2022.06.29

Audit Team : SlowMist Security Team

Summary conclusion : This is Galaxy's NFT gas optimization contract. The total supply of contract tokens is unlimited. The contract does not have the Overflow and the Race Conditions issue.

During the audit we found the following issues and information:

1. Minter roles can mint NFTs to users through the mint/mintBatch function. Since there is no cap on the total token supply, there is a potential risk of overflow
2. The Minter role can burn NFT that the user has approved to the minter role through the burn/burnBatch function.
3. The owner role can modify the contract's _baseURI through the setURI function.
4. The owner role can modify the contract's _transferable through the setTransferable function. When the _transferable status is false, users will not be able to transfer tokens.
5. The owner role can modify the contract's _name through the setName function.
6. The owner role can modify the contract's _symbol through the setSymbol function.

The source code:

```
/*
    Copyright 2022 Project Galaxy.
    Licensed under the Apache License, Version 2.0 (the "License");
```

```
    you may not use this file except in compliance with the License.
    You may obtain a copy of the License at
    http://www.apache.org/licenses/LICENSE-2.0
    Unless required by applicable law or agreed to in writing, software
    distributed under the License is distributed on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    See the License for the specific language governing permissions and
    limitations under the License.
    SPDX-License-Identifier: Apache License, Version 2.0
*/
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity 0.7.6;

import "../../zeppelin-solidity/contracts/token/ERC721/IERC721.sol";
import "../../zeppelin-solidity//contracts/token/ERC721/IERC721Receiver.sol";
import "../../zeppelin-solidity//contracts/token/ERC721/IERC721Metadata.sol";
import "../../zeppelin-solidity//contracts/utils/Address.sol";
import "../../zeppelin-solidity//contracts/utils/Strings.sol";
import "../../zeppelin-solidity/contracts/access/Ownable.sol";
import "../../zeppelin-solidity//contracts/introspection/ERC165.sol";
import "../../interfaces/IStarNFT.sol";

/**
 * @dev Fork https://github.com/generalgalactic/ERC721S and implement IStarNFT
interface
 */
contract StarNFTV4 is Ownable, ERC165, IERC721, IERC721Metadata, IStarNFT {
    using Address for address;
    using Strings for uint256;

    /* ============ State Variables ============ */

    struct Star {
        uint160 owner;
        uint96 cid; // Max value is 7.9E28, enough to store campaign id
    }

    // Token name
    string private _name;

    // Token symbol
    string private _symbol;

    // Total number of tokens burned
```

```
uint256 private _burnCount;

// Array of all tokens storing the owner's address and the campaign id
Star[] private _tokens;

// Mapping owner address to token count
mapping(address => uint256) private _balances;

// Mapping from token ID to approved address
mapping(uint256 => address) private _tokenApprovals;

// Mapping from owner to operator approvals
mapping(address => mapping(address => bool)) private _operatorApprovals;

// Mint and burn star.
mapping(address => bool) private _minters;

// Default allow transfer
bool private _transferable = true;

// Base token URI
string private _baseURI;

/* ============ Events ============ */
// Add new minter
event EventMinterAdded(address indexed newMinter);

// Remove old minter
event EventMinterRemoved(address indexed oldMinter);

/* ============ Modifiers ============ */

/**
 * Only minter.
 */
modifier onlyMinter() {
    require(_minters[msg.sender], "StarNFT: must be minter");
    _;
}

/**
 * Only allow transfer.
 */
modifier onlyTransferable() {
```

```
    require(_transferable, "StarNFT: must transferable");
    _;
}

/**
 * @dev Initializes the contract
 */
constructor() {
    // Initialize zero index value
    Star memory _star = Star(0, 0);
    _tokens.push(_star);
}

/**
 * @dev See {IERC165-supportsInterface}.
 */
function supportsInterface(bytes4 interfaceId)
public
view
override(ERC165, IERC165)
returns (bool)
{
    return
    interfaceId == type(IERC721).interfaceId ||
    interfaceId == type(IERC721Metadata).interfaceId ||
    interfaceId == type(IStarNFT).interfaceId ||
    super.supportsInterface(interfaceId);
}

/**
 * @dev Is this address a minters.
 */
function minters(address account) public view returns (bool) {
    return _minters[account];
}

/**
 * @dev Is this contract allow nft transfer.
 */
function transferable() public view returns (bool) {
    return _transferable;
}

/**
```

```
     * @dev Returns the base URI for nft.
     */
    function baseURI() public view returns (string memory) {
        return _baseURI;
    }

    /**
     * @dev Get Star NFT CID.
     */
    function cid(uint256 tokenId) public view returns (uint256) {
        require(_exists(tokenId), "StarNFT: StarNFT does not exist");
        return _tokens[tokenId].cid;
    }

    /**
     * @dev Get Star NFT owner
     */
    function getNumMinted() public view override returns (uint256) {
        return _tokens.length-1;
    }

    /**
     * @dev See {IERC721Enumerable-totalSupply}.
     */
    function totalSupply() public view returns (uint256) {
        return getNumMinted() - _burnCount;
    }

    /**
     * @dev See {IERC721Enumerable-tokenOfOwnerByIndex}.
     * This is implementation is O(n) and should not be
     * called by other contracts.
     */
    function tokenOfOwnerByIndex(address owner, uint256 index)
    public
    view
    returns (uint256)
    {
        uint256 currentIndex = 0;
        for (uint256 i = 1; i < _tokens.length; i++) {
            if (isOwnerOf(owner, i)) {
                if (currentIndex == index) {
                    return i;
                }
```

```
            currentIndex += 1;
        }
    }
    revert("ERC721Enumerable: owner index out of bounds");
}

/**
 * @dev See {IERC721-balanceOf}.
 */
function balanceOf(address owner)
public
view
override
returns (uint256)
{
    require(
        owner != address(0),
        "ERC721: balance query for the zero address"
    );
    return _balances[owner];
}

/**
 * @dev See {IERC721-ownerOf}.
 */
function ownerOf(uint256 tokenId)
public
view
override
returns (address)
{
    require(
        _exists(tokenId),
        "ERC721: owner query for nonexistent token"
    );
    return address(_tokens[tokenId].owner);
}

/**
 * @dev See {IStarNFT-isOwnerOf}.
 */
function isOwnerOf(address account, uint256 id)
public
view
```

```
override
returns (bool)
{
    address owner = ownerOf(id);
    return owner == account;
}

/**
 * @dev See {IERC721Metadata-name}.
 */
function name() public view virtual override returns (string memory) {
    return _name;
}

/**
 * @dev See {IERC721Metadata-symbol}.
 */
function symbol() public view virtual override returns (string memory) {
    return _symbol;
}

/**
 * @dev See {IERC721Metadata-tokenURI}.
 */
function tokenURI(uint256 tokenId)
public
view
override
returns (string memory)
{
    require(
        _exists(tokenId),
        "ERC721Metadata: URI query for nonexistent token"
    );

    return
    bytes(_baseURI).length > 0
    ? string(abi.encodePacked(_baseURI, tokenId.toString(), ".json"))
    : "";
}

/**
 * @dev See {IERC721-approve}.
 */
```

```
function approve(address to, uint256 tokenId) public override {
    address owner = ownerOf(tokenId);
    require(to != owner, "ERC721: approval to current owner");

    require(
        _msgSender() == owner || isApprovedForAll(owner, _msgSender()),
        "ERC721: approve caller is not owner nor approved for all"
    );

    _approve(to, tokenId);
}

/**
 * @dev See {IERC721-getApproved}.
 */
function getApproved(uint256 tokenId)
public
view
override
returns (address)
{
    require(
        _exists(tokenId),
        "ERC721: approved query for nonexistent token"
    );

    return _tokenApprovals[tokenId];
}

/**
 * @dev See {IERC721-setApprovalForAll}.
 */
function setApprovalForAll(address operator, bool approved)
public
override
{
    require(operator != _msgSender(), "ERC721: approve to caller");

    _operatorApprovals[_msgSender()][operator] = approved;
    emit ApprovalForAll(_msgSender(), operator, approved);
}

/**
 * @dev See {IERC721-isApprovedForAll}.
```

```
 */
function isApprovedForAll(address owner, address operator)
public
view
override
returns (bool)
{
    return _operatorApprovals[owner][operator];
}

/**
 * @dev See {IERC721-transferFrom}.
 */
function transferFrom(
    address from,
    address to,
    uint256 tokenId
) public onlyTransferable override {
    //solhint-disable-next-line max-line-length
    require(
        _isApprovedOrOwner(_msgSender(), tokenId),
        "ERC721: transfer caller is not owner nor approved"
    );

    _transfer(from, to, tokenId);
}

/**
 * @dev See {IERC721-safeTransferFrom}.
 */
function safeTransferFrom(
    address from,
    address to,
    uint256 tokenId
) public onlyTransferable override {
    safeTransferFrom(from, to, tokenId, "");
}

/**
 * @dev See {IERC721-safeTransferFrom}.
 */
function safeTransferFrom(
    address from,
    address to,
```

```
        uint256 tokenId,
        bytes memory _data
    ) public onlyTransferable override {
        require(
            _isApprovedOrOwner(_msgSender(), tokenId),
            "ERC721: transfer caller is not owner nor approved"
        );
        _safeTransfer(from, to, tokenId, _data);
    }

    /**
     * @dev Safely transfers `tokenId` token from `from` to `to`, checking first that contract recipients
     * are aware of the ERC721 protocol to prevent tokens from being forever locked.
     *
     * `_data` is additional data, it has no specified format and it is sent in call to `to`.
     *
     * This internal function is equivalent to {safeTransferFrom}, and can be used to e.g.
     * implement alternative mechanisms to perform token transfer, such as signature-based.
     *
     * Requirements:
     *
     * - `from` cannot be the zero address.
     * - `to` cannot be the zero address.
     * - `tokenId` token must exist and be owned by `from`.
     * - If `to` refers to a smart contract, it must implement {IERC721Receiver-onERC721Received}, which is called upon a safe transfer.
     *
     * Emits a {Transfer} event.
     */
    function _safeTransfer(
        address from,
        address to,
        uint256 tokenId,
        bytes memory _data
    ) internal {
        _transfer(from, to, tokenId);
        require(
            _checkOnERC721Received(from, to, tokenId, _data),
            "ERC721: transfer to non ERC721Receiver implementer"
        );
```

```
    }

    /**
     * @dev Returns whether `tokenId` exists.
     *
     * Tokens can be managed by their owner or approved accounts via {approve} or
{setApprovalForAll}.
     *
     * Tokens start existing when they are minted (`_mint`),
     * and stop existing when they are burned (`_burn`).
     */
    function _exists(uint256 tokenId) internal view returns (bool) {
        return tokenId > 0 && tokenId <= getNumMinted() && _tokens[tokenId].owner !=
0x0;
    }

    /**
     * @dev Returns whether `spender` is allowed to manage `tokenId`.
     *
     * Requirements:
     *
     * - `tokenId` must exist.
     */
    function _isApprovedOrOwner(address spender, uint256 tokenId)
    internal
    view
    returns (bool)
    {
        address owner = ownerOf(tokenId);
        return (spender == owner ||
        getApproved(tokenId) == spender ||
        isApprovedForAll(owner, spender));
    }

    /* ============ External Functions ============ */
    //SlowMist// Minter roles can mint NFTs to users through the mint function. There
is no cap on the total NFT supply
    function mint(address account, uint256 cid)
    external
    override
    onlyMinter
    returns (uint256)
    {
        require(account != address(0), "StarNFT: mint to the zero address");
```

```
        uint256 tokenId = _tokens.length;
        Star memory star = Star(uint160(account), uint96(cid));

        _balances[account] += 1; //SlowMist// Since there is no cap on the total
token supply, there is a potential risk of overflow
        _tokens.push(star);

        require(
            _checkOnERC721Received(address(0), account, tokenId, ""),
            "StarNFT: transfer to non ERC721Receiver implementer"
        );

        emit Transfer(address(0), account, tokenId);
        return tokenId;
    }

    //SlowMist// Minter roles can mint NFTs to users through the mintBatch function.
There is no cap on the total NFT supply
    function mintBatch(
        address account,
        uint256 amount,
        uint256[] calldata cidArr
    ) external override onlyMinter returns (uint256[] memory) {
        require(account != address(0), "StarNFT: mint to the zero address");
        uint256[] memory ids = new uint256[](amount);

        _balances[account] += amount; //SlowMist// Since there is no cap on the total
token supply, there is a potential risk of overflow

        for (uint256 i = 0; i < ids.length; i++) {
            uint256 tokenId = _tokens.length;
            Star memory star = Star(uint160(account), uint96(cidArr[i]));
            ids[i] = tokenId;
            _tokens.push(star);

            require(
                _checkOnERC721Received(address(0), account, tokenId, ""),
                "StarNFT: transfer to non ERC721Receiver implementer"
            );

            emit Transfer(address(0), account, tokenId);
        }
```

```
        return ids;
    }

    //SlowMist// The Minter role can burn NFT that the user has approved to the minter role through the burn function
    function burn(address account, uint256 id) external override onlyMinter {
        require(
            _isApprovedOrOwner(_msgSender(), id),
            "StarNFT: caller is not approved or owner"
        );
        require(isOwnerOf(account, id), "StarNFT: not owner");

        // Clear approvals
        _approve(address(0), id);
        _burnCount++;
        _balances[account] -= 1;
        _tokens[id].owner = 0;
        _tokens[id].cid = 0;

        emit Transfer(account, address(0), id);
    }

    //SlowMist// The Minter role can burn NFT that the user has approved to the minter role through the burnBatch function
    function burnBatch(address account, uint256[] calldata ids)
    external
    override
    onlyMinter
    {
        _burnCount += ids.length;
        _balances[account] -= ids.length;
        for (uint256 i = 0; i < ids.length; i++) {
            uint256 tokenId = ids[i];
            require(
                _isApprovedOrOwner(_msgSender(), tokenId),
                "StarNFT: caller is not approved or owner"
            );
            require(isOwnerOf(account, tokenId), "StarNFT: not owner");
            // Clear approvals
            _approve(address(0), tokenId);
            _tokens[tokenId].owner = 0;
            _tokens[tokenId].cid = 0;

            emit Transfer(account, address(0), tokenId);
```

```
    }
}

/**
 * @dev Transfers `tokenId` from `from` to `to`.
 *  As opposed to {transferFrom}, this imposes no restrictions on msg.sender.
 *
 * Requirements:
 *
 * - `to` cannot be the zero address.
 * - `tokenId` token must be owned by `from`.
 *
 * Emits a {Transfer} event.
 */
function _transfer(
    address from,
    address to,
    uint256 tokenId
) internal virtual {
    require(
        isOwnerOf(from, tokenId),
        "ERC721: transfer of token that is not own"
    );
    require(to != address(0), "ERC721: transfer to the zero address");

    // Clear approvals from the previous owner
    _approve(address(0), tokenId);
    _balances[from] -= 1;
    _balances[to] += 1;
    _tokens[tokenId].owner = uint160(to);

    emit Transfer(from, to, tokenId);
}

/**
 * @dev Approve `to` to operate on `tokenId`
 *
 * Emits a {Approval} event.
 */
function _approve(address to, uint256 tokenId) internal virtual {
    _tokenApprovals[tokenId] = to;
    emit Approval(ownerOf(tokenId), to, tokenId);
}
```

```
    /**
     * @dev Internal function to invoke {IERC721Receiver-onERC721Received} on a
target address.
     * The call is not executed if the target address is not a contract.
     *
     * @param from address representing the previous owner of the given token ID
     * @param to target address that will receive the tokens
     * @param tokenId uint256 ID of the token to be transferred
     * @param _data bytes optional data to send along with the call
     * @return bool whether the call correctly returned the expected magic value
     */
    function _checkOnERC721Received(
        address from,
        address to,
        uint256 tokenId,
        bytes memory _data
    ) private returns (bool) {
        if (to.isContract()) {
            try
            IERC721Receiver(to).onERC721Received(
                _msgSender(),
                from,
                tokenId,
                _data
            )
            returns (bytes4 retval) {
                return retval == IERC721Receiver.onERC721Received.selector;
            } catch (bytes memory reason) {
                if (reason.length == 0) {
                    revert(
                    "ERC721: transfer to non ERC721Receiver implementer"
                    );
                } else {
                    assembly {
                        revert(add(32, reason), mload(reason))
                    }
                }
            }
        }
        return true;
    }

    /* ============ Util Functions ============ */
    /**
```

```
     * @dev Sets a new baseURI for all token types.
     */
    //SlowMist// The owner role can modify the contract's _baseURI through the setURI function
    function setURI(string calldata newURI) external onlyOwner {
        _baseURI = newURI;
    }

    /**
     * @dev Sets a new transferable for all token types.
     */
    //SlowMist// The owner role can modify the contract's _transferable through the setTransferable function
    function setTransferable(bool transferable) external onlyOwner {
        _transferable = transferable;
    }

    /**
     * @dev Sets a new name for all token types.
     */
    //SlowMist// The owner role can modify the contract's _name through the setName function
    function setName(string calldata newName) external onlyOwner {
        _name = newName;
    }

    /**
     * @dev Sets a new symbol for all token types.
     */
    //SlowMist// The owner role can modify the contract's _symbol through the setSymbol function
    function setSymbol(string calldata newSymbol) external onlyOwner {
        _symbol = newSymbol;
    }

    /**
     * @dev Add a new minter.
     */
    function addMinter(address minter) external onlyOwner {
        require(minter != address(0), "minter must not be null address");
        require(!_minters[minter], "minter already added");
        _minters[minter] = true;
        emit EventMinterAdded(minter);
    }
```

```
    /**
     * @dev Remove a old minter.
     */
    function removeMinter(address minter) external onlyOwner {
        require(_minters[minter], "minter does not exist");
        delete _minters[minter];
        emit EventMinterRemoved(minter);
    }
}
```

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