Smart Contract
Security Audit Report

The SlowMist Security Team received the team's application for smart contract security audit of the HangOut on 2022.09.01. The following are the details and results of this smart contract security audit:

Token Name :

HangOut

File name and hash (SHA256) :

HangOutETHFlatland.sol: 201b98b00bad4284863bdd08c8a754708e9ddb9b14924f1064ffbdafa687feb9

HangOut.sol: 618a22d24e150821878eb23c4e7bb38f4e41a3a7a2f6de4a0833c65f5afe147b

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 : 0X002209060001

Audit Date : 2022.09.01 - 2022.09.06

Audit Team : SlowMist Security Team

Summary conclusion : These are the ERC20 token and ERC721 token contract, the ERC20 token contract does not contain the tokenVault and dark coin functions, the total amount of the ERC20 tokens remains unchangeable and the admin role can add signers and the signers can mint ERC721 tokens. These contracts do not have the Overflow and the Race Conditions issue.

During the audit, we found the following information:

1. Only the owner role can change the price and there is a MAX_PRICE limit.
2. The owner role can change the maxSceneOneAddr, maxPlotOneAddr, and txMaxPlots value through the setMaxLimit function.
3. Only the owner role can set the signer.
4. Only the owner role can change the sceneMerkleRoot and plotsMerkleRoot through the setMerkleRoots function.
5. The owner role can withdraw the native tokens in this contract through the withdraw function.
6. The owner role can change the baseURI through the setBaseURI function.
7. The owner role can change the openMint state through the flipMintState function.

The source code:

HangOut.sol

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

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract HangOut is ERC20 {

    constructor()
        ERC20("HangOut", "HO")
    {
        _mint(msg.sender, 1e10 ether);
    }

}
```

HangOutETHFlatland.sol

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

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/cryptography/draft-EIP712.sol";
import "@openzeppelin/contracts/utils/cryptography/SignatureChecker.sol";
import "@openzeppelin/contracts/utils/cryptography/MerkleProof.sol";
import "@openzeppelin/contracts/utils/Strings.sol";

/**
 * @title HangOutETHFlatland contract
 * @dev Extends ERC721 Non-Fungible Token Standard basic implementation
 */
contract HangOutETHFlatland is ERC721, EIP712, Ownable {
```

```
    using Strings for uint256;

    string constant public WHITELIST_MINT_SCENE_CALL_HASH_TYPE =
"mintSceneWhitelist(uint256 sceneId, bytes memory signature, bytes32[] calldata
proof)";
    string constant public WHITELIST_MINT_PLOTS_CALL_HASH_TYPE =
"mintPlotsWhitelist(uint256[] memory plotIDs, bytes memory signature, bytes32[]
calldata proof)";
    string constant public MINT_SCENE_CALL_HASH_TYPE = "mintScene(uint256 sceneId,
bytes memory signature)";
    string constant public MINT_PLOTS_CALL_HASH_TYPE = "mintPlots(uint256[] memory
plotIDs, bytes memory signature)";

    uint256 constant public MAX_PRICE = 2 ether;
    uint256 public superLargePrice = .5 ether;
    uint256 public grandePrice = .26 ether;
    uint256 public standardPrice = .18 ether;

    uint256 public maxSceneOneAddr = 2;
    uint256 public maxPlotOneAddr = 16;
    uint256 public txMaxPlots = 16;

    // plot models
    enum Model{ SUPERLARGE, GRANDE, STANDARD }

    // 1 scene = 4 superLarge + 6 grande + 6 standard
    uint256 constant SUPERLARGE_PER_SCENE = 4;
    uint256 constant GRANDE_PER_SCENE = 6;
    uint256 constant STANDARD_PER_SCENE = 6;
    uint256 constant SCENE_PLOT_NUMBER = SUPERLARGE_PER_SCENE + GRANDE_PER_SCENE +
STANDARD_PER_SCENE;

    Model[SCENE_PLOT_NUMBER] ModelsInScene = [
        Model.SUPERLARGE, Model.SUPERLARGE, Model.SUPERLARGE, Model.SUPERLARGE,
        Model.GRANDE, Model.GRANDE, Model.GRANDE, Model.GRANDE, Model.GRANDE,
Model.GRANDE,
        Model.STANDARD, Model.STANDARD, Model.STANDARD, Model.STANDARD,
Model.STANDARD, Model.STANDARD
    ];

    uint256 constant MAX_SCENE_SUPPLY = 369;
    uint256 public constant totalMaxSupply = MAX_SCENE_SUPPLY*SCENE_PLOT_NUMBER;
    uint256[3] public modelMintCount = [0, 0, 0];
```

```
    // base uri
    string public baseURI;

    // signature signer
    address public signer;

    // mint scene whitelist merkle root
    bytes32 public sceneMerkleRoot;

    // mint plots whitelist merkle root
    bytes32 public plotsMerkleRoot;

    // switch
    bool public openMint = false;

    // minted scene count
    mapping(address => uint256) public mintedSceneCount;

    // minted plot count
    mapping(address => uint256) public mintedPlotCount;

    event OnWithdraw(address to, uint256 amount);

    modifier onlyNotContract() {
        require(tx.origin == msg.sender, "The caller is another contract");
        _;
    }

    constructor(address signer_, bytes32 sceneMerkleRoot_, bytes32 plotsMerkleRoot_)
        ERC721("HangOut Ethereum Flatland", "HOETHFLAND")
        EIP712("HangOut Ethereum Flatland", "1")
    {
        signer = signer_;
        sceneMerkleRoot = sceneMerkleRoot_;
        plotsMerkleRoot = plotsMerkleRoot_;
    }
    //SlowMist// Only the owner role can change the price and there is a MAX_PRICE
limit
    //SlowMist// Missing the event log
    function setPrices(uint256 superLargePrice_, uint256 grandePrice_, uint256
standardPrice_) external onlyOwner {
        require(superLargePrice_ <= MAX_PRICE && grandePrice_ <= MAX_PRICE &&
standardPrice_ <= MAX_PRICE, "invalid price");
        superLargePrice = superLargePrice_;
```

```
        grandePrice = grandePrice_;
        standardPrice = standardPrice_;
    }

    function getPrices() public view returns(uint256, uint256, uint256) {
        return (superLargePrice, grandePrice, standardPrice);
    }
    //SlowMist// The owner role can change the maxSceneOneAddr, maxPlotOneAddr, and txMaxPlots value through the setMaxLimit function
    //SlowMist// Missing the event log
    function setMaxLimit(uint256 maxSceneOneAddr_, uint256 maxPlotOneAddr_, uint256 txMaxPlots_) external onlyOwner {
        maxSceneOneAddr = maxSceneOneAddr_;
        maxPlotOneAddr = maxPlotOneAddr_;
        txMaxPlots = txMaxPlots_;
    }

    function getMaxLimit(address account_) public view returns(uint256, uint256, uint256, uint256, uint256) {
        return (maxSceneOneAddr, maxPlotOneAddr, txMaxPlots, mintedSceneCount[account_], mintedPlotCount[account_]);
    }
    //SlowMist// Only the owner role can set the signer
    //SlowMist// Missing the event log
    function setSigner(address signer_) external onlyOwner {
        signer = signer_;
    }
    //SlowMist// Only the owner role can change the sceneMerkleRoot and plotsMerkleRoot
    //SlowMist// Missing the event log
    function setMerkleRoots(bytes32 sceneMerkleRoot_, bytes32 plotsMerkleRoot_) external onlyOwner {
        sceneMerkleRoot = sceneMerkleRoot_;
        plotsMerkleRoot = plotsMerkleRoot_;
    }

    /**
     * @dev Withdraw ETH
     */
    //SlowMist// The owner role can withdraw the native tokens in this contract through the withdraw function
    function withdraw(address payable to, uint256 amount) external onlyOwner {
        to.transfer(amount);
        emit OnWithdraw(to, amount);
```

```
    }

    /**
     * @dev Set base URI
     */
    //SlowMist// The owner role can change the baseURI
    //SlowMist// Missing the event log
    function setBaseURI(string memory baseURI_) external onlyOwner {
        baseURI = baseURI_;
    }

    /**
     * @dev Flip mint state
     */
    //SlowMist// The owner role can change the openMint state through the
flipMintState function
    //SlowMist// Missing the event log
    function flipMintState() public onlyOwner {
        openMint = !openMint;
    }

    /**
     * @dev Whitelist mint scene
     */
    function mintSceneWhitelist(uint256 sceneId, bytes memory signature, bytes32[]
calldata proof) external payable onlyNotContract {
        require(openMint, "mint not open");
        require(verifySceneMerkle(_accountToLeaf(msg.sender), proof), "invalid
whitelist proof");
        require(sceneId < MAX_SCENE_SUPPLY, "scene id overrange");
        require(mintedSceneCount[msg.sender] + 1 <= maxSceneOneAddr, "mint too many
by one address");
        require(msg.value >=
superLargePrice*SUPERLARGE_PER_SCENE+grandePrice*GRANDE_PER_SCENE+standardPrice*STAND
ARD_PER_SCENE, "not enough ether sent");

        uint256[] memory plotIDs = new uint256[](SCENE_PLOT_NUMBER);
        uint256 beginPlotId = sceneId*SCENE_PLOT_NUMBER;
        for (uint256 i = 0; i < SCENE_PLOT_NUMBER; i++) {
            plotIDs[i] = beginPlotId + i;
        }
        require(_verifyMintSignature(msg.sender, plotIDs,
WHITELIST_MINT_SCENE_CALL_HASH_TYPE, signature), "invalid minting signature");
```

```
        modelMintCount[0] += SUPERLARGE_PER_SCENE;
        modelMintCount[1] += GRANDE_PER_SCENE;
        modelMintCount[2] += STANDARD_PER_SCENE;
        mintedSceneCount[msg.sender] += 1;

        for (uint256 i = 0; i < SCENE_PLOT_NUMBER; i++) {
            _safeMint(msg.sender, plotIDs[i]);
        }
    }

    /**
     * @dev Whitelist mint plots
     */
    function mintPlotsWhitelist(uint256[] memory plotIDs, bytes memory signature,
bytes32[] calldata proof) external payable onlyNotContract {
        require(openMint, "mint not open");
        require(verifyPlotsMerkle(_accountToLeaf(msg.sender), proof), "invalid
whitelist proof");
        require(_verifyMintSignature(msg.sender, plotIDs,
WHITELIST_MINT_PLOTS_CALL_HASH_TYPE, signature), "invalid minting signature");
        require(plotIDs.length <= txMaxPlots, "mint too many at a time");
        require(mintedPlotCount[msg.sender] + plotIDs.length <= maxPlotOneAddr, "mint
too many by one address");

        uint256[3] memory incNumbers;
        for (uint256 i = 0; i < plotIDs.length; i++) {
            require(plotIDs[i] < totalMaxSupply, "plot id overrange");
            Model m = ModelsInScene[plotIDs[i]%SCENE_PLOT_NUMBER];
            incNumbers[uint256(m)] += 1;
        }

        require(msg.value >=
superLargePrice*incNumbers[0]+grandePrice*incNumbers[1]+standardPrice*incNumbers[2],
"not enough ether sent");

        modelMintCount[0] += incNumbers[0];
        modelMintCount[1] += incNumbers[1];
        modelMintCount[2] += incNumbers[2];
        mintedPlotCount[msg.sender] += plotIDs.length;

        for (uint256 i = 0; i < plotIDs.length; i++) {
            _safeMint(msg.sender, plotIDs[i]);
        }
    }
```

```
    /**
     * @dev Mint scene
     */
    function mintScene(uint256 sceneId, bytes memory signature) external payable
onlyNotContract {
        require(openMint, "mint not open");
        require(sceneId < MAX_SCENE_SUPPLY, "scene id overrange");
        require(mintedSceneCount[msg.sender] + 1 <= maxSceneOneAddr, "mint too many
by one address");
        require(msg.value >=
superLargePrice*SUPERLARGE_PER_SCENE+grandePrice*GRANDE_PER_SCENE+standardPrice*STAND
ARD_PER_SCENE, "not enough ether sent");

        uint256[] memory plotIDs = new uint256[](SCENE_PLOT_NUMBER);
        uint256 beginPlotId = sceneId*SCENE_PLOT_NUMBER;
        for (uint256 i = 0; i < SCENE_PLOT_NUMBER; i++) {
            plotIDs[i] = beginPlotId + i;
        }

        require(_verifyMintSignature(msg.sender, plotIDs, MINT_SCENE_CALL_HASH_TYPE,
signature), "invalid minting signature");

        modelMintCount[0] += SUPERLARGE_PER_SCENE;
        modelMintCount[1] += GRANDE_PER_SCENE;
        modelMintCount[2] += STANDARD_PER_SCENE;
        mintedSceneCount[msg.sender] += 1;

        for (uint256 i = 0; i < SCENE_PLOT_NUMBER; i++) {
            _safeMint(msg.sender, plotIDs[i]);
        }
    }

    /**
     * @dev Mint plots
     */
    function mintPlots(uint256[] memory plotIDs, bytes memory signature) external
payable onlyNotContract {
        require(openMint, "mint not open");
        require(_verifyMintSignature(msg.sender, plotIDs, MINT_PLOTS_CALL_HASH_TYPE,
signature), "invalid minting signature");
        require(plotIDs.length <= txMaxPlots, "mint too many at a time");
        require(mintedPlotCount[msg.sender] + plotIDs.length <= maxPlotOneAddr, "mint
too many by one address");
```

```
        uint256[3] memory incNumbers;
        for (uint256 i = 0; i < plotIDs.length; i++) {
            require(plotIDs[i] < totalMaxSupply, "plot id overrange");
            Model m = ModelsInScene[plotIDs[i]%SCENE_PLOT_NUMBER];
            incNumbers[uint256(m)] += 1;
        }

        require(msg.value >=
superLargePrice*incNumbers[0]+grandePrice*incNumbers[1]+standardPrice*incNumbers[2],
"not enough ether sent");

        modelMintCount[0] += incNumbers[0];
        modelMintCount[1] += incNumbers[1];
        modelMintCount[2] += incNumbers[2];
        mintedPlotCount[msg.sender] += plotIDs.length;

        for (uint256 i = 0; i < plotIDs.length; i++) {
            _safeMint(msg.sender, plotIDs[i]);
        }
    }

    /**
     * @dev Mint reserved scenes
     */
    //SlowMist// The owner role can mint all the reserved nfts through the
mintReserved function
    function mintReserved(uint256[] calldata sceneIds) external onlyOwner {
        modelMintCount[0] += SUPERLARGE_PER_SCENE * sceneIds.length;
        modelMintCount[1] += GRANDE_PER_SCENE * sceneIds.length;
        modelMintCount[2] += STANDARD_PER_SCENE * sceneIds.length;

        for (uint256 i = 0; i < sceneIds.length; i++) {
            require(sceneIds[i] < MAX_SCENE_SUPPLY, "scene id overrange");

            uint256 beginPlotId = sceneIds[i]*SCENE_PLOT_NUMBER;
            for (uint256 j = 0; j < SCENE_PLOT_NUMBER; j++) {
                _safeMint(msg.sender, beginPlotId + j);
            }
        }
    }

    /**
     * @dev Get total supply
```

```
     */
    function totalSupply() public view returns (uint) {
        return modelMintCount[0] + modelMintCount[1] + modelMintCount[2];
    }

    /**
     * @dev Get plot model
     */
    function plotModel(uint256 id) public view returns(Model) {
        require(id < totalMaxSupply, "exceed max id");
        require(_exists(id), "token doesn't exist");
        return ModelsInScene[id%SCENE_PLOT_NUMBER];
    }

    /**
     * @dev Verify minting scene merkle tree proof
     */
    function verifySceneMerkle(bytes32 leaf, bytes32[] memory proof) public view
returns (bool) {
        return MerkleProof.verify(proof, sceneMerkleRoot, leaf);
    }

    /**
     * @dev Verify minting plots merkle tree proof
     */
    function verifyPlotsMerkle(bytes32 leaf, bytes32[] memory proof) public view
returns (bool) {
        return MerkleProof.verify(proof, plotsMerkleRoot, leaf);
    }

    /**
     * @dev Base URI for computing {tokenURI}. If set, the resulting URI for each
     * token will be the concatenation of the `baseURI` and the `tokenId`. Empty
     * by default, can be overriden in child contracts.
     */
    function _baseURI() internal view override returns (string memory) {
        return baseURI;
    }

    /**
     * @dev Verify EIP712 signature for minting
     */
    function _verifyMintSignature(address minter, uint256[] memory plotIds, string
memory callHashType, bytes memory signature) internal view returns(bool) {
```

```
        bytes32 digest = _hashTypedDataV4(keccak256(abi.encode(
                keccak256(bytes(callHashType)),
                minter,
                plotIds
            )));

        return SignatureChecker.isValidSignatureNow(signer, digest, signature);
    }

    /**
     * @dev Convert account address to bytes32
     */
    function _accountToLeaf(address account) internal pure returns (bytes32) {
        return bytes32(uint256(uint160(account)));
    }

    function AllSceneMintState() public view returns (bool[] memory) {
        bool[] memory state = new bool[](MAX_SCENE_SUPPLY);
        for (uint256 i = 0; i < MAX_SCENE_SUPPLY; i++) {
            for (uint256 j = 0; j < SCENE_PLOT_NUMBER; j++) {
                if (_exists(i*SCENE_PLOT_NUMBER + j)) {
                    state[i] = true;
                    break;
                }
            }
        }
        return state;
    }

    function SceneMintState(uint256 sceneId) public view returns (bool[] memory) {
        bool[] memory state = new bool[](SCENE_PLOT_NUMBER);
        for (uint256 i = 0; i < SCENE_PLOT_NUMBER; i++) {
            if (_exists(sceneId*SCENE_PLOT_NUMBER + i)) {
                state[i] = true;
            }
        }
        return state;
    }

    function PlotMintState(uint256 plotId) public view returns (bool) {
        return _exists(plotId);
    }
}
```

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