SLOWMIST

Smart Contract Security Audit Report

The SlowMist Security Team received the team's application for smart contract security audit of the EyesFiGhost (GHOST) on 2024.03.08. The following are the details and results of this smart contract security audit:

Token Name :

EyesFiGhost (GHOST)

The contract address :

https://arbiscan.io/address/0xb31add6a99a50b72330689c775ed3d3f7895218e

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
13	Safety Design Audit	Passed
14	Non-privacy/Non-dark Coin Audit	Passed

Audit Result : Passed

Audit Number : 0X002403080002

Audit Date : 2024.03.08 - 2024.03.08

Audit Team : SlowMist Security Team

Summary conclusion : This is a token contract that contains the section and dark coin functions. The total amount of contract tokens can be changed, the users can burn their tokens through the burn functions. SafeMath security module is used, which is a recommended approach. 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 add or remove the Signer through the addSigner and removeSigner functions.
2. Only the Signer can call the mint function to mint tokens.
3. The owner role can update the burnFee but there is no upper limit for the burnFee, and the owner role is an EOA address 0xfFF7d631Fd633130E5c1D9e6d251F51d50155559.
4. The user with enough GHOST tokens can provide the burnFee to remove any tokenId from the tokenIdSets without verifying that the tokenId belongs to the current user.
5. There is no refund logic for users who call the burn function with more than burnFee native tokens, but the owner role can call the release function to withdraw all the native tokens in the contract.

After communicating with the project team, they expressed that the native asset of their business design is stored in BTC. There are no native assets in the EVM. Assets will only be cross-chained from the BTC network to EVM. Once a user chooses to cross-chain, the assets corresponding to the ID on the BTC no longer belong to him, which is equivalent to entering a pool. But users can burn 10,000 GHOST ERC20 tokens to withdraw the assets on the BTC of any ID. If a user wants to withdraw the specified ID NFT when cross-chain, then there must be a user who purchases 10,000 GHOST ERC20 tokens in EVM and cannot withdraw an NFT on BTC.

All cross-chain exchange processes are handled centrally, and the centralized processing part is not within the scope of this audit.

The source code:

EyesfiGhost.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/utils/cryptography/ECDSA.sol";
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
import "@openzeppelin/contracts/utils/Address.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/security/Pausable.sol";
import "@openzeppelin/contracts/utils/structs/EnumerableSet.sol";

contract EyesfiGhost is ERC20, Pausable, Ownable, ReentrancyGuard {

    using ECDSA for bytes32;
    using EnumerableSet for EnumerableSet.UintSet;

    uint256 public maxSupply;
    uint256 public maxCount;
    uint256 public burnFee;
    uint8 public power;
    EnumerableSet.UintSet private tokenIdSets;
    mapping(address => bool) public signerSets;
    mapping(address => uint256) public nonceSets;

    event MintNFT(address indexed receiver, uint256 tokenId, uint256 time);
    event BurnNFT(address indexed burner, uint256 tokenId, string btcAddress);

    constructor(string memory name_, string memory symbol_, uint256 maxCount_, uint8
power_, uint256 burnFee_) ERC20(name_, symbol_) {
        require(maxCount_ > 0, "invalid maxCount");
        maxCount = maxCount_;
        power = power_;
        maxSupply = maxCount_ * _getUnit();
        burnFee = burnFee_;
    }
    //SlowMist// Only the owner role can add the Signer
    function addSigner(address _signer) public onlyOwner {
        signerSets[_signer] = true;
    }
    //SlowMist// Only the owner role can remove the Signer
    function removeSigner(address _signer) public onlyOwner {
        signerSets[_signer] = false;
    }
    //SlowMist// The owner role can update the burnFee and there is no upper limit for
the burnFee
    function updateBurnFee(uint256 _burnFee) public onlyOwner {
        burnFee = _burnFee;
    }

    function _hash(uint256 _tokenId, address _address) internal view returns
(bytes32)
    {
        return keccak256(abi.encode(_tokenId, getChainID(), nonceSets[_address],
address(this), _address));
    }

    function _verify(bytes32 shash, bytes memory token) internal view returns (bool)
    {
        return signerSets[_recover(shash, token)] ;
    }

    function _recover(bytes32 shash, bytes memory token) internal pure returns
(address)
    {
        return shash.toEthSignedMessageHash().recover(token);
    }

    function _getUnit() internal view returns (uint256) {
        return 10 ** (decimals() + power);
    }

    function getChainID() internal view returns (uint256) {
        uint256 id;
        assembly {
            id := chainid()
        }
        return id;
    }

    function _beforeTokenTransfer(address from, address to, uint256 amount) internal
override whenNotPaused {}
    //SlowMist// Suspending all transactions upon major abnormalities is a recommended
approach
    function pause() external onlyOwner {
        _pause();
    }

    function unpause() external onlyOwner {
        _unpause();
    }
    //SlowMist// Only the Signer can call the mint function
    function mint(uint256 tokenId, bytes memory token) external whenNotPaused
nonReentrant {
        uint256 unit = _getUnit();
        require((totalSupply() + unit) <= maxSupply, "reach max");
        require(_verify(_hash(tokenId, msg.sender), token), "Invalid signature");
        nonceSets[msg.sender]++;

        require(tokenId < maxCount, "Invalid tokenId");
        require(!tokenIdSets.contains(tokenId), "tokenId has been used");
        tokenIdSets.add(tokenId);

        _mint(msg.sender, unit);
        emit MintNFT(msg.sender, tokenId, block.timestamp);
    }
    //SlowMist// The user who have the enough GHOST tokens can provide the burnFee to
remove any tokenId from the tokenIdSets
    function burn(uint256 tokenId, string memory btcAddress) external payable
whenNotPaused nonReentrant {
        require(msg.value >= burnFee, "Insufficient burn fee");

        require(bytes(btcAddress).length != 0, "Invalid BTC address");
        require(tokenIdSets.contains(tokenId), "Invalid tokenId");

        tokenIdSets.remove(tokenId);
        _burn(msg.sender, _getUnit());

        emit BurnNFT(msg.sender, tokenId, btcAddress);
    }

    function getNonce(address _address) public view returns (uint256) {
        return nonceSets[_address];
    }

    function release() external onlyOwner {
        Address.sendValue(payable(owner()), address(this).balance);
    }

    function getTokenIds() external view returns (uint256[] memory) {
        return tokenIdSets.values();
    }

    function getTokenIdByIndex(uint256 index) external view returns (uint256) {
        return tokenIdSets.at(index);
    }

    function getTokenIdsLength() external view returns (uint256) {
        return tokenIdSets.length();
    }
}
```

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.