verichains

SECURITY AUDIT OF

MICROPHONE NFT SMART CONTRACTS

Public Report

Sep 05, 2022

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.

EXECUTIVE SUMMARY

This Security Audit Report prepared by Verichains Lab on Sep 05, 2022. We would like to thank the SingSing for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Microphone NFT Smart Contracts. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified some vulnerable issues in the smart contracts code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY 5

1.1. About Microphone NFT Smart Contracts 5

1.2. Audit scope 5

1.3. Audit methodology 5

1.4. Disclaimer 6

2. AUDIT RESULT 8

2.1. Overview 8

2.1.1. Management contract 8

2.1.2. ESing contract 8

2.1.3. MicroNFT contract 8

2.1.4. Breeding contract 8

2.1.5. LootBox contract 8

2.2. Findings 8

2.2.1. Breeding.sol, LootBox.sol - Missing non-contract call checking CRITICAL 9

2.2.2. Breeding.sol - Wrong rate in `init` function MEDIUM 9

2.2.3. Breeding.sol - Breeding fees should be calculated from both `matron` and `siren` or the higher rarity one LOW 10

2.2.4. Breeding.sol - Update `maxBreedTimes` may let the users breed with zero fees INFORMATIVE 10

2.2.5. Breeding.sol, LootBox.sol - Unnecessary delete INFORMATIVE 12

2.2.6. Management.sol - Unnecessary `Ownable` constructor call INFORMATIVE 13

3. VERSION HISTORY 14

1. MANAGEMENT SUMMARY

1.1. About Microphone NFT Smart Contracts

Sing To Earn is one of the special models of the SingSing music platform, combining Game-Fi and Social-Fi elements for the community that loves singing and wants to connect with people with similar interests.

Here, users equip themselves with NFT Micro to be able to sing Karaoke for income. Sing To Earn uses Singing's AI Scoring System technology to score and objectively evaluate based on analysis of millions of previous songs from the community.

In addition, Sing to Earn has many Game Modes that bring unprecedented special experiences in the market when combined with other players through interactive activities on the SingSing (Social-Fi) community.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the Microphone NFT Smart Contracts.

It was conducted on commit `54a38b0c8a42fd250c5a3d0281ef2627a5bb3c82` from git repository *https://github.com/phamsonha/MicroNFT*.

The latest version of the following files were made available in the course of the review:

SHA256 Sum	File
`ef9f00f3815bd9814945f2ab36589c6305a1b8865df5b5618c6a15cd35a5310b`	`Breeding.sol`
`1d05e309cc3056fc00d3e726cd43cc4784c40839a2a040be478779de2dadb57a`	`ESing.sol`
`3d4111d2be021985f5e29629682ab755aa40a9b169fd38d15166f08650e6e23b`	`LootBox.sol`
`671f1f590e41872b7076624fc95ab82eab56a154be11593a54195eb289cce061`	`Management.sol`
`f6711a11a49991fbaf6ebfb4563c2c8bf0ab2e3f80cd13b1aaceaf64d106bae8`	`MicroNFT.sol`

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.

- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract.

However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

2. AUDIT RESULT

2.1. Overview

The Microphone NFT Smart Contracts were written in `Solidity` language, with the required version to be `^0.8.0`. The source code was written based on OpenZeppelin's library.

The Microphone NFT Smart Contracts contains 5 main contracts: `Management`, `ESing`, `MicroNFT`, `Breeding` and `LootBox`.

2.1.1. Management contract

This is the management contract which is used to restricting system access to authorized users (`admin`, `minter`, `verifier`). It also manages the address of other contracts in the Microphone NFT Smart Contracts.

2.1.2. ESing contract

This is the ERC20 token contract in the Microphone NFT Smart Contracts, which extends `ERC20` contracts. `minter` can mint unlimited amount of tokens.

2.1.3. MicroNFT contract

This is ERC721 upgradable NFT contract in the Microphone NFT Smart Contracts, which extends `ERC721EnumerableUpgradeable` contract. Only `minter` and `lootbox` can mint new Micro NFTs.

2.1.4. Breeding contract

This is a breeding contract which lets 2 Micro NFTs to breed a Micro `LootBox` with a random rarity using Verichains' on-chain RNG service. The rarity probability of Micro `LootBox` and the fees to breed can be changed by `admin` at anytime.

2.1.5. LootBox contract

This contract lets users `unbox` their Micro `LootBox` to get a Micro NFT with random appearance (body and head type), `kind` and `class`. The rarity probability of Micro NFT `kind` and `class` can be changed by `admin` at anytime.

2.2. Findings

During the audit process, the audit team found some vulnerabilities in the given version of Microphone NFT Smart Contracts.

SingSing fixed the code according to Verichains's draft report in commit `3a60909ece90e6dc7112504b3f733296c6900d49`.

2.2.1. Breeding.sol, LootBox.sol - Missing non-contract call checking CRITICAL

The `breed` and `unbox` functions are missing non-contract call checking. If functions using RNG does not block contract call, attackers can use contract to call-revert to control the result of random and get the best value.

RECOMMENDATION

Adding `onlyNonContract` modifier to all functions which use the `management.getRandom()` function.

```
modifier onlyNonContract {
    require(tx.origin == msg.sender, "Only non-contract call");
    _;
}
```

UPDATES

- *July 11, 2022*: This issue has been acknowledged and fixed by the SingSing team.

2.2.2. Breeding.sol - Wrong rate in `init` function MEDIUM

In `init` function, the `dropRates[3][3]` is wrong, it should be `dropRates[3][3] = [0, 0, 0, 98, 2];` or whatever higher than `dropRates[2][3]`.

```
function init(address _management)
    public
    AddressZero(_management)
    initializer
{
    ...
    dropRates[0][0] = [100, 0, 0, 0, 0];
    dropRates[0][1] = [50, 49, 1, 0, 0];
    dropRates[0][2] = [50, 0, 49, 1, 0];
    dropRates[0][3] = [50, 0, 0, 49, 1];
    dropRates[0][4] = [50, 0, 0, 0, 50];
    dropRates[1][1] = [0, 98, 2, 0, 0];
    dropRates[1][2] = [0, 49, 59, 1, 0];
    dropRates[1][3] = [0, 49, 1, 49, 1];
    dropRates[1][4] = [0, 49, 1, 0, 50];
    dropRates[2][2] = [0, 0, 98, 2, 0];
    dropRates[2][3] = [0, 0, 49, 50, 1];
    dropRates[2][4] = [0, 0, 49, 1, 50];
    dropRates[3][3] = [0, 0, 98, 2, 0]; // Wrong rate
    dropRates[3][4] = [0, 0, 0, 49, 51];
    dropRates[4][4] = [0, 0, 0, 0, 100];
    ...
}
```

RECOMMENDATION

Fix the wrong rate.

UPDATES

- *July 11, 2022*: This issue has been acknowledged and fixed by the SingSing team.

2.2.3. Breeding.sol - Breeding fees should be calculated from both `matron` and `siren` or the higher rarity one LOW

Breeding fees is currently calculated from `matron` rarity (the first one) so users can call `breed` with the lower rarity one to be `matron` to decrease the breeding fees. The breeding fees should be calculated from either both `matron` and `siren` or the higher rarity one.

```
function breed(
    uint256 _matronId,
    uint256 _sireId,
    bytes calldata _signature
) external override {
    ...
    // Get breeding fees
    uint256 bFee = busdFees[matron.class][matron.cooldownIndex];
    uint256 sFee = eSingFees[matron.class][matron.cooldownIndex];
    ...
}
```

RECOMMENDATION

Calculating breeding fees with the higher rarity one or both.

UPDATES

- *Sep 05, 2022*: This issue has been acknowledged and fixed by the SingSing team.

2.2.4. Breeding.sol - Update `maxBreedTimes` may let the users breed with zero fees INFORMATIVE

The current breeding fees are only set for 5 breed times. When updating `maxBreedTimes` to more than 5 times, the fees are not set so Micros with breeding times more than 5 can breed without any fees.

```
function updateMaxBreedTimes(uint256 _maxBreedTimes)
    external
    override
    onlyAdmin
{
    require(_maxBreedTimes > 0, "_maxBreedTimes must be > 0");
```

```
    maxBreedTimes = _maxBreedTimes;
}

function updateBusdFees(uint256 _microClass, uint256[] calldata _busdFees)
    public
    override
    onlyAdmin
{
    require(
        _microClass < TOTAL_CLASSES,
        "_microClass must be < TOTAL_CLASSES"
    );
    require(
        _busdFees.length == maxBreedTimes,
        "_busdFees length must be = maxBreedTimes"
    );
    delete busdFees[_microClass];
    busdFees[_microClass] = _busdFees;
}

function updateESingFees(uint256 _microClass, uint256[] calldata _eSingFees)
    public
    override
    onlyAdmin
{
    require(
        _microClass < TOTAL_CLASSES,
        "_microClass must be < TOTAL_CLASSES"
    );
    require(
        _eSingFees.length == maxBreedTimes,
        "_eSingFees length must be = maxBreedTimes"
    );
    delete eSingFees[_microClass];
    eSingFees[_microClass] = _eSingFees;
}
```

RECOMMENDATION

- Disable signing signature for breeding before `updateMaxBreedTimes` to a higher one, after that, update the fees and then re-enable signing signature.
- Update the fees when `updateMaxBreedTimes` if necessary.

UPDATES

- *July 11, 2022*: This issue has been acknowledged by the SingSing team.

2.2.5. Breeding.sol, LootBox.sol - Unnecessary delete INFORMATIVE

It is unnecessary to `delete` item in mapping before reassigning it. It will only take more gas.

```
function updateBusdFees(uint256 _microClass, uint256[] calldata _busdFees)
    public
    override
    onlyAdmin
{
    ...
    delete busdFees[_microClass];
    busdFees[_microClass] = _busdFees;
}

function updateESingFees(uint256 _microClass, uint256[] calldata _eSingFees)
    public
    override
    onlyAdmin
{
    ...
    delete eSingFees[_microClass];
    eSingFees[_microClass] = _eSingFees;
}

function updateDropRates(
    uint8 _classCol1,
    uint8 _classCol2,
    uint256[] calldata _dropRates
) external override onlyAdmin {
    ...
    delete dropRates[_classCol1][_classCol2];
    dropRates[_classCol1][_classCol2] = _dropRates;
}

function updateKindDropRates(
    uint8 _typeCol1,
    uint8 _typeCol2,
    uint256[] calldata _dropRates
) external override onlyAdmin {
    ...
    delete kindDropRates[_typeCol1][_typeCol2];
    kindDropRates[_typeCol1][_typeCol2] = _dropRates;
}
```

RECOMMENDATION

Removing `delete` code.

UPDATES

- *July 11, 2022*: This issue has been acknowledged and fixed by the SingSing team.

2.2.6. Management.sol - Unnecessary `Ownable` constructor call INFORMATIVE

`Ownable` constructor take no arguments, so we don't need to call it manually.

```
constructor(
    address _treasury,
    address _verifier,
    address _minter,
    address _randomService
) Ownable() {
    treasury = _treasury;
    verifier = _verifier;
    minter = _minter;
    randomService = IRegistry(_randomService);
}
```

RECOMMENDATION

Removing unnecessary `Ownable()`.

```
constructor(
    address _treasury,
    address _verifier,
    address _minter,
    address _randomService
) {
    treasury = _treasury;
    verifier = _verifier;
    minter = _minter;
    randomService = IRegistry(_randomService);
}
```

UPDATES

- *July 11, 2022*: This issue has been acknowledged and fixed by the SingSing team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*July 11, 2022*	Public Report	Verichains Lab
1.1	*Sep 05, 2022*	Public Report	Verichains Lab

Table 2. Report versions history