PROTOCOL SECURITY AUDIT REPORT

MONKEY KINGDOM

Decurity, 2022

CONTENTS

1 GENERAL INFORMATION

This report contains information about the results of the security audit of the MonkeyKingdom.io smart contracts, conducted by Decurity in the period from 03/24/22 to 03/30/22.

1.1 Introduction

Tasks solved during the work are:

- Review the protocol design and the usage of 3rd party dependencies,
- Audit the contracts implementation,
- Develop the recommendations and suggestions to improve the security of the contracts.

1.2 Scope of Work

The audit scope included the following contracts:

- MKLockRegistry.sol
- MonkeyLegends.sol

Both contracts located in the repository https://github.com/0xBURP/MonkeyKingdom-MonkeyLegends.

Initially analyzed commit hash: 4e1008e1a6f7aa072507e92b53baf1787903a6a4.

Re-tested commit hash: 1f8a98f3500e0b001fd3013328d6b07a200d18a5.

The deployed contract with the etherscan-verified source code: https://etherscan.io/address/0xa8ed317c5a491bbd9127fd2fce6ec7f409b40783#code

We've additionally verified that the source code of *ERC721A.sol*, *MKLockRegistry.sol* and *MonkeyLegends.sol* contracts deployed on etherscan match the source code on Github.

1.3 Threat Model

The assessment presumes actions of an intruder who might have capabilities of an external user. The centralization risks have not been considered upon the request of the Customer.

1.4 Weakness Scoring

An expert evaluation scores the findings in this report, an impact of each vulnerability is calculated based on its ease of exploitation (based on the industry practice and our experience) and severity (for the considered threats).

2 SUMMARY

As a result of this work, we have discovered a single critical exploitable security issue which has been fixed and re-tested in the course of the work.

The other suggestions included fixing the low-risk issues and some best practices (see 3.1).

The MonkeyKingdom team has given the feedback for the suggested changes and explanation for the underlying code.

2.1 Suggestions

The table below contains the discovered issues, their risk level, and their status as of 04/31/2022.

Table 1. Discovered weaknesses

Issue	Risk Level	Probability	Status
Insecure usage of abi.encodePacked in the signature verification	Critical	High	Fixed
No ERC20 token rescue function	Low	Low	Fixed
Signature malleability	Low	Low	Accepted The signature doesn't have to be unique

3 GENERAL RECOMMENDATIONS

This section contains general recommendations how to fix discovered during the testing weaknesses and vulnerabilities and how to improve overall security level.

Section 3.1 contains a list of general mitigations against the discovered weaknesses, technical recommendations for each finding can be found in section 4.

Section 3.2 describes a brief long-term action plan to mitigate further weaknesses and bring the product security to a higher level.

3.1 Current findings remediation

- Fix the issues described in the section 4.2.
- Perform review and verification of all the business scenarios of the contracts,
- Test the production environment of the off-chain service when deployed.

Other best practices recommendations:

- Consider reducing *uint256* to *uint8* for the mappings (such as **BreedCount*) where the value has to be small,
- Consider adding a check for *n != 0* in *claim* (it's checked later in *ERC721A*'s *_mint* method anyway),
- Consider returning the dust *ETH* during the *mintSignedWhitelist* calls.

3.2 Security process improvement

- Keep the whitepaper and documentation updated to make it consistent with the implementation and the intended use cases of the system,
- Launch a public bug bounty campaign for the contracts.

4 DETAILS

4.1 Findings

4.1.1 Insecure usage of abi.encodePacked for the signature verification

Summary:

The *breed* method of the *MonkeyLegends* contract accepts the hashes of Solana assets and the server-generated signature to mint a single Monkey asset for the user. The signed value is comprised of the hashes and generated using the *abi.encodePacked* function:

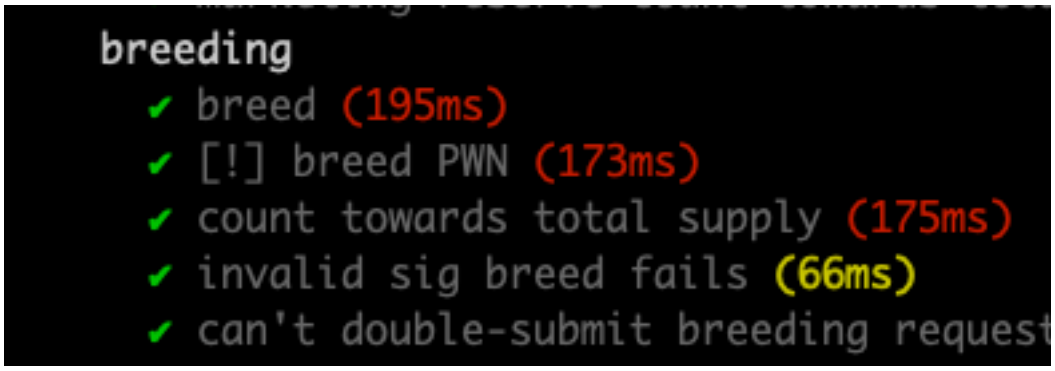
```
104     function breed(
105         string calldata mkHash,
106         string calldata dbHash,
107         string calldata peachHash,
108         bytes calldata sig
109     ) external {
110         require(numMonkeysBreeded + 1 <= MAX_BREED, "Max no. breed reached");
111         bytes memory b = abi.encodePacked(
112             mkHash,
113             dbHash,
114             peachHash,
115             _msgSender()
116         );
117         require(recoverSigner(keccak256(b), sig) == authSigner, "Invalid sig");
118 
119         unchecked {
120             require(wukongsBreedCount[mkHash] < 2, "Wukong breed twice");
121             wukongsBreedCount[mkHash]++;
122             require(baepesBreedCount[dbHash] < 2, "Baepe breed twice");
123             baepesBreedCount[dbHash]++;
124             require(!peachUsed[peachHash], "Peach has been used");
125             peachUsed[peachHash] = true;
126         }
127 
128         super._safeMint(_msgSender(), 1);
129         numMonkeysBreeded++;
130     }
```

An attacker can rearrange the used hash values to get the new ones so that the signature will remain intact. As a result, numerous (the exact number equals the sum of the hashes lengths) redundant assets can be breeded by reusing the same signature.

Sample exploit test code:

```
it("[!] breed PWN", async () => {
  const { monkey, numReserve } = await loadFixture(fixture);
  const d0 = ["abc", "def", "ghi"];
  const d1 = ["abcd", "efg", "hi"];
  const sig1 = genBreedSig(d0, whitelist[0]);
  await monkey.breed(d1[0], d1[1], d1[2], sig1);
  expect(await monkey.balanceOf(acc[0].address)).to.equal(numReserve + 1);
});
```

Result:



Remediation:

Use *abi.encode* instead of *abi.encodePacked*.

Status:

The vulnerability has been fixed along with the bulk breeding feature implementation:

```
66      function breed(string[] calldata hashes, bytes calldata sig) external {
67          uint256 numToMint = hashes.length / 3;
68          require(numBreeded + numToMint <= MAX_BREED, "MAX_BREED reached");
69          require(hashes.length % 3 == 1, "Invalid call");
70          bytes memory b = abi.encode(hashes, _msgSender());
71          require(recoverSigner(keccak256(b), sig) == authSigner, "Invalid sig");
72          unchecked {
73              for (uint256 n = 0; n < 3 * numToMint; ) {
74                  string memory mkHash = hashes[n++];
75                  string memory dbHash = hashes[n++];
76                  string memory peachHash = hashes[n++];
77                  require(
78                      wukongsBreedCount[mkHash]++ < 2 &&
79                          baepesBreedCount[dbHash]++ < 2 &&
80                          !peachUsed[peachHash],
81                      "Check breeding quota"
82                  );
83                  peachUsed[peachHash] = true;
84              }
85          }
86          super._safeMint(_msgSender(), numToMint);
87          numBreeded += numToMint;
88      }
```

Note that the new implementation contains the hashes array length check which seems incorrect but as the Customer explained, it's intentional: the extra field will be used for audit and tracing purposes.

4.1.2 No ERC20 token rescue function

Summary:

The *withdrawAll* method of the *MonkeyLegends* contract sends all the *ETH* and all the *Peach* tokens to the owner. However, there's no similar rescue method for arbitrary *ERC20* tokens.

Remediation:

Add an *ERC20* rescue function.

Status:

The rescue function has been added:

```
142     // withdraw
143     function withdrawAll() external onlyOwner {
144         payable(_msgSender()).transfer(address(this).balance);
145         peach.transferFrom(
146             address(this),
147             _msgSender(),
148             peach.balanceOf(address(this))
149         );
150     }
151
152     function recoverERC20(address tokenAddress) external onlyOwner {
153         IERC20 token = IERC20(tokenAddress);
154         token.transfer(owner(), token.balanceOf(address(this)));
155     }
```

4.1.3 Signature malleability

Summary:

The *splitSignature* and *recoverSigner* methods of the MonkeyLegends contract represent a modified OpenZeppelin implementation of the corresponding mechanism (*OpenZeppelinUpgradesECDSA* library).

However, some of the checks from the original code have been removed. Those checks are needed to make the signatures unique and therefore non-malleable.

However, the current implementation of the *breed* method does not require the signatures to be unique, therefore, there's no direct risk for the Customer.

Remediation:

No action needed if the signatures are not intended to be used as a unique identifier.

5 APPENDIX

5.1 About us

The Decurity (former DeFiSecurity.io) team consists of experienced hackers who have been doing application security assessments and penetration testing for over a decade.

During the recent years, we've gained an expertise in blockchain field and have conducted numerous audits for both centralized and decentralized projects: exchanges, protocols, and blockchain nodes.

Our efforts have helped to protect hundreds of millions of dollars and make web3 a safer place.