CandyKitty-NFT

Smart Contract Security Audit

V1.0

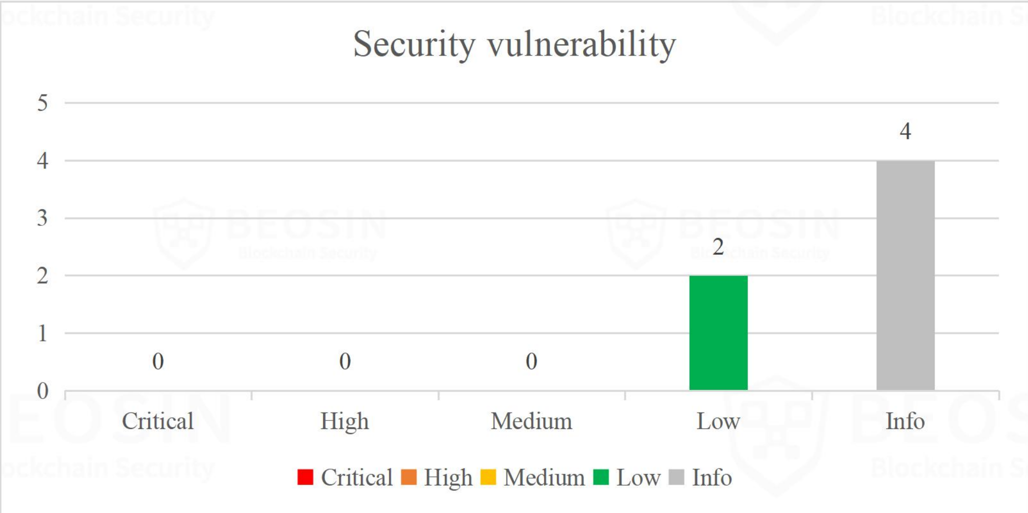
No. 202207081647

Jul 8th, 2022

Contents

Summary of Audit Results

After auditing, 2 Low-risks and 4 Info items were identified in the CandyKitty-NFT project. Specific audit details will be presented in the **Findings section**. Users should pay attention to the following aspects when interacting with this project:



***Notes:**

- **Project Description:**

1. Business overview

The CandyKitty project is an ERC721 contract deployed on Ethereum. The contract has four minting modes. The preSaleMint model uses signature verification and pre-sale to sell tokens. The publicMint mode is a common token sale method, the freeMint mode is to issue a single token through signature verification, and the whitelistMint mode supports signature verification to issue multiple tokens. And the nesting time of the token can be recorded.

1 Overview

1.1 Project Overview

Project Name	CandyKitty-NFT
Platform	Ethereum
Audit scope	https://github.com/candykitty-official/CandyKitty-NFT/
Commit Hash	8ae97d25c03caea157511ff15e0e531998829ba1(Initial) 765331a9b9493e735ab4d65a79b2eb2b0add053f(Latest)

1.2 Audit Overview

Audit work duration：Jul 06, 2022 – Jul 08, 2022

Audit methods: Formal Verification, Static Analysis, Typical Case Testing and Manual Review.

Audit team: Beosin Technology Co. Ltd.

2 Findings

Index	Risk description	Severity level	Status
CandyKitty-1	Duplicate tokenIdx	Low	Fixed
CandyKitty-2	MaxSupply is invalid	Low	Fixed
CandyKitty-3	Missing zero address judgment	Info	Fixed
CandyKitty-4	Missing event trigger	Info	Fixed
CandyKitty-5	Signature verification risk	Info	Fixed
CandyKitty-6	Transaction-ordering dependent	Info	Fixed

Finding Details:

[CandyKitty-1] Duplicate tokenIdx

Severity Level	Low
Type	Business Security
Lines	CandyKitty.sol #L87-103
Description	tokenIdx can be specified when minting token via the whitelistMint function. When switching to other minting model, tokenIdx may be occupied, causing minting failure.

```
function whitelistMint( address _to, uint256[] calldata _tokenIds, bytes32 _nonce, bytes calldata _sig) external payable {
    storage
    require(status == Status.Whitelist, "not in whitelist mint period");
    bytes memory bNonce = new bytes((_tokenIds.length+1)*32);
    for (uint256 idx = 0; idx < _tokenIds.length; idx++) {
        uint256 tokenId = _tokenIds[idx];
        assembly { mstore(add(bNonce, add(0x20, mul(idx, 0x20))), tokenId) }
    }
    assembly { mstore(add(bNonce, mul(add(_tokenIds.length, 1), 32)), _nonce) }

    requireValidSignature(_to, keccak256(bNonce), _sig);

    for (uint256 idx = 0; idx < _tokenIds.length; idx++) {
        _safeMint(_to, _tokenIds[idx], "");
        emit WhitelistMint(_to, _tokenIds[idx]);
    }
}
```

Figure 1 The source code of *whitelistMint* function(Unfixed)

Recommendations	Restricted tokenIdx must be within 1000.
Status	Fixed.

```
function whitelistMint( address _to, uint256[] calldata _tokenIds, bytes32 _nonce, bytes calldata _sig) external payable {
    require(status == Status.Whitelist, "not in whitelist mint period");
    bytes memory bNonce = new bytes((_tokenIds.length+1)*32);
    for (uint256 idx = 0; idx < _tokenIds.length; idx++) {
        uint256 tokenId = _tokenIds[idx];
        require(tokenId <= 1000 && tokenId > 0, "incorrect tokenId");
        assembly { mstore(add(bNonce, add(0x20, mul(idx, 0x20))), tokenId) }
    }
    assembly { mstore(add(bNonce, mul(add(_tokenIds.length, 1), 32)), _nonce) }

    requireValidSignature(_to, keccak256(bNonce), _sig);

    for (uint256 idx = 0; idx < _tokenIds.length; idx++) {
        _safeMint(_to, _tokenIds[idx], "");
        emit WhitelistMint(_to, _tokenIds[idx]);
    }
}
```

Figure 2 The source code of *whitelistMint* function(Fixed)

[CandyKitty-2] MaxSupply is invalid

Severity Level	**Low**
Type	Business Security
Lines	CandyKitty.sol #L87-103
Description	Minting using the whitelistMint function can exceed the maximum constraint and can mint infinitely.

```
function whitelistMint( address _to, uint256[] calldata _tokenIds, bytes32 _nonce, bytes calldata _sig) external payable {
    storage
    require(status == Status.Whitelist, "not in whitelist mint period");
    bytes memory bNonce = new bytes((_tokenIds.length+1)*32);
    for (uint256 idx = 0; idx < _tokenIds.length; idx++) {
        uint256 tokenId = _tokenIds[idx];
        assembly { mstore(add(bNonce, add(0x20, mul(idx, 0x20))), tokenId) }
    }
    assembly { mstore(add(bNonce, mul(add(_tokenIds.length, 1), 32)), _nonce) }

    requireValidSignature(_to, keccak256(bNonce), _sig);

    for (uint256 idx = 0; idx < _tokenIds.length; idx++) {
        _safeMint(_to, _tokenIds[idx], "");
        emit WhitelistMint(_to, _tokenIds[idx]);
    }
}
```

Figure 3 The source code of *whitelistMint* function(Unfixed)

Recommendations	Restricted tokenIdx must be within 1000.
Status	Fixed.

```
function whitelistMint( address _to, uint256[] calldata _tokenIds, bytes32 _nonce, bytes calldata _sig) external payable {
    require(status == Status.Whitelist, "not in whitelist mint period");
    bytes memory bNonce = new bytes((_tokenIds.length+1)*32);
    for (uint256 idx = 0; idx < _tokenIds.length; idx++) {
        uint256 tokenId = _tokenIds[idx];
        require(tokenId <= 1000 && tokenId > 0, "incorrect tokenId");
        assembly { mstore(add(bNonce, add(0x20, mul(idx, 0x20))), tokenId) }
    }
    assembly { mstore(add(bNonce, mul(add(_tokenIds.length, 1), 32)), _nonce) }

    requireValidSignature(_to, keccak256(bNonce), _sig);

    for (uint256 idx = 0; idx < _tokenIds.length; idx++) {
        _safeMint(_to, _tokenIds[idx], "");
        emit WhitelistMint(_to, _tokenIds[idx]);
    }
}
```

Figure 4 The source code of *whitelistMint* function(Fixed)

[CandyKitty-3] Missing zero address judgment

Severity Level	Info
Type	Business Security
Lines	CandyKitty.sol #L216-226
Description	When using *withdraw* method to withdraw ETH, if "_recipient" is zero-address, funds may be lost.

```
function withdraw(address payable _recipient, IERC20 _token) external onlyOwner returns(bool) {
    if (address(_token) == address(0x00)) {
        uint256 balance = address(this).balance;
        _recipient.transfer(balance);
    } else {
        uint256 balance = _token.balanceOf(address(this));
        _token.safeTransfer(_recipient, balance);
    }
    return true;
}
```

Figure 5 The source code of *withdraw* function(Unfixed)

Recommendations	Increase the zero address judgment of the "_recipient".
Status	Fixed.

```
function withdraw(address payable _recipient, IERC20 _token) external onlyOwner returns(bool) {
    require(_recipient != address(0x00), "recipient is zero address");
    if (address(_token) == address(0x00)) {
        uint256 balance = address(this).balance;
        _recipient.transfer(balance);
    } else {
        uint256 balance = _token.balanceOf(address(this));
        _token.safeTransfer(_recipient, balance);
    }
    return true;
}
```

Figure 6 The source code of *withdraw* function(Fixed)

[CandyKitty-4] Missing event trigger

Severity Level	Info
Type	Business Security
Lines	CandyKitty.sol #L118-120
Description	No events triggered when calling *setNestingOpen*, *setPrice*, *setStatus*, *setBaseURI*, *setSignerAddress*.

```
function setSignerAddress(address _newSigner) external onlyOwner {
    signerAddress = _newSigner;
}
```

Figure 7 The source code of *setSignerAddress* function(Unfixed)

```
function setNestingOpen(bool open) external onlyOwner {
    nestingOpen = open;
}
```

Figure 8 The source code of *setNestingOpen* function(Unfixed)

```
function setBaseURI(string memory _uri) public onlyOwner returns(bool) {
    baseURI = _uri;
    return true;
}
```

Figure 9 The source code of *setBaseURI* function(Unfixed)

```
function setStatus(Status _status) external onlyOwner {
    status = _status;
}
```

Figure 10 The source code of *setStatus* function(Unfixed)

```
function setPrice(uint256 _preSalePrice, uint256 _publicSalePrice) external onlyOwner {
    preSalePrice = _preSalePrice;
    publicSalePrice = _publicSalePrice;
}
```

Figure 11 The source code of *setPrice* function(Unfixed)

Recommendations	It is recommended to add related events and trigger them in the corresponding functions.

Status Fixed.

```
function setSignerAddress(address _newSigner) external onlyOwner {
    signerAddress = _newSigner;
    emit NewSignerAddress(_newSigner);
}
```

Figure 12 The source code of *setSignerAddress* function(Fixed)

```
function setNestingOpen(bool open) external onlyOwner {
    nestingOpen = open;
    emit NestingOpened(open);
}
```

Figure 13 The source code of *setNestingOpen* function(Fixed)

```
function setBaseURI(string memory _uri) public onlyOwner returns(bool) {
    baseURI = _uri;
    emit NewBaseURI(_uri);
    return true;
}
```

Figure 14 The source code of *setBaseURI* function(Fixed)

```
function setStatus(Status _status) external onlyOwner {
    status = _status;
    emit NewStatus(_status);
}
```

Figure 15 The source code of *setStatus* function(Fixed)

```
function setPrice(uint256 _preSalePrice, uint256 _publicSalePrice) external onlyOwner {
    preSalePrice = _preSalePrice;
    publicSalePrice = _publicSalePrice;
    emit NewPrice(_preSalePrice, _publicSalePrice);
}
```

Figure 16 The source code of *setPrice* function(Fixed)

[CandyKitty-5] Signature verification risk

Severity Level	Info
Type	Business Security
Lines	CandyKitty.sol #L122-128
Description	Missing zero address filtering in signature verification. There is a risk of being bypassed if the deployer sets the “signerAddress” to zero address.

```
function requireValidSignature(address _to, bytes32 _nonce, bytes memory _sig) internal {
    bytes32 message = ECDSA.toEthSignedMessageHash(abi.encodePacked(_to, _nonce));
    require(!usedMessages[message], "SignatureChecker: Message already used");
    usedMessages[message] = true;
    address signer = ECDSA.recover(message, _sig);
    require(signer == signerAddress, "SignatureChecker: Invalid signature");
}
```

Figure 17 The source code of *requireValidSignature* function(Unfixed)

Recommendations	“signer” cannot be zero address.
Status	Fixed.

```
function requireValidSignature(address _to, bytes32 _nonce, bytes memory _sig) internal {
    require(msg.sender == _to, "_to address must be the caller address");
    bytes32 message = ECDSA.toEthSignedMessageHash(abi.encodePacked(_to, _nonce));
    require(!usedMessages[message], "SignatureChecker: Message already used");
    usedMessages[message] = true;
    address signer = ECDSA.recover(message, _sig);
    require(signer != address(0x00), "signer cannot be zero address");
    require(signer == signerAddress, "SignatureChecker: Invalid signature");
}
```

Figure 18 The source code of *requireValidSignature* function(Fixed)

[CandyKitty-6] Transaction-ordering dependent

Severity Level	Info
Type	Business Security
Lines	CandyKitty.sol #L61-68
Description	The *preSaleMint* function has the risk of transaction-ordering dependence. An attacker can first spend the price of one token to use the user's “message”, so that the user can only get one token and lose the opportunity to buy more tokens.

```
function preSaleMint(address _to, uint256 _quantity, bytes32 _nonce, bytes calldata _sig) external payable {
    require(status == Status.PreSale, "not in presale mint period");
    require(_quantity > 0, "_quantity must greater than zero");
    require(tokenIdx + _quantity <= maxSupply, "exceed max mint amount");
    require(msg.value == _quantity*preSalePrice, "insufficient value");
    requireValidSignature(_to, _nonce, _sig);
    _batchMint(_to, _quantity);
}
```

Figure 19 The source code of *preSaleMint* function(Unfixed)

Recommendations	The *requireValidSignature* function passes in the caller address or “_to” must be the caller address.
Status	Fixed.

```
function requireValidSignature(address _to, bytes32 _nonce, bytes memory _sig) internal {
    require(msg.sender == _to, "_to address must be the caller address");
    bytes32 message = ECDSA.toEthSignedMessageHash(abi.encodePacked(_to, _nonce));
    require(!usedMessages[message], "SignatureChecker: Message already used");
    usedMessages[message] = true;
    address signer = ECDSA.recover(message, _sig);
    require(signer != address(0x00), "signer cannot be zero address");
    require(signer == signerAddress, "SignatureChecker: Invalid signature");
}
```

Figure 20 The source code of *requireValidSignature* function(Fixed)

3 Appendix

3.1 Vulnerability Assessment Metrics and Status in Smart Contracts

3.1.1 Metrics

In order to objectively assess the severity level of vulnerabilities in blockchain systems, this report provides detailed assessment metrics for security vulnerabilities in smart contracts with reference to CVSS 3.1 (Common Vulnerability Scoring System Ver 3.1).

According to the severity level of vulnerability, the vulnerabilities are classified into four levels: "critical", "high", "medium" and "low". It mainly relies on the degree of impact and likelihood of exploitation of the vulnerability, supplemented by other comprehensive factors to determine of the severity level.

Impact / Likelihood	Severe	High	Medium	Low
Probable	Critical	High	Medium	Low
Possible	High	High	Medium	Low
Unlikely	Medium	Medium	Low	Info
Rare	Low	Low	Info	Info

3.1.2 Degree of impact

- **Severe**

Severe impact generally refers to the vulnerability can have a serious impact on the confidentiality, integrity, availability of smart contracts or their economic model, which can cause substantial economic losses to the contract business system, large-scale data disruption, loss of authority management, failure of key functions, loss of credibility, or indirectly affect the operation of other smart contracts associated with it and cause substantial losses, as well as other severe and mostly irreversible harm.

- **High**

High impact generally refers to the vulnerability can have a relatively serious impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a greater economic loss, local functional unavailability, loss of credibility and other impact to the contract business system.

- **Medium**

Medium impact generally refers to the vulnerability can have a relatively minor impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a small amount of economic loss to the contract business system, individual business unavailability and other impact.

- **Low**

Low impact generally refers to the vulnerability can have a minor impact on the smart contract, which can pose certain security threat to the contract business system and needs to be improved.

3.1.4 Likelihood of Exploitation

- **Probable**

Probable likelihood generally means that the cost required to exploit the vulnerability is low, with no special exploitation threshold, and the vulnerability can be triggered consistently.

- **Possible**

Possible likelihood generally means that exploiting such vulnerability requires a certain cost, or there are certain conditions for exploitation, and the vulnerability is not easily and consistently triggered.

- **Unlikely**

Unlikely likelihood generally means that the vulnerability requires a high cost, or the exploitation conditions are very demanding and the vulnerability is highly difficult to trigger.

- **Rare**

Rare likelihood generally means that the vulnerability requires an extremely high cost or the conditions for exploitation are extremely difficult to achieve.

3.1.5 Fix Results Status

Status	Description
Fixed	The project party fully fixes a vulnerability.
Partially Fixed	The project party did not fully fix the issue, but only mitigated the issue.
Acknowledged	The project party confirms and chooses to ignore the issue.

3.2 Audit Categories

No.	Categories	Subitems
1	Coding Conventions	Compiler Version Security
		Deprecated Items
		Redundant Code
		require/assert Usage
		Gas Consumption
2	General Vulnerability	Integer Overflow/Underflow
		Reentrancy
		Pseudo-random Number Generator (PRNG)
		Transaction-Ordering Dependence
		DoS (Denial of Service)
		Function Call Permissions
		call/delegatecall Security
		Returned Value Security
		tx.origin Usage
		Replay Attack
		Overriding Variables
		Third-party Protocol Interface Consistency
3	Business Security	Business Logics
		Business Implementations
		Manipulable Token Price
		Centralized Asset Control
		Asset Tradability
		Arbitrage Attack

Beosin classified the security issues of smart contracts into three categories: Coding Conventions, General Vulnerability, Business Security. Their specific definitions are as follows:

- **Coding Conventions**

Audit whether smart contracts follow recommended language security coding practices. For example, smart contracts developed in Solidity language should fix the compiler version and do not use deprecated keywords.

- **General Vulnerability**

General Vulnerability include some common vulnerabilities that may appear in smart contract projects. These vulnerabilities are mainly related to the characteristics of the smart contract itself, such as integer overflow/underflow and denial of service attacks.

- **Business Security**

Business security is mainly related to some issues related to the business realized by each project, and has a relatively strong pertinence. For example, whether the lock-up plan in the code match the white paper, or the flash loan attack caused by the incorrect setting of the price acquisition oracle.

*Note that the project may suffer stake losses due to the integrated third-party protocol. This is not something Beosin can control. Business security requires the participation of the project party. The project party and users need to stay vigilant at all times.

3.3 Disclaimer

The Audit Report issued by Beosin is related to the services agreed in the relevant service agreement. The Project Party or the Served Party (hereinafter referred to as the "Served Party") can only be used within the conditions and scope agreed in the service agreement. Other third parties shall not transmit, disclose, quote, rely on or tamper with the Audit Report issued for any purpose.

The Audit Report issued by Beosin is made solely for the code, and any description, expression or wording contained therein shall not be interpreted as affirmation or confirmation of the project, nor shall any warranty or guarantee be given as to the absolute flawlessness of the code analyzed, the code team, the business model or legal compliance.

The Audit Report issued by Beosin is only based on the code provided by the Served Party and the technology currently available to Beosin. However, due to the technical limitations of any organization, and in the event that the code provided by the Served Party is missing information, tampered with, deleted, hidden or subsequently altered, the audit report may still fail to fully enumerate all the risks.

The Audit Report issued by Beosin in no way provides investment advice on any project, nor should it be utilized as investment suggestions of any type. This report represents an extensive evaluation process designed to help our customers improve code quality while mitigating the high risks in Blockchain.

3.4 About BEOSIN

Affiliated to BEOSIN Technology Pte. Ltd., BEOSIN is the first institution in the world specializing in the construction of blockchain security ecosystem. The core team members are all professors, postdocs, PhDs, and Internet elites from world-renowned academic institutions.BEOSIN has more than 20 years of research in formal verification technology, trusted computing, mobile security and kernel security, with overseas experience in studying and collaborating in project research at well-known universities. Through the security audit and defense deployment of more than 2,000 smart contracts, over 50 public blockchains and wallets, and nearly 100 exchanges worldwide, BEOSIN has accumulated rich experience in security attack and defense of the blockchain field, and has developed several security products specifically for blockchain.

Official Website

https://www.beosin.com

Telegram

https://t.me/+dD8Bnqd133RmNWNl

Twitter

https://twitter.com/Beosin_com

Email

Contact@beosin.com