Smart Contract Security Audit Report

[2021]

Table Of Contents

1 Executive Summary

On 2021.04.13, the SlowMist security team received the Bunny Park team's security audit application for Bunny Park, developed the audit plan according to the agreement of both parties and the characteristics of the project, and finally issued the security audit report.

The SlowMist security team adopts the strategy of "white box lead, black, grey box assists" to conduct a complete security test on the project in the way closest to the real attack.

The test method information:

Test method	Description
Black box testing	Conduct security tests from an attacker's perspective externally.
Grey box testing	Conduct security testing on code modules through the scripting tool, observing the internal running status, mining weaknesses.
White box testing	Based on the open source code, non-open source code, to detect whether there are vulnerabilities in programs such as nodes, SDK, etc.

The vulnerability severity level information:

Level	Description
Critical	Critical severity vulnerabilities will have a significant impact on the security of the DeFi project, and it is strongly recommended to fix the critical vulnerabilities.
High	High severity vulnerabilities will affect the normal operation of the DeFi project. It is strongly recommended to fix high-risk vulnerabilities.
Medium	Medium severity vulnerability will affect the operation of the DeFi project. It is recommended to fix medium-risk vulnerabilities.
Low	Low severity vulnerabilities may affect the operation of the DeFi project in certain scenarios. It is suggested that the project party should evaluate and consider whether these vulnerabilities need to be fixed.
Weakness	There are safety risks theoretically, but it is extremely difficult to reproduce in engineering.

Level	Description
Suggestion	There are better practices for coding or architecture.

2 Audit Methodology

The security audit process of SlowMist security team for smart contract includes two steps:

Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using automated analysis tools.

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:

- Reentrancy Vulnerability
- Replay Vulnerability
- Reordering Vulnerability
- Short Address Vulnerability
- Denial of Service Vulnerability
- Transaction Ordering Dependence Vulnerability
- Race Conditions Vulnerability
- Authority Control Vulnerability
- Integer Overflow and Underflow Vulnerability
- TimeStamp Dependence Vulnerability
- Uninitialized Storage Pointers Vulnerability
- Arithmetic Accuracy Deviation Vulnerability
- tx.origin Authentication Vulnerability

- "False top-up" Vulnerability
- Variable Coverage Vulnerability
- Gas Optimization Audit
- Malicious Event Log Audit
- Redundant Fallback Function Audit
- Unsafe External Call Audit
- Explicit Visibility of Functions State Variables Aduit
- Design Logic Audit
- Scoping and Declarations Audit

3 Project Overview

3.1 Project Introduction

Deployed on Binance Smart Chain (BSC), BunnyPark is an novel and secure decentralized application ,full of opportunities and enjoyments. The developer friendliness and openness of BunnyPark enables it be compatible with mainstream and innovative DeFi products. It supports more than DEX, oracle machines, NFTs, liquidity proof of work, loan and insurance among other common features, but as well allows to quickly build and flexibly assemble distributed applications (Dapps) of any forms via universal developer protocol.

Audit Information：

Github：https://github.com/renvincentrui/bunnypark

commit: 2b3ba953bc1b476350927244b120b26850b11925;

fix committ: 6d98c5708931a27644e46d5e54ebfd6dc131bed0;

3.2 Vulnerability Information

The following is the status of the vulnerabilities found in this audit:

NO	Title	Category	Level	Status
N1	Excessive Authority issue	Authority Control Vulnerability	Medium	Fixed
N1	Whitelist quota bypass	Design Logic Audit	Critical	Fixed
N2	The series of the activated card slot has not been checked	Design Logic Audit	Medium	Ignored
N3	No restrictions on coin id	Design Logic Audit	Suggestion	Ignored
N4	Event information recording error	Malicious Event Log Audit	Low	Fixed
N5		Others	Suggestion	Fixed

4 Code Overview

4.1 Contracts Description

The main network address of the contract is as follows:

The code was not deployed to the mainnet.

4.2 Visibility Description

The SlowMist Security team analyzed the visibility of major contracts during the audit, the result as follows:

4.3 Vulnerability Summary

[N1] [Medium] Excessive Authority issue

Category: Authority Control Vulnerability

Content

Owner of the NFTCards contract can burn user's nft card, which leads to the Excessive Authority issue

```
    function burn(address owner, uint256 tokenId) public onlyOwner{
    //SlowMist// can burn andy nft card of user, which leads to the Excessive
Authority issue
        _burnNFT(owner, tokenId);
        delete cardSerial[tokenId];
    }
```

Solution

it's suggest to move the owner role to TimeLock contract or the governance

Status

Fixed; The project side has deleted the related logic according to their business conditions

[N1] [Critical] Whitelist quota bypass

Category: Design Logic Audit

Content

a. The buyWarrior function of the CardSales contract does not limit the number of tokenIds purchased, which results in users passing in an empty array. The length of the empty array is 0. Then even if the whitelist is not set, it can grow from scratch and get the quota.

b. The buyWarrior function of the CardSales contract checks that the array length of the tokenIds parameter must meet the size of the whitelist quota, but the whitelist quota is self-increasing, so that users can always purchase Warriors with the whitelist quota in the future. Bypass the 5 quota limit set by the whitelist

```
    function buyWarrior(uint256[] calldata tokenIds, uint256 amount) external lock {
        currSoldWarriorNum = currSoldWarriorNum.add(tokenIds.length);
        require(currSoldWarriorNum + currSoldWarriorActivationSetNum <=
presaleWarriorCount, 'CardSales: warrior sold out');
        //SlowMist// The length of the tokenIds array is not limited, so that an
array with a length of 0 can be passed in for self-increment
        require(whitelist[msg.sender] == tokenIds.length, 'CardSales: not in the
whitelist or quota not right');
        //0 means not set whitelist,6 means already buy once
        //SlowMist// The quota is increased automatically, and users can always
purchase NFT cards with the whitelisted quota, bypassing the whitelist restriction
        whitelist[msg.sender] = warriorQuota + 1;

        //sum price and check amount
        uint256 sum = priceWarrior.mul(tokenIds.length);
        require(sum <= amount, "CardSales: need more usdt amount");

        _safeTransfer(usdt, msg.sender, usdtRecipient, sum);
        for (uint256 i = 0; i < tokenIds.length; i++) {
            require(tokenIds[i] <= 5000 && warriorTokenIds[tokenIds[i]] == 0,
"CardSales: token id is not warrior or already sold");
            INFTCard(shareCardNftToken).mintSerial1Card(msg.sender, tokenIds[i]);
            warriorTokenIds[tokenIds[i]] = 1;
            //user=>serialType=>card info
            userShareCardTokens[msg.sender][1].push(CardInfo(tokenIds[i],
block.number));
        }
        shareCardUsers.add(msg.sender);
        emit BuyWarrior(msg.sender, tokenIds, sum, amount);
    }
```

Solution

Check the length of the passed tokenIds array

Status

Fixed; Problem fixed

[N2] [Medium] The series of the activated card slot has not been checked

Category: Design Logic Audit

Content

The setSlotPrivate function of the CardSales contract does not verify whether the series of the classic card to be set is consistent with the corresponding series of the activated card, which leads to the possibility of putting the family series of cards into the holiday series.

```
function setSlotPrivate(address user, uint8 serial, uint256 cardIndex, uint256
tokenId, uint256 slotIndex, uint256 commonCardTokenId) private lock {
        //SlowMist// The category and serial of comonCardTokenId are not checked. As
a result, the activation card slot of the classic card series can activate the family
series card.
        uint256 len = userActivationCardTokens[user][serial].length;
        require(cardIndex < len, "CardSales: cardIndex is not exist");

        require(userActivationCardTokens[user][serial][cardIndex].tokenId == tokenId,
"CardSales: tokenId is not right");
        require(userActivationCardTokens[user][serial][cardIndex].activated,
"CardSales: not activated");
        require(slotIndex < 4, "CardSales: slotIndex is not exist");
        //check commonCardTokenId's owner is msg.sender
        require(INFTCard(commonNftToken).ownerOf(tokenId) == user, "CardSales: user
not own this common card token id");
        //Transfer user token to this contract
        INFTCard(commonNftToken).transferFrom(user, address(this),
commonCardTokenId);
        userActivationCardTokens[user][serial][cardIndex].slot[slotIndex] =
commonCardTokenId;
        emit SetSlot(msg.sender, user, serial, cardIndex, tokenId, slotIndex,
commonCardTokenId);
    }
```

Solution

Check the serial value

Status

Ignored; After confirming with the project party, the business code here has been deleted, and the business logic here needs to be redesigned after subsequent business changes

[N3] [Suggestion] No restrictions on coin id

Category: Design Logic Audit

Content

The buyWarrior function of the CardSales contract does not limit the coin id. It is recommended to limit the coin id

```
function buyWarrior(uint256[] calldata tokenIds, uint256 amount) external lock {
        require(tokenIds.length > 0, 'CardSales: input tokenIds length is 0');
        currSoldWarriorNum = currSoldWarriorNum.add(tokenIds.length);
        require(currSoldWarriorNum + currSoldWarriorActivationSetNum <=
presaleWarriorCount, 'CardSales: warrior sold out');
        require(tokenIds.length <= warriorQuota && whitelist[msg.sender] ==
tokenIds.length, 'CardSales: not in the whitelist or quota not right');
        //0 means not set whitelist,6 means already buy once
        whitelist[msg.sender] = warriorQuota + 1;

        //sum price and check amount
        uint256 sum = priceWarrior.mul(tokenIds.length);
        require(sum <= amount, "CardSales: need more usdt amount");

        _safeTransfer(usdt, msg.sender, usdtRecipient, sum);
        for (uint256 i = 0; i < tokenIds.length; i++) {
            require(tokenIds[i] <= 5000 && warriorTokenIds[tokenIds[i]] == 0,
"CardSales: token id is not warrior or already sold");
            //SlowMist// Unlimited tokenId, when tokenId already exists, the function
call will fail
           INFTCard(shareCardNftToken).mintSerial1Card(msg.sender, tokenIds[i]);
            warriorTokenIds[tokenIds[i]] = 1;
            //user=>serialType=>card info
            userShareCardTokens[msg.sender][1].push(CardInfo(tokenIds[i],
block.number));
        }
        shareCardUsers.add(msg.sender);
        emit BuyWarrior(msg.sender, tokenIds, sum);
    }
```

Solution

Restrict tokenId

Status

Ignored; After confirming with the project party, adding the limit will consume more gas, and the passed tokenid will call other functions to first determine whether it has been cast

[N4] [Low] Event information recording error

Category: Malicious Event Log Audit

Content

The buyWarrior / buyWarriorActivationSet / buyAstronaut / buyActivation function of the CardSales contract uses the amount passed in the parameter as the purchase amount in the event statement, but this amount is not the actual purchase amount. The user can pass in a large amount, which causes the event information to be recorded incorrectly

e.g. buyWarrior function

```
function buyWarrior(uint256[] calldata tokenIds, uint256 amount) external lock {
        require(tokenIds.length > 0, 'CardSales: input tokenIds length is 0');
        currSoldWarriorNum = currSoldWarriorNum.add(tokenIds.length);
        require(currSoldWarriorNum + currSoldWarriorActivationSetNum <=
presaleWarriorCount, 'CardSales: warrior sold out');
        require(tokenIds.length <= warriorQuota && whitelist[msg.sender] ==
tokenIds.length, 'CardSales: not in the whitelist or quota not right');
        //0 means not set whitelist,6 means already buy once
        whitelist[msg.sender] = warriorQuota + 1;

        //sum price and check amount
        uint256 sum = priceWarrior.mul(tokenIds.length);
        require(sum <= amount, "CardSales: need more usdt amount");

        _safeTransfer(usdt, msg.sender, usdtRecipient, sum);
        for (uint256 i = 0; i < tokenIds.length; i++) {
            require(tokenIds[i] <= 5000 && warriorTokenIds[tokenIds[i]] == 0,
"CardSales: token id is not warrior or already sold");
            INFTCard(shareCardNftToken).mintSerial1Card(msg.sender, tokenIds[i]);
            warriorTokenIds[tokenIds[i]] = 1;
            //user=>serialType=>card info
            userShareCardTokens[msg.sender][1].push(CardInfo(tokenIds[i],
block.number));
        }
        shareCardUsers.add(msg.sender);
        //SlowMist// The amount is passed in by the user, which will cause event
recording errors
        emit BuyWarrior(msg.sender, tokenIds, sum,amount);
    }
```

Solution

Use ”sum“ to record the purchase amount instead of ”amount“

Status

Fixed; The amount field has been removed from the latest code

[N5] [Suggestion]

Category: Others

Content

The stop and start functions of BPToken use the payable modifier, but there is no business logic that needs to be transferred, which is a redundant decorator. Suggest to delete

```
//SlowMist//payable modifier redundancy
 function stop() public payable onlyOwner {
        stopped = true;
    }
function start() public payable onlyOwner {
        stopped = false;
    }
```

Solution

Delete the corresponding redundant code

Status

Fixed

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002104270001	SlowMist Security Team	2021.04.13 - 2021.04.27	Passed

Summary conclusion: The SlowMist security team uses manual and internal tools to analyze the code. Six problems were discovered during the audit. It contains 1 serious vulnerability, 2 medium-risk vulnerabilities, 1 low-risk vulnerability and 2 enhancement suggestions. After communication and feedback with the project party, all the problems found so far have been fixed. Comprehensive assessment without risk

6 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.

SLOWMIST

Official Website

www.slowmist.com

✉

E-mail

team@slowmist.com

Twitter

@SlowMist_Team

Github

https://github.com/slowmist