SMART CONTRACT AUDIT REPORT

for

NFT-GODS

Prepared By: Yiqun Chen

PeckShield
August 15, 2021

Document Properties

Client	NFT-GODS
Title	Smart Contract Audit Report
Target	NFT-GODS
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 15, 2021	Xiaotao Wu	Final Release
1.0-rc	August 12, 2021	Xiaotao Wu	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 | Introduction

Given the opportunity to review the design document and related source code of the NFT-GODS smart contracts, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About NFT-GODS

NFT-GODS, based on global gods cultures as themes, is a decentralization TCG (Trading card game) application of BSC. The first released NFT card series take Chinese classic fairy tale The Investiture of Gods as theme, combining with playing methods, including blind-box card drawing, NFT card mining, card recomposition, strategic athletics, liquidity provider mining, system of master and apprentice as well as hitting lists of public communities.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of NFT-GODS

Item	Description
Name	NFT-GODS
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 15, 2021

In the following, we show the MD5 hash value of the related compressed file with the contracts for audit:

- MD5 (nft-gods-20210805.zip) = c06170b191e1e33153347c6e52a18ce1

And this is the MD5 hash value of the related compressed file with the contracts after all fixes for the issues found in the audit have been checked in:

- MD5 (nft-gods-20210815.zip) = d458159246fae511aca7449b1da5093d

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [8], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `NFT-GODS` smart contracts. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■ ■
Low	1	■
Informational	1	■
Total	4	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Fixed
PVE-002	Low	Improved Sanity Checks in ERC721::tokensOf()	Coding Practices	Fixed
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Fixed
PVE-004	Informational	Redundant Code Removal	Coding Practices	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: MiningPool
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the transfer() routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., ZRX, as our example. We show the related code snippet below. On its entry of transfer(), there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns false. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
```

```
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.1: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `getReward()` routine in the `LPPool` contract. If the `USDT` token is supported as `manager.members("token")`, the unsafe version of `IERC20(manager.members("token")).transfer(msg.sender, reward)` (line 249) may revert as there is no return value in the `USDT` token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
245     function getReward() public updateReward(msg.sender) checkHalve checkStart {
246         uint256 reward = earned(msg.sender);
247         if (reward <= 0) {return;}
248         rewards[msg.sender] = 0;
249         IERC20(manager.members("token")).transfer(msg.sender, reward);
250         emit RewardPaid(msg.sender, reward);
251     }
```

Listing 3.2: `LPPool::getReward()`

Another similar violation can be found in the `adminConfig()` routine of the `LPPool` contract.

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()`.

Status This issue has been fixed.

3.2 Improved Sanity Checks in ERC721::tokensOf()

- ID: PVE-002
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `ERC721`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

In the `ERC721` contract, the `tokensOf()` function returns the corresponding token IDs of a specified user when given the `startIndex` and `endIndex`. While reviewing the implementation of this routine, we notice that it can benefit from additional sanity checks.

To elaborate, we show below the full implementation of the `tokensOf()` function. Specifically, if `endIndex > tokens.length`, the execution of `result[i] = tokens[i]` will revert when `endIndex = tokens.length -1`.

```
    // [startIndex, endIndex)
    function tokensOf(address owner, uint256 startIndex, uint256 endIndex)
        external view returns(uint256[] memory) {

        require(owner != address(0), "owner is zero address");

        uint256[] storage tokens = ownerTokens[owner];
        if (endIndex == 0) {
            return tokens;
        }

        require(startIndex < endIndex, "invalid index");

        uint256[] memory result = new uint256[](endIndex - startIndex);
        for (uint256 i = startIndex; i != endIndex; ++i) {
            result[i] = tokens[i];
        }

        return result;
    }
```

Listing 3.3: `ERC721::tokensOf()`

Recommendation Validate the `endIndex` to ensure `endIndex <= tokens.length`.

Status This issue has been fixed.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: MiningPool
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the MiningPool contract, there are some special accounts, i.e., permitted users. We examine closely the LPPool contract and identify one trust issue on these privileged accounts. To elaborate, we show below the related code snippet. We note that the adminConfig() function allows for these permitted users to transfer a specified _value of manager.members("token") from the LPPool contract to a specified _account. In particular, the manager.members("token") can be configured to be equal to lpToken.

```
    function adminConfig(address _account, uint256 _value, bool _type) public
        CheckPermit("Config") {
        if (_type) {
            IERC20(manager.members("token")).transfer(_account, _value);
        } else {
            IERC20(manager.members("token")).transfer(_account, _value);
        }
    }
```

Listing 3.4: MiningPool::adminConfig()

We understand the need of the privileged function for contract operation, but at the same time the extra power to the permitted users may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among contract users.

Recommendation Make the list of extra privileges granted to permitted users explicit to NFT-GODS users.

Status This issue has been fixed. The NFT-GODS team removed the adminConfig() function from the MiningPool contract.

3.4 Redundant Code Removal

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: MiningPool
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

As mentioned in Section 3.3, the adminConfig() function in LPPool contract allows the permitted users to transfer a specified _value of manager.members("token") from the LPPool contract to a specified _account.

While reviewing the implementation of this routine, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed. Specifically, there is a branch judgment depends on the value of _type (line 273). But the same code is executed in all branches regardless the value of _type is true or false (lines 274 and 276). Therefore, we suggest to remove this redundant branch judgment.

```
272     function adminConfig(address _account, uint256 _value, bool _type) public
            CheckPermit("Config") {
273         if (_type) {
274             IERC20(manager.members("token")).transfer(_account, _value);
275         } else {
276             IERC20(manager.members("token")).transfer(_account, _value);
277         }
278     }
```

Listing 3.5: MiningPool::adminConfig()

Recommendation Consider the removal of the redundant code with a simplified, consistent implementation.

Status The issue has been fixed.

4 Conclusion

In this audit, we have analyzed the NFT-GODS design and implementation. As a decentralization TCG application of BSC, NFT-GODS will bring out immersing NFT gaming experiences with rich content and high playability. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that Solidity-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[9] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[10] PeckShield. PeckShield Inc. https://www.peckshield.com.