SMART CONTRACT AUDIT REPORT

for

GordoNFT

Prepared By: Xiaomi Huang

PeckShield

January 20, 2023

Document Properties

Client	GordoNFT
Title	Smart Contract Audit Report
Target	GordoNFT
Version	1.0-rc
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc	January 20, 2023	Luck Hu	Release Candidate

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the GordoNFT protocol, 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 GordoNFT

GordoNFT is a decentralized NFT lottery that offers an entirely new open and decentralized approach to the creation of jackpots and lottery tickets. Leveraging the power of blockchain technology, all transactions on the network are securely recorded on the public ledger and are always available for the players to review. The basic information of the GordoNFT protocol is as follows:

Table 1.1: Basic Information of The GordoNFT Protocol

Item	Description
Issuer	GordoNFT
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 20, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/GordoNFT/GordoNFT.git (4c36b8f)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/GordoNFT/GordoNFT.git (c23bbba)

1.2 About PeckShield

PeckShield Inc. [7] 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 [6]:

- 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 classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

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

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

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) [5], 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.

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 `GordoNFT` implementation. 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	2	■ ■
Medium	1	■
Low	1	■
Informational	0	
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 that 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 high-severity vulnerabilities, 1 medium-severity vulnerability, and 1 low-severity vulnerability.

Table 2.1: Key GordoNFT Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Incorrect Winner Setting Logic in setWinners()	Business Logic	Fixed
PVE-002	High	Revised Logic in sendRewards()	Business Logic	Fixed
PVE-003	Low	Improved Winners Selection in selectN()	Business Logic	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Incorrect Winner Setting Logic in setWinners()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `GordoVault`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the `GordoNFT` protocol, the `GordoVault` contract is designed to collect fee and also holds the royalties from the buy/sell of `GordoNFT` via the marketplace. The contract gets the selected winners from the `Lottery` contract and evenly distributes the royalties to the winners. While reviewing the logic to set the selected winners to the contract, we notice only 1 of the 4 selected winners can be set to the contract.

To elaborate, we show below the code snippet of the `setWinners()` routine. As the name indicates, it is used by the `Lottery` contract to set the selected winners to the `GordoVault` contract. It uses a `for`-loop to push the input winners one by one to the state variable `winners` (46). However, it comes to our attention that the index of the `_winners[]` is hardcoded to 1 by mistake. As a result, only the second winner is set to this contract, and this winner can be rewarded four times. By design, the index of the `_winners[]` shall be `i`, i.e., `_winners[i]`.

```
41  function setWinners(uint256[] memory _winners) public onlyLottery {
42        require(_winners.length > 0, "invalid winners");
43        // set new winners
44        delete winners;
45        for (uint256 i = 0; i < _winners.length; i++) {
46            winners.push(_winners[1]);
47        }
48    }
```

Listing 3.1: `GordoVault::setWinners()`

Recommendation Revise the index of the `_winners[]` from 1 to `i`.

Status The issue has been fixed by this commit: `9db0aca`.

3.2 Revised Logic in sendRewards()

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: High
- Target: `GordoVault`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the `GordoNFT` protocol, the `GordoVault` contract receives the royalties in `WETH` from the buy/sell of `GordoNFT` via the marketplace, converts `WETH` to `WMATIC`, and rewards `WMATIC` to the selected winners. While reviewing the logic to convert `WETH` to `WMATIC`, we notice it uses the wrong interface to get the output amount.

To elaborate, we show below the code snippet of the `sendRewards()` routine. It transfers all the available `WETH` in the contract to the `DEX` (line 62) and triggers the swap with the desired `WMATIC` amount (line 67). The desired `WMATIC` amount is calculated by calling the `ISwapper(swapper).getAmountsIn()` routine (line 58). However, our analysis shows that the `ISwapper(swapper).getAmountsIn()` routine is designed to return the required input token amount to receive the desired output token amount. In the `sendRewards()` routine, it has `WETH` as the input token and wants to receive `WMATIC` as the output token. Therefore, it needs to call the `ISwapper(swapper).getAmountsOut()` routine to get the maximum `WMATIC` amount per the available `WETH` amount. If the wrong routine is used, it will receive far less `WMATIC` than expected with all `WETH` spent.

```
50  function sendRewards() public onlyLottery {
51    // convert ETH to matic
52    {
53        uint256 _amount = IERC20(WETH).balanceOf(address(this));
54        if (_amount > 0) {
55            address[] memory path = new address[](2);
56            path[0] = WETH;
57            path[1] = WMATIC;
58            uint256[] memory amounts = ISwapper(swapper).getAmountsIn(
59                _amount,
60                path
61            );
62            TransferHelper.safeTransfer(
63                WETH,
64                ISwapper(swapper).GetReceiverAddress(path),
65                _amount
```

```
            );
            ISwapper(swapper)._swap(amounts, path, address(this));
            _amount = IERC20(WMATIC).balanceOf(address(this));
            IWETH(WMATIC).withdraw(_amount);
        }
    }
    ...
}
```

Listing 3.2: `GordoVault::sendRewards()`

Recommendation Revisit the `sendRewards()` routine to use the `ISwapper(swapper).getAmountsOut()` routine to get the maximum output token amount.

Status The issue has been fixed by this commit: `9db0aca`.

3.3 Improved Winners Selection in selectN()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Lottery`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the `GordoNFT` protocol, the `Lottery` contract is responsible for the selection of the winners who can earn the upcoming royalties for the next 10 days. The winners are selected per the requested randomnesses from `Chainlink VRF`. While examining the logic to select the winners, we notice that it may can not select the desired amount (4) of winners in a whole selection round, and the status of the remaining active tokens that are not selected as winners may become wrong.

To elaborate, we show below the code snippets of the `onEachRound()/selectN()` routines. As the name indicates, the `onEachRound()` routine is called every time after the randomnesses are filled by `Chainlink VRF`. Every time the `onEachRound()` routine is triggered, it calls the `selectN()` routine to randomly deactivate 40 `Gordo NFTs` from the total 10000. This is repeated until 4 `Gordo NFTs` will remain, and the remaining 4 `Gordo NFTs` will be selected as the winners of the whole round.

In the `selectN()` routine, it swaps the `Gordo NFTs` at the positions `k` and `curLength-1` in the `tokenIds[]` (lines 110 − 111). And the selected `Gordo NFTs` that are to be deactivated are all moved to the end of the `tokenIds[]`. At last, all the positions in the `tokenIds[]` are filled with the deactivated `Gordo NFTs` except for the first 4 positions. And the `Gordo NFTs` at the first 4 positions in the `tokenIds[]` are selected as the winners (lines 137 − 138).

However, it comes to our attention that the first 4 positions in the `tokenIds[]` are not filled with the 4 remaining `Gordo NFTs`. Because the `k` value in the `selectN()` routine is generated per the randomness and is less than the `curLength`, hence `k` can be any number that is smaller than the remaining amount of active `Gordo NFTs`. But this doesn't mean the `k` value could hit all the numbers in the remaining amount of active `Gordo NFTs`. As a result, if any of the remaining 4 active `Gordo NFTs` is not properly set to the first 4 positions of the `tokenIds[]`, we cannot get 4 winners as expected. Moreover, the status of the missing `Gordo NFTs` can not be properly restored at the end of the whole round (139).

```
95   function selectN(uint256[] memory randomNumbers, uint256 count)
96   private
97   returns (uint256[] memory _tokenIds)
98  {
99   require(
100      curLength >= count && randomNumbers.length >= count,
101      "overflow"
102  );
103  _tokenIds = new uint256[](count);
104  for (uint256 i = 0; i < count; i++) {
105      // selectOne(randomNumbers[i]);
106      uint256 k = randomNumbers[i] % curLength;
107      // swap i, curLength - 1
108      uint256 last = tokenIds[curLength - 1];
109      uint256 selected = tokenIds[k] == 0 ? k + 1 : tokenIds[k];
110      tokenIds[curLength - 1] = selected;
111      tokenIds[k] = last == 0 ? curLength : last;
112      tokenStatus[selected] = !tokenStatus[selected];
113      _tokenIds[i] = selected;
114      if (curLength > 1) curLength = curLength - 1;
115  }
116 }
117
118 function onEachRound(
119  uint256 _lotteryId,
120  uint256 _requestId,
121  uint256[] memory _randomNumbers
122 ) external onlyRandomGenerator {
123  // generate 40 random numbers from chainlink
124  if (_requestId == requestId_) {
125      // curLength <= 40 + 4 =>
126      uint256[] memory selectedUers;
127      if (curLength <= EachRoundInactiveNumber + WinnersNumber) {
128          // last round in each lottery
129          require(curLength > WinnersNumber, "invalid curLength");
130          // select curLength - 4
131          selectedUers = selectN(
132              _randomNumbers,
133              curLength - WinnersNumber
134          );
135          // setWinner,
```

```
            for (uint256 i = 0; i < WinnersNumber; i++) {
                uint256 _tokenId = tokenIds[i];
                winners[i] = _tokenId;
                tokenStatus[_tokenId] = !tokenStatus[_tokenId];
            }
            // resetRound
            if (vault_ != address(0)) IVault(vault_).setWinners(winners);
            if (nft_ != address(0))
                IGordoNFT(nft_).updateTokenMetaData(winners);
            round = round + 1;
            curLength = N;
        } else {
            selectedUers = selectN(_randomNumbers, EachRoundInactiveNumber);
        }
        // change status of NFT meta data
        if (nft_ != address(0))
            IGordoNFT(nft_).updateTokenMetaData(selectedUers);
        if (vault_ != address(0)) IVault(vault_).sendRewards();
    }
    lotteryId += 1;
}
```

Listing 3.3: Lottery.sol

Recommendation Revisit the above mentioned routines to improve the selection of the winners, and make sure all the remaining active tokens can be selected as winners.

Status The issue has been fixed by this commit: `9db0aca`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `GordoNFT` protocol, there is a privilege account, i.e., `owner` that plays a critical role in governing and regulating the system-wide operations (e.g., mint `GordoNFT`). In the following, we use the `GordoNFT` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

Specifically, the privileged functions in the `GordoNFT` contract allow for the `owner` to mint new `GordoNFT`, set the royalty parameters, and set the lottery address.

```
function mint(address _to) public onlyOwner {
  require(_tokenIdTracker.current() < maxSupply, "over max supply");
  _tokenIdTracker.increment();
  uint256 newItemId = _tokenIdTracker.current();
  // super._mint(_to, newItemId);
  _safeMint(_to, newItemId);
}

function batchMint(address _to, uint256 mintCount) public onlyOwner {
  require(
      mintCount > 0 && _tokenIdTracker.current() + mintCount <= maxSupply,
      "amount is invalid"
  );
  for (uint256 x = 0; x < mintCount; x++) {
      _tokenIdTracker.increment();
      uint256 newItemId = _tokenIdTracker.current();
      // super._mint(_to, newItemId);
      _safeMint(_to, newItemId);
  }
}

function setRoyalitiesInfo(
    address newRoyaltiesPaymentAddress,
    uint256 royailiesPercent
) external onlyOwner {
    _royaltiesPaymentsAddress = payable(newRoyaltiesPaymentAddress);
    _royaltiesPercent = royailiesPercent;
}

function setLotteryAddress(address _lottery) external onlyOwner {
    lottery = _lottery;
}
```

Listing 3.4: Example Privileged Operations in the GordoNFT Contract

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the privileged accounts may also be a counter-party risk to the protocol users. It is worrisome if the privileged accounts are plain EOA accounts. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been mitigated as the team clarifies that they will use Gnosis multi-sig as the owner.

4 Conclusion

In this audit, we have analyzed the GordoNFT protocol design and implementation. GordoNFT is a decentralized NFT lottery that offers an entirely new open and decentralized approach to the creation of jackpots and lottery tickets. Leveraging the power of blockchain technology, all transactions on the network are securely recorded on the public ledger and are always available for the players to review. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

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

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

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