SMART CONTRACT AUDIT REPORT

for

Own The Doge

Prepared By: Xiaomi Huang

PeckShield

June 8, 2024

Document Properties

Client	Own The Doge
Title	Smart Contract Audit Report
Target	Own The Doge
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 8, 2024	Xuxian Jiang	Final Release

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 4
- 1.1 About Own The Doge . . . 4
- 1.2 About PeckShield . . . 5
- 1.3 Methodology . . . 5
- 1.4 Disclaimer . . . 7

2 Findings 9
- 2.1 Summary . . . 9
- 2.2 Key Findings . . . 10

3 ERC721 Compliance Checks 11

4 Detailed Results 13
- 4.1 Revisited mintPuppers() Logic in PX/BridgePX . . . 13

5 Conclusion 15

References 16

1 Introduction

Given the opportunity to review the source code of the `Own The Doge` smart contract, we outline in the report our systematic method to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistency between smart contract code and the documentation, and provide additional suggestions or recommendations for improvement. Our results show that the given version of the smart contract exhibits no `ERC721` compliance issues or security concerns. This document outlines our audit results.

1.1 About Own The Doge

`Own The Doge` is the cultural ministry of `Doge` recording the past and writing the future. The token of `DOG` was fractionalized from the `Doge NFT` and `Own The Doge` holds the stewardship so that they are community owned forever by the biggest `Doge` fans around the world. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of `Own The Doge`

Item	Description
Name	Own The Doge
Website	https://www.ownthedoge.com/
Type	ERC721 Smart Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	June 8, 2024

In the following, we show the Git repository and the commit hash value used in this audit:

- https://github.com/kvdenden/doge-pixels.git (8f8ab8e)

1.2 About PeckShield

PeckShield Inc. [5] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the 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 the OWASP Risk Rating Methodology [4]:

- 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 deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would

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
	Approve / TransferFrom Race Condition
ERC721 Compliance Checks	Compliance Checks (Section 3)
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
	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

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.
- ERC721 Compliance Checks: We also validate whether the implementation logic of the audited smart contract(s) follows the standard ERC721 specification and other best practices.
- 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) [3], 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 Logic	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 `Own The Doge` contract design and 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 `ERC721`-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

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

Moreover, we explicitly evaluate whether the given contracts follow the standard `ERC721` specification and other known best practices, and validate its compatibility with other similar `ERC721` tokens and current DeFi protocols. The detailed `ERC721` compliance checks are reported in Section 3. After that, we examine a few identified issues of varying severities that need to be brought up and paid more attention to. (The findings are categorized in the above table.) Additional information can be found in the next subsection, and the detailed discussions are in Section 4.

2.2 Key Findings

Overall, no ERC721 compliance issue was found and our detailed checklist can be found in Section 3. Note that the smart contract implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 informational recommendation.

Table 2.1: Key Audit Findings of Own The Doge

ID	Severity	Title	Category	Status
PVE-001	Informational	Revisited mintPuppers() Logic in PX/BridgePX	Coding Practices	

In the meantime, we also need to 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 our detailed compliance checks.

3 | ERC721 Compliance Checks

The ERC721 standard for non-fungible tokens, also known as deeds. Inspired by the ERC20 token standard, the ERC721 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC721-compliant. Naturally, we examine the list of necessary API functions defined by the ERC721 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Table 3.1: Basic View-Only Functions Defined in The ERC721 Specification

Item	Description	Status
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
ownerOf()	Is declared as a public view function	✓
	Returns the address of the owner of the NFT	✓
getApproved()	Is declared as a public view function	✓
	Reverts while '_tokenId' does not exist	✓
	Returns the approved address for this NFT	✓
isApprovedForAll()	Is declared as a public view function	✓
	Returns a boolean value which check '_operator' is an approved operator	✓

Our analysis shows that there is no ERC721 inconsistency or incompatibility issue found in the audited Own The Doge token contracts. In the surrounding two tables, we outline the respective list of basic view-only functions (Table 3.1) and key state-changing functions (Table 3.2) according to the widely-adopted ERC721 specification.

Table 3.2: Key `State-Changing` Functions Defined in The `ERC721` Specification

Item	Description	Status
safeTransferFrom()	Is declared as a public function	✓
	Reverts while ‘to‘ refers to a smart contract and not implement IERC721Receiver-onERC721Received	✓
	Reverts unless ‘msg.sender‘ is the current owner, an authorized operator, or the approved address for this NFT	✓
	Reverts while ‘_tokenId‘ is not a valid NFT	✓
	Reverts while ‘_from‘ is not the current owner	✓
	Reverts while transferring to zero address	✓
	Emits Transfer() event when tokens are transferred successfully	✓
transferFrom()	Is declared as a public function	✓
	Reverts unless ‘msg.sender‘ is the current owner, an authorized operator, or the approved address for this NFT	✓
	Reverts while ‘_tokenId‘ is not a valid NFT	✓
	Reverts while ‘_from‘ is not the current owner	✓
	Reverts while transferring to zero address	✓
	Emits Transfer() event when tokens are transferred successfully	✓
approve()	Is declared as a public function	✓
	Reverts unless ‘msg.sender‘ is the current owner, an authorized operator, or the approved address for this NFT	✓
	Emits Approval() event when tokens are approved successfully	✓
setApprovalForAll()	Is declared as a public function	✓
	Reverts while not approving to caller	✓
	Emits ApprovalForAll() event when tokens are approved successfully	✓
Transfer() event	Is emitted when tokens are transferred	✓
Approval() event	Is emitted on any successful call to approve()	✓
ApprovalForAll() event	Is emitted on any successful call to setApprovalForAll()	✓

4 | Detailed Results

4.1 Revisited mintPuppers() Logic in PX/BridgePX

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: PX/BridgePX
- Category: Coding Practices [2]
- CWE subcategory: CWE-563 [1]

Description

The PX token contract takes the popular ERC721Upgradeable contact as the base with additional customization on the token mint/burn logic. Moreover, it supports batch processing in minting multiple NFTs together. While examining the related batch processing logic, we notice current implementation may be revisited.

In the following, we show the implementation of the related mintPuppers() routine. This routine has a rather straightforward logic in minting multiple NFTs while enforcing the minter to send in required amount of DOG tokens. Note this routine in essence exchanges the index/pupper mapping between the randomly-chosen index (line 189) and LAST_INDEX (line 195). However, our analysis shows that the pupper of LAST_INDEX, though empty, is better re-mapped to the index. In other words, before moving the used pupper to unavailable pool, we need to properly move pupper from LAST_INDEX to just used pupper as follows: `uint256 last_index_pupper = indexToPupper[LAST_INDEX]; indexToPupper[index] = last_index_pupper; pupperToIndex[last_index_pupper] = index;`.

```
185     function mintPuppers(uint256 qty) public {
186         require(qty > 0, "Non positive quantity");
187         require(qty <= puppersRemaining, "No puppers remaining");
188         for (uint256 i = 0; i < qty; ++i) {
189             uint256 index = INDEX_OFFSET + randYishInRange(puppersRemaining);
190             // if indexToPupper[index] == null, initialize it with `index` pupper
191             // this on-the-go initialization is optimization so gas fees for `
                    indexToPupper` array initialization is delegated to the minter
192             if (indexToPupper[index] == MAGIC_NULL) {
```

```
                indexToPupper[index] = index;
            }
            uint256 LAST_INDEX = INDEX_OFFSET + puppersRemaining - 1;
            if (indexToPupper[LAST_INDEX] == MAGIC_NULL) {
                indexToPupper[LAST_INDEX] = LAST_INDEX;
            }
            // select pupper @ `index`
            uint256 pupper = indexToPupper[index];
            // move pupper from `LAST_INDEX` to just used pupper
            indexToPupper[index] = indexToPupper[LAST_INDEX];
            // move used pupper to `unavailable` pool
            indexToPupper[LAST_INDEX] = pupper;
            pupperToIndex[pupper] = LAST_INDEX;
            _mint(_msgSender(), pupper);
        }
        // transfer collateral to contract's address
        DOG20.transferFrom(_msgSender(), address(this), qty * DOG_TO_PIXEL_SATOSHIS);
    }
```

Listing 4.1: `PX::mintPuppers()`

Recommendation Revise the above routine to better move the pupper from `LAST_INDEX` to just used pupper. Note another contract `BridgePX` shares the same issue.

Status

5 Conclusion

In this security audit, we have examined the `Own The Doge` contract design and implementation. During our audit, we first checked all respects related to the compatibility of the ERC721 specification and other known ERC721 pitfalls/vulnerabilities and found no issue in these areas. We then proceeded to examine other areas such as coding practices and business logics. Overall, we found one low-severity issue and three informational recommendations which are promptly addressed by the team. Meanwhile, as disclaimed in Section 1.4, we appreciate any constructive feedbacks or suggestions about our findings, procedures, audit scope, etc.

References

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

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

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

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

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