SMART CONTRACT AUDIT REPORT

for

CRYPTOZOON

Prepared By: Yiqun Chen

PeckShield

August 25, 2021

Document Properties

Client	CryptoZoon
Title	Smart Contract Audit Report
Target	CryptoZoon
Version	1.0
Author	Xuxian Jiang
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 25, 2021	Xuxian Jiang	Final Release
1.0-rc	August 24, 2021	Xuxian Jiang	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 CryptoZoon 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 CryptoZoon

The CryptoZoon protocol is inspired by Pokemon Story with the main mission to build a comprehensive platform of digital monsters. The goal is to enable millions of individuals to participate in the NFT and blockchain-based gaming world in a simple, creative, and enjoyable way. In particular, it combines the greatest aspects of gaming and digital collectibles, transforming it into the digital creatures universe. With CryptoZoon, players can use their ZOAN tokens to fight monsters, collect, grow, and join training (battle each other). The basic information of audited contracts is as follows:

Table 1.1: Basic Information of CryptoZoon

Item	Description
Name	CryptoZoon
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 25, 2021

In the following, we show the link to the smart contract source code for audit. Note that CryptoZoon assumes a trusted (external) router that provides the required evolvers, battlefields, as well as various protocol-wide arguments (e.g., priceEgg and feeEvolve) and the router itself is not part of this audit.

- https://bscscan.com/address/0x51d7e5022040434328849779762 63aca4ef23f09

1.2 About PeckShield

PeckShield Inc. [9] 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 [8]:

- 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.

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) [7], 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 `CryptoZoon` 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	0	
Low	2	■■
Informational	1	■
Total	3	

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 low-severity vulnerabilities and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Fixed
PVE-002	Informational	Redundant Data/Code Removal	Coding Practices	Fixed
PVE-003	Low	Improved Validation Of Function Arguments	Security Features	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: Low
- Likelihood: Low
- Impact: High
- Target: `CryptoZoan`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

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 `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `recoverZoon()` routine in the `CryptoZoan` contract. If the `USDT` token is supported as `zoonToken`, the unsafe version of `zoonToken.transfer(msg.sender, amount)` (line 313) 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)!

```
311    function recoverZoon(uint256 amount) public {
312        require(msg.sender == dev);
313        zoonToken.transfer(msg.sender, amount); // dont expect we'll hold tokens here
           but might as well
314    }
```

Listing 3.2: `CryptoZoan::recoverZoon()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status This issue has been resolved as the new version removes this `recoverZoon()` function.

3.2 Redundant State/Code Removal

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `CryptoZoan`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

The `CryptoZoan` protocol makes good use of a number of reference contracts, such as `ERC721Upgradeable`, `ERC20`, `SafeMath`, and `SafeERC20`, to facilitate its code implementation and organization. For example, the `CryptoZoan` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the data structures defined in `CryptoZoan`, the `ItemSale` structure is not used throughput the contract. With that, it is suggested to simply remove it.

```
    struct Zoan {
        uint256 collections;
        uint256 generation;
        Tribe tribe;
        uint256 exp;
        uint256 dna;
        uint256 bornTime;
    }

    struct ItemSale {
        uint256 tokenId;
        address owner;
        uint256 price;
    }
```

Listing 3.3: `CryptoZoan::ItemSale`

In addition, a public function `layEgg()` routine can be improved as it contains redundant code. In particular, the `if`-condition (line 209) as well as the `then`-branch (line 209) can be removed. The reason is that the `else`-branch logic (lines $211-213$) can accommodate the entire the `then`-branch.

```
202    function layEgg(
203        address receiver,
204        Tribe[] memory tribes,
205        uint256 _collections
206    ) external onlyEvolver {
207        uint256 amount = tribes.length;
208        require(amount > 0, "require: >0");
209        if (amount == 1) _layEgg(receiver, tribes[0], _collections);
```

```
        else
            for (uint256 index = 0; index < amount; index++) {
                _layEgg(receiver, tribes[index], _collections);
            }
    }
```

Listing 3.4: `CryptoZoan::layEgg()`

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status The issue has been fixed by taking the above suggestions and removing unused/redundant statements.

3.3 Improved Validation Of Function Arguments

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `CryptoZoan`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

The `CryptoZoon` protocol creates `NFT` tokens and each is unique with its own `tokenId`. Moreover, the `tokenId` is widely used as the key to index a number of other states, e.g., `zoans`, `isEvolved`, and `latestBlockTransfer`. With that, there is a constant need in validating whether the given `tokenId` is a valid one or not.

In the following, we show two example routines `exp()` and `evolve()`. These two routines are used by authorized callers to update internal states on `zoan`. It comes to our attention that both routines do not properly validate the input arguments of `_tokenId`.

```
    function exp(uint256 _tokenId, uint256 _exp) public onlyBattlefield {
        require(_exp > 0, "no exp");

        Zoan storage zoan = zoans[_tokenId];
        zoan.exp = zoan.exp.add(_exp);
        emit Exp(_tokenId, _exp);
    }

    function evolve(
        uint256 _tokenId,
        uint256 _dna,
        uint256 _generation
    ) public onlyEvolver {
```

```
        require(
            latestBlockTransfer[_tokenId] < block.number,
            "evolve after transfer"
        );

        Zoan storage zoan = zoans[_tokenId];
        require(!isEvolved[_tokenId], "require: not evolved");

        zoan.bornTime = block.timestamp;
        zoan.dna = _dna;
        zoan.generation = _generation;

        isEvolved[_tokenId] = true;

        emit Evolve(_tokenId, _dna);
    }
```

Listing 3.5: `CryptoZoan::exp()/evolve()`

In other words, though there is a constant need to perform sanity checks on the given `tokenId` , the current implementation simply relies on the trust on the authenticated caller to ensure the `tokenId` index stays within the array range `[0, latestTokenId]`. However, considering the importance of validating the given `tokenId` and its numerous occasions, a better alternative is to make explicit the sanity checks by introducing a new modifier, say `validTokenId()`. This new modifier essentially ensures the given `tokenId` indeed points to a valid `NFT` token, and additionally give semantically meaningful information when it is not!

We highlight that there are a number of functions that can benefit from the new `tokenId`-validating modifier, including `exp()`, `evolve()`, `changeTribe()` and `upgradeGeneration()`.

Recommendation Apply necessary sanity checks to ensure the given `tokenId` is legitimate. Accordingly, a new modifier `validTokenId()` can be developed and appended to each function in the above list.

```
    modifier validTokenId(uint256 _tokenId) {
        require(tokenId > 0 && _tokenId <= latestTokenId, "Invalid tokenId?");
        _;
    }

    function exp(uint256 _tokenId, uint256 _exp) public validTokenId(_tokenId)
        onlyBattlefield {
        require(_exp > 0, "no exp");

        Zoan storage zoan = zoans[_tokenId];
        zoan.exp = zoan.exp.add(_exp);
        emit Exp(_tokenId, _exp);
    }

    function evolve(
        uint256 _tokenId,
```

```
        uint256 _dna,
        uint256 _generation
    ) public validTokenId(_tokenId) onlyEvolver {
        require(
            latestBlockTransfer[_tokenId] < block.number,
            "evolve after transfer"
        );

        Zoan storage zoan = zoans[_tokenId];
        require(!isEvolved[_tokenId], "require: not evolved");

        zoan.bornTime = block.timestamp;
        zoan.dna = _dna;
        zoan.generation = _generation;

        isEvolved[_tokenId] = true;

        emit Evolve(_tokenId, _dna);
    }
```

Listing 3.6: Revised `CryptoZoan::exp()/evolve()`

Status This issue has been fixed by adding the new `validTokenId()` modifier to the above list of related functions.

4 Conclusion

In this audit, we have analyzed the CryptoZoon design and implementation. The CryptoZoon protocol allows for flexible creation and customization of ERC721-based NFT tokens. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

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

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

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

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

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