SMART CONTRACT AUDIT REPORT

for

MY DEFI PET

Prepared By: Yiqun Chen

PeckShield
August 27, 2021

Document Properties

Client	My DeFi Pet
Title	Smart Contract Audit Report
Target	My DeFi Pet
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 27, 2021	Xuxian Jiang	Final Release
1.0-rc1	July 27, 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 smart contract source code of the `My DeFi Pet` 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 have been updated to address security and performance. This document outlines our audit results.

1.1 About My DeFi Pet

`My DeFi Pet` is a virtual pet game that combines `DeFi`, collectibles and the player's own personality. It is operated on supported networks including `Binance Smart Chain (BSC)` and `KardiaChain`. This pet game revolves around a core loop of engaging gaming activities such as collecting, breeding, evolving, battling with, and trading/socializing for pets. The concept of `Season` is used to break down the game progress into smaller parts. This mechanism complements the human tendency for short term rewards.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of My DeFi Pet

Item	Description
Target	My DeFi Pet
Website	https://mydefipet.com/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 27, 2021

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

- https://github.com/mydefipet/my-defi-pet.git (5b60634)

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

- https://github.com/mydefipet/my-defi-pet.git (39fbafe)

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 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) [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. 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 `My DeFi Pet` game. 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	2	■ ■
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 1 medium-severity vulnerability, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Inconsistency Between Document And Implementation	Coding Practices	Fixed
PVE-002	Low	Consistency Between Pet-Transferring Logic	Coding Practices	Fixed
PVE-003	Informational	Improved Gas Efficiency in giveBirth()	Coding Practices	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	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 Inconsistency Between Document and Implementation

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Krc20DPET
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [1]

Description

There is a misleading comment embedded among lines of solidity code, which brings unnecessary hurdles to understand and/or maintain the software.

Specifically, if we examine the `DPETToken::constructor()` routine, the accompanying comment indicates that "`total supply fixed at 1 billion token`". However, the implemented logic (line 654) indicates that the maximum total supply, i.e., `totalTokens`, is actually `100 million`, not `1 billion`.

```
652    constructor() public KRC20Detailed("My DeFi Pet Token", "DPET", 18) {
653
654        totalTokens = 100000000 * 10 ** uint256(decimals());
655        _mint(owner(), totalTokens); // total supply fixed at 1 billion token
656    }
```

Listing 3.1: `DPETToken::constructor()`

Recommendation Ensure the consistency between documents (including embedded comments) and implementation.

Status The issue has been fixed by this commit: `39fbafe`.

3.2 Consistency Between Pet-Transferring Logic

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: NFTMarket
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [1]

Description

My DeFi Pet is a virtual pet game where each pet is internally represented as an NFT token. The NFT representation greatly facilitates a number of gaming activities such as collecting, breeding, evolving, battling with, and trading/socialising for pets. While examining the pet-transferring logic, we notice an inconsistency between transfer() and transferFrom(), the two main approaches for pet-transferring from the current owner to another.

```
    function transfer(
        address _to,
        uint256 _tokenId
    )
        external
        whenNotPaused
    {
        require(_to != address(0));
        require(_to != address(this));
        require(_to != address(saleAuction));
        require(_to != address(siringAuction));

        require(_owns(msg.sender, _tokenId));

        _transfer(msg.sender, _to, _tokenId);
    }
```

Listing 3.2: PetOwnership::transfer()

```
    function transferFrom(
        address _from,
        address _to,
        uint256 _tokenId
    )
        external
        whenNotPaused
    {
        require(_to != address(0));
        require(_to != address(this));
        require(_approvedFor(msg.sender, _tokenId));
        require(_owns(_from, _tokenId));
```

```
395         _transfer(_from, _to, _tokenId);
396     }
```

Listing 3.3: `PetOwnership::transferFrom()`

To elaborate, we show above these two routines. It comes to our attention that the first routine has additional requirements before the pet-transfer can be permitted. Specifically, it requires the recipient cannot be `saleAuction` (line 353) or `siringAuction` (line 354). For consistency, there is a need for the second routine to have the set of requirements.

Recommendation Be consistent between the above two functions `transfer()` and `transferFrom()` for pet-ownership transfer.

Status The issue has been fixed by this commit: `39fbafe`.

3.3 Improved Gas Efficiency in giveBirth()

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `PetBreeding`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [1]

Description

As mentioned in Section 3.2, the virtual pet game has a number of gaming activities. In particular, each virtual pet can be collected, bred, evolved, traded, and even battled with. In the process of examining its `birth`, we notice the current implementation can be improved for gas efficiency.

Specifically, we show below the related `giveBirth()` function. This function allows a pregnant pet to give birth. It comes to our attention that the current implementation may unnecessarily read the same storage multiple times, which can be optimized to read only once. In particular, the storage state `matron.siringWithId` (lines 669 and 680) contains the ID of the sire pet for the given matron that is pregnant. To avoid repeated storage access from the same location, we can simply replace the second access with the internal variable `sireId`, which is populated during the first read.

```
657     function giveBirth(uint256 _matronId)
658         external
659         whenNotPaused
660         returns(uint256)
661     {
662         Pet storage matron = pets[_matronId];
663 
664         // Check that the matron is a valid pet.
665         require(matron.birthTime != 0, "Invalid pet");
```

```
        require(_isReadyToGiveBirth(matron), "Not ready birth");

        uint256 sireId = matron.siringWithId;
        Pet storage sire = pets[sireId];

        uint16 parentGen = matron.generation;
        if (sire.generation > matron.generation) {
            parentGen = sire.generation;
        }

        uint256 childGenes = IGeneScience(geneScience).mixGenes(matron.genes, sire.genes
            , matron.cooldownEndBlock - 1);

        address owner = PetIndexToOwner[_matronId];
        uint256 petId = _createPet(_matronId, matron.siringWithId, parentGen + 1,
            childGenes, owner);

        delete matron.siringWithId;

        pregnantpets--;

        return petId;
    }
```

Listing 3.4: `PetBreeding::giveBirth()`

Recommendation Reduce unnecessary repeated storage reads from the same location.

Status The issue has been fixed by this commit: `39fbafe`.

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 [2]

Description

In the `My DeFi Pet` game, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and token adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

To elaborate, we show below the `emergencyWithdrawalDPETToken()` routine in the `Staking` contract. This routine allows the `owner` account to withdraw all staked `DPET` tokens from the staking contract. Also, notice that the two `DPET` token contracts have the special `owner` to possibly mint previously burned tokens back into circulation. Moreover, the current `owner` is able to mint unlimited pets at his/her discretion.

```
    // Below emergency functions will be never used in normal situations.
    // These function is only prepared for emergency case such as smart contract hacking Vulnerability or smart contract abolishment
    // Withdrawn fund by these function cannot belong to any operators or owners.
    // Withdrawn fund should be distributed to individual accounts having original ownership of withdrawn fund.

    function emergencyWithdrawalDPETToken(uint256 _amount) public onlyOwner {
        require(IKRC20(DPET_ADDRESS).transfer(msg.sender, _amount));
    }
```

Listing 3.5: `Staking::emergencyWithdrawalDPETToken()`

If the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to current game players. A plan needs to be in place to migrate it under community governance. 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. In the meantime, a timelock-based mechanism can also be considered as mitigation.

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 fixed by this commit: `39fbafe`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `My DeFi Pet` game. This pet game revolves around a core loop of engaging gaming activities such as collecting, breeding, evolving, battling with, and trading/socializing for pets. The current code base is clearly organized and those identified issues are promptly confirmed and resolved.

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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

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