SMART CONTRACT AUDIT REPORT

for

Raffle

Prepared By: Xiaomi Huang

PeckShield

May 10, 2023

Document Properties

Client	LooksRare
Title	Smart Contract Audit Report
Target	Raffle
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Patrick Lou, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 10, 2023	Xuxian Jiang	Final Release
1.0-rc	May 8, 2023	Xuxian Jiang	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 `Raffle` 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 Raffle

The `Raffle` protocol allows the creation of a new `raffle` with parameters such as cutoff time, minimum entries, maximum entries per participant, fees, and prizes. The prizes can be deposited into the `raffle`, and participants can enter the `raffle` by purchasing entries. Once the `raffle` is concluded, the winners are selected with the help of some randomness provided by `Chainlink VRF`, after which the winners can claim their prizes. The contract also provides functionalities for the raffle owner to claim the collected fees and for the participants to withdraw their entry fees in case the `raffle` is cancelled. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Raffle

Item	Description
Name	Raffle
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 10, 2023

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

- https://github.com/LooksRare/contracts-raffle.git (4fea5aa)

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

- https://github.com/LooksRare/contracts-raffle.git (adda28a)

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 classified into four categories accordingly, 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) [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.

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 `Raffle` protocol, 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 logic, 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 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 1 medium-severity vulnerability, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Raffle Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Inconsistent Enforcement of whenNot-Paused in cancel()	Coding Practices	Resolved
PVE-002	Informational	Simplified depositPrizes() Logic in Raffle	Coding Practices	Resolved
PVE-003	Low	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-004	Medium	Possible Cancellation Denial-of-Service in Raffle	Time And State	Resolved

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 Inconsistent Enforcement of whenNotPaused in cancel()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Raffle`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

Each `raffle` is created with various parameters (e.g., cutoff time, minimum entries, maximum entries per participant, fees, and prizes) and has its own unique lifecycle. While examining different states and related transitions, we notice the current transition to the `Cancel` state has inconsistent `whenNotPaused` enforcement, which may be resolved for consistency.

In the following, we show the implementation of the related `cancel()` routine. As the name indicates, this routine in essence cancels the given `raffle`, which may only be transitioned from `Created` or `Open` state. In the meantime, we notice this routine has the `nonRecentrant` modifier to guard against possible re-entrancy. However, it does not have the `whenNotPaused` modifier to avoid the cancellation when the protocol is paused. Other state-transitioning routines do have both `nonRecentrant` and `whenNotPaused` modifiers.

```
    function cancel(uint256 raffleId) external nonReentrant {
        Raffle storage raffle = raffles[raffleId];

        RaffleStatus status = raffle.status;
        bool isOpen = status == RaffleStatus.Open;

        if (isOpen) {
            if (raffle.cutoffTime > block.timestamp) {
                revert CutoffTimeNotReached();
            }
        } else {
            _validateRaffleStatus(raffle, RaffleStatus.Created);
```

```
        }

        _cancel(raffleId, raffle, isOpen);
    }
```

Listing 3.1: `Raffle::cancel()`

Recommendation Revise the above routine to add the `whenNotPaused` for consistency.

Status The issue has been fixed by this commit: `d6c377d`.

3.2 Simplified depositPrizes() Logic in Raffle

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Raffle`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

As mentioned earlier, each `raffle` has different states in its lifecycle. While examining the state-transition logic from `Created` to `Open`, we notice the current implementation enforces it can only be triggered by the `raffle` owner, which may be relaxed.

In the following, we show below the implementation of the related `depositPrizes()` routine. It has a rather straightforward logic in depositing the specified prizes into the contract and then updating the `raffle` state to `Open`. It comes to our attention that there is a requirement on `_validateCaller(raffle.owner)` (line 242), which may be safely removed.

```
238    function depositPrizes(uint256 raffleId) external payable nonReentrant whenNotPaused
        {
239        Raffle storage raffle = raffles[raffleId];
240
241        _validateRaffleStatus(raffle, RaffleStatus.Created);
242        _validateCaller(raffle.owner);
243
244        Prize[] storage prizes = raffle.prizes;
245        uint256 prizesCount = prizes.length;
246        uint256 expectedEthValue;
247        for (uint256 i; i < prizesCount; ) {
248            Prize storage prize = prizes[i];
249            TokenType prizeType = prize.prizeType;
250            if (prizeType == TokenType.ERC721) {
251                _executeERC721TransferFrom(prize.prizeAddress, msg.sender, address(this)
                    , prize.prizeId);
252            } else if (prizeType == TokenType.ERC20) {
```

```
                _executeERC20TransferFrom(
                    prize.prizeAddress,
                    msg.sender,
                    address(this),
                    prize.prizeAmount * prize.winnersCount
                );
            } else if (prizeType == TokenType.ETH) {
                expectedEthValue += (prize.prizeAmount * prize.winnersCount);
            } else {
                _executeERC1155SafeTransferFrom(
                    prize.prizeAddress,
                    msg.sender,
                    address(this),
                    prize.prizeId,
                    prize.prizeAmount * prize.winnersCount
                );
            }
            unchecked {
                ++i;
            }
        }

        _validateExpectedEthValueOrRefund(expectedEthValue);

        raffle.status = RaffleStatus.Open;
        emit RaffleStatusUpdated(raffleId, RaffleStatus.Open);
    }
```

Listing 3.2: `Raffle::depositPrizes()`

Recommendation We can remove the caller verfication in the above `depositPrizes()` routine.

Status The issue has been resolved as the team confirms it is part of design.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Raffle`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `Raffle` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure various system parameters, claim protocol fees, and pause/resume protocols). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    function setProtocolFeeRecipient(address _protocolFeeRecipient) external onlyOwner {
        _setProtocolFeeRecipient(_protocolFeeRecipient);
    }

    /**
     * @inheritdoc IRaffle
     */
    function setProtocolFeeBp(uint16 _protocolFeeBp) external onlyOwner {
        _setProtocolFeeBp(_protocolFeeBp);
    }

    /**
     * @inheritdoc IRaffle
     */
    function updateCurrenciesStatus(address[] calldata currencies, bool isAllowed)
        external onlyOwner {
        uint256 count = currencies.length;
        for (uint256 i; i < count; ) {
            isCurrencyAllowed[currencies[i]] = isAllowed;
            unchecked {
                ++i;
            }
        }
        emit CurrenciesStatusUpdated(currencies, isAllowed);
    }

    /**
     * @inheritdoc IRaffle
     */
    function togglePaused() external onlyOwner {
        paused() ? _unpause() : _pause();
    }
```

Listing 3.3: Example Privileged Operations in `Raffle`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it would be better if the privileged account is governed by a DAO-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

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 confirmed by the team. The team intends to manage the admin keys with a multi-sig account.

3.4 Possible Cancellation Denial-of-Service in Raffle

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `Raffle`
- Category: Time and State [6]
- CWE subcategory: CWE-682 [3]

Description

The `Raffle` protocol supports a number of token types as the raffle prizes, including `ERC20`, `ERC721`, `ERC1155`, and `ETH`. While examining the `ERC1155`-based prizes, we notice the current cancellation logic may suffer from a subtle denial-of-service issue.

To elaborate, we show below the related `Raffle::cancel()` routine, which basically invokes the underlying `_cancel()` helper to return back the deposited prizes. In the prize-returning logic, we notice the raffle owner may be able to block the raffle from being cancelled when the fee token type is `ERC1155`. Specifically, the `_cancel()` helper calls the `_transferPrize()` routine, which makes use of `_executeERC1155SafeTransferFrom()` to potentially invoke the callback on the raffle owner. The callback can simply revert to block the cancellation, which essentially locks existing raffle entries.

```
    function cancel(uint256 raffleId) external nonReentrant {
        Raffle storage raffle = raffles[raffleId];

        RaffleStatus status = raffle.status;
        bool isOpen = status == RaffleStatus.Open;

        if (isOpen) {
            if (raffle.cutoffTime > block.timestamp) {
                revert CutoffTimeNotReached();
            }
        } else {
            _validateRaffleStatus(raffle, RaffleStatus.Created);
        }

        _cancel(raffleId, raffle, isOpen);
    }
    function _cancel(
        uint256 raffleId,
        Raffle storage raffle,
        bool shouldWithdrawPrizes
    ) private {
        raffle.status = RaffleStatus.Cancelled;

        if (shouldWithdrawPrizes) {
            uint256 prizesCount = raffle.prizes.length;
            for (uint256 i; i < prizesCount; ) {
```

```
                Prize storage prize = raffle.prizes[i];
                _transferPrize({prize: prize, recipient: raffle.owner, multiplier:
                    uint256(prize.winnersCount)});

                unchecked {
                    ++i;
                }
            }
        }

        emit RaffleStatusUpdated(raffleId, RaffleStatus.Cancelled);
    }
```

Listing 3.4: `Raffle::cancel()`

Recommendation Revisit the above logic to block the denial-of-service issue on the raffle cancellation.

Status The issue has been fixed by this commit: `adda28a`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Raffle` protocol, which allows the creation of a new `raffle` with parameters such as cutoff time, minimum entries, maximum entries per participant, fees, and prizes. The prizes can be deposited into the `raffle`, and participants can enter the `raffle` by purchasing entries. Once the `raffle` is concluded, the winners are selected with the help of some randomness provided by `Chainlink VRF`, after which the winners can claim their prizes. The contract also provides functionalities for the raffle owner to claim the collected fees and for the participants to withdraw their entry fees in case the `raffle` is cancelled. 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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[3] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.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: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.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.