SMART CONTRACT AUDIT REPORT

for

Raffle (V2)

Prepared By: Xiaomi Huang

PeckShield

July 11, 2023

Document Properties

Client	LooksRare
Title	Smart Contract Audit Report
Target	RaffleV2
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	July 11, 2023	Xuxian Jiang	Final Release
1.0-rc	July 1, 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 RaffleV2 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 RaffleV2

The RaffleV2 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 are deposited upon the raffle creation, and participants can enter the raffle by purchasing entries. Once the raffle is concluded, the winners are selected with the help of external 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 created raffle is cancelled. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of RaffleV2

Item	Description
Name	RaffleV2
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 11, 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 (a3be27d)

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 (84aee10)

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.

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.

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 `RaffleV2` 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	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 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 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key RaffleV2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Revisited RandomnessRequest Layout Without Randomness Loss	Coding Practices	Resolved
PVE-002	Informational	Improved Raffle Cancellation Logic After Drawing	Coding Practices	Resolved
PVE-003	Low	Trust Issue of Admin Keys	Security Features	Mitigated

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 Revisited RandomnessRequest Layout Without Randomness Loss

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RaffleV2`
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

Each `raffle` has a cutoff time and users need to participate before the cutoff time. After that, if sufficient entries are entered, the `raffle` will transition into the `Drawing` state and then rely on external randomness provided by `Chainlink VRF` for winner selection. In the process of analyzing the communication with external `Chainlink VRF`, we notice the current implementation can be improved for better randomness.

In the following, we show the implementation of the related `fulfillRandomWords()` routine. As the name indicates, this routine receives the random words returned by `Chainlink VRF`. We notice that the current logic only takes the first returned random word and explicitly ignores its most significant byte (since it packs the random word with a boolean state `exists`) to fulfill the prior random request `RandomnessRequest`. Note this `RandomnessRequest` structure has three member fields: `bool exists`, `uint248 randomWord`, and `uint256 raffleId`.

```
    function fulfillRandomWords(uint256 _requestId, uint256[] memory _randomWords)
        internal override {
        if (randomnessRequests[_requestId].exists) {
            uint256 raffleId = randomnessRequests[_requestId].raffleId;
            Raffle storage raffle = raffles[raffleId];

            if (raffle.status == RaffleStatus.Drawing) {
                _setRaffleStatus(raffle, raffleId, RaffleStatus.RandomnessFulfilled);
```

```
                // We ignore the most significant byte to pack the random word with `
                    exists`
                randomnessRequests[_requestId].randomWord = uint248(_randomWords[0]);
            }
        }
    }
```

Listing 3.1: `RaffleV2::fulfillRandomWords()`

Apparently, the current design packs the first two fields into one single word. However, the third member field `raffleId` has the `uint256` type but its maximum value can only be `2^80 - 1`, which suggests to re-organize the above `RandomnessRequest` structure as follows: `bool exists`, `uint80 raffleId`, and `uint256 randomWord`. The benefit here is the full utilization of the returned random word without entropy loss.

Recommendation Revise the above routine to fully utilize the return random word.

Status The issue has been fixed by this commit: `d582ff2`.

3.2 Improved Raffle Cancellation Logic After Drawing

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Raffle`
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

By design, each `raffle` has different states in its lifecycle. While examining the state-transition logic from the `raffle` cancellation, we notice the design can be revisited. Specifically, there are two occasions when a `raffle` may be cancelled: (1) when it is opened but not drawn after the cutoff; or (2) when it is drawn, but the request for randomness is not fulfilled for 24 hours. Our analysis shows the second cancellation can only be triggered by the contract owner, which may be relaxed.

In the following, we show below the implementation of the related `cancelAfterRandomnessRequest()` routine. It has a rather straightforward logic in cancelling the specified `raffleId` and then updating its state to `Refundable`. It comes to our attention that there is a requirement on `_onlyOwner` (line 708), which may be relaxed to allow anyone to cancel it. If the `raffle` has been drawn, but the request for randomness has not been fulfilled for 24 hours, any user should be entitled to cancel it without affecting the protocol functionality.

```
708     function cancelAfterRandomnessRequest(uint256 raffleId) external onlyOwner
            nonReentrant {
```

```
        Raffle storage raffle = raffles[raffleId];

        _validateRaffleStatus(raffle, RaffleStatus.Drawing);

        if (block.timestamp < raffle.drawnAt + ONE_DAY) {
            revert DrawExpirationTimeNotReached();
        }

        _setRaffleStatus(raffle, raffleId, RaffleStatus.Refundable);
    }
```

Listing 3.2: `RaffleV2::cancelAfterRandomnessRequest()`

Recommendation We can remove the `_onlyOwner` verfication in the above routine. In the meantime, we also suggest to add the `whenNotPaused` modifier for consistency.

Status The issue has been fixed by this commit: `16b56e8`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RaffleV2`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `RaffleV2` 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 IRaffleV2
     */
    function setProtocolFeeBp(uint16 _protocolFeeBp) external onlyOwner {
        _setProtocolFeeBp(_protocolFeeBp);
    }

    /**
     * @inheritdoc IRaffleV2
     */
```

```
    function updateCurrenciesStatus(address[] calldata currencies, bool isAllowed)
        external onlyOwner {
        uint256 count = currencies.length;
        for (uint256 i; i < count; ) {
            isCurrencyAllowed[currencies[i]] = (isAllowed ? 1 : 0);
            unchecked {
                ++i;
            }
        }
        emit CurrenciesStatusUpdated(currencies, isAllowed);
    }
```

Listing 3.3: Example Privileged Operations in `RaffleV2`

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.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `RaffleV2` 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 are deposited upon the `raffle` creation, 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 created `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 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.