SMART CONTRACT AUDIT REPORT

for

PANCAKESWAP LOTTERY

Prepared By: Yiqun Chen

PeckShield

July 7, 2021

Document Properties

Client	PancakeSwap
Title	Smart Contract Audit Report
Target	PancakeSwap Lottery
Version	1.0
Author	Xuxian Jiang
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Final

Version Info

Version	Date	Author(s)	Description
1.0	July 7, 2021	Xuxian Jiang	Final Release
1.0-rc	July 6, 2021	Shulin Bie	Release Candidate

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 `PancakeSwap Lottery` 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 PancakeSwap Lottery

The `PancakeSwap Lottery` is a system that allows users to buy tickets with bets on 6 digit numbers (from 0 to 9). Basically, each ticket has 6 digits and the protocol requires the winning number in order, starting from the first number (from the left). The more numbers the ticket matches in order, the higher the ticket wins from the prize bracket. The implementation consists of two contracts: `PancakeSwapLottery` and `RandomNumberGenerator`. The first one handles the logic for starting and closing lotteries, buying tickets, or viewing lottery information. It also has a function to draw the final numbers, randomly generated with a call to the second contract, which inherits from the `VRFConsumerBase` implementation from `ChainLink`.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of PancakeSwap Lottery

Item	Description
Name	PancakeSwap Lottery
Website	https://pancakeswap.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 7, 2021

In the following, we show the audited contract code deployed at the BSC chain with the following addresses:

- https://bscscan.com/address/0x8c6375Aab6e5B26a30bF241EBBf29AD6e6c503c2#code
- https://bscscan.com/address/0x5aF6D33DE2ccEC94efb1bDF8f92Bd58085432d2c#code

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.

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. 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 `PancakeSwap Lottery` 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	3	■ ■ ■
Informational	0	
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, 3 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Variable Naming in _calculateRewardsForTicketId()	Coding Practices	Resolved
PVE-002	Low	Improved Logic Of claimTickets()	Business Logic	Resolved
PVE-003	Low	Improved Corner Case Handling In changeRandomGenerator()	Coding Practices	Resolved
PVE-004	Medium	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 Improved Variable Naming in _calculateRewardsForTicketId()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: PancakeSwapLottery
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

PancakeSwap Lottery consists of two contracts: PancakeSwapLottery and RandomNumberGenerator: The first one handles the logic for starting and closing lotteries, buying tickets, or viewing lottery information and the second one generates the final random winning number. For each purchased ticket, there is a helper routine to compute its rewards. In the following, we examine this specific helper routine _calculateRewardsForTicketId().

To elaborate, we show below the _calculateRewardsForTicketId() routine. For a given ticket ID, this routine retrieves the user ticket number (line 1303) as well as the winning number (line 1300) and then transforms these two numbers to determine whether there is any match. It turns out that the user ticket number is internally named as winningTicketNumber while the winning number is represented as userNumber. These two internal variables can be exchanged to better present their intended semantic meanings.

```
1288    /**
1289     * @notice Calculate rewards for a given ticket
1290     * @param _lotteryId: lottery id
1291     * @param _ticketId: ticket id
1292     * @param _bracket: bracket for the ticketId to verify the claim and calculate
            rewards
1293     */
1294    function _calculateRewardsForTicketId(
1295        uint256 _lotteryId,
```

```
        uint256 _ticketId,
        uint32 _bracket
    ) internal view returns (uint256) {
        // Retrieve the winning number combination
        uint32 userNumber = _lotteries[_lotteryId].finalNumber;

        // Retrieve the user number combination from the ticketId
        uint32 winningTicketNumber = _tickets[_ticketId].number;

        // Apply transformation to verify the claim provided by the user is true
        uint32 transformedWinningNumber = _bracketCalculator[_bracket] +
            (winningTicketNumber % (uint32(10)**(_bracket + 1)));

        uint32 transformedUserNumber = _bracketCalculator[_bracket] + (userNumber % (
            uint32(10)**(_bracket + 1)));

        // Confirm that the two transformed numbers are the same, if not throw
        if (transformedWinningNumber == transformedUserNumber) {
            return _lotteries[_lotteryId].cakePerBracket[_bracket];
        } else {
            return 0;
        }
    }
```

Listing 3.1: `PancakeSwapLottery::_calculateRewardsForTicketId()`

Recommendation Choose variable names that better represent the intended purpose.

Status The issue has been fixed by switching these two variable names: `userNumber` and `winningTicketNumber`.

3.2 Improved Logic Of claimTickets()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PancakeSwapLottery`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

At the core the `PancakeSwap Lottery` protocol is the `PancakeSwapLottery` contract, which has a key function `claimTickets()`. This key function is designed to allow users to claim a set of winning tickets for a lottery. For that, there is a need to validate that the user is indeed claiming the correct bracket.

To elaborate, we show below the `claimTickets()` function. The validation on claiming the correct bracket is achieved by calling an internal helper routine `_calculateRewardsForTicketId()` (line 873).

If this internal helper routine returns no rewards for a higher number of bracket, the given ticket is considered as claiming the right bracket. However, it does not consider the situation when the higher bracket has 0 as its reward breakdown. Moreover, if the even higher number of bracket has a non-0 `_calculateRewardsForTicketId()`, the current user may not claim the right bracket.

```
    function claimTickets(
        uint256 _lotteryId,
        uint256[] calldata _ticketIds,
        uint32[] calldata _brackets
    ) external override notContract nonReentrant {
        require(_ticketIds.length == _brackets.length, "Not same length");
        require(_ticketIds.length != 0, "Length must be >0");
        require(_ticketIds.length <= maxNumberTicketsPerBuyOrClaim, "Too many tickets");
        require(_lotteries[_lotteryId].status == Status.Claimable, "Lottery not claimable");

        // Initializes the rewardInCakeToTransfer
        uint256 rewardInCakeToTransfer;

        for (uint256 i = 0; i < _ticketIds.length; i++) {
            require(_brackets[i] < 6, "Bracket out of range"); // Must be between 0 and 5

            uint256 thisTicketId = _ticketIds[i];

            require(_lotteries[_lotteryId].firstTicketIdNextLottery > thisTicketId, "TicketId too high");
            require(_lotteries[_lotteryId].firstTicketId <= thisTicketId, "TicketId too low");
            require(msg.sender == _tickets[thisTicketId].owner, "Not the owner");

            // Update the lottery ticket owner to 0x address
            _tickets[thisTicketId].owner = address(0);

            uint256 rewardForTicketId = _calculateRewardsForTicketId(_lotteryId, thisTicketId, _brackets[i]);

            // Check user is claiming the correct bracket
            require(rewardForTicketId != 0, "No prize for this bracket");

            if (_brackets[i] != 5) {
                require(
                    _calculateRewardsForTicketId(_lotteryId, thisTicketId, _brackets[i] + 1) == 0,
                    "Bracket must be higher"
                );
            }

            // Increment the reward to transfer
            rewardInCakeToTransfer += rewardForTicketId;
        }

```

```
        // Transfer money to msg.sender
        cakeToken.safeTransfer(msg.sender, rewardInCakeToTransfer);

        emit TicketsClaim(msg.sender, rewardInCakeToTransfer, _lotteryId, _ticketIds.
            length);
    }
```

Listing 3.2: `PancakeSwapLottery::claimTickets()`

Recommendation Properly revise the `claimTickets()` routine to consider the above-mentioned corner case.

Status The team has confirmed that the above situation should not happen with the way the lottery is operated and the way the brackets are set.

3.3 Improved Corner Case Handling In changeRandomGenerator()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PancakeSwapLottery`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

The `PancakeSwap Lottery` protocol allows the privileged `owner` to customize a number of protocol parameters. In the following, we examine one specific privileged `changeRandomGenerator()` function that can be used to change the random number generator.

```
    function changeRandomGenerator(address _randomGeneratorAddress) external onlyOwner {
        require(_lotteries[currentLotteryId].status == Status.Claimable, "Lottery not in
             claimable");

        // Request a random number from the generator based on a seed
        IRandomNumberGenerator(_randomGeneratorAddress).getRandomNumber(
            uint256(keccak256(abi.encodePacked(currentLotteryId, currentTicketId)))
        );

        // Calculate the finalNumber based on the randomResult generated by ChainLink's
            fallback
        IRandomNumberGenerator(_randomGeneratorAddress).viewRandomResult();

        randomGenerator = IRandomNumberGenerator(_randomGeneratorAddress);

        emit NewRandomGenerator(_randomGeneratorAddress);
```

```
1008    }
```

Listing 3.3: `PancakeSwapLottery::changeRandomGenerator()`

To elaborate, we show above the `changeRandomGenerator()` function. The function has an entry requirement (line 995) that disallows the change when the current lottery is not in the claimable status. However, it also blocks the change when there is no lottery yet.

Recommendation Improve the above `changeRandomGenerator()` function by adding the support of the corner case. An example revision will be the following requirement `require((currentLotteryId == 0)|| (_lotteries[currentLotteryId].status == Status.Claimable))`.

Status This issue has been fixed by taking the above recommendation.

3.4 Trust Issue Of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `PancakeSwapLottery`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `PancakeSwap Lottery` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., changing operators and configuring various system parameters). In the following, we show the representative functions potentially affected by the privilege of the `owner` account.

```
     function setMinAndMaxTicketPriceInCake(uint256 _minPriceTicketInCake, uint256
         _maxPriceTicketInCake)
        external
        onlyOwner
    {
        require(_minPriceTicketInCake <= _maxPriceTicketInCake, "minPrice must be <
            maxPrice");

        minPriceTicketInCake = _minPriceTicketInCake;
        maxPriceTicketInCake = _maxPriceTicketInCake;
    }

    /**
     * @notice Set max number of tickets
     * @dev Only callable by owner
     */
    function setMaxNumberTicketsPerBuy(uint256 _maxNumberTicketsPerBuy) external
        onlyOwner {
```

```
        require(_maxNumberTicketsPerBuy != 0, "Must be > 0");
        maxNumberTicketsPerBuyOrClaim = _maxNumberTicketsPerBuy;
    }

    function setOperatorAndTreasuryAndInjectorAddresses(
        address _operatorAddress,
        address _treasuryAddress,
        address _injectorAddress
    ) external onlyOwner {
        require(_operatorAddress != address(0), "Cannot be zero address");
        require(_treasuryAddress != address(0), "Cannot be zero address");
        require(_injectorAddress != address(0), "Cannot be zero address");

        operatorAddress = _operatorAddress;
        treasuryAddress = _treasuryAddress;
        injectorAddress = _injectorAddress;

        emit NewOperatorAndTreasuryAndInjectorAddresses(_operatorAddress,
            _treasuryAddress, _injectorAddress);
    }
```

Listing 3.4: A number of representative `setters` in `PancakeSwapLottery`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. At the same time the extra power to the owner may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these `onlyOwner` privileges explicit or raising necessary awareness among lottery users.

Recommendation Make the list of extra privileges granted to `owner` explicit to lottery users.

Status This issue has been confirmed by the team. The privileged account will be managed by a multi-sig account.

4 Conclusion

In this audit, we have analyzed the design and implementation of `PancakeSwap Lottery`, which is a system that allows users to buy tickets with bets on 6 digit numbers (from 0 to 9). The implementation consists of two contracts: `PancakeSwapLottery` and `RandomNumberGenerator`. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

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-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.