SMART CONTRACT AUDIT REPORT

for

PancakeSwap CakePool

Prepared By: Patrick Lou

PeckShield

April 19, 2022

Document Properties

Client	PancakeSwap Finance
Title	Smart Contract Audit Report
Target	PancakeSwap CakePool
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 19, 2022	Xuxian Jiang	Final Release
1.0-rc	April 3, 2022	Xuxian Jiang	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Patrick Lou
Phone	+86 156 0639 2692
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the CakePool contract in the PancakeSwap 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 contract can be further improved due to the presence of several issues related to business Logic or security. This document outlines our audit results.

1.1 About PancakeSwap CakePool

PancakeSwap is the leading decentralized exchange on BNB Smart Chain (previously BSC), with very high trading volumes in the market. The audited CakePool is an extension to the original PancakeSwap MasterChef protocol for liquidity mining, which allows users to earn CAKE rewards while supporting PancakeSwap by staking the same CAKE token. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of the PancakeSwap

Item	Description
Name	PancakeSwap Finance
Website	https://pancakeswap.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 19, 2022

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

- https://github.com/ChefSnoopy/pancake-contracts.git (070cddd)

1.2 About PeckShield

PeckShield Inc. [11] 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 [10]:

- 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.

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

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

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) [9], 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 `PancakeSwap CakePool` smart contract 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 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 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, this smart contract is 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 PancakeSwap CakePool Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possibly Open Repeated Pool Initialization	Business Logic	Resolved
PVE-002	Informational	Removal of Unused State And Code	Coding Practices	Resolved
PVE-003	Low	Suggested Reentrancy Protection in Deposit and Withdraw	Time and State	Confirmed
PVE-004	Medium	Trust Issue Of Admin Keys	Security Features	Resolved

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Possibly Open Repeated Pool Initialization

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `CakePool`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In `CakePool`, it initializes the protocol by depositing a dummy token into the `MasterChefV2` contract. By doing this, it could earn a constant number of `CAKE` tokens per block from `MasterChefV2`. The `CakePool` contract then handles the distribution of the `CAKE` rewards to staking users. While reviewing the current initialization logic of `CakePool`, we notice the current implementation can be improved.

To elaborate, we show below the implementation of the `init()` routine. It has a rather straightforward logic in depositing the intended `dummyToken` to `MasterChefV2`. However, it comes to our attention that there is no access control for the `init()` routine, and it could be called more than once. By design, `init()` should be called only once by the protocol owner.

```
    function init(IERC20 dummyToken) external {
        uint256 balance = dummyToken.balanceOf(msg.sender);
        require(balance != 0, "Balance must exceed 0");
        dummyToken.safeTransferFrom(msg.sender, address(this), balance);
        dummyToken.approve(address(masterchefV2), balance);
        masterchefV2.deposit(cakePoolPID, balance);
        emit Init();
    }
```

Listing 3.1: `CakePool::init()`

Recommendation Ensure the `init()` routine could only be called once by applying the `initializer` or `onlyOwner` modifiers.

Status The issue has been fixed by this commit: `1c629bf`.

3.2 Removal of Unused State And Code

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `CakePool`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [2]

Description

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

If we examine closely the key storage states in `CakePool`, there are several states that are defined but not used. Examples include `lastHarvestedTime` and `MAX_CALL_FEE`. These two states can be safely removed.

```
    uint256 public totalShares;
    uint256 public lastHarvestedTime;
    address public admin;
    address public treasury;
    address public operator;
    uint256 public cakePoolPID;
    uint256 public totalBoostDebt; // total boost debt.
    uint256 public totalLockedAmount; // total lock amount.

    uint256 public constant MAX_PERFORMANCE_FEE = 2000; // 20%
    uint256 public constant MAX_CALL_FEE = 100; // 1%
    uint256 public constant MAX_WITHDRAW_FEE = 500; // 5%
    uint256 public constant MAX_WITHDRAW_FEE_PERIOD = 1 weeks; // 1 week
    uint256 public constant MIN_LOCK_DURATION = 1 weeks; // 1 week
    uint256 public constant MAX_LOCK_DURATION_LIMIT = 1000 days; // 1000 days
    uint256 public constant BOOST_WEIGHT_LIMIT = 500 * 1e10; // 500%
    uint256 public constant PRECISION_FACTOR = 1e12; // precision factor.
    uint256 public constant PRECISION_FACTOR_SHARE = 1e28; // precision factor for share
        .
    uint256 public constant MIN_DEPOSIT_AMOUNT = 0.00001 ether;
    uint256 public constant MIN_WITHDRAW_AMOUNT = 0.00001 ether;
```

Listing 3.2: The `CakePool` Contract

Recommendation Consider the removal of the redundant states with a simplified, consistent implementation.

Status The issue has been fixed by this commit: `1c629bf`.

3.3 Suggested Reentrancy Protection in Deposit and Withdraw

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CakePool`
- Category: Time and State [8]
- CWE subcategory: CWE-663 [3]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [13] exploit, and the recent `Uniswap/Lendf.Me` hack [12].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `CakePool` as an example, the `withdrawOperation()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (lines 424 and 428) starts before effecting the update on internal states, hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
389     function withdrawOperation(uint256 _shares, uint256 _amount) internal {
390         UserInfo storage user = userInfo[msg.sender];
391         require(_shares <= user.shares, "Withdraw amount exceeds balance");
392         require(user.lockEndTime < block.timestamp, "Still in lock");
393
394         // Calculate the percent of withdraw shares, when unlocking or calculating the
                Performance fee, the shares will be updated.
395         uint256 currentShare = _shares;
396         uint256 sharesPercent = (_shares * PRECISION_FACTOR_SHARE) / user.shares;
397
398         // Harvest token from MasterchefV2.
```

```
        harvest();

        // Update user share.
        updateUserShare(msg.sender);

        if (_shares == 0 && _amount > 0) {
            uint256 pool = balanceOf();
            currentShare = (_amount * totalShares) / pool; // Calculate equivalent shares
            if (currentShare > user.shares) {
                currentShare = user.shares;
            }
        } else {
            currentShare = (sharesPercent * user.shares) / PRECISION_FACTOR_SHARE;
        }
        uint256 currentAmount = (balanceOf() * currentShare) / totalShares;
        user.shares -= currentShare;
        totalShares -= currentShare;

        // Calculate withdraw fee
        if (!freeFeeUsers[msg.sender] && (block.timestamp < user.lastDepositedTime + withdrawFeePeriod)) {
            uint256 feeRate = withdrawFee;
            if (_isContract(msg.sender)) {
                feeRate = withdrawFeeContract;
            }
            uint256 currentWithdrawFee = (currentAmount * feeRate) / 10000;
            token.safeTransfer(treasury, currentWithdrawFee);
            currentAmount -= currentWithdrawFee;
        }

        token.safeTransfer(msg.sender, currentAmount);

        if (user.shares > 0) {
            user.cakeAtLastUserAction = (user.shares * balanceOf()) / totalShares;
        } else {
            user.cakeAtLastUserAction = 0;
        }

        user.lastUserActionTime = block.timestamp;

        // Update user info in Boost Contract.
        updateBoostContractInfo(msg.sender);

        emit Withdraw(msg.sender, currentAmount, currentShare);
    }
```

Listing 3.3: `CakePool::withdrawOperation()`

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`. However, it is important to take precautions in making use of `nonReentrant` to block possible re-entrancy. Note a similar issue exists in another routine depositOperation().

Recommendation Apply necessary reentrancy prevention by utilizing the nonReentrant modifier to block possible re-entrancy.

Status The issue has been confirmed.

3.4 Trust Issue Of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: CakePool
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

Description

In CakePool contract, there are privileged accounts (including owner and admin) that play critical roles in governing and regulating the protocol-related operations. To elaborate, we show below the sensitive operations that are related to owner/admin.

```
    function setOperator(address _operator) external onlyOwner {
        require(_operator != address(0), "Cannot be zero address");
        operator = _operator;
    }

    /**
     * @notice Set Boost Contract address
     * @dev Callable by the contract admin.
     */
    function setBoostContract(address _boostContract) external onlyAdmin {
        require(_boostContract != address(0), "Cannot be zero address");
        boostContract = _boostContract;
    }

    /**
     * @notice Set free fee address
     * @dev Only callable by the contract admin.
     * @param _user: User address
     * @param _free: true:free false:not free
     */
    function setFreeFeeUser(address _user, bool _free) external onlyAdmin {
        require(_user != address(0), "Cannot be zero address");
        freeFeeUsers[_user] = _free;
    }
```

Listing 3.4: Example Privileged Operations in CakePool

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts 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 privileges explicit or raising necessary awareness among protocol users.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changes 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 This issue has been confirmed by the team. They will use time lock and multi-signature scheme to ensure admin key security.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `CakePool` contract in the `PancakeSwap` protocol. The protocol is designed to allow users to earn `CAKE` rewards while supporting `PancakeSwap` by staking the same `CAKE` token. During the audit, we notice that the current code base is well organized.

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-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

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

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

[8] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

[12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[13] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.