SMART CONTRACT AUDIT REPORT

for

PancakeSwap Cross Farming

Prepared By: Xiaomi Huang

PeckShield

September 28, 2022

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	September 28, 2022	Luck Hu	Final Release
1.0-rc	September 27, 2022	Luck Hu	Release Candidate

Contact

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

Name	Xiaomi Huang
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 `PancakeSwap Cross Farming` 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 Cross Farming

`PancakeSwap` deploys the `V2Swap Dex Exchange` on an `EVM`-compatible chain and the `Masterchef` farm pool on the `BSC` chain. The audited `Cross Farming` protocol allows for the users of the exchanges on the `EVM` chain to stake their `LP` tokens to the vault and get in return the same amount of `LP` tokens that will be deposited to the `Masterchef` farm pool on the `BSC` chain. In this way, users can enjoy the `CAKE` rewards from the `Masterchef`. The `Cross Farming` is built on the `bridge/messagebus` solution of `Celer`. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The `Cross Farming`

Item	Description
Name	PancakeSwap Finance
Website	https://pancakeswap.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 28, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the audit only covers the contracts under the `projects/cross-chain/contracts` directory.

- https://github.com/chefcooper/pancake-contracts/tree/feature/cross-chain (3ab582b)

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

- https://github.com/chefcooper/pancake-contracts/tree/feature/cross-chain (f56a59f)

1.2 About PeckShield

PeckShield Inc. [8] 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 [7]:

- 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) [6], 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 Cross Farming` 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 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	1	■
Informational	1	■
Total	3	

We have so far identified a list of potential issues. 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, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key PancakeSwap Cross Farming Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Suggested immutable Usages for Gas Efficiency	Coding Practices	Fixed
PVE-002	Informational	Removal of Redundant Code	Coding Practices	Fixed
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

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 Suggested immutable Usages for Gas Efficiency

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: CrossFarmingProxy
- Category: Coding Practices [5]
- CWE subcategory: CWE-1099 [1]

Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

In the following, we show the key state variable `factory` in the `CrossFarmingProxy` contract. If there is no need to dynamically update this key variable, it can be declared as either constant or `immutable` for gas efficiency. In particular, the above state variable can be defined as `immutable` as it will not be changed after its initialization in `constructor()`.

```
contract CrossFarmingProxy is ReentrancyGuard {
  using SafeERC20 for IERC20;

  // cross-chain user address.
  address public user;
  // CAKE token.
```

```
    address public CAKE;
    // cross-farming receiver contract on BSC chain.
    address public factory;
    // MCV2 contract.
    IMasterChefV2 public MASTER_CHEF_V2;
    ...
    constructor() {
        factory = msg.sender;
    }
```

Listing 3.1: `CrossFarmingProxy.sol`

Recommendation Revisit the state variable definition and make extensive use of `immutable` states for gas efficiency.

Status This issue has been fixed in the following commit: `f56a59f`.

3.2 Removal of Redundant Code

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `CrossFarmingVault`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [3]

Description

As the core of `PancakeSwap Cross Farming` protocol, the `CrossFarmingVault` contract provides the main entry point for interacting with users. It makes good use of a number of reference contracts, such as `Ownable`, `Pausable`, and `ReentrancyGuard`, to facilitate its code implementation and organization. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the events defined in the `CrossFarmingVault` contract, there is a `FarmingContractUpdated()` event that is not used anywhere (line 73).

```
69  event AckWithdraw(address indexed sender, uint256 pid, uint256 amount, uint64 nonce);
70  event FallbackDeposit(address indexed sender, uint256 pid, uint256 amount, uint64
        nonce);
71  event FallbackWithdraw(address indexed sender, uint256 pid, uint256 amount, uint64
        nonce);
72  event AckEmergencyWithdraw(address indexed sender, uint256 pid, uint256 amount, uint64
        nonce);
73  event FarmingContractUpdated(address indexed sender, address senderContract, address
        receiverContract);
```

Listing 3.2: `CrossFarmingVault::events`

In addition, the `CrossFarmingVault` contract allows for the owner to pause/unpause the interacting with users via the `pause()/unpause()` routines. However, there is a redundant involving of the `whenNotPaused/whenPaused` modifiers, as these modifiers have been invoked in the internal `_pause()/_unpause()` routines. Based on this, the involving of the `whenNotPaused/whenPaused` modifiers can be removed.

```
/**
* @notice Triggers stopped state
* @dev Only possible when contract not paused.
*/
function pause() external onlyOwner whenNotPaused {
    _pause();
    emit Pause();
}

/**
* @notice Returns to normal state
* @dev Only possible when contract is paused.
*/
function unpause() external onlyOwner whenPaused {
    _unpause();
    emit Unpause();
}
```

Listing 3.3: `CrossFarmingVault::pause()/unpause()`

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

Status This issue has been fixed in the following commit: `f56a59f`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In `PancakeSwap Cross Farming` protocol, there are certain privileged accounts (including `owner` and `operator`, etc.) that play critical roles in governing and regulating the protocol-related operations. Our analysis shows that these privileged accounts need to be scrutinized. In the following, we use the

CrossFarmingSender contract as an example and show the representative functions potentially affected by the privilege of the owner.

Specifically, the privileged functions in CrossFarmingSender allow for the owner to set the price oracle and the oracleUpdateBuffer which are used to calculate the exchange rate between BNB/ETH, set the gas cost for specific operation in different chains, set the compensation rate for the differences of gas price/gas limit between different chains, etc.

```
    /// set oracle data feeds.
    function setOracle(Feeds _feed, address _oracle) external onlyOwner {
        require(_oracle != address(0), "Oracle feed can't be zero address");
        oracle[_feed] = AggregatorV3Interface(_oracle);

        // Dummy check to make sure the interface implements this function properly
        oracle[_feed].latestRoundData();

        emit NewOracle(_oracle);
    }

    /// set oracle update buffer, different oracle feeds have different update frequency,
    /// so the buffer should also change accordingly
    function setOracleUpdateBuffer(Feeds _feed, uint256 _oracleUpdateBuffer) external onlyOwner {
        require(_oracleUpdateBuffer > 0, "oracle update time buffer should > 0");
        oracleUpdateBuffer[_feed] = _oracleUpdateBuffer;
        emit OracleBufferUpdated(_feed, _oracleUpdateBuffer);
    }

    /// set gas cost for specific operation in different chain.
    function setGaslimits(
        Chains _chain,
        DataTypes.MessageTypes _type,
        uint256 _gaslimit
    ) external onlyOwner {
        require(_gaslimit > 0, "Gaslimit should be > zero");
        gaslimits[_chain][_type] = _gaslimit;
        emit GasLimitUpdated(_chain, _type, _gaslimit);
    }

    /// @notice gas price and gas limit is different in different EVM chain, compensation rate
    /// is for hedging the risk of this difference caused the executor signer lose gas fee in execution
    function setCompensationRate(uint256 _rate) external onlyOwner {
        require(_rate >= MIN_COMPENSATION_RATE && _rate <= MAX_COMPENSATION_RATE, "Invalid compensation rate");
        compensationRate = _rate;
        emit CompensationRateUpdated(compensationRate);
    }

    /// set BNB change amount for new BSC chain user.
```

```
    function setBnbChange(uint256 _change) external onlyOwner {
        require(_change > 0, "BNB change for new user should greater than zero");
        BNB_CHANGE = _change;
        emit BnbChangeUpdated(_change);
    }

    /// create farming-proxy contract gas limit cost in BSC chain.
    function setCreateProxyGasLimit(uint256 _gaslimit) external onlyOwner {
        createProxyGasLimit = _gaslimit;
        emit CreateProxyGasLimitUpdated(_gaslimit);
    }
```

Listing 3.4: Example Privileged Operations in the `CrossFarmingSender` Contract

It would be worrisome if the `owner` or other privileged accounts are plain EOAs. A multi-sig account could greatly alleviate this concern, though it is far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered for mitigation.

Recommendation Suggest a multi-sig account plays each privileged account to mitigate this issue. Additionally, all changes to privileged operations may need to be mediated with necessary timelocks.

Status This issue has been confirmed by the team.

4 | Conclusion

In this audit, we have analyzed the PancakeSwap Cross Farming protocol design and implementation. PancakeSwap deploys the V2Swap Dex Exchange on an EVM-compatible chain and the Masterchef farm pool on the BSC chain. The audited Cross Farming allows for the users of the exchanges on the EVM chain to stake their LP tokens to the vault and get in return the same amount of LP tokens that will be deposited to the Masterchef farm pool on BSC chain. In this way, users can enjoy the CAKE rewards from the Masterchef on BSC chain. Our analysis shows that the current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

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

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.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 VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

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

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