



SMART CONTRACT AUDIT REPORT

for

PancakeSwap FarmBooster



Prepared By: Xiaomi Huang

PeckShield
July 30, 2022

Document Properties

Client	PancakeSwap Finance
Title	Smart Contract Audit Report
Target	PancakeSwap FarmBooster
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 30, 2022	Luck Hu	Final Release
1.0-rc	July 29, 2022	Luck Hu	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	4
1.1	About PancakeSwap FarmBooster	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Incorrect Index Value Used in remove()	11
3.2	Potential Reentrancy Risk in deposit() and withdraw()	12
3.3	Trust Issue Of Admin Keys	14
4	Conclusion	16
	References	17

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the PancakeSwap FarmBooster 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 FarmBooster

PancakeSwap is the leading decentralized exchange on BNB Smart Chain (previously BSC), with very high trading volumes in the market. The PancakeSwap FarmBooster protocol is one of the core functions of PancakeSwap, which controls the boost multiplier for users in the main MasterChef, using parameters fetched from MasterChef pools and CakePool (cake staking pool). 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	July 30, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the audited project is `farm-booster` which is under directory `projects/farm-booster/contracts/`.

- <https://github.com/chefcooper/pancake-contracts.git> (69b821a)

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.git> (2f98da5)

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	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

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 PancakeSwap FarmBooster 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	2	
Low	1	
Informational	0	
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, the 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 medium-severity vulnerabilities and 1 low-severity vulnerability.

Table 2.1: Key PancakeSwap FarmBooster Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incorrect Index Value Used in remove()	Coding Practices	Fixed
PVE-002	Low	Potential Reentrancy Risk in deposit() and withdraw()	Time and State	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 Incorrect Index Value Used in remove()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: IterableMapping
- Category: Coding Practices [5]
- CWE subcategory: CWE-682 [3]

Description

The FarmBooster contract maintains a state variable (i.e. `mapping(address => ItMap) public userInfo`) which records the pid-multiplier pairs for each users proxy. The pid-multiplier pair is represented with the ItMap struct which is implemented in the IterableMapping library. While examining the logic to add and remove a pid-multiplier entry, we notice the existence of using an incorrect key index.

To elaborate, we show below code snippet from the IterableMapping library. As the names indicate, the `insert()` routine is used to insert a key/value pair, and the `remove()` routine is used to remove the pair of the input key. In the `insert()` routine, the key is pushed into an array `keys` and the index of key in `keys` is recorded by the `indexs[key]`. Note that the index value starts from 1, not 0. In the `remove()` routine, it gets the index of the input key and moves the last key from the end of the `keys` to the index of the key to be removed. And then, it updates the `indexs[lastKey]` to `index - 1` (line 37) which is incorrect. By design, the `indexs[lastKey]` shall be updated to `index` which is the index of the removed key.

```
16     function insert(  
17         ItMap storage self,  
18         uint256 key,  
19         uint256 value  
20     ) internal {  
21         uint256 keyIndex = self.indexs[key];  
22         self.data[key] = value;  
23         if (keyIndex > 0) return;  
24         else {
```

```

25         self.indexs[key] = self.keys.length + 1;
26         self.keys.push(key);
27         return;
28     }
29 }
30
31 function remove(IterableMapping self, uint256 key) internal {
32     uint256 index = self.indexs[key];
33     if (index == 0) return;
34     uint256 lastKey = self.keys[self.keys.length - 1];
35     if (key != lastKey) {
36         self.keys[index - 1] = lastKey;
37         self.indexs[lastKey] = index - 1;
38     }
39     delete self.data[key];
40     delete self.indexs[key];
41     self.keys.pop();
42 }

```

Listing 3.1: IterableMapping.sol

Recommendation Revise the above mentioned `remove()` routine to correctly update the `indexs[key]`.

Status The issue has been fixed in the following commit: [2f98da5](#).

3.2 Potential Reentrancy Risk in `deposit()` and `withdraw()`

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: FarmBoosterProxy
- Category: Time and State [\[6\]](#)
- CWE subcategory: CWE-663 [\[2\]](#)

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 [\[11\]](#) exploit, and the recent Uniswap/Lendf.Me hack [\[10\]](#).

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `FarmBoosterProxy` as an example, the `withdraw()` function (see the code snippet below) is provided to withdraw LP token from the `Masterchef`. However, the invocation of an external contract requires extra care in avoiding the above re-entrancy.

Apparently, the interaction with the external contract (lines 99 and 100) starts before effecting the update on internal states (line 101), hence violating the principle. In this particular case, the external contract may have certain hidden logic that may be capable of launching re-entrancy via the same entry function.

```

88  /**
89   * @notice Withdraw LP tokens from pool.
90   * @dev It can only be called by admin.
91   * @param _pid The id of the pool.
92   * @param _amount Amount of LP tokens to withdraw.
93   */
94  function withdraw(uint256 _pid, uint256 _amount) external onlyAdmin {
95      uint256 poolLength = masterchefV2.poolLength();
96      require(_pid < poolLength, "Pool is not exist");
97      masterchefV2.withdraw(_pid, _amount);
98      address lpAddress = masterchefV2.lpToken(_pid);
99      IERC20(lpAddress).safeTransfer(msg.sender, _amount);
100     harvestCake();
101     farmBooster.updatePoolBoostMultiplier(msg.sender, _pid);
102     emit WithdrawByProxy(msg.sender, address(this), _pid, _amount);
103 }

```

Listing 3.2: `FarmBoosterProxy::withdraw()`

Note another routine `deposit()` in this contract shares the same issue.

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

Status The issue has been fixed in the following commit: 2005d07.

3.3 Trust Issue Of Admin Keys

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

Description

In the PancakeSwap FarmBooster protocol, there is a privileged account, i.e., `owner` that plays a critical role in governing and regulating the system-wide operations (e.g., set the `BOOSTER_FACTORY`, set `cA/cB`, etc.). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `FarmBooster` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

Specifically, the privileged functions in the `FarmBooster` contract allow for the `owner` to set the `BOOSTER_FACTORY` which can set the proxy for any new user. And the `owner` can also set the pool status in the `whiteList` where only the allowed pools can support farm boosting, etc.

```

130     /// @notice set boost factory contract.
131     function setBoostFactory(address _factory) external onlyOwner {
132         require(_factory != address(0), "setBoostFactory: Invalid factory");
133         BOOSTER_FACTORY = _factory;

135         emit UpdateBoostFactory(_factory);
136     }

138     /// @notice Set user boost proxy contract, can only invoked by boost contract.
139     /// @param _user boost user address.
140     /// @param _proxy boost proxy contract.
141     function setProxy(address _user, address _proxy) external onlyFactory {
142         require(_proxy != address(0), "setProxy: Invalid proxy address");
143         require(proxyContract[_user] == address(0), "setProxy: User has already set
            proxy");

145         proxyContract[_user] = _proxy;

147         emit UpdateBoostProxy(_user, _proxy);
148     }

150     /// @notice Only allow whitelisted pids for farm boosting
151     /// @param _pid pool id(MasterchefV2 pool).
152     /// @param _status farm pool allowed boosted or not
153     function setBoosterFarms(uint256 _pid, bool _status) external onlyOwner {
154         whiteList[_pid] = _status;
155         emit UpdateBoostFarms(_pid, _status);

```

156

}

Listing 3.3: FarmBooster

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not 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 `owner` privileges to the intended governance contract. And 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 the admin key security.



4 | Conclusion

In this audit, we have analyzed the `PancakeSwap FarmBooster` protocol design and implementation. The protocol is designed to control the boost multiplier for users in the main `MasterChef`, using parameters fetched from `MasterChef` pools and `CakePool` (cake staking pool). 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-663: Use of a Non-reentrant Function in a Concurrent Context. <https://cwe.mitre.org/data/definitions/663.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: Concurrency. <https://cwe.mitre.org/data/definitions/557.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>.
- [10] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. <https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09>.

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

