SMART CONTRACT AUDIT REPORT

for

PancakeSwap MasterChefV2

Prepared By: Patrick Lou

PeckShield

March 23, 2022

Document Properties

Client	PancakeSwap Finance
Title	Smart Contract Audit Report
Target	PancakeSwap MasterChefV2
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	March 23, 2022	Luck Hu	Final Release
1.0-rc	March 18, 2022	Luck Hu	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 `PancakeSwap MasterChefV2` 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 MasterChefV2

`PancakeSwap` is the leading decentralized exchange on `BNB Smart Chain` (previously `BSC`), with very high trading volumes in the market. The `PancakeSwap MasterChefV2` protocol is one of the core functions of `PancakeSwap`, which allows users to earn `CAKE` rewards while supporting `PancakeSwap` by staking respective LP tokens. 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	March 23, 2022

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

- https://github.com/chefcooper/pancake-contracts/blob/dev/MasterChefV2/projects/masterchef/v2/contracts/MasterChefV2.sol (af1c18d)

1.2 About PeckShield

PeckShield Inc. [7] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [6]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

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) [5], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the `PancakeSwap MasterChefV2` 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	2	■■
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, 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 2 medium-severity vulnerabilities and 2 low-severity vulnerabilities.

Table 2.1: Key PancakeSwap MasterChefV2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Potential Re-Initialization Risks in init()	Business Logic	Fixed
PVE-002	Medium	Timely massUpdatePools During Cake Rate Changes	Business Logic	Confirmed
PVE-003	Low	Duplicate Pool Detection And Prevention	Business Logic	Confirmed
PVE-004	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 Potential Re-Initialization Risks in init()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: MasterChefV2
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In MasterChefV2, it initializes the protocol by depositing a dummy token into the MasterChef V1 (MCV1) contract. By doing this, it could earn a constant number of CAKE tokens per block from MCV1. The MasterChefV2 contract then handles the distribution of the CAKE rewards to all the PancakeSwap products. Among all the earned CAKE tokens, certain percentage (controlled by cakeRateToRegularFarm) of the CAKE is distributed to the regular farm pools and others (controlled by cakeRateToSpecialFarm and cakeRateToBurn) are distributed to the special farm pools or may be burned by transferring them to the burnAdmin account.

To elaborate, we show below the related code snippet of the init() routine. Specifically, it sets the variable lastBurnedBlock (line 156) with the block.number. The lastBurnedBlock variable records the block number when the last CAKE burn action is executed. With the lastBurnedBlock variable, the protocol could calculate the block numbers since the last CAKE burn action and then calculate the amount of CAKE pending for burn to burnAdmin.

However, it comes to our attention that there is no access control for the init() routine, and it could be called more than once. As the function name suggests, init() should be called only once by the protocol owner. In addition, if the init() is called more than once, the lastBurnedBlock will be reset to the current block.number, which makes the amount of CAKE pending for burn inaccurate.

```
149    function init(IBEP20 dummyToken) external {
150        uint256 balance = dummyToken.balanceOf(msg.sender);
151        require(balance != 0, "MasterChefV2: Balance must exceed 0");
```

```
        dummyToken.safeTransferFrom(msg.sender, address(this), balance);
        dummyToken.approve(address(MASTER_CHEF), balance);
        MASTER_CHEF.deposit(MASTER_PID, balance);
        // MCV2 start to earn CAKE reward from current block in MCV1 pool
        lastBurnedBlock = block.number;
        emit Init();
    }
```

Listing 3.1: MasterChefV2::init()

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

Status This issue has been fixed in the following commit: `58102f2`.

3.2 Timely massUpdatePools During Cake Rate Changes

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `MasterChefV2`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The `MasterChefV2` protocol provides an incentive mechanism that rewards the staking of supported assets with the `CAKE` token. The `CAKE` rewards are earned from the `MasterChef V1 (MCV1)` by depositing a dummy token into `MCV1`. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. The pools are classified to two different pool types (general and special) each of which takes different `CAKE` rate from the total `CAKE` rewards. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The `CAKE` rates of different pool types can be dynamically changed via `updateCakeRate()`. When analyzing the `CAKE` rates update, we notice the need of timely invoking `massUpdatePools()` to update the reward distribution before the new `CAKE` rate becomes effective.

```
    function updateCakeRate(
        uint256 _burnRate,
        uint256 _regularFarmRate,
        uint256 _specialFarmRate,
        bool _withUpdate
    ) external onlyOwner {
        require(
            _burnRate > 0 && _regularFarmRate > 0 && _specialFarmRate > 0,
            "MasterChefV2: Cake rate must be greater than 0"
```

```
        );
        require(
            _burnRate.add(_regularFarmRate).add(_specialFarmRate) ==
                CAKE_RATE_TOTAL_PRECISION,
            "MasterChefV2: Total rate must be 1e12"
        );
        if (_withUpdate) {
            massUpdatePools();
        }
        // burn cake base on old burn cake rate
        burnCake(false);

        cakeRateToBurn = _burnRate;
        cakeRateToRegularFarm = _regularFarmRate;
        cakeRateToSpecialFarm = _specialFarmRate;

        emit UpdateCakeRate(_burnRate, _regularFarmRate, _specialFarmRate);
    }
```

Listing 3.2: MasterChefV2::updateCakeRate()

Similarly, the reward pools can be dynamically added via `add()` and the weights of supported pools can be adjusted via `set()`. There is also the need of timely invoking `massUpdatePools()` to update the reward distribution before the new pool weight becomes effective.

```
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate
    ) external onlyOwner {
        // No matter _withUpdate is true or false, we need to execute updatePool once
            before set the pool parameters.
        updatePool(_pid);

        if (_withUpdate) {
            massUpdatePools();
        }

        if (poolInfo[_pid].isRegular) {
            totalRegularAllocPoint = totalRegularAllocPoint.sub(poolInfo[_pid].
                allocPoint).add(_allocPoint);
        } else {
            totalSpecialAllocPoint = totalSpecialAllocPoint.sub(poolInfo[_pid].
                allocPoint).add(_allocPoint);
        }
        poolInfo[_pid].allocPoint = _allocPoint;
        emit SetPool(_pid, _allocPoint);
    }
```

Listing 3.3: MasterChefV2::set()

If the call to `massUpdatePools()` is not immediately invoked before updating the `CAKE` rates or the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, these interfaces are restricted to the owner (via the `onlyOwner` modifier), which greatly alleviates the concern.

Recommendation Timely invoke `massUpdatePools()` when either any `CAKE` rate or any pool weight has been updated. In fact, the `_withUpdate` parameter to the `set()`, `add()` and `updateCakeRate()` routines can be simply ignored or removed.

```
    function set(
        uint256 _pid,
        uint256 _allocPoint
    ) external onlyOwner {
        // No matter _withUpdate is true or false, we need to execute updatePool once before set the pool parameters.
        massUpdatePools();

        if (poolInfo[_pid].isRegular) {
            totalRegularAllocPoint = totalRegularAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
        } else {
            totalSpecialAllocPoint = totalSpecialAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
        }
        poolInfo[_pid].allocPoint = _allocPoint;
        emit SetPool(_pid, _allocPoint);
    }
```

Listing 3.4: Revised MasterChefV2::set()

Status This issue has been confirmed by the team. And the team clarifies that, the `massUpdatePools()` may fail due to running out of gas, because the pool number can not be predicted. Considering this, they decide to set the `_withUpdate` to true if the pool number is not very huge. And if the pool number become very big, they will timely update specific pool by a script. By doing this, they try to ensure the fairness of the cake distribution.

3.3 Duplicate Pool Detection And Prevention

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `MasterChefV2`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The `MasterChefV2` protocol provides incentive mechanisms that reward the staking of supported assets with certain reward tokens. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. Each pool has its `allocPoint*100%/totalAllocPoint` share of scheduled rewards and the rewards for stakers are proportional to their share of `LP tokens` in the pool.

In current implementation, there are a number of concurrent pools that share the rewarded tokens and more can be scheduled for addition (via a privileged function). To accommodate these new pools, the design has the necessary mechanism in place that allows for dynamic additions of new staking pools that can participate in being incentivized as well.

The addition of a new pool is implemented in `add()`, whose code logic is shown below. It turns out that it did not perform necessary sanity checks in preventing a new pool with a duplicate token from being added. Though it is a privileged interface (protected with the modifier `onlyOwner`), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong pool introduction from human omissions.

```
    /// @notice Add a new pool. Can only be called by the owner.
    /// DO NOT add the same LP token more than once. Rewards will be messed up if you do.
    /// @param _allocPoint Number of allocation points for the new pool.
    /// @param _lpToken Address of the LP BEP-20 token.
    /// @param _isRegular Whether the pool is regular or special. LP farms are always "regular". "Special" pools are
    /// @param _withUpdate Whether call "massUpdatePools" operation.
    /// only for CAKE distributions within PancakeSwap products.
    function add(
        uint256 _allocPoint,
        IBEP20 _lpToken,
        bool _isRegular,
        bool _withUpdate
    ) external onlyOwner {
        require(_lpToken.balanceOf(address(this)) >= 0, "None BEP20 tokens");
        // stake CAKE token will cause staked token and reward token mixed up,
        // may cause staked tokens withdraw as reward token,never do it.
        require(_lpToken != CAKE, "CAKE token can't be added to farm pools");

        if (_withUpdate) {
            massUpdatePools();
        }

        if (_isRegular) {
            totalRegularAllocPoint = totalRegularAllocPoint.add(_allocPoint);
        } else {
            totalSpecialAllocPoint = totalSpecialAllocPoint.add(_allocPoint);
        }
        lpToken.push(_lpToken);
```

```
        poolInfo.push(
            PoolInfo({
                allocPoint: _allocPoint,
                lastRewardBlock: block.number,
                accCakePerShare: 0,
                isRegular: _isRegular,
                totalBoostedShare: 0
            })
        );
        emit AddPool(lpToken.length.sub(1), _allocPoint, _lpToken, _isRegular);
    }
```

Listing 3.5: `MasterChefV2::add()`

Recommendation Detect whether the given pool for addition is a duplicate of an existing pool or not. The pool addition is only successful when there is no duplicate.

```
    function checkPoolDuplicate(IBEP20 _lpToken) private {
        uint256 length = lpToken.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            require(lpToken[pid] != _lpToken, "add: existing pool?");
        }
    }

    /// @notice Add a new pool. Can only be called by the owner.
    /// DO NOT add the same LP token more than once. Rewards will be messed up if you do.
    /// @param _allocPoint Number of allocation points for the new pool.
    /// @param _lpToken Address of the LP BEP-20 token.
    /// @param _isRegular Whether the pool is regular or special. LP farms are always "regular". "Special" pools are
    /// @param _withUpdate Whether call "massUpdatePools" operation.
    /// only for CAKE distributions within PancakeSwap products.
    function add(
        uint256 _allocPoint,
        IBEP20 _lpToken,
        bool _isRegular,
        bool _withUpdate
    ) external onlyOwner {
        require(_lpToken.balanceOf(address(this)) >= 0, "None BEP20 tokens");
        checkPoolDuplicate(_lpToken);
        // stake CAKE token will cause staked token and reward token mixed up,
        // may cause staked tokens withdraw as reward token,never do it.
        require(_lpToken != CAKE, "CAKE token can't be added to farm pools");

        if (_withUpdate) {
            massUpdatePools();
        }

        if (_isRegular) {
            totalRegularAllocPoint = totalRegularAllocPoint.add(_allocPoint);
```

```
        } else {
            totalSpecialAllocPoint = totalSpecialAllocPoint.add(_allocPoint);
        }
        lpToken.push(_lpToken);

        poolInfo.push(
            PoolInfo({
                allocPoint: _allocPoint,
                lastRewardBlock: block.number,
                accCakePerShare: 0,
                isRegular: _isRegular,
                totalBoostedShare: 0
            })
        );
        emit AddPool(lpToken.length.sub(1), _allocPoint, _lpToken, _isRegular);
    }
```

Listing 3.6: Revised `MasterChefV2::add()`

We point out that if a new pool with a duplicate LP token can be added, it will likely cause a havoc in the distribution of rewards to the pools and the stakers.

Status This issue has been confirmed by the team. They decide to reserve the duplicate LP token pools possibility. They will count the tokens number in different pools separately, so that the pool's cake distribution will not be affected by other pools.

3.4 Trust Issue Of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `MasterChefV2`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In `MasterChefV2` contract, there are privileged accounts (including `owner` and `boostContract`) 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`. Specifically, it has the authority to add/set the reward pool, update the `CAKE` rates, update the `burnAdmin` (line 472) to which the burned `CAKE` rewards will be sent, update the whitelist for the special pools, and update the `boostContract`. Moreover, the `boostContract` has the authority to set user's boost factor (lines 460 and 461) which will change the rewards share of the user in the reward pool.

```
    /// @notice Update the % of CAKE distributions for burn, regular pools and special
        pools.
    /// @param _burnRate The % of CAKE to burn each block.
    /// @param _regularFarmRate The % of CAKE to regular pools each block.
    /// @param _specialFarmRate The % of CAKE to special pools each block.
    /// @param _withUpdate Whether call "massUpdatePools" operation.
    function updateCakeRate(
        uint256 _burnRate,
        uint256 _regularFarmRate,
        uint256 _specialFarmRate,
        bool _withUpdate
    ) external onlyOwner {
        require(
            _burnRate > 0 && _regularFarmRate > 0 && _specialFarmRate > 0,
            "MasterChefV2: Cake rate must be greater than 0"
        );
        require(
            _burnRate.add(_regularFarmRate).add(_specialFarmRate) ==
                CAKE_RATE_TOTAL_PRECISION,
            "MasterChefV2: Total rate must be 1e12"
        );
        if (_withUpdate) {
            massUpdatePools();
        }
        // burn cake base on old burn cake rate
        burnCake(false);

        cakeRateToBurn = _burnRate;
        cakeRateToRegularFarm = _regularFarmRate;
        cakeRateToSpecialFarm = _specialFarmRate;

        emit UpdateCakeRate(_burnRate, _regularFarmRate, _specialFarmRate);
    }

    // @notice Update burn admin address.
    /// @param _newAdmin The new burn admin address.
    function updateBurnAdmin(address _newAdmin) external onlyOwner {
        require(_newAdmin != address(0), "MasterChefV2: Burn admin address must be valid
            ");
        require(_newAdmin != burnAdmin, "MasterChefV2: Burn admin address is the same
            with current address");
        address _oldAdmin = burnAdmin;
        burnAdmin = _newAdmin;
        emit UpdateBurnAdmin(_oldAdmin, _newAdmin);
    }
```

Listing 3.7: MasterChefV2::updateCakeRate()/updateBurnAdmin()

It would be worrisome if the `owner` or the `boostContract` is plain EOA account. 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 Promptly transfer the `owner` and the `boostContract` 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 admin key security.

4 Conclusion

In this audit, we have analyzed the `PancakeSwap` `MasterChefV2` protocol design and implementation. The protocol is designed to allow users to earn `CAKE` rewards while supporting `PancakeSwap` by staking incentivized LP tokens. 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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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

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