Security Audit Report for MasterChefV2 and CakePool of PancakeSwap

Date: April 18, 2022

Version: 1.0

Contact: contact@blocksecteam.com

Contents

Report Manifest

Item	Description
Client	Pancake
Target	MasterChefV2 and CakePool of PancakeSwap

Version History

Version	Date	Description
1.0	April 15, 2022	First Release

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high-impact security incidents. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The smart contracts that are audited in this report include the following ones.

Smart Contract	Github URL
MasterChefV2	`https://github.com/chefcooper/pancake-contracts/blob/dev/MasterChefV2/projects/masterchef/v2/contracts/MasterChefV2.sol`
CakePool	`https://github.com/ChefSnoopy/pancake-contracts/blob/master/projects/cake-pool/contracts/CakePool.sol`

The two smart contracts aim to update the existing *MasterChefV1* contract with the following new features:

- There have two different types of pools: regular pool and special pool. The special one may receive more rewards than the regular one, and it can be used by only permitted accounts.
- The project can increase the rewards of specified users (in specified pools) by changing the `user.userBoostMultiplier`.
- The CAKE pool is implemented with single smart contract: *CakePool* that implements more complicated staking rules. For example, it divides staking into demand-deposit and lock with different interest rate. Furthermore, it charges different fees (i.e., overdue fee, performance fee, and withdraw fee) for users in different situation.

The auditing process is iterative. Specifically, we will audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following. Our audit report is responsible for the only initial version (`Version 1`), as well as new codes (in the following versions) to fix issues in the audit report.

Smart Contract	Commit SHA
`Version 1`	
MasterChefV2	`14d5943681a9ffdd5493e4166183a2f8de093dad`
CakePool	`1c629bff63c356a2b96a64de9bf497ce6caaa5fd`
`Version 2`	
CakePool	`3eeff3954e5fdf5f7d1a2ab6f915cd7ec6dafbcb`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Access control
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [1] and Common Weakness Enumeration [2]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered issue will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The issue has been received by the client, but not confirmed yet.
- **Confirmed** The issue has been recognized by the client, but not fixed yet.

[1] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[2] https://cwe.mitre.org/

Table 1.1: Vulnerability Severity Classification

Impact		Likelihood: *High*	Likelihood: *Low*
	High	High	Medium
	Low	Medium	Low

- **Fixed** The issue has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we find zero potential issue in the smart contracts. We also have five recommendations, as follows:

- High Risk: 0
- Medium Risk: 0
- Low Risk: 0
- Recommendations: 5

ID	Severity	Description	Category	Status
1	-	*Add a method to update the variable* `overdueFee`	Recommendation	Fixed
2	-	*Add the* `whenNotPaused` *modifier to withdrawal-related functions*	Recommendation	Fixed
3	-	*Address the concern of centralization risk*	Recommendation	Confirmed
4	-	*Save gas*	Recommendation	Confirmed
5	-	*Add check to prevent duplicated pools*	Recommendation	Confirmed

The details are provided in the following sections.

2.1 Additional Recommendation

2.1.1 Add a method to update the variable `overdueFee`

Status Fixed in `Version 2`.

Introduced by `Version 1`.

Description The *CakePool* contract lacks a method to update the variable `overdueFee`.

Impact The overdue fee rate in the *CakePool* contract can not be updated.

Suggestion Add a method to update the variable `overdueFee`.

2.1.2 Add the `whenNotPaused` modifier to withdrawal-related functions

Status Fixed in `Version 2`.

Introduced by `Version 1`.

Description The *CakePool* contract uses the `whenNotPaused` modifier for two functions (i.e., `unlock` and `deposit`) which prevents users from depositing CAKE into the paused *CakePool*. However, we believe that the `pause` function is usually invoked in an emergency, under which it does not make sense to invoke withdrawal-related functions to withdraw CAKE.

Impact The `whenNotPaused` modifier can not prevent attackers withdrawing money from a paused *CakePool*.

Suggestion Add the `whenNotPaused` modifier to withdrawal-related functions: `withdrawByAmount`, `withdraw`.

2.1.3 Address the concern of centralization risk

Status Confirmed.

Introduced by `Version 1`

Description There are a few critical functions that can be invoked by only authorized accounts, such as `init`, `add`, `set`, `setOperator`, `setboostContract`, and `setFreeFreeUser`.

Impact If the private keys of the authorized EOAs are compromised, the money in the project is at risk.

Suggestion Please make sure that you assign the *TimeLock* contract [1] (DAO contract) of PancakeSwap as the authorized accounts.

Feedback from the project Yes, we really use time lock multisig address as the contract owner, and use Gnosis safe APP to control admin.

2.1.4 Save gas

Status Confirmed

Introduced by `Version 1`

Description The following code is gas wasting.

```
210 function set(
211     uint256 _pid,
212     uint256 _allocPoint,
213     bool _withUpdate
214 ) external onlyOwner {
215     // No matter _withUpdate is true or false, we need to execute updatePool once before set
            the pool parameters.
216     updatePool(_pid);
217
218     if (_withUpdate) {
219         massUpdatePools();
220     }
221     ......
```

Listing 2.1: MasterChefV2.sol

Impact NA.

Suggestion Use the following code instead to save gas.

```
function set(
    uint256 _pid,
    uint256 _allocPoint,
    bool _withUpdate
) external onlyOwner {
    require("poolInfo[_pid].allocPoint != _allocPoint", "XXX");
    if (_withUpdate) {
        massUpdatePools();
    } else {
        updatePool(_pid);
    }
```

[1] https://bscscan.com/address/0xa1f482dc58145ba2210bc21878ca34000e2e8fe4

Feedback from the project About "require("poolInfo[_pid].allocPoint != _allocPoint", "XXX");", it is good suggestion, but it's not critical, we will deploy the contract soon, so we don't want to add it.

About '_withUpdate' option, your suggesion is if(xx)massUpdatePools() else updatePool(_pid) actually massUpdatePools() exectution logic will ignore alloc point is 0 pool, if we don't manual call updatePool(), this will cause the cake distribute not accurate, in our test case, we have met the issues, that's why we insist keep 'updatePool' always exectue, may be a little more gas cost, but it's a onlyowner function, it's not a big problems.

2.1.5 Add check to prevent duplicated pools

Status Confirmed.

Introduced by `Version 1`

Description The `add` function in *MasterChefV2* adds a new pool that supports a new LP token. We notice that it does not check duplicated LP tokens.

Impact Different pools can use the same LP token, which may violates the original design.

Suggestion Use the following code to check duplicated pools.

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

Feedback from the project Actually, we noticed that, we aim to keep add duplicate pool 'feature'