SMART CONTRACT AUDIT REPORT

For

PANCAKE FOLIO

Prepared By: Shuxiao Wang Hangzhou, China

Sep. 11, 2020

Document Properties

Client	Pancake Folio
Title	Smart Contract Audit Report
Target	PCF Staking
Version	1.0
Author	Chiachih Wu
Auditors	Chiachih Wu, Huaguo Shi
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	Sep. 11, 2020	Chiachih Wu	Final Release
1.0-rc2	Sep. 5, 2020	Chiachih Wu	Release Candidate #2
1.0-rc1	Sep. 4, 2020	Chiachih Wu	Release Candidate #1

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the source code of **PCF Staking** smart contract, we in the report outline 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. This document outlines our audit results.

1.1 About PCF Staking

Pancake Folio is designed as a suite of DeFi protocols to make crypto banking simple. It provides a full-suite of DeFi products, including automated yield farming tool, lending protocol, 1-click savings account, customized robo-advisor, and more. Liquidity reward program (including PCF Staking) marks the initial circulation of PCF, governance token of Pancake Folio, and the purpose is to raise awareness and gather momentum from the DeFi community prior to the launch of Pancake products. Good liquidity allows people to exchange with low slippage, essential for the launch of PCF token.

The basic information of PCF Staking is as follows:

Table 1.1: Basic Information of PCF Staking

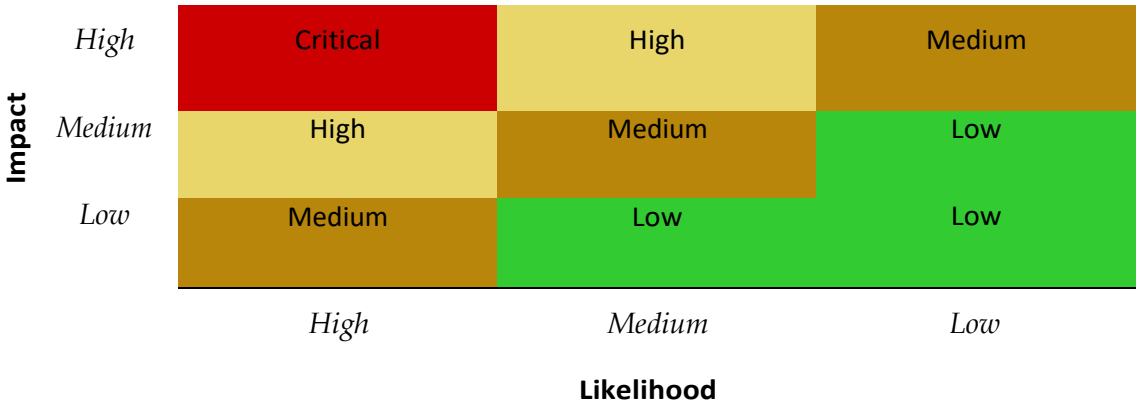
Item	Description
Issuer	Pancake Folio
Type	TRON Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	Sep. 11, 2020

In the following, we show the repository of reviewed code used in this audit.

1.2 About PCFShield

PCFShield Inc. [14] 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).

Table 1.2: Vulnerability Severity Classification



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

- 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 witha 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
	TRC20 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) [8], 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 audit does not give any warranties on finding all possible security issues of the given smart contract(s), 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 an 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 PCF Staking implementation. During the first phase of our audit, we studied the smart contract source code and ran 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	0	
Low	3	■■■
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 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 ofthem 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 3 low-severity vulnerabilities, and 1 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Info.	Improving Sanity Checks in claim()	Coding Practices	Confirmed
PVE-002	Low	Suggested Adherence of Checks-Effects-Interactions	Time and State	Confirmed
PVE-003	Low	Improved Address Validation in _transferOwnership()	Coding Practices	Confirmed
PVE-004	Low	Business Logic Error in the Interactions with Voting Escrow	Business Logics	Confirmed

Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improving Sanity Checks in claim()

- ID: PVE-001
- Severity: Informational
- Likelihood: Low
- Impact: N/A
- Target: PCF Staking_sol
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [2]

Description

In the PCF Staking contract, the claim() function allows users to claim their rewards based onthe amount and time they have staked. When a user has not staked or has already unstaked allyCRV tokens, the claim() function still performs all the calculation with zero reward transfers and Claimed event emitted. If we could check the totalStakedFor(msg_sender) in the beginning of claim(), non-staked users could save some gas with no Transfer() and Claim() events generated. Same issue applies to claimAndUnstake().

Moreover, we noticed that the claim() function tends to avoid the contract accounts using Address_isContract(msg_sender) in line 126. However, the Address_isContract() uses extcodesize opcode to get the code size of msg_sender, which has no effect if the msg_sender calls claim() from its constructor. Similar problems are found in claimAndUnstake() and stake().

```
function claim() external {
    require(!Address.isContract(msg.sender), "No harvest thanks");
    // cumulate user and global time*amount
    _updateTotalStaking(0);
    _updateUserStaking(0, msg.sender);

    _poolUnlock();
```

```
        curve.safeTransfer(msg.sender, curveReward);

        emit Claimed(msg.sender, bellaReward);
    }
```

Listing 3.1: PCF Staking.sol

Recommendation Ensure the msg_sender has staked or been staked for by totalStakedFor().

Get rid of contract accounts calling from the constructor by ensuring tx_origin == msg_sender

```
    function claim() external {

        require(tx.origin == msg.sender && !Address.isContract(msg.sender), "No harvest
            thanks");

        require( totalStakedFor(msg.sender) > 0, "nothing to claim");

        // cumulate user and global time*amount

        _updateTotalStaking(0);

        _updateUserStaking(0, msg.sender);


        _poolUnlock();


        (uint256 bellaReward, uint256 curveReward) = _calculateRewardAndBurnAll(msg.
```

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: PCF Staking_sol
- Category: Time and State [7]
- 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 TRON history, including the DAO [17] exploit, and the recent Uniswap/Lendf.Me hack [15]. We notice there are several occasions the checks-effects-interactions principle is violated. Forexample, the invest() function (see the code snippet below) is provided to externally call a tokencontract 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 (line 331) starts before effecting the update on internal states (lines 333), hence violating the principle. In this particular case, if the external contract has some hidden logic that may be capable of launching re-entrancy via the very same invest() function.

```
322     function invest() public {
323
324         uint256 totalBalance = totalInvested.add(yCurve.balanceOf(address(this)));
325         uint256 investableBalance = totalBalance.mul(DEPOSIT_BUFFER).div(ONE);
326
327         uint256 furtherDeposit = investableBalance.sub(totalInvested);
328
329         yCurve.approve(ycrv_gauge, 0);
330         yCurve.approve(ycrv_gauge, furtherDeposit);
331         ICrvDeposit(ycrv_gauge).deposit(furtherDeposit);
332
333         totalInvested = totalInvested.add(furtherDeposit);
334
335     }
```

Listing 3.3: PCF Staking.sol

Specifically, each invest() call moves 90% of the PCF Curves balance into ycrv_guage. Let's say we have 100 PCFCurves in the PCF Staking contract in the beginning. The first invest() call moves 90 PCFCurves out. When someone invokes the invest() function again right after the first invest(), furtherDeposit would be 0 since investableBalance and totalInvested are both 90. However, if the deposit() call is somehow re-entrying invest(), totalInvested is not updated yet (=0). Therefore, furtherDeposit would be 9 and another 9 yCurves would be deposit()'ed. The bad actor could do it again and again until all PCFCurve in the PCF Staking contract are gone.

Recommendation Apply necessary reentrancy prevention by following the checks-effects-interactions best practice. The above three functions can be revised as follows:

```
322     function invest() public {
323
324         uint256 totalBalance = totalInvested.add(yCurve.balanceOf(address(this)));
```

Listing 3.4: PCF Staking.sol

3.3 Improved Address Validation in _transferOwnership()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: Ownable_sol
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [2]

Description

The _transferOwnership() function in Ownable contract allows the current admin to transfer her privileges to another address. However, inside _transferOwnership(), the newOwner is directly stored into the storage, _owner, after validating the newOwner is a non-zero address, which may not be enough.

```
function _transferOwnership(address newOwner) internal {
    require(newOwner != address(0), "Owner should not be 0 address");
    emit OwnershipTransferred(_owner, newOwner);
    _owner = newOwner;
}
```

Listing 3.5: Ownable.sol

As shown in the above code snippets, newOwner is only validated against the zero address in line 70. However, if current admin enters a wrong address by mistake, she would never be able to take the management permissions back. Besides, if the newOwner is the same as the current admin address stored in _owner, it's a waste of gas.

Recommendation It would be much safer that if the transition is managed by implementing a two-step approach: _transferOwnership() and _updateOwnership(). Specifically, the _transferOwnership () function keeps the new address in the storage, _newOwner, instead of modifying the _owner() directly. The updateOwnership() function checks whether _newOwner is msg.sender, which means _newOwner signs the transaction and verifies herself as the new owner. After that, _newOwner could be set into _owner.

```
function _transferOwnership(address newOwner) internal {

    require(newOwner != address(0), "Owner should not be 0 address");

    require(newOwner != _owner, "The current and new owner cannot be the same");

    require(newOwner != _newOwner, "Cannot set the candidate owner to the same
        address");

    _newOwner = newOwner;

}


function _updateOwnership() public {
```

Listing 3.6: Ownable.sol

3.4 Business Logic Error in the Interactions with Voting Escrow

- ID: PVE-004

- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: PCF Staking_sol
- Category: Business Logics [6]
- CWE subcategory: CWE-841 [4]

Description

In PCF Staking, some functions are implemented for boosting the CRV rewards. For example, the lock_crv() function allows the owner to lock amount of CRV balance withheld by the PCF Staking contract until unlockTime to gain more CRV rewards. However, the create_lock() call to the voting_escrow (line 407) contract is likely to be reverted due to the fact that voting_escrow tends to prevent contract accounts from calling the function unless the caller contract address is in its whitelist.

```
    function lock_crv(uint256 amount, uint256 unlockTime) external onlyOwner {
        curve.approve(voting_escrow, 0);

        curve.approve(voting_escrow, amount);                          I
        VotingEscrow(voting_escrow).create_lock(amount, unlockTime);
```

Listing 3.7: PCF Staking.sol

Specifically, as shown in the code snippets below, voting_escrow checks msg_sender with the assert_not_contract() in the beginning of create_lock() (line 418).

```
def create_lock(_value: uint256, _unlock_time: uint256):
```

```
    """

    @notice Deposit `_value` tokens for `msg.sender` and lock until `_unlock_time`
    @param _value Amount to deposit

    @param _unlock_time Epoch time when tokens unlock, rounded down to whole weeks
    """

    self.assert_not_contract(msg.sender)

    """

    @notice Check if the call is from a whitelisted smart contract, revert if not
    @param addr Address to be checked

    """

    if addr != tx.origin:

        checker: address = self.smart_wallet_checker
```

Listing 3.9: voting_escrow

Based on that, lock_crv() always reverts until the PCF Staking contract is whitelisted by voting_escrow. Same theory applies to increase_crv_amount() and increase_crv_unlock_time().

Recommendation Try to be whitelisted or remove those functions since they have no effect.

3.5 Other Suggestions

We strongly suggest not to use experimental Solidity features (e.g., pragma experimental ABIEncoderV2) or third-party unaudited libraries. If necessary, refactor current code base to only use stable features or trusted libraries.

Last but not least, 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 the mainnet.

4 | Conclusion

In this audit, we thoroughly analyzed the PCF Staking implementation. During the audit, we noticed that the current code base is well organized and those identified issues are promptly confirmed and fixed.

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.

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found

- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [10, 11, 12, 13, 16].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [18] is an issue when code can call back into your contract and change state, such as withdrawing TRONs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks TRON indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected revert.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External Call

- Description: Whether the contract has any external call without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless Send

- Description: Whether the contract is vulnerable to gasless send.
- Result: Not found
- Severity: Medium

5.1.12 Send Instead Of Transfer

- Description: Whether the contract uses send instead of transfer.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to Out-Of-Gas exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated tx.origin to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of byte[], as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword var to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version $0.4.23$) enforces strict ABI length checks for return data from calls() [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing transfer() of ERC20 tokens).
- Result: Not found
- Severity: Low