SMART CONTRACT AUDIT REPORT

for

PancakeBunny Prediction

Prepared By: Yiqun Chen

Hangzhou, China
August 30, 2021

Document Properties

Client	PancakeBunny
Title	Smart Contract Audit Report
Target	PancakeBunny Prediction
Version	1.0
Author	Shulin Bie
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 30, 2021	Shulin Bie	Final Release
1.0-rc	August 26, 2021	Shulin Bie	Release Candidate

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction **4**

- 1.1 About PancakeBunny Prediction 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 Redundant State/Code Removal 11
- 3.2 Suggested oracleLatestRoundId Reset In setOracle() 12
- 3.3 Trust Issue Of Admin Keys 14
- 3.4 Potential Overflow/Underflow In PricePrediction Implementation 15

4 Conclusion **17**

References **18**

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the PancakeBunny Prediction 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 contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About PancakeBunny Prediction

PancakeBunny is a yield aggregator on Binance Smart Chain (BSC), which allows users to earn higher interest on their underlying crypto assets through its advanced yield optimization strategies. The PancakeBunny Prediction protocol is one of the core functions of PancakeBunny, which is designed as a decentralized BNB price prediction platform. It allows the user to profit from the BNB price rises and falls. The PancakeBunny Prediction protocol enriches the PancakeBunny ecosystem and also presents a unique contribution to current DeFi ecosystem.

The basic information of PancakeBunny Prediction is as follows:

Table 1.1: Basic Information of PancakeBunny Prediction

Item	Description
Target	PancakeBunny Prediction
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 30, 2021

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

- https://github.com/bunnyprediction/prediction-service/tree/develop/prediction-onchain (77fa56c)

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

- https://github.com/bunnyprediction/prediction-service/tree/master/prediction-onchain (ee3a8d8)

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 \ 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 [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 `PancakeBunny Prediction` 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 logic, 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	
Undetermined	0	
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 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 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key PancakeBunny Prediction Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Redundant State/Code Removal	Coding Practices	Fixed
PVE-002	Low	Suggested oracleLatestRoundId Reset In setOracle()	Coding Practices	Fixed
PVE-003	Low	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-004	Low	Potential Overflow/Underflow In PricePrediction Implementation	Numeric Errors	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, 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 should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Redundant State/Code Removal

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: PricePrediction
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [3]

Description

In the PancakeBunny Prediction protocol, the genesisLockRound() function is designed to lock the genesis prediction round. After the prediction round is locked, the transactions for the prediction round will be denied. According to the current design, the prediction round should be locked between the start lock block specified by the lockBlock and the end lock block specified by the lockBlock plus the bufferBlocks. Once the prediction round is not locked in the given period of time, the prediction round will never be locked. While examining the logic of the genesisLockRound() function, we notice there are some redundant code that can be safely removed.

To elaborate, we show below the related code snippet of the PricePrediction contract. In the genesisLockRound() function, the require(block.number <= rounds[currentEpoch].lockBlock.add(bufferBlocks), "Can only lock round within bufferBlocks")") is called (line 233 - line 236) to ensure the prediction round can only be locked in the given period of time. It comes to our attention that there is the same protection logic in the _safeLockRound() function, which will be subsequently called (line 239) in the genesisLockRound() function. We suggest to remove the redundant protection (line 233 - line 236) in the genesisLockRound() function.

```
230     function genesisLockRound() external onlyOperator whenNotPaused {
231         require(genesisStartOnce, "Can only run after genesisStartRound is triggered");
232         require(!genesisLockOnce, "Can only run genesisLockRound once");
233         require(
234             block.number <= rounds[currentEpoch].lockBlock.add(bufferBlocks),
235             "Can only lock round within bufferBlocks"
```

```
        );

        int256 currentPrice = _getPriceFromOracle();
        _safeLockRound(currentEpoch, currentPrice);

        currentEpoch = currentEpoch + 1;
        _startRound(currentEpoch);
        genesisLockOnce = true;
    }

    ...

    function _safeLockRound(uint256 epoch, int256 price) internal {
        require(rounds[epoch].startBlock != 0, "Can only lock round after round has
            started");
        require(block.number >= rounds[epoch].lockBlock, "Can only lock round after
            lockBlock");
        require(block.number <= rounds[epoch].lockBlock.add(bufferBlocks), "Can only
            lock round within bufferBlocks");
        _lockRound(epoch, price);
    }
```

Listing 3.1: `PricePrediction::genesisLockRound()&&_safeLockRound()`

Recommendation Consider the removal of the redundant code.

Status The issue has been addressed by the following commit: `8fa7f8d`.

3.2 Suggested oracleLatestRoundId Reset In setOracle()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PricePrediction`
- Category: Coding Practices [5]
- CWE subcategory: CWE-287 [2]

Description

In the `PancakeBunny Prediction` protocol, the `oracle` storage variable points to the price oracle that provides the price for the protocol. The `oracleLatestRoundId` storage variable stores the oracle update round identity that the protocol retrieved from the `oracle` last time. The `setOracle()` function is designed to update the `oracle`. While examining the related logic of the `oracle`, we observe an improper logic that can be improved.

To elaborate, we show below the related code snippet of the `PricePrediction` contract. In the internal `_getPriceFromOracle()` function used by other functions of the contract to get the current price, the oracle query, i.e., `(uint80 roundId, int256 price, , uint256 timestamp, )= oracle.`

`latestRoundData()` (line 582), is called to retrieve the current `roundId` and `price`. The requirement on `require(roundId > oracleLatestRoundId, "Oracle update roundId must be larger than oracleLatestRoundId")` is called (line 584) to ensure the current oracle update round identity is larger than last time. In other words, the current price is newer than last time. After that, the `roundId` is assigned to the `oracleLatestRoundId` storage variable (line 585). Based on the above analysis, if we assume that we update the `oracle` with the `setOracle()` function and the update round identity of the new `oracle` is less than the `oracleLatestRoundId`, the `_getPriceFromOracle()` function will be failed and the transactions will be reverted. We suggest to reset the `oracleLatestRoundId` storage variable in the `setOracle()` function.

```
    function setOracle(address _oracle) external onlyAdmin {
        require(_oracle != address(0), "Cannot be zero address");
        oracle = AggregatorV3Interface(_oracle);
    }
```

Listing 3.2: `PricePrediction::setOracle()`

```
580     function _getPriceFromOracle() internal returns (int256) {
581         uint256 leastAllowedTimestamp = block.timestamp.add(oracleUpdateAllowance);
582         (uint80 roundId, int256 price, , uint256 timestamp, ) = oracle.latestRoundData()
            ;
583         require(timestamp <= leastAllowedTimestamp, "Oracle update exceeded max
                timestamp allowance");
584         require(roundId > oracleLatestRoundId, "Oracle update roundId must be larger
                than oracleLatestRoundId");
585         oracleLatestRoundId = uint256(roundId);
586         return price;
587     }
```

Listing 3.3: `PricePrediction::_getPriceFromOracle()`

Recommendation Reset the `oracleLatestRoundId` storage variable during updating the `oracle` with the `setOracle()` function.

Status The issue has been addressed by the following commit: `8fa7f8d`.

3.3 Trust Issue Of Admin Keys

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: PricePrediction
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the PancakeBunny Prediction protocol, there is a privileged account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    function setOracle(address _oracle) external onlyAdmin {
        require(_oracle != address(0), "Cannot be zero address");
        oracle = AggregatorV3Interface(_oracle);
    }

    ...

    /**
     * @dev set reward rate
     * callable by admin
     */
    function setRewardRate(uint256 _rewardRate) external onlyAdmin {
        require(_rewardRate <= TOTAL_RATE, "rewardRate cannot be more than 100%");
        rewardRate = _rewardRate;
        treasuryRate = TOTAL_RATE.sub(_rewardRate);

        emit RatesUpdated(currentEpoch, rewardRate, treasuryRate);
    }

    /**
     * @dev set treasury rate
     * callable by admin
     */
    function setTreasuryRate(uint256 _treasuryRate) external onlyAdmin {
        require(_treasuryRate <= TOTAL_RATE, "treasuryRate cannot be more than 100%");
        rewardRate = TOTAL_RATE.sub(_treasuryRate);
        treasuryRate = _treasuryRate;

        emit RatesUpdated(currentEpoch, rewardRate, treasuryRate);
    }
```

Listing 3.4: PricePrediction::setOracle()&&setRewardRate()&&setTreasuryRate()

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 `PancakeBunny Prediction` design.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, 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.

3.4 Potential Overflow/Underflow In PricePrediction Implementation

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PricePrediction`
- Category: Numeric Errors [6]
- CWE subcategory: CWE-190 [1]

Description

`SafeMath` is a Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, we find that it is not widely used in `PricePrediction` contract.

In particular, while examining the logic of the `PricePrediction` contract, we notice that there are several functions without the overflow/underflow protection. In the following, we use the `getUserRounds()` routine as an example. To elaborate, we show below the related code snippet of the `getUserRounds()` routine in the `PricePrediction` contract. The `getUserRounds()` function is designed to retrieve the prediction rounds that a user has participated in. In the `getUserRounds()` function, it comes to our attention that all the arithmetic operations (lines 367, 374, 377) do not use `SafeMath` library to prevent overflows or underflows, which may introduce unexpected behavior. We suggest to use `SafeMath` to avoid unexpected overflows or underflows.

```
359     function getUserRounds(
360         address user,
361         uint256 cursor,
362         uint256 size
363     ) external view returns (uint256[] memory, uint256) {
364         uint256 length = size;
```

```
        uint256 userRoundLength = userRounds[user].length;

        if (length > userRoundLength - cursor) {
            length = userRoundLength - cursor;
        }

        uint256[] memory values = new uint256[](length);

        for (uint256 i = 0; i < length; i++) {
            values[i] = userRounds[user][cursor + i];
        }

        return (values, cursor + length);
    }
```

Listing 3.5: `PricePrediction::getUserRounds()`

Note the `getUserRoundsInReverseOrder()` and `getRoundBetInfo()` routines in the `PricePrediction` contract can be similarly improved.

Recommendation Use `SafeMath` to avoid unexpected overflows or underflows.

Status The issue has been addressed by the following commit: `8fa7f8d`.

4 Conclusion

In this audit, we have analyzed the `PancakeBunny Prediction` design and implementation. `PancakeBunny` is a yield aggregator on `Binance Smart Chain` (`BSC`), which allows users to earn higher interest on their underlying crypto assets through its advanced yield optimization strategies. The `PancakeBunny Prediction` protocol is one of the core functions of `PancakeBunny`, which is designed as a decentralized BNB price prediction platform. It allows the user to profit from the BNB price rises and falls. The `PancakeBunny Prediction` protocol enriches the `PancakeBunny` ecosystem and also presents a unique contribution to current DeFi ecosystem. 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-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.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 CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.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.