SMART CONTRACT AUDIT REPORT

for

PancakeSwap V3

Prepared By: Xiaomi Huang

PeckShield

March 28, 2023

Document Properties

Client	PancakeSwap Finance
Title	Smart Contract Audit Report
Target	PancakeSwap V3
Version	1.0
Author	Stephen Bie
Auditors	Stephen Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 28, 2023	Stephen Bie	Final Release
1.0-rc	March 22, 2023	Stephen Bie	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

Given the opportunity to review the design document and related smart contract source code of the PancakeSwap V3 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 PancakeSwap

PancakeSwap is the leading decentralized exchange on BNB Smart Chain (previously BSC), with very high trading volumes in the market. The audited PancakeSwap V3 is designed as an evolutional improvement of Uniswap V3. PancakeSwap V3 uses Uniswap V3's core design, but extends with liquidity provider incentives, which allows the liquidity provider to farm their Pancake V3 Positions NFT-V1 to earn CAKE. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of PancakeSwap V3

Item	Description
Target	PancakeSwap V3
Website	https://pancakeswap.finance/
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	March 28, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that this audit only covers the projects/router, projects/v3-core, and projects/v3-periphery sub-directories.

- https://github.com/pancakeswap/pancake-v3.git (84fd2f9)

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

- https://github.com/pancakeswap/pancake-v3.git (a447743)

1.2 About PeckShield

PeckShield Inc. [10] 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 [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.

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) [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 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 V3` 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	1	🟦
Low	0	
Informational	3	🟦 🟦 🟦
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 1 medium-severity vulnerability and 3 informational recommendations.

Table 2.1: Key PancakeSwap V3 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Improper Event Information in PancakeV3FactoryOwner::setOwner()	Business Logic	Fixed
PVE-002	Informational	Redundant State/Code Removal	Coding Practices	Fixed
PVE-003	Informational	Immutable States If Only Set at Constructor()	Coding Practices	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

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 Improper Event Information in PancakeV3FactoryOwner::setOwner()

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: PancakeV3FactoryOwner
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

While examining the events that reflect the PancakeV3FactoryOwner dynamics, we notice there is an incorrect event information in the setOwner() routine. To elaborate, we show below the related code snippet of the contract. By design, the setOwner() routine is used to transfer ownership to another address if needed. Within the routine, the event OwnerChanged(address indexed oldOwner, address indexed newOwner) will be emitted to reflect the operation. According to the event definition, the first oldOwner parameter indicates the old owner and the second newOwner parameter represents the new owner. However, we notice both parameters are the new owner in the emitted event (line 28).

```
24      event OwnerChanged(address indexed oldOwner, address indexed newOwner);
25      function setOwner(address _owner) external onlyOwner {
26          owner = _owner;

28          emit OwnerChanged(owner, _owner);
```

```
    }
```

Listing 3.1: `PancakeV3FactoryOwner::setOwner()`

Recommendation Properly emit the above-mentioned event with accurate information to timely reflect state changes. This is very helpful for external analytics and reporting tools.

Status The issue has been addressed by the following commit: `a447743`.

3.2 Redundant State/Code Removal

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `PancakeV3Factory`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

In the `PancakeSwap V3` protocol, the `PancakeV3Factory` contract is one of the main entries. In particular, one entry routine, i.e., `createPool()`, is designed to create a new pool. While examining its logic, we observe the presence of unnecessary redundancies that can be safely removed.

To elaborate, we show below the related code snippet of the `PancakeV3Factory` contract. Inside the `createPool()` routine, the requirement of `require(tickSpacing != 0)` (line 57) is called to ensure that the user specified swap `fee` for the pool has been enabled by the privileged `owner`. It comes to our attention that there is the same sanity check inside the routine (line 59). We suggest to remove the redundant code safely (line 57).

```
48      function createPool(
49          address tokenA,
50          address tokenB,
51          uint24 fee
52      ) external override returns (address pool) {
53          require(tokenA != tokenB);
54          (address token0, address token1) = tokenA < tokenB ? (tokenA, tokenB) : (tokenB,
                tokenA);
55          require(token0 != address(0));
56          int24 tickSpacing = feeAmountTickSpacing[fee];
57          require(tickSpacing != 0);
58          TickSpacingExtraInfo memory info = feeAmountTickSpacingExtraInfo[fee];
59          require(tickSpacing != 0 && info.enabled, "fee is not available yet");
60          if (info.whitelistRequested) {
61              require(_whiteListAddresses[msg.sender], "user should be in the white list
                    for this fee tier");
62          }
```

```
        require(getPool[token0][token1][fee] == address(0));
        pool = IPancakeV3PoolDeployer(poolDeployer).deploy(address(this), token0, token1
            , fee, tickSpacing);
        getPool[token0][token1][fee] = pool;
        // populate mapping in the reverse direction, deliberate choice to avoid the
            cost of comparing addresses
        getPool[token1][token0][fee] = pool;
        emit PoolCreated(token0, token1, fee, tickSpacing, pool);
    }
```

Listing 3.2: PancakeV3Factory::createPool()

Recommendation Consider the removal of the redundant code.

Status The issue has been addressed by the following commit: a447743.

3.3 Immutable States If Only Set at Constructor()

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: PancakeV3Factory
- Category: Coding Practices [6]
- CWE subcategory: CWE-561 [2]

Description

Since version 0.6.5, Solidity introduces the feature of declaring a state as immutable. An immutable state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as immutable is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an immutable state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of immutable states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

While examining all the state variables defined in the PancakeSwap V3 protocol, we observe the poolDeployer storage variable defined in the PancakeV3Factory contract needs not to be updated dynamically. It can be declared as immutable for gas efficiency.

```
    contract PancakeV3Factory is IPancakeV3Factory {
        /// @inheritdoc IPancakeV3Factory
        address public override owner;

        /// @inheritdoc IPancakeV3Factory
        address public override poolDeployer;
        ...
    }
```

Listing 3.3: `PancakeV3Factory`

Recommendation Revisit the state variable definition and make good use of `immutable`/`constant` states.

Status The issue has been addressed by the following commit: `a447743`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

Description

In the `PancakeSwap V3` protocol, there is a privileged `owner` 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 setFactoryOwner(address _owner) external onlyOwner {
        factory.setOwner(_owner);
    }

    function enableFeeAmount(uint24 fee, int24 tickSpacing) external onlyOwner {
        factory.enableFeeAmount(fee, tickSpacing);
    }

    function setFeeProtocol(IPancakeV3Pool pool, uint8 feeProtocol0, uint8 feeProtocol1)
         external onlyOwner {
        pool.setFeeProtocol(feeProtocol0, feeProtocol1);
    }

    function collectProtocol(
        IPancakeV3Pool pool,
        address recipient,
        uint128 amount0Requested,
        uint128 amount1Requested
    ) external onlyOwner returns (uint128 amount0, uint128 amount1) {
        return pool.collectProtocol(recipient, amount0Requested, amount1Requested);
    }

    function setLmPool(address pool, address lmPool) external onlyOwnerOrLmPoolDeployer {
        IPancakeV3PoolWithLmPool(pool).setLmPool(IPancakeV3LmPool(lmPool));
    }
```

Listing 3.4: `PancakeV3FactoryOwner`

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 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 The issue has been confirmed by the team.

4 Conclusion

In this audit, we have analyzed the design and implementation of the PancakeSwap V3 protocol, which is designed as an evolutional improvement of Uniswap V3 - a major decentralized exchange (DEX) running on top of Ethereum blockchain. PancakeSwap V3 uses Uniswap V3's core design, but extends with liquidity provider incentives. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

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

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

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

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