SMART CONTRACT AUDIT REPORT

for

DackieSwap

Prepared By: Xiaomi Huang

PeckShield

August 28, 2023

Document Properties

Client	DackieSwap
Title	Smart Contract Audit Report
Target	DackieSwap
Version	1.0
Author	Xuxian Jiang
Auditors	Colin Zhong, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 28, 2023	Xuxian Jiang	Final Release
1.0-rc	August 26, 2023	Xuxian Jiang	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 DackieSwap 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 DackieSwap

DackieSwap provides the decentralized exchange on Base with the goal of having high trading volumes in the market. It is designed as an evolutional improvement of Pancake V3, by making use of the Uniswap V3's core design with additional extensions on liquidity provider incentives. The audited DackieSwap allows the liquidity provider to farm their DackieSwap Positions NFT to earn rewards. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of DackieSwap

Item	Description
Target	DackieSwap
Website	https://www.dackieswap.xyz/
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	August 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 following three directories: dackie-pad/, pools/, and v3-lm-pool/. The other directories v3-core/, v3-periphery/, masterchef-v3/, and router/ share the same logic with the `pancakes-v3` fork and are not part of this audit.

- https://github.com/DackieSwap/dackieswap-contracts-v3.git (e26c4db)

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

- https://github.com/DackieSwap/dackieswap-contracts-v3.git (0b1c8c8)

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.

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

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 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 `DackieSwap` 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	2	■ ■
Medium	2	■ ■
Low	1	■
Informational	0	
Total	5	

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 2 high-severity vulnerabilities, 2 medium-severity vulnerabilities, and 1 low-severity vulnerability.

Table 2.1: Key DackieSwap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Lack of Timelocked finalWithdraw() in DackiePadInitializableV5/V6	Business Logic	Resolved
PVE-002	High	Incorrect NFT Removal Logic in SmartChefNFT	Business Logic	Resolved
PVE-003	Low	Improved Ether Transfer With Necessary Reentrancy Guard	Code Practices	Resolved
PVE-004	Medium	Incorrect Liquidity Mining in DackieV3LmPool	Business Logic	Resolved
PVE-005	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 Lack of Timelocked finalWithdraw() in DackiePadInitializableV5/V6

- ID: PVE-001
- Severity: High
- Likelihood: Low
- Impact: High
- Target: DackiePadInitializableV5/V6
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The DackieSwap protocol features the IDO support with the deployable DackiePadInitializableV5 and DackiePadInitializableV6 contracts. In the process of examining the IDO support, we notice a potential issue of an admin function finalWithdraw() which needs to be better revisited.

To elaborate, we show below the implementation of this admin routine finalWithdraw(). While it is protected with the onlyOwner modifier, we notice this routine may be called to withdraw all funds from the IDO contracts. This capability may be concerning to the protocol users. A better approach is to add a timelock so that it may not be able to invoke until the vesting duration is over.

```
    function finalWithdraw(uint256 _lpAmount, uint256 _offerAmount) external override onlyOwner {
        require(_lpAmount <= raiseToken.balanceOf(address(this)), "Operations: Not enough LP tokens");
        require(_offerAmount <= offeringToken.balanceOf(address(this)), "Operations: Not enough offering tokens");

        if (_lpAmount > 0) {
            raiseToken.safeTransfer(msg.sender, _lpAmount);
        }

        if (_offerAmount > 0) {
            offeringToken.safeTransfer(msg.sender, _offerAmount);
        }
```

```
        emit AdminWithdraw(_lpAmount, _offerAmount);
    }
```

Listing 3.1: `DackiePadInitializableV5::finalWithdraw()`

Recommendation Revisit the above `finalWithdraw()` routine so that even the privileged admin may not withdraw the funds from the `IDO` contract at will.

Status The issue has been fixed by this commit: `0b1c8c8`.

3.2 Incorrect NFT Removal Logic in SmartChefNFT

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `SmartChefNFT`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `DackieSwap` protocol has a built-in `SmartChefNFT` contract that allows the liquidity provider to farm their `Pancake V3 NFT Positions` to earn rewards. In the process of examining the `NFT` addition/removal logic, we notice a potential issue that needs to be fixed.

To elaborate, we show below the implementation of the related `removeTokenId()` routine. This routine is used to remove a specific `tokenId` from a user. However, it comes to our attention that this routine may be abused to withdraw a `tokenId` that does not belong to the withdrawing user – even though it may have the cost of losing one of his `NFT` positions. In other words, the user may create a `NFT` position with tiny liquidity and steal another user's `NFT` position with larger liquidity.

```
    function removeTokenId(UserInfo storage _user, uint256 _tokenId) internal {
        uint256[] storage tokenIds = _user.tokenIds;
        uint256 indexToBeDeleted;

        for (uint256 i = 0; i < tokenIds.length; i++) {
            if (tokenIds[i] == _tokenId) {
                indexToBeDeleted = i;
                break;
            }
        }

        if (indexToBeDeleted < tokenIds.length - 1) {
            tokenIds[indexToBeDeleted] = tokenIds[tokenIds.length - 1];
        }
        // decrease array length, this will delete the last item
```

```
        tokenIds.pop();

        // Withdraw NFT from contract
        stakedToken.transferFrom(address(this), msg.sender, _tokenId);
        uint256 rarity = getRarity(_tokenId);
        _user.amount -= rarity * BASE_FACTOR;
        totalStake -= rarity * BASE_FACTOR;
    }
```

Listing 3.2: `SmartChefNFT::removeTokenId()`

Recommendation Revisit the above `removeTokenId()` routine to ensure the requested `tokenId` belongs to the withdrawing user.

Status The issue has been fixed by this commit: `697f98b`.

3.3 Improved Ether Transfer With Necessary Reentrancy Guard

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `DackieNFT`, `DackiEggNFT`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1109 [1]

Description

The `DackieSwap` swap has its `NFT` implementation which allows protocol users to purchase. The collected funds can be later withdrawn from the `NFT` contract. In fact, the funds can be retrieved by calling the `withdraw()` routine. While reviewing the implementation of this `withdraw()` routine, we notice that the `ETH` transfer may fail because of the possible `Out-of-Gas` issue.

To elaborate, we show below the code snippet of the `withdraw()` routine, which allows to transfer `ETH` to the contract owner. We notice that the `withdraw()` routine directly calls the native `transfer()` routine (line 126) to transfer `ETH`. However, the `transfer()` is not recommended to use any more since the `EIP-1884` may increase the gas cost and the 2300 gas limit may be exceeded. There is a helpful blog stop-using-soliditys-transfer-now that explains why the `transfer()` is not recommended any more.

As a result, the `transfer()` may revert and the `ETH` could be locked in the contract. Based on this, we suggest to use the low-level `call()` directly with value attached to transfer `ETH`.

```
124     function withdraw() public onlyOwner {
125         uint256 balance = address(this).balance;
126         payable(owner()).transfer(balance);
```

```
    }
```

Listing 3.3: `DackieNFT::withdraw()`

Recommendation Revisit the `withdraw()` routine to transfer `ETH` using `call()`. The same issue is also applicable to the `DackiEggNFT` contract.

Status The issue has been fixed by this commit: `697f98b`.

3.4 Incorrect Liquidity Mining in DackieV3LmPool

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `DackieV3LmPool`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `DackieSwap` protocol features a liquidity mining support with `Pancake V3 NFT`-like positions. In the process of examining the actual implementation, we notice a potential issue that may block a legitimate user from claiming the rewards.

To elaborate, we show below the implementation of the related `getRewardGrowthInside()` routine. As the name indicates, this routine is used to compute the reward growth data. However, it does not consider a possible underflow situation that may make the associated `MasterChef V3` smart contract unable to calculate reward for the positions whose initial `rewardGrowthInsideX128` values were negative underflow.

```
    function getRewardGrowthInside(
        mapping(int24 => LmTick.Info) storage self,
        int24 tickLower,
        int24 tickUpper,
        int24 tickCurrent,
        uint256 rewardGrowthGlobalX128
    ) internal view returns (uint256 rewardGrowthInsideX128) {
        Info storage lower = self[tickLower];
        Info storage upper = self[tickUpper];

        // calculate reward growth below
        uint256 rewardGrowthBelowX128;
        if (tickCurrent >= tickLower) {
            rewardGrowthBelowX128 = lower.rewardGrowthOutsideX128;
        } else {
            rewardGrowthBelowX128 = rewardGrowthGlobalX128 - lower.
                rewardGrowthOutsideX128;
```

```
        }

        // calculate reward growth above
        uint256 rewardGrowthAboveX128;
        if (tickCurrent < tickUpper) {
            rewardGrowthAboveX128 = upper.rewardGrowthOutsideX128;
        } else {
            rewardGrowthAboveX128 = rewardGrowthGlobalX128 - upper.
                rewardGrowthOutsideX128;
        }

        rewardGrowthInsideX128 = rewardGrowthGlobalX128 - rewardGrowthBelowX128 -
            rewardGrowthAboveX128;
    }
```

Listing 3.4: `LmTick::getRewardGrowthInside()`

Recommendation Revisit the above `getRewardGrowthInside()` routine to handle the possible underflow situation. Here comes a possible extension to check whether an underflow situation occurs:

```
    function _getRewardGrowthInsideInternal(
        int24 tickLower,
        int24 tickUpper
    ) internal view returns (uint256 rewardGrowthInsideX128, bool isNegative) {
        (, int24 tick, , , , , ) = pool.slot0();
        LmTick.Info memory lower = lmTicks[tickLower];

        LmTick.Info memory upper = lmTicks[tickUpper];

        // calculate reward growth below
        uint256 rewardGrowthBelowX128;
        if (tick >= tickLower) {
            rewardGrowthBelowX128 = lower.rewardGrowthOutsideX128;
        } else {
            rewardGrowthBelowX128 = rewardGrowthGlobalX128 - lower.
                rewardGrowthOutsideX128;
        }

        // calculate reward growth above
        uint256 rewardGrowthAboveX128;
        if (tick < tickUpper) {
            rewardGrowthAboveX128 = upper.rewardGrowthOutsideX128;
        } else {
            rewardGrowthAboveX128 = rewardGrowthGlobalX128 - upper.
                rewardGrowthOutsideX128;
        }

        rewardGrowthInsideX128 = rewardGrowthGlobalX128 - rewardGrowthBelowX128 -
            rewardGrowthAboveX128;
        isNegative = (rewardGrowthBelowX128 + rewardGrowthAboveX128) >
            rewardGrowthGlobalX128;
```

```
    }
```

Listing 3.5: Revised `_getRewardGrowthInsideInternal()`

Status The issue has been fixed by this commit: `697f98b`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `DackieSwap` 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 and managing the reward pools). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    function setEmergency(bool _emergency) external onlyOwner {
        emergency = _emergency;
        emit SetEmergency(emergency);
    }

    function setReceiver(address _receiver) external onlyOwner {
        if (_receiver == address(0)) revert ZeroAddress();
        if (CAKE.allowance(_receiver, address(this)) != type(uint256).max) revert();
        receiver = _receiver;
        emit NewReceiver(_receiver);
    }

    function setLMPoolDeployer(ILMPoolDeployer _LMPoolDeployer) external onlyOwner {
        if (address(_LMPoolDeployer) == address(0)) revert ZeroAddress();
        LMPoolDeployer = _LMPoolDeployer;
        emit NewLMPoolDeployerAddress(address(_LMPoolDeployer));
    }
```

Listing 3.6: Example Privileged Operations in `MasterChefV3`

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 `DackieSwap` protocol, which is designed as an evolutional improvement of `Uniswap V3` - a major decentralized exchange (`DEX`) running on top of `Base` blockchain. The protocol makes use of the `Uniswap V3`'s core design with additional extensions on liquidity provider incentives and allows the liquidity provider to farm their `DackieSwap Positions NFT` to earn rewards. 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.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: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.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.