



SMART CONTRACT AUDIT REPORT

for

Cakepie Protocol



Prepared By: Xiaomi Huang

PeckShield
December 13, 2023

Document Properties

Client	Cakepie
Title	Smart Contract Audit Report
Target	Cakepie Protocol
Version	1.0
Author	Xuxian Jiang
Auditors	Jianzuo Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 13, 2023	Xuxian Jiang	Final Release
1.0-rc1	December 1, 2023	Xuxian Jiang	Release Candidate #1

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	4
1.1	About Cakepie	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	Incorrect emergencyWithdraw() Logic in CakepieBribeRewardDistributor	11
3.2	Improved nonReentrant Protection in PancakeV3Helper	12
3.3	Lack of CKPRatio Initialization in RewardDistributor	13
3.4	Trust Issue of Admin Keys	14
3.5	Revisited Upgrade Logic in TransparentUpgradeableproxy	15
4	Conclusion	17
	References	18

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the `Cakepie` 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 Cakepie

As part of `Magpie XYZ`'s commitment to enhancing the yield in the DeFi ecosystem, `Cakepie` allows users to stake assets in different types of pools and convert their `Cake` on `Cakepie` with `CAKE` or locked `Cake` positions on `PancakeSwap`. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The `Cakepie` Protocol

Item	Description
Name	<code>Cakepie</code>
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 13, 2023

In the following, we show the Git repository of reviewed files and the commit hash values used in the audit. This audit covers all contracts in the given repository, excluding the following contracts: `MasterCakepie.sol`, `CakeRush.sol`, and `mCake.sol`.

- https://github.com/magpiexyz/cakepie_contract.git (3abfc8d)

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

- https://github.com/magpiexyz/cakepie_contract.git (8d82d24)

1.2 About PeckShield

PeckShield Inc. [11] 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	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- 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 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

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

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) [9], 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 `Cakepie` protocol 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 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	1	
Low	4	
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 1 medium-severity vulnerability and 4 low-severity vulnerabilities.

Table 2.1: Key Cakepie Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incorrect emergencyWithdraw() Logic in CakepieBribeRewardDistributor	Business Logic	Resolved
PVE-002	Low	Improved nonReentrant Protection in PancakeV3Helper	Time And State	Resolved
PVE-003	Low	Lack of CKPRatio Initialization in RewardDistributor	Coding Practices	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-005	Low	Revisited Upgrade Logic in TransparentUpgradeableproxy	Business Logic	Resolved

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Incorrect emergencyWithdraw() Logic in CakepieBribeRewardDistributor

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: CakepieBribeRewardDistributor
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The Cakepie protocol has a `CakepieBribeRewardDistributor` contract that is used for distributing rewards from voting. In the process of examining its emergency withdrawal logic, we notice the current implementation needs to be improved.

To elaborate, we show below the related code snippet from the related `emergencyWithdraw()` routine. As the name indicates, this routine is used to perform emergency withdrawal of the given asset from the contract. However, it mis-interprets the intention of retrieving the native coin with `bribeManager` (line 254). In other words, it should be replaced with `NATIVE`.

```

253     function emergencyWithdraw(address _token, address _receiver) external onlyOwner {
254         if (_token == bribeManager) {
255             address payable recipient = payable(_receiver);
256             recipient.transfer(address(this).balance);
257         } else {
258             IERC20(_token).safeTransfer(_receiver, IERC20(_token).balanceOf(address(this)));
259         }
260     }

```

Listing 3.1: `CakepieBribeRewardDistributor::emergencyWithdraw()`

Recommendation Revisit the above logic to use `NATIVE` as the intention to retrieve native coin in the contract.

Status The issue has been fixed by this commit: [df7b602](#).

3.2 Improved nonReentrant Protection in PancakeV3Helper

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: PancakeV3Helper
- Category: Time and State [\[8\]](#)
- 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 Ethereum history, including the DAO [\[13\]](#) exploit, and the Uniswap/Lendf.Me hack [\[12\]](#).

We notice there are occasions where the checks-effects-interactions principle is violated. Using the PancakeV3Helper as an example, the `withdrawNFT()` function (see the code snippet below) is provided to externally call a token contract to transfer NFT asset. However, the invocation of an external contract requires extra care in avoiding the above re-entrancy. For example, the interaction with the external contract (line 105) start before effecting the update on internal states, hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching re-entrancy via the same entry function.

```

101     function withdrawNFT(address _pool, uint256 _tokenId) external {
102         uint256[] memory tokenId = toArray(_tokenId);
103         if (!_isTokenOwner(msg.sender, tokenId)) revert TokenNotOwned();
104
105         IPancakeStaking(pancakeStaking).withdrawV3For(msg.sender, _pool, _tokenId);
106
107         delete userPositionInfos[_tokenId];
108         removeToken(msg.sender, _tokenId);
109
110         emit NewWithdraw(msg.sender, _pool, _tokenId);
111     }

```

Listing 3.2: PancakeV3Helper::withdrawNFT()

While the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for re-entrancy, it is important

to take precautions to thwart possible re-entrancy. Meanwhile, the ERC721 support may naturally have the built-in support for callbacks, which deserve the special attention to guard against possible re-entrancy.

In addition, the above routine can also benefit from an additional check to ensure the given pool is valid, i.e., `_checkForValidPool(msg.sender, _pool, _tokenId, false)`.

Recommendation Apply necessary reentrancy prevention by following the checks-effects-interactions principle and utilizing the necessary `nonReentrant` modifier to block possible re-entrancy. Also for consistency, we suggest to do the same for the `depositNFT()` routine.

Status The issue has been fixed by this commit: `df7b602`.

3.3 Lack of CKPRatio Initialization in RewardDistributor

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: RewardDistributor
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

The RewardDistributor contract has a key parameter `CKPRatio`. This parameter is used to emit Cakepie tokens for PancakeV3 pools. However, it comes to our attention that this `CKPRatio` parameter is never initialized.

In the following, we show the code snippet from the related routine, i.e., `_handleTransfer()`. This routine is used to send rewards to supported rewarders. This should only be called for rewards from Pancake for staked LPs (not revenue share nor reward for `VeCake`). With that, there is a need to properly initialize this `CKPRatio` parameter. Otherwise, there is no intended extra Cakepie reward.

```

385     function _handleTransfer(
386         address _pool,
387         address _finalDestination,
388         address _rewardToken,
389         uint256 _amount,
390         bool _isRewarder
391     ) internal {
392         // For V2 and AML pools
393         if (_isRewarder) {
394             IERC20(_rewardToken).safeIncreaseAllowance(_finalDestination, _amount);
395             IBaseRewardPool(_finalDestination).queueNewRewards(_amount, _rewardToken);
396         } else {
397             IERC20(_rewardToken).safeTransfer(_finalDestination, _amount);
398         }

```

```

399         // assuming only reward from V3 pool type does not go through rewarder
400         uint256 CKPrewardAmount = (_amount * CKPRatio) / DENOMINATOR;
401
402         if (cakepie != address(0)) {
403             IERC20(cakepie).safeTransfer(_finalDestination, CKPrewardAmount);
404
405             emit RewardPaidTo(_pool, _finalDestination, address(cakepie),
406                             CKPrewardAmount);
407         }
408     }
409 }

```

Listing 3.3: RewardDistributor::_handleTransfer()

Recommendation Properly initialize the CKPRatio parameter.

Status The issue has been fixed by this commit: 8efabdd.

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 [2]

Description

In the Cakepie protocol, there is a privileged account, i.e., owner, that plays a critical role in governing and regulating the protocol-wide operations (e.g., pause/unpause protocol, configure reward multipliers). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the PancakeStakingBaseUpg contract as an example and show the representative functions potentially affected by the privileges of the owner account.

```

457     function config(
458         address _masterChefV3,
459         address _nonfungiblePositionManager,
460         address _rewardDistributor,
461         address _pancakeV3Helper,
462         address _pancakeV2LPHelper,
463         address _pancakeAMLHelper
464     ) external onlyOwner {
465         masterChefV3 = IMasterChefV3(_masterChefV3);
466         nonfungiblePositionManager = INonfungiblePositionManager(
467             _nonfungiblePositionManager);
468         rewardDistributor = IRewardDistributor(_rewardDistributor);

```

```

468     pancakeV3Helper = _pancakeV3Helper;
469     pancakeV2LPHelper = _pancakeV2LPHelper;
470     pancakeAMLHelper = _pancakeAMLHelper;
471 }
472
473 function setVoteManager(address _newVoteManager) external onlyOwner {
474     address oldVoteManager = voteManager;
475     voteManager = _newVoteManager;
476
477     emit VoteManagerSet(oldVoteManager, _newVoteManager);
478 }
479
480 function setAllowedOperator(address _operator, bool _active) external onlyOwner {
481     allowedOperator[_operator] = _active;
482     emit AllowedOperatorSet(_operator, _active);
483 }

```

Listing 3.4: Example Privileged Operation in `PancakeStakingBaseUpg`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the `owner` may also be a counter-party risk to the protocol users. It is worrisome if the privileged `owner` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

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 mitigated as the team confirms the use of a multi-sig for all admin roles.

3.5 Revisited Upgrade Logic in `TransparentUpgradeableProxy`

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `TransparentUpgradeableProxy`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

To facilitate the protocol upgrade, `Cakepie` makes use of `TransparentUpgradeableProxy` to deploy actual back-end implementation behind the front-end proxy. Moreover, the front-end proxy has the added

support of a timelock to avoid sudden upgrade or mitigate potential admin risk. While examining the timelock support, we notice the implementation needs to be improved.

To elaborate, we show below the related code snippet from the `submitUpgrade()` routine. As the name indicates, this routine is used to submit the upgrade request. However, this request is never honored by validating the upgrade timelock `timelockEndForUpgrade` (line 38). As a result, the current implementation still allows for instant upgrade of the back-end implementation.

```
36     function submitUpgrade(address newImplementation) external ifAdmin {  
37         nextImplementation = newImplementation;  
38         timelockEndForUpgrade = block.timestamp + timelockLength;  
39     }
```

Listing 3.5: `TransparentUpgradeableProxy::submitUpgrade()`

Recommendation Revisit the above logic to make use of `timelockEndForUpgrade` to upgrade the contract implementation.

Status The issue has been resolved as the team confirms it is part of design.



4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Cakepie` protocol, which aims to support the long-term commitment to enhancing the yield in the DeFi ecosystem. Specifically, `Cakepie` allows users to stake assets in different types of pools and convert their `Cake` on `Cakepie` with `CAKE` or `locked Cake` positions on `PancakeSwap`. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based 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-663: Use of a Non-reentrant Function in a Concurrent Context. <https://cwe.mitre.org/data/definitions/663.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 CATEGORY: Concurrency. <https://cwe.mitre.org/data/definitions/557.html>.
- [9] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.

- [10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [11] PeckShield. PeckShield Inc. <https://www.peckshield.com>.
- [12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. <https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09>.
- [13] David Siegel. Understanding The DAO Attack. <https://www.coindesk.com/understanding-dao-hack-journalists>.

