Security Audit Report for Cakepie Contracts

Date: Dec 10, 2023

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2023.12.10 20:38:29 +08'00'

Contents

Report Manifest

Item	Description
Client	Cakepie
Target	Cakepie Contracts

Version History

Version	Date	Description
1.0	Dec 10, 2023	First Release

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 5 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the code repository of Cakepie Contracts[1] of Cakepie. Cakepie is a yield optimization protocol built upon PancakeSwap. It enables users to manage PancakeSwap V2/V3 positions and claim rewards. CAKE holders can convert their CAKE token or locked CAKE positions from PancakeSwap on Cakepie. In addition, Cakepie grants users governance rights. Users with voting powers can vote in Cakepie and votes will be cast to Pancake's GaugeVoting. Cakepie also incorporates a bribe market where users can add bribes that are distributed to active voters.

Please note that this audit only covers the following contracts:

- PancakeV3Helper.sol
- PancakeV2Helper.sol
- PancakeAMLHelper.sol
- PancakeStakingBNBChain.sol
- mCakeConvertorBNBChain.sol
- mCakeConvertorSideChain.sol
- PancakeStakingBaseUpg.sol
- mCakeConvertorBaseUpg.sol
- DustRefunder.sol
- Enumerable.sol
- CakepieReceiptToken.sol
- Cakepie.sol
- CakepieBribeManager.sol
- CakepieBribeDistributor.sol
- PancakeVoteManager.sol
- RewardDistributor.sol
- BaseRewardPoolV3.sol

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Cakepie Contracts	`Version 1`	`3abfc8d14d473eb53947963c0fcbdc4af2653eaa`
	`Version 2`	`da2b39eb120affdca2cc6b930efd6312bedea5a6`

[1] https://github.com/magpiexyz/cakepie_contract

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we find **six** potential issues. Besides, we also have **five** recommendations and **one** note.

- Medium Risk: 3
- Low Risk: 3
- Recommendation: 5
- Note: 1

ID	Severity	Description	Category	Status
1	Medium	Uninitialized state variables	Software Security	Fixed
2	Medium	Incorrect interface used for `CakepieBribeManager`	Software Security	Fixed
3	Medium	Potential DoS to native token transfer due to insufficient gas	Software Security	Fixed
4	Low	Potential inconsistent pool identifier	Software Security	Fixed
5	Low	Potential inconsistent token pairs	DeFi Security	Fixed
6	Low	Insufficient check on the Pancakeswap pool	DeFi Security	Fixed
7	-	Implement usages for unused `isActive` flags	Recommendation	Acknowledged
8	-	Add sanity checks before setting parameters	Recommendation	Fixed
9	-	Remove unused logic	Recommendation	Fixed
10	-	Fix typo in `CakepieBribeRewardDistributor`	Recommendation	Fixed
11	-	Remove unused `payable` attribute	Recommendation	Fixed
12	-	Centralization risk	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Uninitialized state variables

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the following contracts, there are several state variables that cannot be set through constructors or privileged functions:

- The `CKPRatio` variable of the `RewardDistributor` contract, which specifies the distribution ratio of Cakepie rewards.
- The `gaugeVoting` variable of the `PancakeStakingBNBChain` contract, which specifies the address for the `GaugeVoting` contract from PancakeSwap.
- The `voter` variable of the `PancakeVoteManager` contract, which specifies the address for the `GaugeVoting` contract from PancakeSwap.

Impact The Cakepie rewards cannot be distributed and the voting cannot be cast on the `GaugeVoting` contract.

Suggestion Configure the uninitialized state variables accordingly.

2.1.2 Incorrect interface used for `CakepieBribeManager`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `PancakeVoteManager` contract attempts to invoke the `exactCurrentEpoch` function in the `CakepieBribeManager` contract (Line 284). However, there is no such interface in the target `CakepieBribeManager` contract.

```
278    function _afterVoteUpdate(
279        address _user,
280        address _pool,
281        uint256 _pid,
282        int256 _weight
283    ) internal virtual {
284        uint256 epoch = ICakepieBribeManager(bribeManager).exactCurrentEpoch();
285        emit Voted(epoch, _user, _pool, _pid, _weight);
286    }
```

Listing 2.1: PancakeVoteManager.sol

Impact Users cannot vote due to the revert in the `_afterVoteUpdate` function.

Suggestion Revise the incorrect interface accordingly.

2.1.3 Potential DoS to native token transfer due to insufficient gas

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `_refundETH` function in the `DustRefunder` contract utilizes `send` to transfer native tokens. However, this can fail if the recipient is a proxy contract whose fallback function consumes significant gas, causing a DoS.

```
34    function _refundETH(address payable _dustTo, uint256 _refundAmt) internal {
35        if (_refundAmt > 0) {
36            bool success = _dustTo.send(_refundAmt);
37            require(success, "ETH transfer failed");
38        }
39    }
```

Listing 2.2: DustRefunder.sol

Impact Contract users using a proxy cannot increase liquidity due to the revert in the `_refundETH` function.

Suggestion Revise the code logic accordingly.

2.1.4 Potential inconsistent pool identifier

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `CakepieBribeManager` contract, a pool identifier consists of an `epoch` and a `pid`.

```
317    function _getPoolIdentifier(uint256 _epoch, uint256 _pid) internal pure returns (bytes32) {
318        return keccak256(abi.encodePacked(_epoch, _pid));
319    }
```

Listing 2.3: CakepieBribeManager.sol

However, this contract allows the admin to force reset a pool's `pid`, introducing potential data inconsistency issues. Specifically, the bribe information is indexed based on these pool identifiers. If a pool's `pid` is reset improperly, it can break this index and cause unexpected behavior.

```
392    function forceSetMarketPID(address _pool, uint256 _newPid) external onlyOwner {
393        poolToPid[_pool] = _newPid;
394    }
```

Listing 2.4: CakepieBribeManager.sol

Impact N/A

Suggestion Revise the code logic accordingly.

2.2 DeFi Security

2.2.1 Potential inconsistent token pairs

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `PancakeStakingBaseUpg` contract, the `increaseLiquidityV3For` function assumes that `token1` is the native token when a pool info's `isNative` field is set to true. However, for token pairs with the native token as `token0`, this assumption is violated. In such cases, the passed `token1` will no longer be the native token, causing the liquidity operation to fail.

The same issue also exists in the `DustRefunder` contract. It assumes only `_token1` can be the native token.

```
219    function increaseLiquidityV3For(
220        address _for,
221        address _v3Pool,
222        IMasterChefV3.IncreaseLiquidityParams calldata params
223    ) external payable nonReentrant whenNotPaused _onlyPoolHelper(pancakeV3Helper) {
224        Pool storage poolInfo = pools[_v3Pool];
225        address token0 = IPancakeV3PoolImmutables(poolInfo.poolAddress).token0();
226        address token1 = IPancakeV3PoolImmutables(poolInfo.poolAddress).token1();
227
```

```
228         (uint256 balBeforeToken0, uint256 balBeforeToken1) = _checkTokenBalances(token0, token1);
229
230         IERC20(token0).safeTransferFrom(_for, address(this), params.amount0Desired);
231         IERC20(token0).safeIncreaseAllowance(address(masterChefV3), params.amount0Desired);
232
233         if (poolInfo.isNative) {
234             balBeforeToken1 = address(this).balance - msg.value;
235             masterChefV3.increaseLiquidity{ value: msg.value }(params);
236         } else {
237             IERC20(token1).safeTransferFrom(_for, address(this), params.amount1Desired);
238             IERC20(token1).safeIncreaseAllowance(address(masterChefV3), params.amount1Desired);
239
240             masterChefV3.increaseLiquidity(params);
241         }
242
243         refundOrTransfer(token0, token1, _for, poolInfo.isNative, balBeforeToken0, balBeforeToken1)
                ;
244     }
```

Listing 2.5: PancakeStakingBaseUpg.sol

```
10      function refundOrTransfer(
11          address _token0,
12          address _token1,
13          address _dustTo,
14          bool _isNative,
15          uint256 _balBeforeToken0,
16          uint256 _balBeforeToken1
17      ) internal {
18          uint256 dustToken0Amt = IERC20(_token0).balanceOf(address(this)) - _balBeforeToken0;
19          if (dustToken0Amt > 0) {
20              IERC20(_token0).safeTransfer(_dustTo, dustToken0Amt);
21          }
22          uint256 dustToken1Amt;
23          if (_isNative) {
24              dustToken1Amt = address(this).balance - _balBeforeToken1;
25              _refundETH(payable(_dustTo), dustToken1Amt);
26          } else {
27              dustToken1Amt = IERC20(_token1).balanceOf(address(this)) - _balBeforeToken1;
28              if (dustToken1Amt > 0) {
29                  IERC20(_token1).safeTransfer(_dustTo, dustToken1Amt);
30              }
31          }
32      }
```

Listing 2.6: DustRefunder.sol

Impact The liquidity cannot be added to token pairs with the native token as `token0`.

Suggestion Revise the code logic accordingly.

2.2.2 Insufficient check on the Pancakeswap pool

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `PancakeV3Helper` contract, the `_checkForValidPool` function verifies that `token0` and `token1` of the position match the pool before position operations. However, this check may be inadequate as the same token pair can correspond to multiple pools on PancakeSwap. According to PancakeSwap, a pool is uniquely identified by `token0`, `token1`, and `fee`.

The code logic should not be affected, since the contract interacts with the correct pool. However, it may lead to potential security issues if the event emitted is used off-chain.

```
193    function _checkForValidPool(
194        address _for,
195        address _pool,
196        uint256 _tokenId,
197        bool _isStore
198    ) internal {
199        (
200            address token0,
201            address token1,
202            int24 tickLower,
203            int24 tickUpper,
204            uint128 liquidity,
205            uint256 feeGrowthInside0LastX128,
206            uint256 feeGrowthInside1LastX128
207        ) = getCurrentPosition(_tokenId);
208
209        address token0Pool = IPancakeV3PoolImmutables(_pool).token0();
210        address token1Pool = IPancakeV3PoolImmutables(_pool).token1();
211
212        if (token0Pool != token0 || token1Pool != token1) revert InvalidPool();
213
214        if (_isStore) {
215            userPositionInfos[_tokenId] = Position({
216                poolAddress: _pool,
217                tokenId: _tokenId,
218                token0: token0,
219                token1: token1,
220                tickLower: tickLower,
221                tickUpper: tickUpper,
222                liquidity: liquidity,
223                feeGrowthInside0LastX128: feeGrowthInside0LastX128,
224                feeGrowthInside1LastX128: feeGrowthInside1LastX128
225            });
226
227            addToken(_for, _tokenId);
228        }
229    }
```

Listing 2.7: PancakeV3Helper.sol

Impact N/A

Suggestion Add the check on `fee` for the pool.

2.3 Additional Recommendation

2.3.1 Implement usages for unused `isActive` flags

Status Acknowledged

Introduced by `Version 1`

Description In the following contracts, there are `isActive` flags used to indicate whether a pool or other identity is activated or enabled, but are rarely checked:

1. In contract `PancakeStakingBaseUpg`, the `isActive` flag for the pools is not checked when harvesting rewards for V3 pools.
2. In contract `PancakeVoteManager`, the `isActive` flag for pools is set but not checked anywhere.
3. In contract `RewardDistributor`, the `isActive` flag for whether the fee is enabled is only set but not checked anywhere.

Impact N/A

Suggestion Implement usages for those flags.

2.3.2 Add sanity checks before setting parameters

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The following functions set parameters for the contract, but no proper sanity check is implemented:

1. The function `pushVotingEpoch` in contract `CakepieBribeManager` does not check whether the new `epochStartTime` is chronologically arranged. And so as `forcePushEpoch`.
2. The function `setFeeRatio` should check the upper limit (`DENOMINATOR`, which is 10000) of the `feeRatio`. Likewise, the `forcePushEpoch` function should conduct sanity checks about the epoch time.

Impact Missing sanity check may allow misconfiguration.

Suggestion Add sanity checks accordingly.

2.3.3 Remove unused logic

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the following contracts, there are redundant logic or functions that can be removed to reduce code size and gas usage.

1. In contract `PancakeStakingBaseUpg`, the `_onlyAllowedOperator` modifier is unused. Besides, the `require` statement in this modifier is incorrect.

```
167         modifier _onlyAllowedOperator() {
168             if (allowedOperator[msg.sender]) revert OnlyAllowedOperator();
169             _;
170         }
```

Listing 2.8: PancakeStakingBaseUpg.sol

2. In contract `CakepieBribeManager`, the `pushBribingEpoch` function and `currentBribingEpoch` state variable is not used.
3. In contract `PancakeVoteManager`, the functions `getUserVoteForMarketsInVlCakepie` and `getUserVoteForPoolsInVlCakepie` are exactly the same.

Impact N/A.

Suggestion Remove the unused logic.

2.3.4 Fix typo in `CakepieBribeRewardDistributor`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There is a typo in function `emergencyWithdraw` of contract `CakepieBribeRewardDistributor`. The `bribeManager` should be `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 2.9: CakepieBribeRewardDistributor.sol

Impact N/A.

Suggestion Fix the typo accordingly.

2.3.5 Remove unused `payable` attribute

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `mCakeConvertorBNBChain`, the function `lockAllCake` does not need the `payable` attribute. Removing this attribute prevents callers from mistakenly transferring native tokens to this contract.

Impact N/A.

Suggestion Remove the `payable` attribute for this function.

2.4 Note

2.4.1 Centralization risk

Description In the Cakepie contracts, there are privileged functions that can cause severe consequences when misused by the project maintainers.

1. In contract `CakepieBribeRewardDistributor`, the function `emergencyWithdraw` can withdraw any token from the contract.
2. In contract `mCakeConvertorBNBChain` , the `mintFor` function can arbitrarily mint mCake token to any account. There is no contract to call this function, so it is assumed that an operator EOA calls this function.
3. In contract `BaseRewardPoolV3`, the function `emergencyWithdraw` can withdraw any token from the contract. Besides, the function only allows calls from `MasterCakepie`, but the contract does not seem to implement this function.
4. In contract `CakepieBribeManager`, the function `manualClaimFees` effectively withdraw any token from the contract, instead of only withdrawing the unclaimed fees stored in the `unCollectedFee` state variable.
5. In contract `CakepieBribeRewardDistributor`, rewards are distributed to the users by verifying the merkle proofs submitted and the merkle roots stored in the contract. The operator of the contract can update the merkle roots so that users may be unable to claim their rewards if the roots are reset mistakenly.

Feedback from the Project

1. Yes, we want to withdraw all tokens in the contract if we got any emergency situation, this is the `onlyOwner` function we only can call it through the multiSig account.
2. `mintFor` should be called by `pancakeStaking`, since in the future, if someone delegates to `pancakeStaking`, mCake should be minted for the delegator.
3. Intended for now, not adding on `MasterCakepie` unless we really need it when some reward stuck in rewarder.
4. In our implementation, only fee might leave in the manager contract if collector not set, reward will be transferred to the distributor directly.
5. When rolling merkle tree, off-chain logic does reserve the unclaimed rewards for users, so user won't lose whatever unclaimed.