Security Audit Report for VECake Gauges Contracts

Date: November 28, 2023

Version: 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Pancake
Target	VECake Gauges Contracts

Version History

Version	Date	Description
1.0	November 28, 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 for VECake Gauges Contracts[1] of Pancake. The VECake Gauges Contracts include both the `VECake` and `GaugeVoting` contracts. Users can deposit LP tokens into `VECake` to acquire voting power, and subsequently vote for gauge weights in the `GaugeVoting` contract. Additionally, the `VECake` contract provides interfaces for users migrating from the `CakePool` contract to the `VECake` contract. It is worth noting that external dependencies, such as OpenZeppelin's library, are assumed to be reliable and are therefore not included in the scope of this audit.

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
VECake Gauges Contracts	`Version 1`	`7974a13e369fee4f4eb04143d54cf14535cab3c1`
	`Version 2`	`3a66761d091a7ecb2e41d4c6c08ce5f5c95f7b88`
	`Version 3`	`93a746c8fb3e0d23dd0292f4fa42866c565a6275`

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] https://github.com/Chef-Snoopy/gauges-contracts

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.

Table 1.1: Vulnerability Severity Classification

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

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

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

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

Chapter 2 Findings

In total, we find **six** potential issues. Besides, we also have **two** recommendations and **two** notes.

- High Risk: 3
- Low Risk: 3
- Recommendation: 2
- Note: 2

ID	Severity	Description	Category	Status
1	Low	Inconsistent lock time limits	Software Security	Fixed
2	High	Incorrect operator precedence	Software Security	Fixed
3	Low	Flawed code logic that cannot update the first added gauge info	Software Security	Fixed
4	Low	Lack of sanity check on admin voting weight	Software Security	Fixed
5	High	Lack of updates on `gaugeChangesWeight` and `gaugeTypeChangesSum`	Software Security	Fixed
6	High	Inconsistent designs related to `boostMultiplier`	DeFi Security	Fixed
7	-	Fix typos	Recommendation	Fixed
8	-	Remove debugging codes	Recommendation	Fixed
9	-	Potential centralization risk	Note	-
10	-	Ensure the proper use of function `totalSupplyAtTime` and `balanceOfAtTime`	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Inconsistent lock time limits

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `MAX_LOCK` in the `VECake` contract is defined as 53 weeks, while the `MAX_LOCK_TIME` is 104 weeks in the `GaugeVoting` contract, causing an inconsistency.

```
98    // MAX_LOCK 53 weeks - 1 seconds
99    uint256 public constant MAX_LOCK = (53 * WEEK) - 1;
```

Listing 2.1: VECake.sol

```
28    uint256 constant MAX_LOCK_TIME = WEEK * 104;
```

Listing 2.2: GaugeVoting.sol

Impact N/A

Suggestion Revise the code accordingly.

2.1.2 Incorrect operator precedence

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `_vote1` function in the `GaugeVoting` contract incorrectly updates the bias value.

```
693    function _vote1(bytes32 gauge_hash, VotedSlope memory old_slope, VotedSlope memory new_slope,
           uint256 old_bias, uint256 new_bias) internal {
694        uint256 next_time = (block.timestamp + WEEK) / WEEK * WEEK;
695
696        uint256 gauge_type = gaugeTypes_[gauge_hash] - 1;
697        require(gauge_type >= 0, "Gauge not added");
698
699        // Remove old and schedule new slope changes
700        // Remove slope changes for old slopes
701        // Schedule recording of initial slope for next_time
702        uint256 old_weight_bias = _getWeight(gauge_hash);
703        uint256 old_weight_slope = gaugePointsWeight[gauge_hash][next_time].slope;
704        uint256 old_sum_bias = _getTypeSum(gauge_type);
705        uint256 old_sum_slope = gaugeTypePointsSum[gauge_type][next_time].slope;
706
707        gaugePointsWeight[gauge_hash][next_time].bias = old_weight_bias + new_bias > old_bias ?
               old_weight_bias + new_bias : old_bias - old_bias;
708        gaugeTypePointsSum[gauge_type][next_time].bias = old_sum_bias + new_bias > old_bias ?
               old_sum_bias + new_bias : old_bias - old_bias;
```

Listing 2.3: GaugeVoting.sol

The bias is calculated as:

```
bias = max(old_weight_bias + new_bias, old_bias) - old_bias
```

Therefore, Line 707 should be:

```
(old_weight_bias + new_bias > old_bias ? old_weight_bias + new_bias : old_bias) - old_bias
```

The same issue also exists in the `_vote2` and `_vote3` functions.

Impact Incorrect operator precedence will lead to unexpected behaviors.

Suggestion Revise the code accordingly.

2.1.3 Flawed code logic that cannot update the first added gauge info

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `updateGaugeInfo` function in the `GaugeVoting` contract updates the gauge info. Line 213 requires `gaugeIndex[gauge_hash]-1` to be greater than or equal to 0, which will underflow for the first gauge at index 0. This causes the function to revert when trying to update `gauges[0]`.

```
207    function updateGaugeInfo(address gauge_addr, uint256 _pid, address _masterChef, uint256
           _chainId, uint256 _boostMultiplier, uint256 _maxVoteCap) external onlyOwner {
208        require(_masterChef != address(0), "masterChef address is empty");
209        require(_boostMultiplier >= 100 && _boostMultiplier <= 500);
210        require(_maxVoteCap >= 0 && _maxVoteCap <= 10000);
211
212        bytes32 gauge_hash = keccak256(abi.encodePacked(gauge_addr, _chainId));
213        uint256 idx = gaugeIndex_[gauge_hash] - 1;
214        require(idx >= 0, "Gauge not added");
215
216        gauges[idx] = GaugeInfo({
217            pairAddress: gauge_addr,
218            pid: _pid,
219            masterChef: _masterChef,
220            chainId: _chainId,
221            boostMultiplier: _boostMultiplier,
222            maxVoteCap: _maxVoteCap
223        });
224
225        emit UpdateGaugeInfo(gauge_hash, _pid, _masterChef, _chainId, _boostMultiplier, _maxVoteCap
               );
226    }
```

Listing 2.4: GaugeVoting.sol

```
155    function addGauge(address gauge_addr, uint256 _type, uint256 _weight, uint256 _pid, address
           _masterChef, uint256 _chainId, uint256 _boostMultiplier, uint256 _maxVoteCap) external
           onlyOwner {
156        require(_type >= 0 && _type < gaugeTypes, "Invalid gauge type");
157        bytes32 gauge_hash = keccak256(abi.encodePacked(gauge_addr, _chainId));
158        require(gaugeTypes_[gauge_hash] == 0, "Gauge already added"); // dev: cannot add the same
               twice
159        require(_masterChef != address(0), "masterChef address is empty");
160        require(_boostMultiplier >= 100 && _boostMultiplier <= 500);
161        require(_maxVoteCap >= 0 && _maxVoteCap <= 10000);
162
163        uint256 n = gaugeCount;
164        gaugeCount = n + 1;
165        gauges[uint256(n)] = GaugeInfo({
166            pairAddress: gauge_addr,
167            pid: _pid,
168            masterChef: _masterChef,
169            chainId: _chainId,
170            boostMultiplier: _boostMultiplier,
171            maxVoteCap: _maxVoteCap
172        });
173
174        gaugeIndex_[gauge_hash] = n;
175        gaugeTypes_[gauge_hash] = _type + 1;
```

Listing 2.5: GaugeVoting.sol

Impact The first added gauge info can be never updated.

Suggestion Revise the code accordingly.

2.1.4 Lack of sanity check on admin voting weight

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `voteFromAdmin` function in the `GaugeVoting` contract currently lacks a sanity check on the `_admin_weight` parameter. This value is used to specify the voting weight proportion, which should be between 0 and 10,000.

```
323    function voteFromAdmin(address _gauge_addr, uint256 _admin_weight, uint256 _end, uint256
           _chainId) external onlyOwner {
324        uint256 nextTime = (block.timestamp + WEEK) / WEEK * WEEK;
325        require(_end > nextTime, "Your end timestamp expires too soon");
326
327        bytes32 gauge_hash = keccak256(abi.encodePacked(_gauge_addr, _chainId));
328
329        // Prepare slopes and biases in memory
330        VotedSlope memory old_slope = voteUserSlopes[address(0)][gauge_hash];
331        uint256 old_bias = old_slope.slope;
332
333        uint256 idx = gaugeIndex_[gauge_hash];
334        require(idx >= 0, "Gauge not added");
335
336        GaugeInfo memory info = gauges[idx];
337        uint256 _admin_weight2 = _admin_weight;
338
339        VotedSlope memory new_slope = VotedSlope({
340            slope: gaugePointsTotal[totalLastScheduled] * _admin_weight2 * 20 / 1000000,
341            end: _end,
342            power: _admin_weight2
343        });
344        uint256 new_bias = new_slope.slope;
345
346        if (old_slope.end > nextTime) {
347            _vote1(gauge_hash, old_slope, new_slope, old_bias, new_bias);
348        } else {
349            _vote2(gauge_hash, new_slope, old_bias, new_bias);
350        }
351        if (old_slope.end > block.timestamp) {
352            _vote3(gauge_hash, old_slope, old_bias, new_bias);
353        }
354
355        _getTotal();
356
357        emit VoteForGaugeFromAdmin(block.timestamp, msg.sender, gauge_hash, new_slope.power);
358    }
```

Listing 2.6: GaugeVoting.sol

Impact The admin may mistakenly gain too much voting power.

Suggestion Add sanity check on the admin voting weight.

2.1.5 Lack of updates on `gaugeChangesWeight` and `gaugeTypeChangesSum`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Two mappings, `gaugeChangesWeight` and `gaugeTypeChangesSum`, of the `GaugeVoting` contract, are never modified. Specifically, they should be updated within the `_vote1`, `_vote2`, and `_vote3` functions to track changes in the slope. Although these variables are used in functions such as `_getTypeSum` and `_getWeight`, their values persistently remain at zero.

```
92    /// @dev type_id -> time -> slope
93    mapping(uint256 => mapping(uint256 => uint256)) public gaugeTypeChangesSum;
```

Listing 2.7: GaugeVoting.sol

```
85    /// @dev gauge_hash -> time -> slope
86    mapping(bytes32 => mapping(uint256 => uint256)) public gaugeChangesWeight;
```

Listing 2.8: GaugeVoting.sol

Impact Corresponding calculations will be wrong due to the lack of updates.

Suggestion Revise the code accordingly.

2.2 DeFi Security

2.2.1 Inconsistent designs related to `boostMultiplier`

Severity High

Status Fixed in `Version 3`

Introduced by `Version 1`

Description The `GaugeVoting` contract allocates a `boostMultiplier` for each gauge to calculate the weights. However, there are some inconsistent designs around the use of `boostedMultiplier`. For example, both `gaugeTypePointsSum` and `gaugePointsTotal` are calculated based on the `boostedMultiplier` when adding a new gauge.

```
179    if (_weight > 0) {
180        uint256 typeWeight = _getTypeWeight(_type);
181        uint256 oldTypeSum = _getTypeSum(_type);
182        uint256 oldTotal = _getTotal();
183
184        gaugeTypePointsSum[_type][nextTime].bias = _weight * _boostMultiplier + oldTypeSum;
185        gaugeTypeSumLastScheduled[_type] = nextTime;
186        gaugePointsTotal[nextTime] = oldTotal + typeWeight * _weight * _boostMultiplier;
187        totalLastScheduled = nextTime;
188
```

```
189         gaugePointsWeight[gauge_hash][nextTime].bias = _weight;
190     }
```

Listing 2.9: GaugeVoting.sol

Therefore, if a `boostedMultiplier` is updated in the `updateGaugeInfo` function, these two variables should be recalculated correspondingly. However, at present, they are not updated as necessary.

```
207     function updateGaugeInfo(address gauge_addr, uint256 _pid, address _masterChef, uint256
            _chainId, uint256 _boostMultiplier, uint256 _maxVoteCap) external onlyOwner {
208         require(_masterChef != address(0), "masterChef address is empty");
209         require(_boostMultiplier >= 100 && _boostMultiplier <= 500);
210         require(_maxVoteCap >= 0 && _maxVoteCap <= 10000);
211
212         bytes32 gauge_hash = keccak256(abi.encodePacked(gauge_addr, _chainId));
213         uint256 idx = gaugeIndex_[gauge_hash] - 1;
214         require(idx >= 0, "Gauge not added");
215
216         gauges[idx] = GaugeInfo({
217             pairAddress: gauge_addr,
218             pid: _pid,
219             masterChef: _masterChef,
220             chainId: _chainId,
221             boostMultiplier: _boostMultiplier,
222             maxVoteCap: _maxVoteCap
223         });
224
225         emit UpdateGaugeInfo(gauge_hash, _pid, _masterChef, _chainId, _boostMultiplier, _maxVoteCap
                );
226     }
```

Listing 2.10: GaugeVoting.sol

Additionally, there are inconsistencies in other places related to weight points about whether or not to use the `boostedMultiplier`.

Impact Inconsistent usages could potentially lead to unexpected consequences.

Suggestion Revise the code accordingly.

2.3 Additional Recommendation

2.3.1 Fix typos

Status Fixed in `Version 2`

Introduced by `Version 1`

Description "inilization should be "initialization".

```
113     // Cake pool migation inilization flag
114     bool public inilization;
```

Listing 2.11: VECake.sol

Impact N/A

Suggestion Fix the typo.

2.3.2 Remove debugging codes

Status Fixed in `Version 3`

Introduced by `Version 2`

Description There are some debugging codes that should be removed.

```
8   import "hardhat/console.sol";
```

Listing 2.12: GaugeVoting.sol

Impact N/A

Suggestion Remove the debugging codes.

2.4 Note

2.4.1 Potential centralization risk

Description The owner of the `VECake` and `GaugeVoting` contracts possesses notable privileges to modify critical configurations. For instance, the owner can enable emergency withdrawals, add gauges, update weights, and more. This concentration of power introduces a single point of failure. If an attacker were to compromise the owner, the entire system could potentially be incapacitated.

Feedback from the Project Owner will be controlled by multisig wallet.

2.4.2 Ensure the proper use of function `totalSupplyAtTime` and `balanceOfAtTime`

Description In the `VECake` contract, the `totalSupply` function should, in theory, not return the total supply at a timestamp beyond `block.timestamp`. However, in `Version 2`, the contract introduces a `totalSupply` function (renamed to `totalSupplyAtTime` in `Version 3`) without timestamp limitations that accommodates special usages in the `GaugeVoting` contract. Other external contracts utilizing this function should take care to invoke it properly. Passing a future timestamp could produce incorrect results. The same issue also holds for function `balanceOfAtTime`.

```
883   function totalSupplyAtTime(uint256 _timestamp) external view returns (uint256) {
884       return _totalSupplyAt(pointHistory[epoch], _timestamp);
885   }
```

Listing 2.13: VECake.sol

```
347   function balanceOfAtTime(address _user, uint256 _timestamp) external view returns (uint256) {
348       return _balanceOf(_user, _timestamp);
349   }
```

Listing 2.14: VECake.sol

Feedback from the Project Should be ok, if other smart contract used the `totalSupplyAtTime(uint256 _timestamp)`, they should know what are they doing.