SMART CONTRACT AUDIT REPORT

for

Gauge & GaugeProxy

Prepared By: Shuxiao Wang

PeckShield
March 4, 2021

Document Properties

Client	Andre Cronje
Title	Smart Contract Audit Report
Target	Gauge/GaugeProxy
Version	1.0
Author	Xuxian Jiang
Auditors	Ruiyi Zhang, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 4, 2021	Xuxian Jiang	Final Release
1.0-rc	March 4, 2021	Xuxian Jiang	Release Candidate
0.3	March 1, 2021	Xuxian Jiang	Additional Findings #2
0.2	February 27, 2021	Xuxian Jiang	Additional Findings #1
0.1	February 26, 2021	Xuxian Jiang	Initial Draft

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of Gauge/GaugeProxy, 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 Gauge/GaugeProxy

The Gauge/GaugeProxy implementation presents a unique offering in enabling voting-based rewarding mechanism. The reward amount is weighted according to user votes, which collectively determines the percentage for reward dissemination to each individual reward pool. The design of each reward pool, i.e., `Gauge`, is heavily influenced by the `Synthetix` liquidity mining reward contract. However, it has its own innovation in taking into account time-weighted-locked share to improve the reward efficiency and effectiveness.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Gauge/GaugeProxy

Item	Description
Client	Andre Cronje
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 4, 2021

In the following, we show the repository of reviewed files and the MD5 checksum hash value used in this audit.

- URL: https://gist.github.com/0xkoffee/88b57d2e745086a1b968a5338f6103dd
- MD5: 67d1b13f579fcb0f87a39c6969cd0409

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

- URL: https://gist.github.com/0xkoffee/88b57d2e745086a1b968a5338f6103dd
- MD5: 1f2b809be3f350c4005cae0111745fe5

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

Likelihood

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.

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 accordingly classified into four categories, 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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

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

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.

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.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the given Gauge/GaugeProxy contracts. 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	1	■
Medium	1	■
Low	2	■ ■
Informational	3	■ ■ ■
Total	7	

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 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 high-severity vulnerability, 1 medium-severity vulnerability, 2 low-severity vulnerabilities, and 3 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Logic in poke()	Business Logic	Fixed
PVE-002	High	Inconsistent Votes Usage in GaugeProxy	Business Logic	Fixed
PVE-003	Low	Possible Unitialized Pool ID Use in deposit()	Coding Practices	Fixed
PVE-004	Medium	Revisited setPID() Logic	Business Logic	Fixed
PVE-005	Informational	Improved Validation Of Function Arguments	Coding Practices	Fixed
PVE-006	Informational	Suggested Adherence of Checks-Effects-Interactions	Time and State	Fixed
PVE-007	Informational	Removal Of Unused Events	Coding Practices	Fixed

Beside the identified issues, we note that the staking support assumes the staked tokens are not deflationary. Also, 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 Improved Logic in poke()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: GaugeProxy
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

In order to timely update the user votes, the GaugeProxy contract provides a public function, i.e., poke(), that can be used by anyone to refresh the votes from a given user. To elaborate, we show below the full implementation of this routine.

```
    // Reset votes to 0
    function poke(address _owner) public {
        address[] memory _tokenVote = tokenVote[msg.sender];
        uint256 _tokenCnt = _tokenVote.length;
        uint256[] memory _weights = new uint[](_tokenCnt);

        uint256 _prevUsedWeight = usedWeights[_owner];
        uint256 _weight = DILL.balanceOf(_owner);


        for (uint256 i = 0; i < _tokenCnt; i ++) {
            uint256 _prevWeight = votes[_tokenVote[i]][_owner];
            _weights[i] = _prevWeight.mul(_weight).div(_prevUsedWeight);
        }

        _vote(_owner, _tokenVote, _weights);
    }
```

Listing 3.1: GaugeProxy::poke()

Our analysis with this routine shows the function header is misleading as it does not reset the votes to 0. Moreover, the list of tokens that have been voted should be retrieved by tokenVote[_owner], not current tokenVote[msg.sender] (line 579).

Recommendation Improved the poke() logic by correcting the function header and using the given _owner to access tokenVote. An example revision is shown below:

```
577     // reflesh user votes
578     function poke(address _owner) public {
579         address[] memory _tokenVote = tokenVote[_owner];
580         uint256 _tokenCnt = _tokenVote.length;
581         uint256[] memory _weights = new uint[](_tokenCnt);
582 
583         uint256 _prevUsedWeight = usedWeights[_owner];
584         uint256 _weight = DILL.balanceOf(_owner);
585 
586 
587         for (uint256 i = 0; i < _tokenCnt; i ++) {
588             uint256 _prevWeight = votes[_tokenVote[i]][_owner];
589             _weights[i] = _prevWeight.mul(_weight).div(_prevUsedWeight);
590         }
591 
592         _vote(_owner, _tokenVote, _weights);
593     }
```

Listing 3.2: GaugeProxy::poke()

Status This issue has been fixed in this commit: 88b57d2.

3.2 Inconsistent Votes Usage in GaugeProxy

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: GaugeProxy
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

To properly record user votes, the GaugeProxy contract has its internal states votes that are indexed by user and token. However, it comes to our attention that the way to retrieve the user votes is inconsistent in current implementation.

To elaborate, we show below three routines: _reset(), poke(), and _vote(). The first routine updates the votes via votes[_owner][_token] (line 570), while the other two access the votes via votes[_token][_owner] (lines 588 and 617).

```
557     // Reset votes to 0
558     function _reset(address _owner) internal {
```

```
        address[] storage _tokenVote = tokenVote[_owner];
        uint256 _tokenVoteCnt = _tokenVote.length;

        for (uint i = 0; i < _tokenVoteCnt; i ++) {
            address _token = _tokenVote[i];
            uint _votes = votes[_owner][_token];

            if (_votes > 0) {
                totalWeight = totalWeight.sub(_votes);
                weights[_token] = weights[_token].sub(_votes);

                votes[_owner][_token] = 0;
            }
        }

        delete tokenVote[_owner];
    }
```

Listing 3.3: GaugeProxy::_reset()

```
    // Reset votes to 0
    function poke(address _owner) public {
        address[] memory _tokenVote = tokenVote[msg.sender];
        uint256 _tokenCnt = _tokenVote.length;
        uint256[] memory _weights = new uint[](_tokenCnt);

        uint256 _prevUsedWeight = usedWeights[_owner];
        uint256 _weight = DILL.balanceOf(_owner);


        for (uint256 i = 0; i < _tokenCnt; i ++) {
            uint256 _prevWeight = votes[_tokenVote[i]][_owner];
            _weights[i] = _prevWeight.mul(_weight).div(_prevUsedWeight);
        }

        _vote(_owner, _tokenVote, _weights);
    }
```

Listing 3.4: GaugeProxy::poke()

```
    function _vote(address _owner, address[] memory _tokenVote, uint256[] memory
        _weights) internal {
        // _weights[i] = percentage * 100
        _reset(_owner);
        uint256 _tokenCnt = _tokenVote.length;
        uint256 _weight = DILL.balanceOf(_owner);
        uint256 _totalVoteWeight = 0;
        uint256 _usedWeight = 0;

        for (uint256 i = 0; i < _tokenCnt; i ++) {
            _totalVoteWeight = _totalVoteWeight.add(_weights[i]);
        }
```

```

        for (uint256 i = 0; i < _tokenCnt; i ++) {
            address _token = _tokenVote[i];
            address _gauge = gauges[_token];
            uint256 _tokenWeight = _weights[i].mul(_weight).div(_totalVoteWeight);

            if (_gauge != address(0x0)) {
                _usedWeight = _usedWeight.add(_tokenWeight);
                totalWeight = totalWeight.add(_tokenWeight);
                weights[_token] = weights[_token].add(_tokenWeight);
                tokenVote[_owner].push(_token);
                votes[_token][_owner] = _tokenWeight;
            }
        }

        usedWeights[_owner] = _usedWeight;
    }
```

Listing 3.5: GaugeProxy::_vote()

Recommendation Be consistent when indexing the internal states `votes`.

Status This issue has been fixed in this commit: `88b57d2`.

3.3 Possible Unitialized Pool ID Use in deposit()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `GaugeProxy`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [1]

Description

The `GuageProxy` contract is programmed to stake the holding `mDILL` tokens into `MasterChef` for possible `PICKLE` rewards. To properly stake, there is a need to specify the pool ID managed by `MasterChef`. With that, the `GuageProxy` contract provides two functions, `setPID()` and `deposit()`, to specify the pool ID and stake the holding `mDILL` tokens into the pool.

```
    // Sets MasterChef PID
    function setPID(uint _pid) external {
        require(msg.sender == governance, "!gov");
        pid = _pid;
    }


    // Deposits mDILL into MasterChef
```

```
    function deposit() public {
        IERC20 _token = TOKEN;
        uint _balance = _token.balanceOf(address(this));
        _token.safeApprove(address(MASTER), 0);
        _token.safeApprove(address(MASTER), _balance);
        MASTER.deposit(pid, _balance);
    }
```

Listing 3.6: GaugeProxy::setPID() and GaugeProxy::deposit()

However, since the `deposit()` function is a public one, which allows any one to invoke. As a result, if the pool ID is not assigned, the holding `mDILL` tokens will be staked into the pool with ID as 0, which is the uninitialized pool ID.

Recommendation Ensure the pool ID is initialized before `deposit()` can be invoked.

Status This issue has been fixed in this commit: `88b57d2`.

3.4 Revisited setPID() Logic

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `GaugeProxy`
- Category: Business Logic [5]
- CWE subcategory: N/A

Description

As mentioned in Section 3.3, the `GuageProxy` contract is programmed to stake the holding `mDILL` tokens into `MasterChef` for possible `PICKLE` rewards. We have examined the need to initialize the pool ID before staking the holding `mDILL` tokens into the pool. In the following, we re-examine the `setPID()` logic and find that the update of the pool ID may not be able to retrieve back previously staked `mDILL` tokens.

To elaborate, we show below the `setPID()` implementation. It simply validates the caller and properly updates the pool ID. However, the new pool ID will inevitably affect other routines. One of them is `collect()` that is designed to collect possible `PICKLE` rewards.

```
    // Sets MasterChef PID
    function setPID(uint _pid) external {
        require(msg.sender == governance, "!gov");
        pid = _pid;
    }
```

Listing 3.7: GaugeProxy::setPID()

Specifically, the `collect()` will attempt to retrieve back the rewards from the new pool ID, instead of the old pool ID. However, once the pool ID is updated, there is no way to retrieve back the previously staked `mDILL` tokens.

```
    // Fetches Pickle
    function collect() public {
        (uint _locked,) = MASTER.userInfo(pid, address(this));
        MASTER.withdraw(pid, _locked);
        deposit();
    }
```

Listing 3.8: GaugeProxy::collect()

Recommendation Revise current execution logic of `setPID()` to properly retrieve back previously staked `mDILL` tokens so that they can be re-staked into the new pool.

Status This issue has been fixed in this commit: `88b57d2`.

3.5 Improved Validation Of Function Arguments

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `GaugeProxy`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [1]

Description

The `GaugeProxy`'s `voting` capability allows each user to better specify their intentions on voted `gauges`. Specifically, it provides a public `vote()` function, which delegates to the internal handler `_vote()`. To elaborate, we show below these two routines. It comes to our attention that the `_vote()` has the inherent assumption on the same length of the given two arrays, i.e., `_tokenVote`, and `_weights`. However, this is not enforced at the `vote()` function.

```
    function _vote(address _owner, address[] memory _tokenVote, uint256[] memory
        _weights) internal {
        // _weights[i] = percentage * 100
        _reset(_owner);
        uint256 _tokenCnt = _tokenVote.length;
        uint256 _weight = DILL.balanceOf(_owner);
        uint256 _totalVoteWeight = 0;
        uint256 _usedWeight = 0;

        for (uint256 i = 0; i < _tokenCnt; i++) {
            _totalVoteWeight = _totalVoteWeight.add(_weights[i]);
        }
```

```
        for (uint256 i = 0; i < _tokenCnt; i ++) {
            address _token = _tokenVote[i];
            address _gauge = gauges[_token];
            uint256 _tokenWeight = _weights[i].mul(_weight).div(_totalVoteWeight);

            if (_gauge != address(0x0)) {
                _usedWeight = _usedWeight.add(_tokenWeight);
                totalWeight = totalWeight.add(_tokenWeight);
                weights[_token] = weights[_token].add(_tokenWeight);
                tokenVote[_owner].push(_token);
                votes[_token][_owner] = _tokenWeight;
            }
        }

        usedWeights[_owner] = _usedWeight;
    }


    // Vote with DILL on a gauge
    function vote(address[] calldata _tokenVote, uint256[] calldata _weights) external {
        _vote(msg.sender, _tokenVote, _weights);
    }
```

Listing 3.9: GaugeProxy::vote()

Recommendation Add the length check on all given arguments of `vote()`. An example revision is shown below:

```
    // Vote with DILL on a gauge
    function vote(address[] calldata _tokenVote, uint256[] calldata _weights) external {
        require(_tokenVote.length == _weights.length, "!length");
        _vote(msg.sender, _tokenVote, _weights);
    }
```

Listing 3.10: GaugeProxy::vote()

Status This issue has been fixed in this commit: `88b57d2`.

3.6 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Gauge
- Category: Time and State [6]
- CWE subcategory: CWE-663 [2]

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` [11] exploit, and the recent `Uniswap/Lendf.Me` hack [10].

We notice an occasion where the `checks-effects-interactions` principle is violated. Using the `Gauge` as an example, the `_deposit()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 307) starts before effecting the update on internal states (lines 308−309), 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 very same `_deposit()` function.

```
305     function _deposit(uint amount) internal nonReentrant updateReward(msg.sender) {
306         require(amount > 0, "Cannot stake 0");
307         TOKEN.safeTransferFrom(msg.sender, address(this), amount);
308         _totalSupply = _totalSupply.add(amount);
309         _balances[msg.sender] = _balances[msg.sender].add(amount);
310         emit Staked(msg.sender, amount);
311     }
```

Listing 3.11: Gauge::_deposit()

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` best practice.

Status This issue has been fixed in this commit: `88b57d2`.

3.7 Removal Of Unused Events

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Gauge
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [1]

Description

Meaningful events are an important part in smart contract design as they can not only greatly expose the runtime dynamics of smart contracts, but also allow for better understanding about their behavior and facilitate off-chain analytics. These events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed.

We have examined the events emitted from the protocol and notice that there are two events that are defined but never emitted: RewardsDurationUpdated and Recovered.

```
    event RewardAdded(uint256 reward);
    event Staked(address indexed user, uint256 amount);
    event Withdrawn(address indexed user, uint256 amount);
    event RewardPaid(address indexed user, uint256 reward);
    event RewardsDurationUpdated(uint256 newDuration);
    event Recovered(address token, uint256 amount);
```

Listing 3.12: Gauge.sol

Recommendation Remove the above-mentioned two events that are not used.

Status This issue has been fixed in this commit: 88b57d2.

4 Conclusion

In this audit, we have analyzed the Gauge/GaugeProxy design and implementation. The system presents a unique offering in voting-based rewarding support. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

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

[6] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.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.

[10] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[11] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.