SMART CONTRACT AUDIT REPORT

for

SWERVE FINANCE

Prepared By: Shuxiao Wang

Hangzhou, China
September 26, 2020

Document Properties

Client	Swerve Finance
Title	Smart Contract Audit Report
Target	Swerve
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Huaguo Shi, Jeff Liu
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	September 26, 2020	Xuxian Jiang	Final Release
0.4	September 22, 2020	Xuxian Jiang	Additional Findings #3
0.3	September 20, 2020	Xuxian Jiang	Additional Findings #2
0.2	September 19, 2020	Xuxian Jiang	Additional Findings #1
0.1	September 17, 2020	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 **5**
- 1.1 About Swerve Finance . . . 5
- 1.2 About PeckShield . . . 6
- 1.3 Methodology . . . 6
- 1.4 Disclaimer . . . 8

2 Findings **10**
- 2.1 Summary . . . 10
- 2.2 Key Findings . . . 11

3 Detailed Results **12**
- 3.1 Inaccurate blocksPerYear Estimate in APYOracle . . . 12
- 3.2 Lack of Protection Against Oversized Gauge/Type Weights . . . 13
- 3.3 Better Handling of Privilege Transfers . . . 15
- 3.4 Unused Import/Code/Interface Removal . . . 17
- 3.5 Incompatibility with Deflationary/Rebasing Tokens . . . 18
- 3.6 Improved Binary Search in find_block_epoch() . . . 21
- 3.7 Implicit Threshold On Supported Distinct Gauge Types . . . 23
- 3.8 Lack of Rigorous Sanity Checks Against Gauge/Type Weight Updates . . . 25
- 3.9 Improved AddType() Event Generation . . . 28
- 3.10 Improved Precision By Multiplication-Before-Division . . . 30
- 3.11 Better Consistency Between deposit() and withdraw() . . . 32
- 3.12 Viewable Adjustment of claimable_tokens() . . . 34

4 Conclusion **35**

5 Appendix **36**
- 5.1 Basic Coding Bugs . . . 36
  - 5.1.1 Constructor Mismatch . . . 36
  - 5.1.2 Ownership Takeover . . . 36

5.1.3 Redundant Fallback Function . . . 36
5.1.4 Overflows & Underflows . . . 36
5.1.5 Reentrancy . . . 37
5.1.6 Money-Giving Bug . . . 37
5.1.7 Blackhole . . . 37
5.1.8 Unauthorized Self-Destruct . . . 37
5.1.9 Revert DoS . . . 37
5.1.10 Unchecked External `Call` . . . 38
5.1.11 Gasless `Send` . . . 38
5.1.12 `Send` Instead Of `Transfer` . . . 38
5.1.13 Costly Loop . . . 38
5.1.14 (Unsafe) Use Of Untrusted Libraries . . . 38
5.1.15 (Unsafe) Use Of Predictable Variables . . . 39
5.1.16 Transaction Ordering Dependence . . . 39
5.1.17 Deprecated Uses . . . 39
5.2 Semantic Consistency Checks . . . 39
5.3 Additional Recommendations . . . 39
5.3.1 Avoid Use of Variadic Byte Array . . . 39
5.3.2 Make Visibility Level Explicit . . . 40
5.3.3 Make Type Inference Explicit . . . 40
5.3.4 Adhere To Function Declaration Strictly . . . 40

References **41**

1 | Introduction

Given the opportunity to review the Swerve protocol design document and related smart contract source code, we in the report outline 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 Swerve Finance

Swerve is a community fork of `Curve` that is designed for efficient stablecoin trading with low trading fees and (extremely) low slippage. By performing a community fork, Swerve presents its own `SwerveDAO` to govern the development and evolvement. The main differences from `Curve` include the removal of pre-mines, the removal of shareholder/team allocation, and arguably a better decentralization without dominating voting powers held by a selected few of entities. The governance tokens are designed to be 100% issued to liquidity providers as reward incentives, instead of 62% in Curve. Swerve can be considered as a representative demonstrating the recent rise of a more open, community-led trend/paradigm.

The basic information of Swerve is as follows:

Table 1.1: Basic Information of Swerve

Item	Description
Issuer	Swerve Finance
Website	https://swerve.fi/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 26, 2020

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit:

- https://github.com/SwerveFinance/SwerveContracts (487b410)

1.2 About PeckShield

PeckShield Inc. [21] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [16]:

- 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.

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

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) [15], 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 audit does not give any warranties on finding all possible security issues of the given smart contract(s), 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 Swerve implementation. During the first phase of our audit, we studied the smart contract source code and ran 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	3	■ ■ ■
Low	3	■ ■ ■
Informational	5	■ ■ ■ ■ ■
Total	12	

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, 3 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 5 informational recommendations.

Table 2.1: Key Audit Findings of Swerve Protocol

ID	Severity	Title	Category	Status
PVE-001	Medium	Inaccurate blocksPerYear Estimate in APYOracle	Numeric Errors	Fixed
PVE-002	High	Lack of Protection Against Oversized Gauge/Type Weights	Numeric Errors	Confirmed
PVE-003	Low	Better Handling of Privilege Transfers	Business Logics	Confirmed
PVE-004	Informational	Unused Import/Code/Interface Removal	Coding Practices	Confirmed
PVE-005	Informational	Incompatibility with Deflationary/Rebasing Tokens	Business Logics	Confirmed
PVE-006	Low	Improved Binary Search in find_block_epoch()	Business Logics	Confirmed
PVE-007	Medium	Implicit Threshold On Supported Distinct Gauge Types	Business Logics	Confirmed
PVE-008	Medium	Lack of Rigorous Sanity Checks Against Gauge/Type Weight Updates	Coding Practices	Confirmed
PVE-009	Informational	Improved AddType() Event Generation	Error Conditions, Return Values, Status Codes	Confirmed
PVE-010	Low	Improved Precision By Multiplication-Before-Division	Numeric Errors	Confirmed
PVE-011	Informational	Better Consistency Between deposit() and withdraw()	Coding Practices	Confirmed
PVE-012	Informational	Viewable Adjustment of claimable_tokens()	Expression Issues	Confirmed

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 Inaccurate blocksPerYear Estimate in APYOracle

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact:Medium
- Target: `APYOracle`
- Category: Numeric Errors [14]
- CWE subcategory: CWE-190 [3]

Description

The Swerve protocol contains a contract `APYOracle` that defines a main function, i.e., `getAPY()`. This function is used to report the `annual percentage yield (APY)` of a `Swerve` pool. Note `APY` is a normalized representation of an interest rate, based on a compounding period of one year and allows for a reasonable and meaningful comparison with other different offerings (likely with varying compounding schedules).

In the following, we show the code snippet of the `APYOracle` contract. This contract defines a constant `blocksPerYear`, which, as the name indicates, shows the number of estimated blocks within a year. Considering the non-constant block production of the Ethereum blockchain, we typically choose an average block production time (e.g., $13s$, $14s$, or $15s$) for the estimation. However, the constant used in the contract is 242_584 (line 10) shows a bizarre block production time. In fact, for the block production time of $13s$, $14s$, and $15s$, this constant should be 2_425_846, 2_252_571, and 2_102_400 respectively. Our best guess is the average $13s$ block production is taken for the estimate. However, the last digit is somehow omitted when updating the contract, resulting in a report of `APY` with one order of magnitude less.

```
7  contract APYOracle {
8      YPool public pool;
9      uint256 public poolDeployBlock;
10     uint256 constant blocksPerYear = 242584;
11
12     constructor(YPool _pool, uint256 _poolDeployBlock) public {
```

```
        pool = _pool;
        poolDeployBlock = _poolDeployBlock;
    }

    function getAPY() external view returns (uint256) {
        uint256 blocks = block.number - poolDeployBlock;
    uint256 price = pool.get_virtual_price() - 1e18;
        return price * blocksPerYear / blocks;
    }
}
```

Listing 3.1: APYOracle.sol

By having a wrong `blocksPerYear` constant, the contract could report a wrong `APY` of current returns. The current constant is an order of magnitude less than the normal number. Therefore, it reports much less yield for the configured pool (ever since the contract is deployed).

We emphasize that this issue only affects the display UI and does not mean the protocol `APY` was wrong. In fact, as mentioned earlier, it reports less `APY`, putting itself in a disadvantageous position when compared with other similar offerings.

Recommendation Change the `blocksPerYear` constant with a normal one and report the correct `APY`.

Status This issue has been confirmed and fixed. A new `APYOracle` contract with the correct number of `blocksPerYear` has been accordingly deployed.

3.2 Lack of Protection Against Oversized Gauge/Type Weights

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `GaugeController`
- Category: Numeric Errors [14]
- CWE subcategory: CWE-190 [3]

Description

The `SwerveDAO` is based on the `CurveDAO` where the `GaugeController` contract is the central of the entire governance subsystem. In particular, this `GaugeController` contract is responsible for adding new `gauges` and their `types`, changing their `weights`, as well as casting votes on different `gauges`.

Our analysis leads to the discovery of a potential pitfall when a new oversized `gauge` (or `type`) weight is updated on current pools. In particular, as the `gauge_relative_weight()` routine involves the multiplication of three `uint256` integer, it is possible for their multiplication to have an undesirable overflow (`MULTIPLIER * _type_weight * _gauge_weight` in `GaugeController.vy` at line 366), especially when `_type_weight` or `_gauge_weight` is largely controlled by an external entity. Fortunately, an authentication check is in place that effectively restricts the caller to be `admin` and thus greatly alleviates such concern.

```
def _change_type_weight(type_id: int128, weight: uint256):
    """
    @notice Change type weight
    @param type_id Type id
    @param weight New type weight
    """
    old_weight: uint256 = self._get_type_weight(type_id)
    old_sum: uint256 = self._get_sum(type_id)
    _total_weight: uint256 = self._get_total()
    next_time: uint256 = (block.timestamp + WEEK) / WEEK * WEEK

    _total_weight = _total_weight + old_sum * weight - old_sum * old_weight
    self.points_total[next_time] = _total_weight
    self.points_type_weight[type_id][next_time] = weight
    self.time_total = next_time
    self.time_type_weight[type_id] = next_time

    log NewTypeWeight(type_id, next_time, weight, _total_weight)
```

Listing 3.2: GaugeController.vy

However, any mis-configuration on the given weight may block the reward-claiming attempts of users who have staked on the affected `gauges` or `types`. If we use the `change_type_weight()` as an example, this issue is made possible if the `weight` amount is given as the argument to `_change_type_weight()` such that the calculation of `MULTIPLIER * _type_weight * _gauge_weight` always overflows, hence reverting every `gauge_relative_weight()` calculation of affected `gauges` in reward-claiming attempts. Note either only the specific `gauge` with the misconfigured oversized weight or all `gauges` sharing the same oversized `gauge type` will be affected.

```
def _gauge_relative_weight(addr: address, time: uint256) -> uint256:
    """
    @notice Get Gauge relative weight (not more than 1.0) normalized to 1e18
            (e.g. 1.0 == 1e18). Inflation which will be received by it is
            inflation_rate * relative_weight / 1e18
    @param addr Gauge address
    @param time Relative weight at the specified timestamp in the past or present
    @return Value of relative weight normalized to 1e18
    """
    # short circuit if single gauge and just give full weight
    if self.n_gauges == 1 and self.gauges[0] == addr:
        return MULTIPLIER
    t: uint256 = time / WEEK * WEEK
    _total_weight: uint256 = self.points_total[t]

    if _total_weight > 0:
        gauge_type: int128 = self.gauge_types_[addr] - 1
```

```
        _type_weight: uint256 = self.points_type_weight[gauge_type][t]
        _gauge_weight: uint256 = self.points_weight[addr][t].bias
        return MULTIPLIER * _type_weight * _gauge_weight / _total_weight
    else:
        return 0
```

Listing 3.3: GaugeController.vy

To mitigate, it is best to apply a threshold check on the allowed weight update on a current `gauge` or a supported `gauge type`. Specifically, we can define `TOTAL_WEIGHT_THRESHOLD` that aims to restrict the total weight calculated from all current `gauges`. Therefore, for any change on a `gauge` weight or a `type` weight, we can guarantee that the total weight is within an appropriate range. A candidate choice should be no larger than `TOTAL_WEIGHT_THRESHOLD: constant(uint256)= convert(-1, uint256)/ MULTIPLIER`.

Recommendation Add sanity checks to prevent the changed weight of a `gauge` or an existing `gauge type` from leading to an overflow calculation.

Status This issue has been confirmed. However, since the current protocol has been deployed, this specific issue is best mitigated through an off-chain vetting process to avoid configuring with an over-weighted number. Moreover, the team decides to address it in the next round of protocol development (or update).

3.3 Better Handling of Privilege Transfers

- ID: PVE-003
- Severity: Low
- Likelihood: N/A
- Impact: High
- Targets: `GaugeController`, `ERC20CRV`
- Category: Security Features [9]
- CWE subcategory: CWE-282 [4]

Description

The Swerve protocol implements a rather basic access control mechanism that allows a privileged account, i.e., `admin`, to be granted exclusive access to typically sensitive functions (e.g., setting the new `minter` and adding a new `gauge/type`). Because of the privileged access and the implications of these sensitive functions, the `admin` account is essential for the protocol-level safety and operation.

Within the contract `GaugeController`, a specific function, i.e., `commit_transfer_ownership(addr: address)`, is provided to allow for possible `admin` updates. However, current implementation achieves its goal by providing another companion function `apply_transfer_ownership()`. This is reasonable under the assumption that the `future_admin` parameter is always correctly provided. However, in the unlikely situation, when an incorrect `future_admin` is provided, the contract owner may be forever lost, which might be devastating for protocol-wide operation and maintenance.

As a common best practice, instead of achieving the owner update only from one party, i.e., the current `admin`, it is suggested to get both parties involved. For example, the first step is initiated by the current `admin` and the second step is initiated by the configured `future_admin` who accepts and materializes the update. Both steps should be executed in two separate transactions. By doing so, it can greatly alleviate the concern of accidentally transferring the contract ownership to an uncontrolled address. By doing so, we can guarantee that an owner public key cannot be nominated unless there is an entity that has the corresponding private key. This is explicitly designed to prevent unintentional errors in the owner transfer process.

```
@external
def commit_transfer_ownership(addr: address):
    """
    @notice Transfer ownership of GaugeController to `addr`
    @param addr Address to have ownership transferred to
    """
    assert msg.sender == self.admin  # dev: admin only
    self.future_admin = addr
    log CommitOwnership(addr)


@external
def apply_transfer_ownership():
    """
    @notice Apply pending ownership transfer
    """
    assert msg.sender == self.admin  # dev: admin only
    _admin: address = self.future_admin
    assert _admin != ZERO_ADDRESS  # dev: admin not set
    self.admin = _admin
    log ApplyOwnership(_admin)
```

Listing 3.4: GaugeController.vy

Recommendation Implement a two-step approach that involves actions from both relevant parties, i.e., `admin` and `future_admin`. In particular, the current `admin` initiates `commit_transfer_ownership()` and the intended `future_admin` executes `accept_transfer_ownership()`. Note that the current ownership transfer only involves actions from one party. The same is also applicable for other privileged accounts.

```
@external
def commit_transfer_ownership(addr: address):
    """
    @notice Transfer ownership of GaugeController to `addr`
    @param addr Address to have ownership transferred to
    """
    assert msg.sender == self.admin  # dev: admin only
```

```
    self.future_admin = addr
    log CommitOwnership(addr)


@external
def accept_transfer_ownership():
    """
    @notice Accept pending ownership transfer
    """
    assert msg.sender == self.future_admin  # dev: future_admin only
    assert _admin != ZERO_ADDRESS  # dev: admin not set
    self.admin = msg.sender
    log ApplyOwnership(msg.sender )
```

Listing 3.5: GaugeController.vy

Status This issue has been confirmed. However, considering the fact that the current protocol has been deployed and it is an operation-level issue, the team decides to address it in the future upgrade.

3.4 Unused Import/Code/Interface Removal

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: VotingEscrow, GaugeController
- Category: Coding Practices [10]
- CWE subcategory: CWE-561 [5]

Description

Swerve makes use of a number of reference contracts, such as ERC20, cERC20, USDT, Curve, and SmartWalletChecker, to facilitate the protocol implementation and organization. For instance, the VotingEscrow smart contract interacts with at least two other reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the VotingEscrow contract, the following import is not necessary: SmartWalletChecker.[1]. Therefore, the import of interface SmartWalletChecker can be safely removed.

```
# Interface for checking whether address belongs to a whitelisted
# type of a smart wallet.
# When new types are added - the whole contract is changed
# The check() method is modifying to be able to use caching
```

[1]The original CurveDAO codebase indeed makes use of SmartWalletChecker.

```
# for individual wallet addresses
interface SmartWalletChecker:
    def check(addr: address) -> bool: nonpayable
```

Listing 3.6: VotingEscrow.vy

In addition, we notice the `interface Controller` import in `GaugeController` can be more concise in only defining required interfaces, i.e., `gauge_relative_weight()`, `voting_escrow()`, and `checkpoint_gauge()`. Other unneeded interfaces can therefore be removed.

```
interface Controller:
    def period() -> int128: view
    def period_write() -> int128: nonpayable
    def period_timestamp(p: int128) -> uint256: view
    def gauge_relative_weight(addr: address, time: uint256) -> uint256: view
    def voting_escrow() -> address: view
    def checkpoint(): nonpayable
    def checkpoint_gauge(addr: address): nonpayable
```

Listing 3.7: GaugeController.vy

Another example is the `ZapDelegator` contract that defines an abstract contract `YToken`, which is not used. We also notice that the `token` variable in `GaugeController` is unused even it is provided at construction. From the software engineering perspective, we normally recommend removing unused code. However, we suggest to retain this particular unused `token` as it could provide additional context information on the operated governance token.

Recommendation Remove unnecessary imports of reference contracts and revise existing ones if necessary.

Status This issue has been confirmed. Since the current protocol has been deployed and these unused interfaces and code do not affect the protocol operation in any way, the team decides to leave it as is for the time being.

3.5 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `LiquidityGauge`, `VotingEscrow`
- Category: Business Logics [11]
- CWE subcategory: CWE-841 [8]

Description

In Swerve, the `LiquidityGauge` contract is designed to be the main entry for interaction with farming users. In particular, one entry routine, i.e., `deposit()`, accepts user deposits of supported assets (e.g., swUSD). Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the `LiquidityGauge` contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contracts.

```
def deposit(_value: uint256, addr: address = msg.sender):
    """
    @notice Deposit `_value` LP tokens
    @param _value Number of tokens to deposit
    @param addr Address to deposit for
    """
    if addr != msg.sender:
        assert self.approved_to_deposit[msg.sender][addr], "Not approved"

    self._checkpoint(addr)

    if _value != 0:
        _balance: uint256 = self.balanceOf[addr] + _value
        _supply: uint256 = self.totalSupply + _value
        self.balanceOf[addr] = _balance
        self.totalSupply = _supply

        self._update_liquidity_limit(addr, _balance, _supply)

        assert ERC20(self.lp_token).transferFrom(msg.sender, self, _value)

    log Deposit(addr, _value)


@external
@nonreentrant('lock')
def withdraw(_value: uint256):
    """
    @notice Withdraw `_value` LP tokens
    @param _value Number of tokens to withdraw
    """
    self._checkpoint(msg.sender)

    _balance: uint256 = self.balanceOf[msg.sender] - _value
    _supply: uint256 = self.totalSupply - _value
    self.balanceOf[msg.sender] = _balance
    self.totalSupply = _supply

    self._update_liquidity_limit(msg.sender, _balance, _supply)

    assert ERC20(self.lp_token).transfer(msg.sender, _value)

    log Withdraw(msg.sender, _value)
```

Listing 3.8: LiquidityGauge.vy

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every transfer() or transferFrom(). (Another type is rebasing tokens such as YAM.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as deposit() and withdraw(), may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of Swerve and affects protocol-wide operation and maintenance. A similar issue can also be found in VotingEscrow.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in transfer() or transferFrom() will always result in full transfer, we need to ensure the increased or decreased amount in the gauge before and after the transfer() or transferFrom() is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into Swerve. In Swerve, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., USDT) that may have control switches that can be dynamically exercised to suddenly become one.

We emphasize that the current deployment of LiquidityGauge and VotingEscrow are safe as they use swUSD and SWRV for deposits and withdrawals. (And both assets are not deflationary or rebasing.) However, the current code implementation is generic in supporting various tokens and there is a need to highlight the possible pitfall from the audit perspective. Also, current codebase is not compatible with other non-compliant ERC20 tokens, including BNB.[2]

Recommendation If current codebase needs to support possible deflationary tokens, it is better to check the balance before and after the transfer()/transferFrom() call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted USDT.

Status This issue has been confirmed. However, considering the fact that the current protocol has been deployed and this specific issue does not affect the normal operation, the team decides to address it when the need of supporting deflationary/rebasing tokens arises.

[2]The use of BNB in LiquidityGauge and VotingEscrow may result in funds being locked and unrecoverable!

3.6 Improved Binary Search in find_block_epoch()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: VotingEscrow
- Category: Business Logics [11]
- CWE subcategory: CWE-841 [8]

Description

Following CurveDAO, the SwerveDAO takes the approach of measuring the vote in a amount-time-weighted manner where the time counted is the time left to unlock, i.e., how long the tokens cannot be moved in the future. (Note the maximum selectable locktime is 4 years.)

Because of the above vote measurement, it becomes necessary to measure current voting power at a particular block and convert from a block number to the corresponding timestamp. To this end, the current codebase takes a binary search algorithm to estimate the related timestamp for a given block number. To elaborate, we show below the related code snippet of the binary search routine, i.e., find_block_epoch().

The routine implements a rather standard binary search algorithm and we find the current implementation can be (slightly) improved by iterating one round less. Specifically, if the comparison with current _block (line 475) shows it is identical with the _mid block number, we can simply return _mid, hence allowing for early termination of the iteration.

```
461 def find_block_epoch(_block: uint256, max_epoch: uint256) -> uint256:
462     """
463     @notice Binary search to estimate timestamp for block number
464     @param _block Block to find
465     @param max_epoch Don't go beyond this epoch
466     @return Approximate timestamp for block
467     """
468     # Binary search
469     _min: uint256 = 0
470     _max: uint256 = max_epoch
471     for i in range(128):  # Will be always enough for 128-bit numbers
472         if _min >= _max:
473             break
474         _mid: uint256 = (_min + _max + 1) / 2
475         if self.point_history[_mid].blk <= _block:
476             _min = _mid
477         else:
478             _max = _mid - 1
479     return _min
```

Listing 3.9: VotingEscrow.vy

In addition, the `balanceOfAt()` routine has an internal `for` loop that can be similarly optimized. Moreover, we notice that the internal `for` loop has an upper bound of at most 128 times. This number can be reduced to 30 (`VotingEscrow.vy` at line 520). The reason is that `user_point_history` holds at most $1,000,000,000$ points and $1,000,000,000 < 2^{30}$. In the same vein, the `for` loop in `find_block_epoch()` can set the upper limit of 100, instead of current 128.

Recommendation Optimize the `find_block_epoch()` implementation as shown below. Note that a similar optimization is also applicable to the `balanceOfAt()` implementation.

```
def find_block_epoch(_block: uint256, max_epoch: uint256) -> uint256:
    """
    @notice Binary search to estimate timestamp for block number
    @param _block Block to find
    @param max_epoch Don't go beyond this epoch
    @return Approximate timestamp for block
    """
    # Binary search
    _min: uint256 = 0
    _max: uint256 = max_epoch
    _tmp_block: uint256 = 0
    for i in range(100):  # Will be always enough for 128-bit numbers
        if _min >= _max:
            break

        _mid: uint256 = (_min + _max + 1) / 2
        _tmp_block = self.point_history[_mid].blk

        if _tmp_block == _block:
            return _mid
        elif _tmp_block < _block:
            _min = _mid
        else:
            _max = _mid - 1
    return _min
```

Listing 3.10: VotingEscrow.vy (revised)

Status This issue has been confirmed. However, for the same reason as outlined in Section 3.5, the team decides to address it in the future iteration of development.

3.7 Implicit Threshold On Supported Distinct Gauge Types

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: GaugeController
- Category: Business Logics [11]
- CWE subcategory: CWE-837 [7]

Description

In Swerve, there is an implicit restriction on the number of gauge types that can be supported. However, this restriction is not enforced when a new gauge type is being added. As a result, if a new gauge type is assigned with an type id that exceeds the limit, the new gauge type as well as all gauges of this gauge type will not be able to participate in the governance token distribution.

To elaborate, we show below the _get_total() routine that is responsible for calculating and maintaining the total weighted-sum of all current gauges. For each gauge, its weight is counted by multiplying the gauge weight with the corresponding gauge type weight.

```
def _get_total() -> uint256:
    """
    @notice Fill historic total weights week-over-week for missed checkins
            and return the total for the future week
    @return Total weight
    """
    t: uint256 = self.time_total
    _n_gauge_types: int128 = self.n_gauge_types
    if t > block.timestamp:
        # If we have already checkpointed - still need to change the value
        t -= WEEK
    pt: uint256 = self.points_total[t]

    for gauge_type in range(100):
        if gauge_type == _n_gauge_types:
            break
        self._get_sum(gauge_type)
        self._get_type_weight(gauge_type)

    for i in range(500):
        if t > block.timestamp:
            break
        t += WEEK
        pt = 0
        # Scales as n_types * n_unchecked_weeks (hopefully 1 at most)
        for gauge_type in range(100):
            if gauge_type == _n_gauge_types:
                break
            type_sum: uint256 = self.points_sum[gauge_type][t].bias
```

```
249             type_weight: uint256 = self.points_type_weight[gauge_type][t]
250             pt += type_sum * type_weight
251         self.points_total[t] = pt

253         if t > block.timestamp:
254             self.time_total = t
255     return pt
```

Listing 3.11: GaugeController.vy

Apparently, as shown in the line 233, only the first 100 `gauge types` are taken into consideration, excluding all other `gauge types` and their `gauges` from participating in the distribution of governance tokens.

```
@external
def add_type(_name: String[64], weight: uint256 = 0):
    """
    @notice Add gauge type with name `_name` and weight `weight`
    @param _name Name of gauge type
    @param weight Weight of gauge type
    """
    assert msg.sender == self.admin
    type_id: int128 = self.n_gauge_types
    self.gauge_type_names[type_id] = _name
    self.n_gauge_types = type_id + 1
    if weight != 0:
        self._change_type_weight(type_id, weight)
        log AddType(_name, type_id)
```

Listing 3.12: GaugeController.vy

Meanwhile, the `add_type()` routine that handles the addition of new types is not enforcing the above (implicit) limit. With that, it is strongly suggested to define the `MAX_GAUGE_TYPES` and make the limit explicit. This explicit limit is necessary as we observe blurred or confused declaration of the number of `gauge types` reflected in other data structures. For example, both `time_sum` and `time_type_weight` denote the mapping from a specific `gauge type` to the last scheduled time of all `gauges` of the same type and the type weight respectively. The current declarations (lines 103 and 109) misleadingly indicate the protocol support 1,000,000,000 types!

By having the explicit limit, we can re-define both `time_sum` and `time_type_weight` in an unambiguous manner that greatly reduces the storage reservation from 1,000,000,000 to 100, i.e., `time_sum: public(uint256[100])` and `time_type_weight: public(uint256[100])`.

```
103 time_sum: public(uint256[1000000000]) # type_id -> last scheduled time (next week)

105 points_total: public(HashMap[uint256, uint256]) # time -> total weight
106 time_total: public(uint256) # last scheduled time

108 points_type_weight: public(HashMap[int128, HashMap[uint256, uint256]]) # type_id ->
    time -> type weight
```

```
time_type_weight: public(uint256[1000000000]) # type_id -> last scheduled time (next
    week)
```

Listing 3.13: GaugeController.vy

Recommendation Explicitly limit the number of gauge types that can be supported in the protocol and enforce the limit when a new type is being added.

```
MAX_GAUGE_TYPES: constant(uint256) = 100

@external
def add_type(_name: String[64], weight: uint256 = 0):
    """
    @notice Add gauge type with name `_name` and weight `weight`
    @param _name Name of gauge type
    @param weight Weight of gauge type
    """
    assert msg.sender == self.admin

    type_id: int128 = self.n_gauge_types
    assert type_id < MAX_GAUGE_TYPES

    self.gauge_type_names[type_id] = _name
    self.n_gauge_types = type_id + 1
    if weight != 0:
        self._change_type_weight(type_id, weight)
        log AddType(_name, type_id)
```

Listing 3.14: GaugeController.vy

Status This issue has been confirmed. However, for the same reason as outlined in Section 3.5, the team decides to address it in the future iteration of development.

3.8 Lack of Rigorous Sanity Checks Against Gauge/Type Weight Updates

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: GaugeController
- Category: Coding Practices [10]
- CWE subcategory: CWE-1099 [2]

Description

The distribution of SWRV governance tokens requires proper setup of participating gauges, gauge types as well as their respective weights. The share of each gauge is proportional to each gauge weight multiplied with the `gauge type` weight and then divided by the total weighted sum (Section 3.7).

In this section, we examine the logic related to the updates to these weights. Our result shows the update logic can be improved by applying more rigorous sanity checks. Based on current implementation, certain corner cases may be exploited to lead to undesirable consequences, including reporting a lower `gauge_relative_weight()` and a higher `get_total_weight()`. These two routines are essential for the calculation of `gauge` proportions for reward distribution.

To elaborate, we show its code snippet below of two essential functions, i.e., `_change_type_weight()` and `_change_gauge_weight()`. These two functions handles the weight updates to `gauges` and `gauge types`, respectively. Both routines do not validate the given `gauge` or `gauge type` as part of input arguments.

```
def _change_type_weight(type_id: int128, weight: uint256):
    """
    @notice Change type weight
    @param type_id Type id
    @param weight New type weight
    """
    old_weight: uint256 = self._get_type_weight(type_id)
    old_sum: uint256 = self._get_sum(type_id)
    _total_weight: uint256 = self._get_total()
    next_time: uint256 = (block.timestamp + WEEK) / WEEK * WEEK

    _total_weight = _total_weight + old_sum * weight - old_sum * old_weight
    self.points_total[next_time] = _total_weight
    self.points_type_weight[type_id][next_time] = weight
    self.time_total = next_time
    self.time_type_weight[type_id] = next_time

    log NewTypeWeight(type_id, next_time, weight, _total_weight)
```

Listing 3.15: GaugeController.vy

```
def _change_gauge_weight(addr: address, weight: uint256):
    # Change gauge weight
    # Only needed when testing in reality
    gauge_type: int128 = self.gauge_types_[addr] - 1
    old_gauge_weight: uint256 = self._get_weight(addr)
    type_weight: uint256 = self._get_type_weight(gauge_type)
    old_sum: uint256 = self._get_sum(gauge_type)
    _total_weight: uint256 = self._get_total()
    next_time: uint256 = (block.timestamp + WEEK) / WEEK * WEEK

    self.points_weight[addr][next_time].bias = weight
    self.time_weight[addr] = next_time

    new_sum: uint256 = old_sum + weight - old_gauge_weight
    self.points_sum[gauge_type][next_time].bias = new_sum
    self.time_sum[gauge_type] = next_time

```

```
468     _total_weight = _total_weight + new_sum * type_weight - old_sum * type_weight
469     self.points_total[next_time] = _total_weight
470     self.time_total = next_time
471
472     log NewGaugeWeight(addr, block.timestamp, weight, _total_weight)
```

Listing 3.16: GaugeController.vy

Without validating the `type_id`, it is possible to assign the weight of the `minusOne` (or -1) `gauge type`. Later on, if an unregistered `gauge` is updated, the `_change_gauge_weight()` may eventually contaminate the calculation of `points_total[next_time]` and `time_total` (lines 469 – 470). We have not exhaustively searched through all possible exploitations. However, the lack of thorough validation itself is worrisome and we strongly apply necessary sanity checks to block updating invalid `gauges` and `gauge types`.

Similarly, we can improve the validation of `vote_for_gauge_weights()` by ensuring `assert (gauge_type >= 0)and (gauge_type < self.n_gauge_types)`, instead of current `assert gauge_type >= 0, "Gauge not added"` (line 503). Also, enhance `_gauge_relative_weight()` to return 0 for a given `gauge` but with an invalid `gauge_type` (line 363).

Last but not least, it is also suggested to enhance the `add_gauge()` logic by ensuring the total number of `gauges` is no more than $1,000,000,000$ – the hard-coded limit in the system. As already suggested in Section 3.7, the `add_type()` logic needs to be revised by ensuring the total number of `gauge types` is no more than 100 – an implicit limit in the system.

Recommendation Validate the given `gauge` address or the `gauge type` before updating their weights in the system. An example validation is shown below.

```
def _change_type_weight(type_id: int128, weight: uint256):
    """
    @notice Change type weight
    @param type_id Type id
    @param weight New type weight
    """
    assert (type_id >= 0) and (type_id < self.n_gauge_types)
    old_weight: uint256 = self._get_type_weight(type_id)
    old_sum: uint256 = self._get_sum(type_id)
    _total_weight: uint256 = self._get_total()
    next_time: uint256 = (block.timestamp + WEEK) / WEEK * WEEK

    _total_weight = _total_weight + old_sum * weight - old_sum * old_weight
    self.points_total[next_time] = _total_weight
    self.points_type_weight[type_id][next_time] = weight
    self.time_total = next_time
    self.time_type_weight[type_id] = next_time

    log NewTypeWeight(type_id, next_time, weight, _total_weight)
```

Listing 3.17: GaugeController.vy (revised)

```
def _change_gauge_weight(addr: address, weight: uint256):
    # Change gauge weight
    # Only needed when testing in reality
    gauge_type: int128 = self.gauge_types_[addr] - 1
    assert (gauge_type >= 0) and (gauge_type < self.n_gauge_types)
    old_gauge_weight: uint256 = self._get_weight(addr)
    type_weight: uint256 = self._get_type_weight(gauge_type)
    old_sum: uint256 = self._get_sum(gauge_type)
    _total_weight: uint256 = self._get_total()
    next_time: uint256 = (block.timestamp + WEEK) / WEEK * WEEK

    self.points_weight[addr][next_time].bias = weight
    self.time_weight[addr] = next_time

    new_sum: uint256 = old_sum + weight - old_gauge_weight
    self.points_sum[gauge_type][next_time].bias = new_sum
    self.time_sum[gauge_type] = next_time

    _total_weight = _total_weight + new_sum * type_weight - old_sum * type_weight
    self.points_total[next_time] = _total_weight
    self.time_total = next_time

    log NewGaugeWeight(addr, block.timestamp, weight, _total_weight)
```

Listing 3.18: GaugeController.vy (revised)

Status This issue has been confirmed. The teams plans to add necessary validation logics in the next iteration of development.

3.9 Improved AddType() Event Generation

- ID: PVE-009
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `GaugeController`
- Category: Status Codes [12]
- CWE subcategory: CWE-682 [6]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools.

`Events` can be emitted in a number of scenarios, e.g., when updating system-wide parameters or adding new components. For example, Swerve defines new `gauges` and `types`. However, the current implementation can be improved by correctly emitting related events when they are being changed.

In the following, we use the `AddType` event as an example. This event is defined in the `GaugeController` contract and represents the state of adding a new `gauge type`.

```
423 @external
424 def add_type(_name: String[64], weight: uint256 = 0):
425     """
426     @notice Add gauge type with name `_name` and weight `weight`
427     @param _name Name of gauge type
428     @param weight Weight of gauge type
429     """
430     assert msg.sender == self.admin
431     type_id: int128 = self.n_gauge_types
432     self.gauge_type_names[type_id] = _name
433     self.n_gauge_types = type_id + 1
434     if weight != 0:
435         self._change_type_weight(type_id, weight)
436         log AddType(_name, type_id)
```

Listing 3.19: GaugeController.vy

However, we notice that this event is not emitted if `weight==0` (line 434). It may cause issues for off-chain components to monitor the set of `gauge types` being supported in the system. Moreover, the event can be improved by encoding the weight information as well, which is currently missing.

In the same vein, we suggest to refine the `NewGauge`, `NewGaugeWeight`, and `VoteForGauge` events by indexing the related `gauge_address`. In addition, we can enhance the `Minted` event by encoding `to_mint` as well. By doing so, we can better facilitate off-chain analytics and reporting tools.

Recommendation Emit necessary events to timely reflect system dynamics.

```
423 @external
424 def add_type(_name: String[64], weight: uint256 = 0):
425     """
426     @notice Add gauge type with name `_name` and weight `weight`
427     @param _name Name of gauge type
428     @param weight Weight of gauge type
429     """
430     assert msg.sender == self.admin
431     type_id: int128 = self.n_gauge_types
432     self.gauge_type_names[type_id] = _name
433     self.n_gauge_types = type_id + 1
434     if weight != 0:
435         self._change_type_weight(type_id, weight)
436     log AddType(_name, type_id, weight)
```

Listing 3.20: GaugeController.vy

Status This issue has been confirmed. The teams plans to emit the above events in the next iteration of development.

3.10 Improved Precision By Multiplication-Before-Division

- ID: PVE-010
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `LiquidityGauge`
- Category: Numeric Errors [14]
- CWE subcategory: CWE-190 [3]

Description

`SafeMath` is a widely-used `Solidity` `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. The `Vyper` language has built-in support of performing bounds and overflow checking on array accesses and arithmetic operations. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (`mul`) and division (`div`) are involved.

In particular, we use the `_update_liquidity_limit()` as an example. This routine is used to incentivize users to participate in governance by allowing voting users to get boosted in their weight calculation.

```
@internal
def _update_liquidity_limit(addr: address, l: uint256, L: uint256):
    """
    @notice Calculate limits which depend on the amount of CRV token per-user.
            Effectively it calculates working balances to apply amplification
            of CRV production by CRV
    @param addr User address
    @param l User's amount of liquidity (LP tokens)
    @param L Total amount of liquidity (LP tokens)
    """
    # To be called after totalSupply is updated
    _voting_escrow: address = self.voting_escrow
    voting_balance: uint256 = ERC20(_voting_escrow).balanceOf(addr)
    voting_total: uint256 = ERC20(_voting_escrow).totalSupply()

    lim: uint256 = l * TOKENLESS_PRODUCTION / 100
    if (voting_total > 0) and (block.timestamp > self.period_timestamp[0] + BOOST_WARMUP
        ):
        lim += L * voting_balance / voting_total * (100 - TOKENLESS_PRODUCTION) / 100

    lim = min(l, lim)
    old_bal: uint256 = self.working_balances[addr]
    self.working_balances[addr] = lim
    _working_supply: uint256 = self.working_supply + lim - old_bal
    self.working_supply = _working_supply
```

```
137     log UpdateLiquidityLimit(addr, l, L, lim, _working_supply)
```

Listing 3.21: LiquidityGauge.vy

We notice the calculation of the boosted amount of `lim` (line 129) involves both multiplication and division: `L * voting_balance / voting_total * (100 - TOKENLESS_PRODUCTION)/ 100` (line 129). For improved precision, it is better to calculate the multiplication before the division, i.e., `lim += L * voting_balance * (100 - TOKENLESS_PRODUCTION)/ (voting_total * 100)`. Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible.

Recommendation Revise the above calculations as follows to better mitigate possible precision loss.

```
112 @internal
113 def _update_liquidity_limit(addr: address, l: uint256, L: uint256):
114     """
115     @notice Calculate limits which depend on the amount of CRV token per-user.
116             Effectively it calculates working balances to apply amplification
117             of CRV production by CRV
118     @param addr User address
119     @param l User's amount of liquidity (LP tokens)
120     @param L Total amount of liquidity (LP tokens)
121     """
122     # To be called after totalSupply is updated
123     _voting_escrow: address = self.voting_escrow
124     voting_balance: uint256 = ERC20(_voting_escrow).balanceOf(addr)
125     voting_total: uint256 = ERC20(_voting_escrow).totalSupply()

127     lim: uint256 = l * TOKENLESS_PRODUCTION / 100
128     if (voting_total > 0) and (block.timestamp > self.period_timestamp[0] + BOOST_WARMUP
            ):
129         lim += L * voting_balance * (100 - TOKENLESS_PRODUCTION) / (voting_total * 100)

131     lim = min(l, lim)
132     old_bal: uint256 = self.working_balances[addr]
133     self.working_balances[addr] = lim
134     _working_supply: uint256 = self.working_supply + lim - old_bal
135     self.working_supply = _working_supply

137     log UpdateLiquidityLimit(addr, l, L, lim, _working_supply)
```

Listing 3.22: LiquidityGauge.vy

Status This issue has been confirmed. The teams plans to address this issue in the next iteration of development.

3.11 Better Consistency Between deposit() and withdraw()

- ID: PVE-011
- Severity: Informational
- Likelihood: Informational
- Impact: Informational
- Target: `LiquidityGauge`
- Category: Coding Practices [10]
- CWE subcategory: CWE-1099 [2]

Description

In Section 3.5, we have examined the two important routines in `LiquidityGauge`: i.e., `deposit()` and `withdraw()`. In particular, the first routine accepts user deposits of supported assets (e.g., `swUSD`) while the second routine allows for withdrawals of the deposited assets.

We notice that `deposit()` optimizes the handling of a corner case, i.e., when the given `_value` is 0. This optimization has certain benefits in reducing gas cost of the execution path. However, this optimization can be similarly applied to the `withdraw()` routine (that does not explicitly optimize the handling of this corner case). This avoids unnecessary gas cost and achieves better consistency between `deposit()` and `withdraw()`.

```
@external
@nonreentrant('lock')
def deposit(_value: uint256, addr: address = msg.sender):
    """
    @notice Deposit `_value` LP tokens
    @param _value Number of tokens to deposit
    @param addr Address to deposit for
    """
    if addr != msg.sender:
        assert self.approved_to_deposit[msg.sender][addr], "Not approved"

    self._checkpoint(addr)

    if _value != 0:
        _balance: uint256 = self.balanceOf[addr] + _value
        _supply: uint256 = self.totalSupply + _value
        self.balanceOf[addr] = _balance
        self.totalSupply = _supply

        self._update_liquidity_limit(addr, _balance, _supply)

        assert ERC20(self.lp_token).transferFrom(msg.sender, self, _value)

    log Deposit(addr, _value)


@external
@nonreentrant('lock')
```

```
def withdraw(_value: uint256):
    """
    @notice Withdraw `_value` LP tokens
    @param _value Number of tokens to withdraw
    """
    self._checkpoint(msg.sender)

    _balance: uint256 = self.balanceOf[msg.sender] - _value
    _supply: uint256 = self.totalSupply - _value
    self.balanceOf[msg.sender] = _balance
    self.totalSupply = _supply

    self._update_liquidity_limit(msg.sender, _balance, _supply)

    assert ERC20(self.lp_token).transfer(msg.sender, _value)

    log Withdraw(msg.sender, _value)
```

Listing 3.23: LiquidityGauge.vy

Recommendation Ensure the consistency of the underlying logic of processing `deposit()` and `withdraw()`.

```
@external
@nonreentrant('lock')
def withdraw(_value: uint256):
    """
    @notice Withdraw `_value` LP tokens
    @param _value Number of tokens to withdraw
    """
    self._checkpoint(msg.sender)

    if _value != 0:
      _balance: uint256 = self.balanceOf[msg.sender] - _value
      _supply: uint256 = self.totalSupply - _value
      self.balanceOf[msg.sender] = _balance
      self.totalSupply = _supply

      self._update_liquidity_limit(msg.sender, _balance, _supply)

      assert ERC20(self.lp_token).transfer(msg.sender, _value)

    log Withdraw(msg.sender, _value)
```

Listing 3.24: LiquidityGauge.vy

Status This issue has been confirmed. The teams plans to address this issue in the next iteration of development.

3.12 Viewable Adjustment of claimable_tokens()

- ID: PVE-012
- Severity: Informational
- Likelihood: Informational
- Impact: Informational
- Target: LiquidityGauge
- Category: Expression Issues [13]
- CWE subcategory: N/A

Description

The LiquidityGauge contract defines a querier, i.e., claimable_tokens, on the number of claimable tokens for a particular user. This routine is useful for direct hands-on manual queries (from the etherscan.io website) or programmatic third-party integrations with other DeFi protocols, including various DEXes.

However, this particular querier requires taking a checkpoint on the internal bookkeeping maintained in the LiquidityGauge contract, which makes the function in the automatically generated ABI file not viewable. However, the accompanying comment above the code suggests to manually change the modifier in the ABI to be a view function. This is indeed a great suggestion that allows a farmer (with minimal web-surfing background) to be able to query the claimable tokens without making an online transaction.

```
@external
def claimable_tokens(addr: address) -> uint256:
    """
    @notice Get the number of claimable tokens per user
    @dev This function should be manually changed to "view" in the ABI
    @return uint256 number of claimable tokens per user
    """
    self._checkpoint(addr)
    return self.integrate_fraction[addr] - Minter(self.minter).minted(addr, self)
```

Listing 3.25: LiquidityGauge.vy

Recommendation Manually adjust the claimable_tokens() function and make it viewable.

Status This issue has been confirmed. However, considering the fact that the current protocol has been deployed, the team decides to make the viewable adjustment in the next deployment or upgrade when such opportunity arises.

4 Conclusion

In this audit, we thoroughly analyzed the design and implementation of the Swerve protocol. The system presents a community fork of `Curve` with the removal of questionable pre-mines and other stakeholder/team allocation of governance tokens. By taking a proxy-based architectural approach, Swerve readily reuses the `Curve` protocol with its own `SwerveDAO`, which is itself a fork of `CurveDAO` with different pool/weight setups and varying distribution schedules of governance tokens.

As a final precaution, 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.

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [17, 18, 19, 20, 22].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [23] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

5.1.12 `Send` Instead Of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

[2] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

[3] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[4] MITRE. CWE-282: Improper Ownership Management. https://cwe.mitre.org/data/definitions/282.html.

[5] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

[6] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[7] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

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

[9] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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

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

[12] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[13] MITRE. CWE CATEGORY: Expression Issues. https://cwe.mitre.org/data/definitions/569.html.

[14] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[15] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[16] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[17] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[18] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[19] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[20] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

[21] PeckShield. PeckShield Inc. https://www.peckshield.com.

[22] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[23] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.