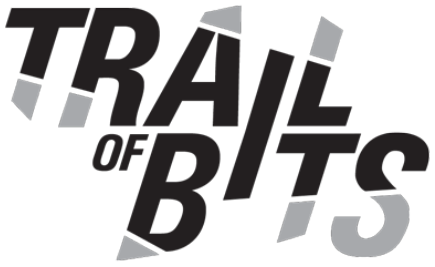

Curve DAO

Security Assessment

July 10th 2020

Prepared For:
Michael Egorov | *Swiss Stake*
michael@swiss-stake.com

Prepared By:
Josselin Feist | *Trail of Bits*
josselin@trailofbits.com

Gustavo Grieco | *Trail of Bits*
gustavo.grieco@trailofbits.com

Michael Colburn | *Trail of Bits*
michael.colburn@trailofbits.com

Executive Summary

Project Dashboard

Code Maturity Evaluation

Engagement Goals

Coverage

Recommendations Summary
- Short term
- Long term

Findings Summary
- 1. LiquidityGauge does not account for VotingEscrow's balance updates
- 2. LiquidityGauge does not account for VotingEscrow's totalSupply updates
- 3. Early users will have a unfair advantage
- 4. GaugeController allows for quick vote and withdraw voting strategy
- 5. Adding the same gauge multiple times will lead to incorrect sum of weights
- 6. Spam attack would prevent LiquidityGauge's integral from being updated
- 7. Minter user can confiscate any user tokens
- 8. Mint and Burn events cannot be trusted
- 9. VotingEscrow's Admin can take whitelisted accounts hostage
- 10. ERC20CRV is not initiated correctly with large name and symbol
- 11. Lack of two-step procedure for critical operations is error-prone
- 12. Lack of value verification on decimals is error-prone
- 13. Lack of events is error-prone
- 14. Race condition in removing addresses from whitelist and withdrawing
- 15. Lack of documentation is error-prone
- 16. VotingEscrow's balanceOfAt and totalSupplyAt can return different values for the same block
- 17. No incentive to vote early in GaugeController
- 18. Several loops are not executable due to gas limitation
- 19. Testing smart contract code in Brownie can be unreliable
- 20. Aragon's voting does not follow voting best practices
- 21. Race condition may result in users earning less interest than expected

A. Vulnerability Classifications

B. Code Maturity Classifications

C. Code Quality

D. Token Integration Checklist
ERC Conformity
Contract Composition
Owner privileges
Token Scarcity

E. Fix Log
Detailed Fix Log

Executive Summary

From June 22 through July 10, 2020, Swiss-Stake engaged Trail of Bits to review the security of Curve DAO. We conducted this assessment over the course of six person-weeks with three engineers working from `f1c8f43` from the `curve-dao-contracts` repository.

In the first two weeks, we focused on understanding the codebase and reviewing the contracts against the most common smart contract flaws. In the final week, we reviewed the checkpoint functions and `LiquidityGauge` bookkeeping, and looked for corner cases in the most complex contract's interactions.

Our review resulted in 21 findings ranging from high to informational severity. The most significant findings are related to incorrect updating of the `LiquidityGauge` bonus, which can allow attackers to earn unfair interest. Moreover, we found that the code would benefit from better documentation, function composition, and code readability. We also found potential risks related to out-of-gas consumption, and external risk introduced by the use of Aragon's contracts. See additional code quality issues in Appendix C, and see recommendations to follow when adding arbitrary tokens in Appendix D.

Overall, the codebase meets most of its security expectations. A significant effort has been made to identify potential risks and to develop suitable mitigations and tests. However, the codebase is very complex, numerous behaviors are not documented, and the arithmetic operations would benefit from high-level clarifications.

Moving forward, Trail of Bits recommends addressing the findings presented and increasing the documentation. Curve Dao must be careful with the deployment of the contracts and the interactions of its early users and their advantages. We also recommend considering an alternative to the Aragon voting contract. Finally, we recommend performing an economic assessment to make sure the monetary incentives are properly designed.

Project Dashboard

Application Summary

Name	Curve Dao
Version	f1c8f43
Type	Vyper contracts
Platforms	Ethereum

Engagement Summary

Dates	June 22–July 10
Method	Whitebox
Consultants Engaged	3
Level of Effort	6 person-weeks

Vulnerability Summary

Total High-Severity Issues	4	■ ■ ■ ■
Total Medium-Severity Issues	8	■ ■ ■ ■ ■ ■ ■ ■
Total Low-Severity Issues	4	■ ■ ■ ■
Total Informational-Severity Issues	4	■ ■ ■ ■
Total Undetermined-Severity Issues	1	■
Total	21	

Category Breakdown

Access Controls	2	■ ■
Auditing and Logging	3	■ ■ ■
Data Validation	13	■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■
Patching	1	■
Timing	2	■ ■
Total	21	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Satisfactory.** The codebase has a strong access control mechanism and we found only minor concerns.
Arithmetic	**Moderate.** The system relies on complex arithmetic. While the use of Vyper prevents overflow and underflow flaws, we found several issues related to interest computation.
Centralization	**Moderate.** The contracts' owners have significant privileges. Additionally, the deployer of `ERC20CRV` will own all the tokens at deployment and will have a significant advantage.
Upgradeability	**Not Applicable.**
Function Composition	**Moderate.** Some components are written multiple times, and the codebase would benefit from greater code reuse.
Front-Running	**Satisfactory.** Most functions are not impacted by front-running, or the impact is expected. We found only one minor issue.
Monitoring	**Weak.** We found that Mint and Burn events could be compromised. Additionally, several components do not emit events. Finally, we were not aware of any off-chain components that monitor the contracts.
Specification	**Moderate.** The provided documentation omitted several behaviors, and the codebase would benefit from more thorough documentation.
Testing & Verification	**Moderate.** The codebase has several unit tests, but it is missing gas evaluation. No verification of code was present.

Engagement Goals

The engagement was scoped to provide a security assessment of Curve DAO protocol smart contracts in the `curve-dao-contracts` repository.

Specifically, we sought to answer the following questions:

- Are appropriate access controls set for the admin/user roles?
- Does arithmetic for internal bookkeeping operations hold?
- Is there any arithmetic overflow or underflow affecting the code?
- Can participants manipulate or block gauge or voting operations?
- Is it possible to manipulate gauges or voting by front-running transactions?
- Is it possible for participants to steal or lose tokens?
- Can participants perform denial-of-service attacks against any of the gauges or voting escrow?

Coverage

The engagement focused on the following components:

- **Liquidity gauges:** These allow users to deposit liquidity using different ERC20 tokens and get CRV tokens based on the amount locked and other factors. We reviewed the contract's interactions with users depositing liquidity to ensure proper behavior. We looked for flaws that would allow an attacker to withdraw more than deposited and prevent users from withdrawing their assets. We also focused on interest rate computation and history catch-up.
- **Controller gauge:** Liquidity gauges are created and managed by a special contract called the controller gauge. We reviewed the access control of this contract as well as interaction with the gauges once deployed. We looked for flaws in voting and checked for the proper increase of period and epoch.
- **Voting escrow:** Once users deposit liquidity, they can use mint tokens locked for a period of time in the voting escrow contract. We reviewed the consistency and the corner cases in computation of weights and verified that the locks are held in each case. We looked for flaws that would allow an attacker to unlock a deposit early, withdraw more than deposited, or prevent users from withdrawing their deposits.
- **CRV Token and Minter:** The liquidity gauge mints a CRV token every time it adds liquidity to the gauge. This contract implements a standard ERC20 token. We verified that all the expected properties are correctly implemented. We also looked for flaws that would allow a minter to mint more than the time-limited supply, and we reviewed the CRV token for its conformity to the ERC20 standard.
- **Access controls.** Many parts of the system expose privileged functionality, i.e., setting protocol parameters or managing gauges. We reviewed these functions to ensure they can only be triggered by the intended actors and that they do not contain unnecessary privileges that may be abused.
- **Arithmetic.** We reviewed calculations for logical consistency, as well as rounding issues and scenarios where reverts due to overflow may negatively impact use of the protocol.

We briefly reviewed the Curve DAO external interactions with the Aragon contracts, however, their upgradability and external dependency risks were considered out of scope.

Additionally, we briefly reviewed the Airdrop contract and looked for the most common smart contract flaws.

Off-chain code components were outside the scope of this assessment.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short term

❑ **Prevent users from earning interest after their `VotingEscrow` lock expires.** Consider either:

- Removing the bonus based on the locked tokens, or
- Adding watchers that will penalize users cheating the system, or
- Integrating the locking end time in the bonus computation.

Users are able to sell or re-lock expired tokens while still earning interest for these tokens (TOB-CURVE-DAO-001).

❑ **Since `VotingEscrow`'s total supply constantly changes in the interest rate bonus, consider either:**

- **Removing the bonus based on locked tokens, or**
- **Updating the formulas to take updates of the total supply into account.**

The interest rate percentage is based on `VotingEscrow`'s total supply, which changes constantly, and users can game the system to earn more of a bonus than expected (TOB-CURVE-DAO-002).

❑ **Since early users will have an unfair interest rate advantage, consider either:**

- **Removing the bonus based on the locked tokens, or**
- **Clearly documenting that early users have a system advantage.**

Any user advantage must be properly considered (TOB-CURVE-DAO-003).

❑ **Prevent the quick vote and withdraw strategy in `GaugeController`. Consider implementing either:**

1. **A weighted stake, with the weight decreasing over time, or**
2. **A locking period after weight's update.**

A quick vote and withdraw strategy allows a votes weight to be higher than expected in all the gauge's votes (TOB-CURVE-DAO-004).

❑ **Disallow adding the same gauge twice and add proper documentation to ensure the administrator is aware of the procedure to change some gauge weight liquidity.**

Adding the same gauge multiple times will corrupt the gauges' weights (TOB-CURVE-DAO-005).

❑ **Ensure that the `LiquidityGauge's` parameters always lead `rate * last_weight` to be greater than `_working_supply`.** Rounding to zero will allow attackers to spam the gauge and prevent users from earning interest (TOB-CURVE-DAO-006).

❑ **Remove the minter's permission to take tokens from other users, or properly document why this is necessary.** This will prevent users from distrusting the contracts (TOB-CURVE-DAO-007).

❑ **Use dedicated events for minting and burning, or don't allow users to fake `Transfer` events**. This will prevent confusion when events are used by off-chain components (TOB-CURVE-DAO-008).

❑ **Make sure users are aware that admin privileges can take whitelisted accounts hostage.** This will help users better understand the risks of interacting with this contract (TOB-CURVE-DAO-009).

❑ **Check the length of the token's name and symbol in `ERC20CRV`.** This will prevent the contract from returning an unexpected name or symbol (TOB-CURVE-DAO-010).

❑ **Use a two-step procedure for all non-recoverable critications.** This will reduce the possibility of mistakes when the users are executing critical operations (TOB-CURVE-DAO-011).

❑ **Either use a bit mask on the return of decimals, or revert if the value is greater than 255 in `VotingEscrow`.** This will prevent the contract from returning an unexpected number of decimals (TOB-CURVE-DAO-012).

❑ **Add events for all critical operations** to monitor the contracts and detect suspicious behavior. Missing events are listed in TOB-CURVE-DAO-013.

❑ **Document how to deal with whitelist removal.** Consider:

- Calling `remove_from_whitelist` when tokens are still locked (so the attacker cannot withdraw them, even after the lock expires).
- Increase the amount of gas when calling `remove_from_whitelist` to reduce the window of opportunity for this issue.

This will help reduce an attacker's window of opportunity to move their tokens (TOB-CURVE-DAO-014).

❑ **Increase the documentation, including all the identified missing behavior descriptions.** This will help users and auditors understand the system better (TOB-CURVE-DAO-015).

❑ **Document `balanceOfAt` and `totalSupplyAt` must not be called on the current block.** This will prevent users from misusing the `balanceOfAt` and `totalSupplyAt` functions (TOB-CURVE-DAO-016).

❑ **Create an incentive to vote early in `GaugeController`.** Consider using either:
- A decreasing weight to create an advantage for early voters, or
- A blind vote.

The lack of an incentive encourages voting at the very last minute and penalizes early voters (TOB-CURVE-DAO-017).

❑ **Reduce the risks associated with out-of-gas issues.**
- Allow users to execute the history catch-up in `VotingEscrow._checkpoint` without depositing or withdrawing the lock.
- Create a bot that will call `LiquidityGauge.user_checkpoint` and the `VotingEscrow's` history catch-up function at least once per week.
- Consider allowing iteration over the periods in multiple transactions in `GaugeController.`

Several contracts can be trapped if they are not called for a long time, or if `GaugeController` lists too many gauges (TOB-CURVE-DAO-018).

❑ **Improve Brownie test capabilities:**
- Modify Brownie to disallow automatic increase of the block timestamp and number.
- Set a reasonable default for the maximum gas used per transaction during tests.

This will improve testing of corner cases in the code where operations are executed in the same block or use a large amount of gas (TOB-CURVE-DAO-019).

❑ **Do not use the original Aragon contract. Consider:**
- Improving Aragon's voting to mitigate the issues listed in TOB-CURVE-DAO-020.
- Implementing a voting contract to replace Aragon's. Perform a security assessment on the contract before deployment.

Aragon's voting contract does not meet the security requirements for Curve Dao (TOB-CURVE-DAO-020).

❑ **Add a parameter to `LiquidityGauge.deposit` to specify the minimal amount of interest to receive, or make sure off-chain components take changes in the bonus into account.** This will prevent users from receiving less interest than expected (TOB-CURVE-DAO-021).

Long term

❑ **Write clear documentation of the different components' interactions and the dependencies of the assets, and consider an economical assessment.** This will help users and auditors to better understand how the contracts work (TOB-CURVE-DAO-001, TOB-CURVE-DAO-002, TOB-CURVE-DAO-003).

❑ **Properly document the `GaugeController`'s voting process.** This will help prevent misconceptions of how users are allowed to use their voting weight (TOB-CURVE-DAO-004, TOB-CURVE-DAO-017 TOB-CURVE-DAO-020).

❑ **Follow closely the progress made by the community on on-chain voting**. Blockchain-based online voting is a known challenge. No perfect solution has been found so far and the domain evolves quickly (TOB-CURVE-DAO-004, TOB-CURVE-DAO-017 TOB-CURVE-DAO-020).

❑ **Identify, review, and minimize the permissions assigned to each privileged user, and make sure users can access the information.** This will mitigate any potential private key compromise and increase the trust users have in your contracts (TOB-CURVE-DAO-007, TOB-CURVE-DAO-009, TOB-CURVE-DAO-011).

❑ **Use a blockchain monitoring system to track any suspicious behavior in the contracts.** The system relies on the correct behavior of several contracts. A monitoring system that tracks critical events and upfront-running would quickly detect any compromised system components (TOB-CURVE-DAO-008, TOB-CURVE-DAO-013, TOB-CURVE-DAO-014).

❑ **Carefully review Vyper's security advisories, open issues, and the current language limitations.** This will mitigate the risk of introducing issues caused by the compiler (TOB-CURVE-DAO-010, TOB-CURVE-DAO-012).

❑ **Create an incident response plan**. This will help reduce response time in case of security incidents (TOB-CURVE-DAO-014).

❑ **Review the contract's complete documentationand simplify its use.** This will mitigate the possibility of function misuse (TOB-CURVE-DAO-015).

❑ **Properly test system properties when functions are called in the same block or within a short period.** This will prevent unexpected results when functions are called with a small time interval (TOB-CURVE-DAO-016).

❑ **Improve the support of out-of-gas scenarios due to loop iterations:**

- Test the functions for their gas limit.
    - Use `brownie test` with the `--gas` flag.
    - Use the Echidna gas fuzzing feature.
- Update `GaugeController`'s logic to work with a large number of periods.

This will help detect issues caused by very high gas consumption before deployment (TOB-CURVE-DAO-018).

❑ **Carefully consider the unpredictable nature of Ethereum transactions and design your contracts so they don't depend on the transaction's ordering.** An attacker can control the order of the transactions to attack the system (TOB-CURVE-DAO-021).

❑ **Use a lower or higher bound on asset conversions.** An attacker can control the order of the transactions to change the outcome of asset conversion (TOB-CURVE-DAO-021).

❑ **Use use Echidna and Manticore to test and verify:**

- Time-dependent code (TOB-CURVE-DAO-006, TOB-CURVE-DAO-019)
- High–gas-consuming code (TOB-CURVE-DAO-019)
- Gauge administration functions (TOB-CURVE-DAO-005)

Several issues were found in these areas, and automated testing and verification will prevent similar issues.

Findings Summary

#	Title	Type	Severity
1	`LiquidityGauge` does not account for `VotingEscrow`'s balance updates	Data Validation	Medium
2	`LiquidityGauge` does not account for `VotingEscrow`'s `totalSupply` updates	Data Validation	Medium
3	Early users will have a unfair advantage	Data Validation	Medium
4	`GaugeController` allows for quick vote and withdraw voting strategy	Data Validation	Medium
5	Adding the same gauge multiple times will lead to incorrect sum of weights	Data Validation	Medium
6	Spam attack would prevent `LiquidityGauge`'s integratal from being updated	Timing	Medium
7	`Minter` user can confiscate any user tokens	Access Controls	High
8	`Mint` and `Burn` events cannot be trusted	Auditing and Logging	Low
9	`VotingEscrow`'s Admin can take whitelisted accounts hostage	Access Controls	Medium
10	`ERC20CRV` is not initiated correctly with large name and symbol	Data Validation	Low
11	Lack of two-step procedure for critical operations is error-prone	Data Validation	High
12	Lack of value verification on decimals is error-prone	Data Validation	Low
13	Lack of events is error-prone	Auditing and Logging	Informational
14	Race condition in removing addresses from whitelist and withdrawing	Timing	Informational
15	Lack of documentation is error-prone	Auditing and Logging	Informational

16	`VotingEscrow`'s `balanceOfAt` and `totalSupplyAt` can return different values for the same block	Data Validation	Low
17	No incentive to vote early in `GaugeController`	Data Validation	Medium
18	Several loops are not executable due to gas limitation	Data Validation	High
19	Testing smart contract code in Brownie can be unreliable	Patching	Undetermined
20	Aragon's voting does not follow voting best practices	Data Validation	High
21	Race condition may result in users earning less interest than expected	Data Validation	Informational

1. `LiquidityGauge` does not account for `VotingEscrow`'s balance updates

Severity: Medium
Type: Data Validation
Target: `LiquidityGauge.vy`

Difficulty: Low
Finding ID: TOB-CURVE-DAO-001

Description
`VotingEscrow`'s balance update is not accounted for in `LiquidityGauge`, so an attacker can earn more interest than they should by moving their `VotingEscrow` tokens.

`LiquidityGauge` computes the interest earned by users. A bonus is applied for `VotingEscrow` token holders:

```
def _update_liquidity_limit(addr: address, l: uint256, L: uint256):
    # 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 * 20 / 100
    if voting_total > 0:
        lim += L * voting_balance / voting_total * 80 / 100

    lim = min(l, lim)
```

Figure 1.1: `LiquidityGauge.vy#L75-L88`.

Users receive `VotingEscrow` tokens by locking their CRV tokens for a given period of time. Once the locking period is complete, they can withdraw their tokens.

The withdrawal of `VotingEscrow` tokens does not decrease the bonus applied to the interest rate in `LiquidityGauge`. As a result, an attacker can make a profit by re-using the tokens in the system to earn more interest, or by selling them while still earning the interest.

Exploit Scenario
The system has four users. Three of them have the same amount of liquidity tokens (100) and CRV locked (100):

- Alice: 100 LT, 100 Locked: working_balance = 60
- Bob: 100 LT, 100 Locked: working_balance = 60
- Eve 1: 50 LT, 100 Locked: working_balance = 50
- Eve 2: 50 LT, 0 Locked: working_balance = 10
- Carl: 0 LT, 300 Locked: working_balance = 60

Once the lock on Eve's first account ends, she deposits the CRV tokens in her second account. As a result, she has two accounts with a total working balance of 100 units when she should earn only 60 units.

Recommendation
Short term, consider either:

1. Removing the bonus based on the locked tokens,
2. Adding watchers that will penalize users cheating the system, or
3. Integrating the locking end time in the bonus computation.

Solutions (2) and (3) require significant modifications in the codebase and should be implemented with caution. Issues TOB-CURVE-DAO-002 and TOB-CURVE-DAO-003 must be considered when implementing the fix.

Long term, write clear documentation of the different components' interactions and the dependencies of the assets. Consider an economical assessment.

2. `LiquidityGauge` does not account for `VotingEscrow`'s `totalSupply` updates

Severity: Medium
Type: Data Validation
Target: `LiquidityGauge.vy`

Difficulty: Low
Finding ID: TOB-CURVE-DAO-002

Description
`VotingEscrow`'s `totalSupply` update is not accounted for in `LiquidityGauge`. As a result, users will not earn the expected interest.

`LiquidityGauge` computes the interest earned by users. A bonus is applied to `VotingEscrow` token's holder:

```
def _update_liquidity_limit(addr: address, l: uint256, L: uint256):
    # 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 * 20 / 100
    if voting_total > 0:
        lim += L * voting_balance / voting_total * 80 / 100

    lim = min(l, lim)
```

Figure 2.1: `LiquidityGauge.vy#L75-L88`.

The bonus is based on a percentage of a user's `VotingEscrow`'s tokens. `VotingEscrow` can be minted and burned at any moment, changing `totalSupply`.

As a result, the interest bonus given when `LiquidityGauge` is called does not reflect the real percentage over time. This might result in unexpected opportunities.

Exploit Scenario
Bob has 20% of the `VotingEscrow` locked tokens. Bob starts earning interest in `LiquidityGauge`. After a few days, the other users unlock their tokens. Bob now has 40% of the locked tokens, but he continues to earn interest based on 20%.

Recommendation
Short term, consider either:

1. Removing the bonus based on locked tokens, or
2. Updating the formulas to account for the total supply updates.

The second option may not be straightforward to implement and may require significant change. Issues TOB-CURVE-DAO-001 and TOB-CURVE-DAO-003 must be considered when implementing the fix.

Long term, write clear documentation of the different components' interactions and the asset dependencies. Consider an economical assessment.

3. Early users will have a unfair advantage

Severity: Medium
Type: Data Validation
Target: `LiquidityGauge.vy`

Difficulty: Medium
Finding ID: TOB-CURVE-DAO-003

Description
The `VotingEscrow`'s bonus for earned interest gives an unfair advantage to early users.

`LiquidityGauge` distributes a bonus based on the user's `VotingEscrow` token percentage:

```
def _update_liquidity_limit(addr: address, l: uint256, L: uint256):
    # 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 * 20 / 100
    if voting_total > 0:
        lim += L * voting_balance / voting_total * 80 / 100

    lim = min(l, lim)
```

Figure 3.1: `LiquidityGauge.vy#L75-L88`.

At launch, the `ERC20CRV` contract has 100% of the token supply, so it and the first token receivers can receive a significant and unfair bonus on their interest.

Combined with TOB-CURVE-DAO-001, this issue will allow early users to earn significant profits.

Exploit Scenario
Eve deploys the system, locks half of the supply, and only puts the other half in distribution. As a result, Eve earns significantly more interest than any other user.

Recommendation
Short term, consider either:

- Removing the bonus based on the locked tokens, or
- Clearly documenting that early users will have an advantage in the system.

Issues TOB-CURVE-DAO-001 and TOB-CURVE-DAO-002 must be considered when implementing the fix.

Long term, write clear documentation of the different components' interactions and the asset dependencies. Consider an economical assessment.

4. `GaugeController` allows for quick vote and withdraw voting strategy

Severity: Medium
Type: Data Validation
Target: `GaugeController.vy`

Difficulty: Low
Finding ID: TOB-CURVE-DAO-004

Description
The `GaugeController` voting can be abused to apply all of the user's weight in every gauge's vote.

`GaugeController`'s voting changes the weight of the gauges. Each user can split their voting weight power between the gauges:

```
def vote_for_gauge_weights(_gauge_id: int128, _user_weight: int128):
     [..]
     @param _user_weight Weight for a gauge in bps (units of 0.01%). Minimal is
0.01%. Ignored if 0
     [..]
     assert (_user_weight >= 0) and (_user_weight <= 10000), "You used all your
voting power"
     [..]
     new_slope: VotedSlope = VotedSlope({
         slope: slope * _user_weight / 10000,
         end: lock_end,
         power: _user_weight
     })
     [..]
```

Figure 4.1: `GaugeController.vy#L359-L384`.

The sum of all the weight used must not exceed 10,000:

```
    # Check and update powers (weights) used
     power_used: int128 = self.vote_user_power[msg.sender]
     power_used += (new_slope.power - old_slope.power)
     self.vote_user_power[msg.sender] = power_used
     assert (power_used >= 0) and (power_used <= 10000), 'Used too much power'
```

Figure 4.2: `GaugeController.vy#L388-L392`.

A gauge's weight can be updated every week:

```
def _enact_vote(_gauge_id: int128):
    now: uint256 = as_unitless_number(block.timestamp)
    ts: uint256 = self.vote_enacted_at[_gauge_id]
```

```
if (ts + WEEK) / WEEK * WEEK <= block.timestamp:
    # Update vote_point
```

Figure 4.3: `GaugeController.vy#L324-L329`.

There is no incentive to vote early, and no lock to prevent a user from removing their weight after a vote. As a result, an attacker can put 100% of its voting power (10,000) on a gauge's vote, and remove it right afterwards to re-use all its voting power on another vote.

Exploit Scenario
The system has three gauges. Eve has 1,000,000 tokens locked for two months. On every gauge's vote:

- Eve calls `vote_for_gauge_weights` with a voting power of 10,000 (100%) just before the vote ends.
- Once the vote ends, Eve calls `vote_for_gauge_weights` with a voting power of 0.

Eve uses all her voting power for all the gauges' votes. As a result, Eve manipulates the weights' updates more than she should.

Recommendation
Blockchain-based online voting is a known challenge. No perfect solution has been found so far. Short term, consider either:

1. Implementing a weighted stake, with weight decreasing over time, or
2. Implementing a locking period after the weight update.

Long term, properly document and test the voting process. Closely follow the progress made by the community on on-chain voting.

References
- Aragon vote shows the perils of on-chain governance

5. Adding the same gauge multiple times will lead to incorrect sum of weights

Severity: Medium
Type: Data Validation
Target: `GaugeController.vy`

Difficulty: High
Finding ID: TOB-CURVE-DAO-005

Description
The administrator can add the same gauge multiple times in the controller, leaving the contract in an invalid state.

The Gauge Controller contract allows its administrator to add liquidity gauges using the `add_gauge` function:

```
@public
def add_gauge(addr: address, gauge_type: int128, weight: uint256 = 0):
    assert msg.sender == self.admin
    assert (gauge_type >= 0) and (gauge_type < self.n_gauge_types)
    # If someone adds the same gauge twice, it will override the previous one
    # That's probably ok


    if self.gauge_types_[addr] == 0:
        n: int128 = self.n_gauges
        self.n_gauges = n + 1
        self.gauges[n] = addr


    self.gauge_types_[addr] = gauge_type + 1
```

Figure 5.1: `GaugeController.vy#L120-L132`*.*

However, contrary to what the code comment suggests, it is possible to lead the contract into an invalid state if the administrator adds the same gauge twice:

```
@public
def add_gauge(addr: address, gauge_type: int128, weight: uint256 = 0):
       ...
       self.type_weights[gauge_type][p] = _type_weight
       self.gauge_weights[addr][p] = weight
       self.weight_sums_per_type[gauge_type][p] = weight + old_sum
       if epoch_changed:
           self.total_weight[p-1] = self.total_weight[p-2]
       self.total_weight[p] = self.total_weight[p-1] + _type_weight * weight
       self.period_timestamp[p] = block.timestamp
```

Figure 5.2: `GaugeController.vy#L154-L160`*.*

The `total_weight` and the `weight_sums_per_type` will be incorrectly computed, since they will be increased by the weight a second time.

Exploit Scenario
Eve is the administrator of the gauge controller contract. Eve adds the same gauge twice and corrupts the other weight's percentage. As a result, users receive less interest than expected.

Recommendation
Short term, disallow adding the same gauge twice. Add proper documentation to ensure the administrator is aware of the procedure to change some gauge weight liquidity.

Long term, use Echidna and Manticore to ensure that the gauge administration functions are properly implemented.

6. Spam attack would prevent `LiquidityGauge`'s integral from being updated

Severity: Medium
Type: Timing
Target: `LiquidityGauge.vy`

Difficulty: High
Finding ID: TOB-CURVE-DAO-006

Description
An attacker spamming `LiquidityGauge` can prevent the integral from being updated. As a result, users will not earn interest.

On every balance's update, `LiquidityGauge._checkpoint` is executed and updates the integral based on the time elapsed since the last update:

```
@private
def _checkpoint(addr: address):
    _integrate_checkpoint: timestamp = self.integrate_checkpoint
    [..]
        dt = as_unitless_number(block.timestamp - _integrate_checkpoint)
    [..]
        _integrate_inv_supply += rate * last_weight * dt / _working_supply
```

Figure 6.1: `LiquidityGauge.vy#L92-L146`.

If `rate * last_weight * dt < _working_supply`, the integral will not be updated. `Dt` is the time elapsed since the last call to `_checkpoint` and is directly controllable by the caller.

An attacker can prevent the integral from being updated by calling the contract frequently. The attack is partially mitigated by the gas cost, but miners can perform the attack without paying any gas.

Exploit Scenario
Eve is a malicious miner, and adds a call to LiquidityGauge on every block. As a result, Eve prevents the `LiquidityGauge` from earning interest.

Recommendation
Short term, ensure that the system's parameters always make `rate * last_weight` greater than `_working_supply`.

Long term, take in consideration short and long times period increase in the tests, and consider using Echidna and Manticore to identify unexpected behaviors allowed by these increases.

7. `Minter` user can confiscate any user tokens

Severity: High
Type: Access Controls
Target: `ERC20CV.vy`

Difficulty: High
Finding ID: TOB-CURVE-DAO-007

Description
`ERC20CV`'s `minter` has the unexpected right to move tokens from any users, increasing the risks associated with the `minter` account.

The administrator of the contract can design a special user called a `minter`:

```
@public
def set_minter(_minter: address):
    assert msg.sender == self.admin  # dev: admin only
    self.minter = _minter
```

Figure 7.1: `ERC20CV.vy#L143-L146.`

This privileged user can be wielded to mint new tokens:

```
@public
def mint(_to: address, _value: uint256):
    """
    @dev Mint an amount of the token and assigns it to an account.
         This encapsulates the modification of balances such that the
         proper events are emitted.
    @param _to The account that will receive the created tokens.
    @param _value The amount that will be created.
    """
    assert msg.sender == self.minter  # dev: minter only
    assert _to != ZERO_ADDRESS  # dev: zero address


    if block.timestamp >= self.start_epoch_time + RATE_REDUCTION_TIME:
        self._update_mining_parameters()


    _total_supply: uint256 = self.total_supply + _value
    assert _total_supply <= self._available_supply()  # dev: exceeds allowable mint amount
    self.total_supply = _total_supply
```

```
    self.balanceOf[_to] += _value
    log.Transfer(ZERO_ADDRESS, _to, _value)
```

Figure 7.2: ERC20CRV.vy#L230-L250.

However, it is also possible to use the `minter` to take tokens from other user accounts, since the `transferFrom` function has an allowance bypass hardcoded for the `minter` user:

```
@public
def transferFrom(_from : address, _to : address, _value : uint256) -> bool:
    """
     @dev Transfer tokens from one address to another.
          Note that while this function emits a Transfer event, this is not required as per
the specification,
          and other compliant implementations may not emit the event.
     @param _from address The address which you want to send tokens from
     @param _to address The address which you want to transfer to
     @param _value uint256 the amount of tokens to be transferred
    """
    # NOTE: vyper does not allow underflows
    #       so the following subtraction would revert on insufficient balance
    self.balanceOf[_from] -= _value
    self.balanceOf[_to] += _value
    if msg.sender != self.minter:  # minter is allowed to transfer anything
        # NOTE: vyper does not allow underflows
        # so the following subtraction would revert on insufficient allowance
        self.allowances[_from][msg.sender] -= _value
    log.Transfer(_from, _to, _value)
    return True
```

Figure 7.3: ERC20CRV.vy#L253-L263.

Exploit Scenario

A malicious admin can silently change the minter address to steal tokens from users.

Recommendation

Short term, remove the `minter`'s permission to take tokens from other users or properly document why this is necessary.

Long term, review and minimize the permissions assigned to each privileged user. This will mitigate any potential private key compromise and increase the trust from users in your contracts.

8. `Mint` and `Burn` events cannot be trusted

Severity: Low
Type: Auditing and Logging
Target: `ERC20CV.vy`

Difficulty: Low
Finding ID: TOB-CURVE-DAO-008

Description
Events associated with `mint` and `burn` calls can be produced even if these functions are not called.

The `ERC20CRV` contract uses special `Transfer` events to signal the call to `mint`:

```
@public
def mint(_to: address, _value: uint256):
    """
    @dev Mint an amount of the token and assigns it to an account.
         This encapsulates the modification of balances such that the
         proper events are emitted.
    @param _to The account that will receive the created tokens.
    @param _value The amount that will be created.
    """
    assert msg.sender == self.minter  # dev: minter only
    assert _to != ZERO_ADDRESS  # dev: zero address


    if block.timestamp >= self.start_epoch_time + RATE_REDUCTION_TIME:
        self._update_mining_parameters()


    _total_supply: uint256 = self.total_supply + _value
    assert _total_supply <= self._available_supply()  # dev: exceeds allowable mint amount
    self.total_supply = _total_supply


    self.balanceOf[_to] += _value
    log.Transfer(ZERO_ADDRESS, _to, _value)
```

Figure 8.1: `ERC20CV.vy#L230-L250`.

And `burn`:

```
@public
```

```
def burn(_value: uint256) -> bool:
    """
    @dev Burn an amount of the token of msg.sender.
    @param _value The amount that will be burned.
    """
    self.balanceOf[msg.sender] -= _value
    self.total_supply -= _value


    log.Transfer(msg.sender, ZERO_ADDRESS, _value)
    return True
```

Figure 8.2: `ERC20CV.sol#L253-L263`.

However, in certain situations, these events can be produced even without calling such functions:

- `Transfer(..., 0x0, ...)` can be produced by any user transferring to the `0x0` address.
- `Transfer(0x0, ..., ...)` can be produced by the `minter` user when it employs the `transferFrom` function to recover tokens from `0x0`.

Exploit Scenario
Alice implements an off-chain component to interact with the Curve contract relying on the events. However, Eve triggers a transfer to `0x0`, so Alice's code does not work as expected.

Recommendation
Short term, use dedicated events for minting and burning, or don't allow users to fake `Transfer` events.

Long term, consider using a blockchain monitoring system to track any suspicious behavior in the contracts. The system relies on the correct behavior of several contracts, and a monitoring system that tracks critical events would quickly detect of any compromised system components.

9. `VotingEscrow`'s Admin can take whitelisted accounts hostage

Severity: Medium
Type: Access Controls
Target: `VotingEscrow.vy`

Difficulty: High
Finding ID: TOB-CURVE-DAO-009

Description
`VotingEscrow`'s admin can allow or disallow any contract to interact with `VotingEscrow`. A malicious owner can use this feature to ask for a ransom from `VotingEscrow`'s users.

```
@public
def add_to_whitelist(addr: address):
    assert msg.sender == self.admin
    self.contracts_whitelist[addr] = True


@public
def remove_from_whitelist(addr: address):
    assert msg.sender == self.admin
    self.contracts_whitelist[addr] = False
```

Figure 9.1: `VotingEscrow.vy#L90-L99`.

Exploit Scenario
Eve is a malicious `VotingEscrow` owner. Eve allows Bob to use `VotingEscrow's multisig` wallet. Bob deposits $1,000,000 worth of assets in the contract. Eve revokes Bob from the whitelist, and asks him to pay $100,000 in ransom to withdraw its funds, which he does.

Recommendation
Short term, make sure users are aware of this risk.

Long term, identify and document all possible actions for privileged accounts. Ensure users can easily identify the risks associated with every privileged account.

10. ERC20CRV is not initiated correctly with large name and symbol

Severity: Low
Type: Data Validation
Target: `ERC20CRV.vy`

Difficulty: High
Finding ID: TOB-CURVE-DAO-010

Description
Vyper does not check the length of the string it receives and only keeps the destination size's number of elements. As a result, if `ERC20CRV` is initiated with a large `name` or `symbol`, it will have an incorrect value.

```
name: public(string[64])
symbol: public(string[32])
```

Figure 10.1: `ERC20CRV.vy#L12-L13`.

Exploit Scenario
Bob deploys `ERC20CRV` with a name of 65 characters, but only the first 64 characters are kept, so the token is deployed incorrectly.

Recommendation
Short term, check the length of the string.

Long term, carefully review Vyper's open issues and current language limitations.

References
- vyper#1840

11. Lack of two-step procedure for critical operations is error-prone

Severity: High | Difficulty: High
Type: Data Validation | Finding ID: TOB-CURVE-DAO-011
Target: `VotingEscrow.vy, PoolProxy.vy, GaugeController.vy`

Description
Several critical operations are done in one function call. This schema is error-prone and can lead to irrevocable mistakes.

For example, `VotingEscrow.transfer_ownership` changes the contract's owner without any verification:

```
@public
def transfer_ownership(addr: address):
    assert msg.sender == self.admin
    self.admin = addr
```

Figure 11.1: `VotingEscrow.vy#L84-L87`.

As a result, if the admin sends an incorrect value, it will not be possible to recover the system.

Functions that would benefit from a two-step procedure include:

- `VotingEscrow.transfer_ownership` (`VotingEscrow.vy#L84-L87`)
- `PoolProxy.set_admins (PoolProxy.vy#L40)`
- `GaugeController.transfer_ownership (GaugeController.vy#L80)`

Exploit Scenario
Bob calls `VotingEscrow.transfer_ownership` but does not set the `addr` parameter. As a result, the new admin is the address `0x0`, and Bob loses ownership of the contract.

Recommendation
Short term, use a two-step procedure for all non-recoverable critications.

Long term, identify and document all possible actions and their associated risks for privileged accounts.

12. Lack of value verification on decimals is error-prone

Severity: Low
Type: Data Validation
Target: `VotingEscrow.vy`

Difficulty: High
Finding ID: TOB-CURVE-DAO-012

Description
The lack of `uint8` type in Vyper requires that all return values of erc20.`decimals()` calls are checked.

`VotingEscrow` calls `decimals()` without checking the return value:

```
self.decimals = ERC20(token_addr).decimals()
```

Figure 12.1: `VotingEscrow.vy#L78`.

`ERC20.decimals()` returns a `uint8`, but this type is not handled by Vyper. As a result, the decimal value used could be invalid.

Exploit Scenario
Eve deploys a token with decimals of 520. It's decimals are read as 8 by the Solidity contract, but 520 by `VotingEscrow`. As a result, `VotingEscrow`'s usage is incorrect.

Recommendation
Short term, either use a bit mask on the return of decimals, or revert if the value is greater than 255.

Long term, carefully review Vyper's security advisories and the current language limitations.

References
- VVE-2020-0001: Interfaces returning integer types less than 256 bits can be manipulated if `uint256` is used

13. Lack of events is error-prone

Severity: Informational
Type: Auditing and Logging
Target: All contracts

Difficulty: Low
Finding ID: TOB-CURVE-DAO-013

Description
Several critical operations do not trigger events. As a result, it will be difficult to review the correct behavior of the contracts once deployed.

Critical operations that would benefit from triggering events include:

- `PoolProxy.set_admins (PoolProxy.vy#L40)`
- `PoolProxy.set_burner (PoolProxy.vy#50)`
- `ERC20CRV.update_mining_parameters (ERC20CRV.vy#L71)`
- `ERC20CRV.set_minter (ERC20CRV.vy#144)`
- `ERC20CRV.set_admin (ERC20CRV.vy#150)`
- `GaugeController.transfer_ownership (GaugeController.vy#L80)`
- `GaugeController._change_type_weight (GaugeController.vy#L224)`
- `GaugeController._change_gauge_weight (GaugeController.vy#L272)`
- `GaugeController.vote_for_gauge_weights (GaugeController.vy#L359)`
- `LiquidityGauge._update_liquidity_limit (LiquidityGauge#75)`
- `VotingEscrow.transfer_ownership (VotingEscrow.vy#L85)`
- `VotingEscrow.add_to_whitelist (VotingEscrow.vy#L103)`
- `VotingEscrow.remove_from_whitelist (VotingEscrow.vy#L110)`

Users and blockchain monitoring systems can't easily detect suspicious behaviors without events.

Exploit Scenario
Eve compromises the `PoolProxy` contract. Bob does not notice the compromise and Eve is able to change the parameter of the pool.

Recommendation
Short term, add events for all critical operations to help monitor the contracts and detect suspicious behavior.

Long term, consider using a blockchain monitoring system to track any suspicious behavior in the contracts. The system relies on the correct behavior of several contracts. A monitoring system that tracks critical events would allow quick detection of any compromised system components.

14. Race condition in removing addresses from whitelist and withdrawing

Severity: Informational
Type: Timing
Target: `VotingEscrow.vy`

Difficulty: High
Finding ID: TOB-CURVE-DAO-014

Description
The `VotingEscrow` contract provides a set of functions to add and remove contract addresses in a whitelist. Once the admin calls `remove_from_whitelist` with a user's address, that user should no longer be able to perform any operation with tokens.

```
@public
def remove_from_whitelist(addr: address):
   assert msg.sender == self.admin
   self.contracts_whitelist[addr] = False
```

Figure 14.1: `VotingEscrow.vy#L96-L99`.

This approach could be used by the admin to stop a contract that was upgraded by malicious code. However, it is vulnerable to a race condition if the user removed from the whitelist is monitoring unconfirmed transactions on the blockchain. If this user sees the transaction containing the call before it has been mined, they can call `withdraw` to claim their tokens (given that locks are expired), effectively circumventing the restrictions imposed by this whitelist.

Exploit Scenario
Alice is the administrator of `VotingEscrow`. She whitelists Bob's `multisig` wallet. However, an attacker takes control of it (either using a vulnerability in the contract or compromising their users' keys).

1. Alice calls `remove_from_whitelist(Bob)`. This forbids Bob's contract from withdrawing his tokens.
2. The attacker sees the unconfirmed transaction and calls `withdraw` to claim his tokens before Alice's transaction has been mined. He pays a higher fee to ensure that his call will be mined before the `remove_from_whitelist` call.
3. If the attacker's transaction is mined before Alice's, the removal of Bob's contract from the whitelist will be ineffective since the attacker can still spend his tokens.

Recommendation
Short term, document how to deal with this kind of situation:

- Call `remove_from_whitelist` when tokens are still locked (so the attacker cannot withdraw them, even after the locked expires).
- Increase the amount of gas when calling `remove_from_whitelist` in order to reduce the window of opportunity.

Long term, carefully monitor the blockchain to prevent and mitigate these kinds of front-running attacks, and create an incident response plan.

15. Lack of documentation is error-prone

Severity: Informational
Type: Auditing and Logging
Target: `several contracts and readme`

Difficulty: Low
Finding ID: TOB-CURVE-DAO-015

Description
The overall codebase lacks code documentation, high-level description, and examples. As a result, the contracts are difficult to review and the likelihood of user mistakes is high.

Several behaviors are not documented, including:

- `VotingEscrow.withdraw(_value)` will withdraw the whole balance if `_value` is zero.
    - Additionally, allowing the withdrawal of only part of the locked amount is error-prone and it is unclear whether this functionality is needed.
- `VotingEscrow.deposit(value, unlock_time)` has no documentation regarding the expected value for `unlock_time`. It also fails if used with a `value` larger `than 2**128` because the locked amounts are internally converted to `int128`.
- `_user_weight` in `GaugeController.vote_for_gauge_weights(_gauge_id, _user_weight)` should be between 0 and 10,000.
- The lock time in `VotingEscrow.deposit` is rounded down to weeks.
- `Last_point.bias` in `VotingEscrow._checkpoint` can be negative due to arithmetic rounding.

The current high-level documentation would benefit from more details, including:

- User-level examples that describe who the different users are, how they interact with the contracts, and concrete scenarios highlighting usage.
- The reasoning behind some design choices, such as:
    - `EscrowVoting` must not be tokenized.
    - Partial withdrawals from escrow are possible.

Exploit Scenario
Bob develops a `multisig` contract that calls `VotingEscrow.withdraw`. Bob is not aware that `withdraw(0)` withdraws the whole balance. As a result, Bob's contract does not work as expected.

Recommendation
Short term, review and properly document these corner cases.

Long term, review the complete documentation of the contract and simplify itto prevent misuse.

16. `VotingEscrow`'s `balanceOfAt` and `totalSupplyAt` can return different values for the same block

Severity: Low
Type: Data Validation
Target: `VotingEscrow.vy`

Difficulty: Low
Finding ID: TOB-CURVE-DAO-016

Description

`VotingEscrow`'s `balanceOfAt` and `totalSupplyAt` return their corresponding values for a given block. Because the balance and supply can vary within the same block, these functions can return different values when called on the current block.

`VotingEscrow`'s `balanceOfAt(addr, block)` and `totalSupplyAt(block)` use a binary search to return their values associated with the `block`:

```
    # Binary search
    _min: int128 = 0
    _max: int128 = self.user_point_epoch[addr]
    for i in range(128):  # Will be always enough for 128-bit numbers
        if _min >= _max:
            break
        _mid: int128 = (_min + _max + 1) / 2
        if self.user_point_history[addr][_mid].blk <= _block:
            _min = _mid
        else:
            _max = _mid - 1
```

Figure 16.1: `VotingEscrow.vy#L359-L369`.

```
    _min: int128 = 0
    _max: int128 = max_epoch
    for i in range(128):  # Will be always enough for 128-bit numbers
        if _min >= _max:
            break
        _mid: int128 = (_min + _max + 1) / 2
        if self.point_history[_mid].blk <= _block:
            _min = _mid
        else:
            _max = _mid - 1
    return _min
```

Figure 16.2: `VotingEscrow.vy#L324-L335`.

If a block is contained in `point_history`, the latest one will be used.

Points on the current block can be added indefinitely in `point_history`. As a result, a user calling `balanceOfAt` or `totalSupplyAt` on the current block might not receive the latest value.

The issue does not impact Aragon's usage, as vote creation uses the previous block number for its snapshot:

```
uint64 snapshotBlock = getBlockNumber64() - 1; // avoid double voting in this very
block
```

Figure 16.2: Voting.sol#L284.

Exploit Scenario
Bob creates a voting contract that relies on `balanceOfAt` and `totalSupplyAt`. Eve creates a vote using `block.number` as a snapshot and corrupts the quorum percentage.

Recommendation
Short term, document that `balanceOfAt` and `totalSupplyAt` must not be called on the current block.

Long term, properly test system properties when functions called in the same block or within a short period.

17. No incentive to vote early in `GaugeController`

Severity: Medium
Type: Data Validation
Target: `GaugeController.vy`

Difficulty: Medium
Finding ID: TOB-CURVE-DAO-017

Description
`GaugeController` voting offers no incentive to vote early, so late-voting users have a benefit over early voters.

Sinces all the votes are public, users who vote earlier are penalized because their votes are known by the other participants. An attacker can learn exactly how many tokens are necessary to change the outcome of the voting just before it ends.

Exploit Scenario
Bob votes for a vote gauge with half of its weight. His vote is winning, so he does not put in the other half of its weight. Eve votes at the last second and changes the outcome of the vote. As a result, Bob loses the vote.

Recommendation
Blockchain-based online voting is a known challenge. No perfect solution has been found so far.

Short term consider either:

- Using a decreasing weight to create an early voting advantage
- Using a blind vote

Long term, properly document and test the voting process and closely follow the community's progress regarding on-chain voting.

References
- Aragon vote shows the perils of on-chain governance

18. Several loops are not executable due to gas limitation

Severity: High
Type: Data Validation
Target:

Difficulty: High
Finding ID: TOB-CURVE-DAO-018

Description
The codebase relies on several loops that can iterate hundreds of times with costly gas consumption. This design is error-prone and may cause the contract to be trapped because it runs out of gas.

For example, the `LiquidityGauge` and `VotingEscrow` `_checkpoint` functions both have loops that can be iterated hundreds of times while changing state:

```
        for i in range(500):
             [..]
             self.period_checkpoints[p] = new_period_time
```

`LiquidityGauge.vy#L115-L128`.

```
    for i in range(255):
         [...]
             self.point_history[_epoch] = last_point
```

`VotingEscrow.vy#L158-L181`.

These loops have code that writes state variables, which is the operation that consumes the most gas.

Both loops are executed with every interaction of the contract. `VotingEscrow` may not be called as often as `LiquidityGauge` over a long period of time. However, it's unlikely there will be a long period of time in which these contracts are not called.
Additionally, `GaugeController` iterates over the contract's whole period in several locations, such as:

```
        for i in range(500):
             _p += 1
             if _p == p:
                 break
             self.type_weights[gauge_type][_p] = type_weight
             self.weight_sums_per_type[gauge_type][_p] = old_sum
```

`GaugeController.vy#L203-L206`.

If the number of periods is large, the contract is trapped.

Exploit Scenario
Bob adds hundreds of gauges. As a result, most of the functions in `GaugeController` cannot be executed anymore.

Recommendations
Short term

- Allow users to execute the history catch-up in `VotingEscrow._checkpoint` without depositing or withdrawing the lock.
- Create a bot that will call `LiquidityGauge.user_checkpoint` and the `VotingEscrow's` history catch-up function at least once per week.
- Consider allowing iteration over the periods in multiple transactions in `GaugeController`, and make sure the partial updates are sound.

Long term:

- Test functions for their gas limit:
  - Use `brownie test` with the `--gas` flag.
  - Use Echidna's gas fuzzing feature.
- Update `GaugeController`'s logic to work with a large number of periods.

19. Testing smart contract code in Brownie can be unreliable

Severity: Undetermined
Type: Patching
Target: All the smart contracts and tests

Difficulty: Medium
Finding ID: TOB-CURVE-DAO-019

Description
The Brownie testing system should be improved to make it more robust when dealing with time-dependent and high-consumption gas tests.

When Brownie tests code that depends on the block number and timestamp in smart contracts, it provides specific functions to simulate how they're produced by the simulated blockchain.

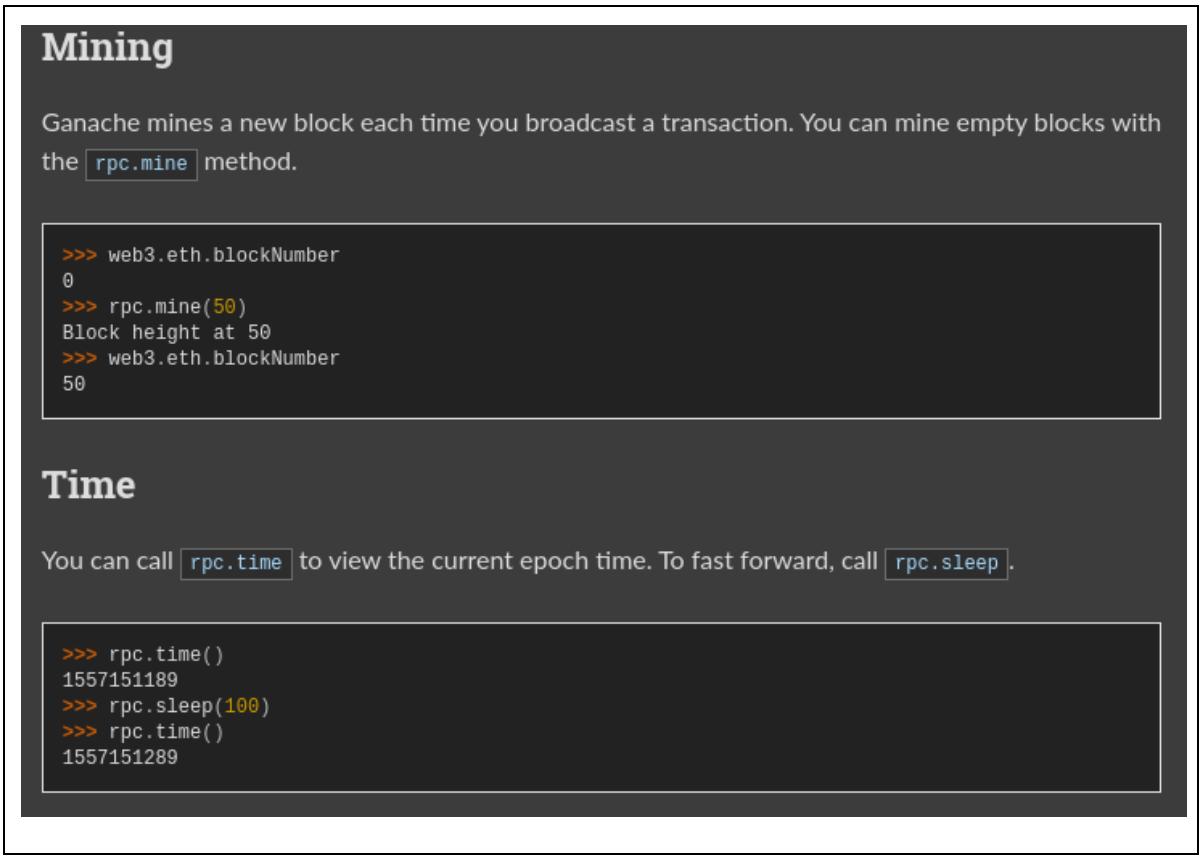

Figure 19.1: Simulating blocks in Brownie tests.

However, we found that the timestamp and block number increase even if the developer does not use the instrumentation functions. This means any test that requires checking whether the code can be executed correctly in the same block will not operate reliably.

Additionally, during testing, Brownie uses a default value for maximum gas which is determined using the `Eth.estimateGas` function. This estimate could allow tests to pass

even if they consume a very large amount of gas, making them impractical to use when deployed.

Exploit Scenario
Curve DAO contracts are developed without proper testing and as a result, the code is deployed with a critical bug in it.

Recommendation
Short term:

- Modify Brownie to disallow automatic block timestamp and number increases.
- Set a reasonable default for the maximum gas used per transaction during tests.

Long term, use Echidna and Manticore to test your time-dependent and high–gas-consuming code.

20. Aragon's voting does not follow voting best practices

Severity: High
Type: Data Validation
Target: Aragon's `Voting.sol`

Difficulty: Medium
Finding ID: TOB-CURVE-DAO-020

Description
Curve Dao uses Aragon for voting. Its voting logic is simple, but does not preventseveral abuses that can occur with on-chain voting.

In particular, the voting contract has the following issues:

- No mitigation for quick vote and withdraw (similar to issue TOB-CURVE-DAO-004).
- No incentive to vote earlier (similar to issue TOB-CURVE-DAO-017).
- No mitigation for spam attacks. An attacker with vote creation rights can create hundreds of thousands of votes, and will need only one to pass to succeed.

Exploit Scenario
Eve is a miner. She creates new votes to set a new `minter` on `ERC20CRV` on every block. The other users cannot vote on all the votes. As a result, one vote is accepted, and Eve takes control of `ERC20CRV`'s minting.

Recommendation
Blockchain-based online voting is a known challenge. No perfect solution has been found so far.

Short term, consider either:

- Improving Aragon's voting to mitigate the listed issues, or
- Implementing a voting contract to replace Aragon's. Perform a security assessment on the contract before deployment.

Long term, properly document and test the voting process. Closely Follow the community's progress regarding on-chain voting.

References
- Security Disclosure: Aragon 0.6 Voting ("Voting v1")
- Aragon vote shows the perils of on-chain governance

21. Race condition may result in users earning less interest than expected

Severity: Informational
Type: Data Validation
Target: `LiquidityGauge.vy`

Difficulty: Medium
Finding ID: TOB-CURVE-DAO-021

Description
The absence of a minimal interest rate might return a lower bonus for users than expected.

`LiquidityGauge` computes the interest earned by users. A bonus is applied to `VotingEscrow` token's holder:

```
def _update_liquidity_limit(addr: address, l: uint256, L: uint256):
     # 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 * 20 / 100
     if voting_total > 0:
         lim += L * voting_balance / voting_total * 80 / 100

     lim = min(l, lim)
```

Figure 2.1: `LiquidityGauge.vy#L75-L88`.

The bonus depends on `VotingEscrow`'s total supply, which can increase over time. If a user makes a deposit in `LiquidityGauge` and his transaction is mined after the total supply is increased, they can receive less bonus as expected.

Exploit Scenario
Bob calls `LiquidityGauge` and expects to receive a bonus of 10%. At the same time, Alice locks a significant amount of tokens in `VotingEscrow`. Alice's transaction is accepted before Bob's, so Bob receives a bonus of only 9%.

Recommendation
Short term, add a parameter to `LiquidityGauge.deposit` specifying the minimal amount of interest to be received, or make sure off-chain components take this scenario into account.

Long term, carefully consider the unpredictable nature of Ethereum transactions and design your contracts so they don't depend on transactions order. Additionally, always use a lower or higher bound on asset conversions.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking, or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important
Medium	Individual user's information is at risk, exploitation would be bad for

	client's reputation, moderate financial impact, possible legal implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses in order to exploit this issue

B. Code Maturity Classifications

Code Maturity Classes	
Category Name	**Description**
Access Controls	Related to the authentication and authorization of components.
Arithmetic	Related to the proper use of mathematical operations and semantics.
Assembly Use	Related to the use of inline assembly.
Centralization	Related to the existence of a single point of failure.
Upgradeability	Related to contract upgradeability.
Function Composition	Related to separation of the logic into functions with clear purpose.
Front-Running	Related to resilience against front-running.
Key Management	Related to the existence of proper procedures for key generation, distribution, and access.
Monitoring	Related to use of events and monitoring procedures.
Specification	Related to the expected codebase documentation.
Testing & Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.).

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.

Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

C. Code Quality

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

General suggestions:

- **Do not use one-letter variable names.** The smart contract code uses variables with very short names that can be difficult to parse when the code is modified or reviewed. Use full names, e.g., `weight` instead of `w`.
- **Split large functions into internal functions.** Large functions such as `LiquidityGauge._checkpoint` and `VotingEscrow._checkpoint` can be split into internal functions (e.g., history catch-up, user value update, etc.). Having smaller and simpler functions will simplify review and verification of the code.
- **Do not use unnamed constants.** The smart contract code uses certain constants without naming them. Use proper names, e.g., `BASE` instead of `10 ** 18`.

`ERC20CRV.vy`:

- **Consider correcting the `RATE_REDUCTION_COEFFICIENT` constant to be more accurate.** The exact coefficient used is `1414213562373095168`, and the comment accompanying its declaration indicates it should be equal to `sqrt(2) * 1e18`. However, a more accurate approximation of `sqrt(2) * 1e18` would actually be `1414213562373095049`, which differs in the last three decimal places.

VotingEscrow.vy:

- **Split the deposit functions into deposit creation, amount increase, and time increase functions.** `Deposit` handles the creation and increase of a deposit's amount and time simultaneously. As a result, the function has to handle too many cases and is error-prone.
- **Use `find_block_epoch` in `balanceOfAt`.** `BalanceOfAt` duplicates the code of `find_block_epoch`.

D. Token Integration Checklist

The following checklist provides recommendations when interacting with arbitrary tokens. Every unchecked item should be justified and its associated risks understood.

For convenience, all Slither utilities can be run directly on a token address, such as:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken
```

General Security Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (aka "level of effort"), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on blockchain-security-contacts.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

ERC Conformity

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use slither-check-erc to review that:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **`The name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and might not be present.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. If this is the case, ensure the value returned is below 255.
- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.
- ❏ **The token is not an ERC777 token and has no external function call in `transfer` and `transferFrom`.** External calls in the transfer functions can lead to reentrancies.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use slither-prop to review that:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests, then check the properties with Echidna and Manticore.

Finally, there are certain characteristics that are difficult to identify automatically. Review for these conditions by hand:

- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest might be trapped in the contract if not taken into account.

Contract Composition

- ❏ **The contract avoids unneeded complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.
- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.
- ❏ **The contract has only a few non–token-related functions.** Non–token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.

Owner privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts might change their rules over time. Use Slither's `human-summary` printer to determine if the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.
- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pauseable code by hand.
- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams, or that reside in legal shelters should require a higher standard of review.

Token Scarcity

Reviews for issues of token scarcity requires manual review. Check for these conditions:

- ❏ **No user owns most of the supply.** If a few users own most of the tokens, they can influence operations based on the token's repartition.

- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange can compromise the contract relying on the token.
- ❏ **Users understand the associated risks of large funds or flash loans.** Contracts relying on the token balance must carefully take in consideration attackers with large funds or attacks through flash loans.

E. Fix Log

Swiss-Stake addressed issues TOB-CURVE-DAO-001 to TOB-CURVE-DAO-013 in their codebase as a result of the assessment. Each of the fixes was verified by Trail of Bits. The reviewed code is available in git revision [9ba007a5013dd46e66401bc552933407f0bee044](9ba007a5013dd46e66401bc552933407f0bee044).

ID	Title	Severity	Status
01	`LiquidityGauge` does not account for `VotingEscrow`'s balance updates	Medium	Mitigated
02	`LiquidityGauge` does not account for `VotingEscrow`'s `totalSupply` updates	Medium	Not fixed
03	Early users have a unfair advantage	Medium	Not fixed
04	`GaugeController` allows for quick vote and withdraw voting strategy	Medium	Mitigated
05	Adding the same gauge multiple times will lead to incorrect sum of weights	Medium	Fixed
06	Spam attack would prevent `LiquidityGauge`'s integral from being updated	Medium	Risk accepted
07	`Minter` user can confiscate any user tokens	High	Fixed
08	`Mint` and `Burn` events cannot be trusted	Low	Fixed
09	`VotingEscrow`'s Admin can take whitelisted accounts hostage	Medium	Fixed
10	`ERC20CRV` is not initiated correctly with large name and symbol	Low	Fixed
11	Lack of two-step procedure for critical operations is error-prone	High	Fixed
12	Lack of value verification on decimals is error-prone	Low	Fixed
13	Lack of events is error-prone	Informational	Mitigated
14	Race condition in removing addresses from whitelist and withdrawing	Informational	WIP
15	Lack of documentation is error-prone	Informational	WIP
16	`VotingEscrow`'s `balanceOfAt` and `totalSupplyAt` can	Low	WIP

	return different values for the same block		
17	No incentive to vote early in `GaugeController`	Medium	WIP
18	Several loops will not be executable due to gas limitation	High	WIP
19	Testing smart contract code in Brownie can be unreliable	Undetermined	WIP
20	Aragon's voting does not follow voting best practices	High	WIP
21	Race condition can lead users to earn less interest than expected	Informational	WIP

Detailed Fix Log

This section includes brief descriptions of fixes implemented by Swiss-Stake after the end of this assessment that were reviewed by Trail of Bits.

Finding 1: `LiquidityGauge` does not account for `VotingEscrow`'s balance updates

This issue is mitigated by:

- Reducing the bonus created by the vote locks from 5 to 2.5.
- Adding a public `kick` function to adjust the working balance of any user abusing the bonus.

We recommend updating the documentation to ensure users are aware of `kick`. Curve DAO should consider developing a bot that will scan the account and call `kick` when appropriate. This bot should be publically available to prevent TOB-CURVE-DAO-001 being exploited.

Finding 2: `LiquidityGauge` does not account for `VotingEscrow`'s `totalSupply` updates

This issue is not fixed.

Finding 3: Early users have a unfair advantage

To fix the issue, Curve DAO added a check preventing the bonus from being applies during the first two weeks:

```
(block.timestamp > self.period_checkpoints[0] + BOOST_WARMUP)
```

`LiquidityGauge.vy#L101.`

`Self.period_checkpoints[0]` will be zero if the liquidity gauge is deployed when the period on the gauge controller is greater than or equal to 1. As a result, the check is incorrectly implemented.

Additionally, the delay in the bonus activation will only work if early users share their tokens enough to create a well-distributed reparition.

Finding 4: `GaugeController` allows for quick vote and withdraw voting strategy
This appears to be mitigated by disallowing changing weight votes more often than once in 10 days.

Finding 5: Adding the same gauge multiple times leads to incorrect sum of weights
This appears to be fixed by disallowing adding the same gauge twice.

Finding 6: Spam attack would prevent `LiquidityGauge`'s integral from being updated
The client estimated the impact of this issue and accepted the risk.

Finding 7: `Minter` user can confiscate any user tokens

This appears to be fixed by:
- Disallowing the transfer of unapproved tokens by the `minter`.
- Disallowing setting the `minter` address more than once.

Finding 8: `Mint` and `Burn` events cannot be trusted

This appears to be fixed by:

- Disallowing transfer of unapproved tokens by the `minter`.
- Disallowing users to transfer to `0x0`.

Finding 9: `VotingEscrow`'s Admin can take whitelisted accounts hostage
This appears to be fixed by allowing un-whitelisted addresses to withdraw from the voting escrow contract.

Finding 10: `ERC20CRV` is not initiated correctly with large name and symbol
This appears to be fixed by requiring the use of Vyper 0.2.0 to resolve this issue.

Finding 11: Lack of two-step procedure for critical operations is error-prone
This appears to be fixed by implementing a two-step procedure in the following functions:

- `VotingEscrow.transfer_ownership`
- `PoolProxy.set_admins`
- `GaugeController.transfer_ownership`

Finding 12: Lack of value verification on decimals is error-prone
This appears to be fixed by validating the values obtained from calling the `decimals` function.

Finding 13: Lack of events is error-prone
This appears to be mitigated by adding suitable events in the following functions:

- `PoolProxy.set_admins`
- `PoolProxy.set_burner`
- `ERC20CRV.update_mining_parameters`
- `ERC20CRV.set_minter`
- `ERC20CRV.set_admin`

- `GaugeController.transfer_ownership`
- `GaugeController._change_type_weight`
- `GaugeController._change_gauge_weight`
- `GaugeController.vote_for_gauge_weights`
- `LiquidityGauge._update_liquidity_limit`

However, events associated with important operations in `VotingEscrow` are missing.

Swiss-Stake is still working to fix the remaining issues.