CANTINA

Eco Association

Security Review

Cantina Managed review by:

Riley Holterhus, Lead Security Researcher

Slowfi, Security Researcher

April 23, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Eco is a currency with a growing community building products and services to drive its adoption. Eco's design is informed by the following assumptions: Given better economic data and a more transparent incentive structure for governance, it is possible to govern and grow a reserve currency that is digitally-native and more clearly aligned with its users' (our) collective best interests.

From Jan 22nd to Feb 9th the Cantina team conducted a review of currency-1.5 on commit hash 2c7ad91c. The team identified a total of **26** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 3
- Medium Risk: 3
- Low Risk: 8
- Gas Optimizations: 2
- Informational: 10

3 Findings

3.1 High Risk

3.1.1 Trustees can abuse a bit-shift overflow during reveals

Severity: High Risk

Context: CurrencyGovernance.sol#L786-L787

Description: When a trustee reveals their committed votes, the `uint256 scoreDuplicateCheck` value is used to ensure their vote follows the intended Borda count scheme. Each check is implemented as follows:

```
uint256 duplicateCompare = (2 ** _support - 1) << (_score - _support);

if (scoreDuplicateCheck & duplicateCompare > 0) {
    revert InvalidVoteBadScore(v);
}

scoreDuplicateCheck += duplicateCompare;

scores[_proposalId] += _score;
```

Notice that the `duplicateCompare` variable is using a left bit-shift based on the number of votes the trustee submitted. In Solidity, this bit-shift will never revert, even if the resulting value has truncated bits that are shifted past the `uint256` boundary.

As a consequence, a trustee can submit extremely large scores for a proposal of their choice, and the bit-shift overflow will bypass the `InvalidVoteBadScore()` error. This can lead to unfair behavior, because one malicious trustee can easily decide the outcome of the entire cycle.

Recommendation: Ensure that this left bit-shift does not overflow. This can be accomplished using a `safeLeftShift()` helper function (one implementation exists in `ECOxExchange.sol`) or by placing an upper-bound on the value of `_score`. Since `2 ** _support - 1` is `_support` number of 1 bits starting from the lsb, it is sufficient to require that `_score - _support <= 256 - _support`, or equivalently, that `_score <= 256`.

Eco: Fixed in PR 82.

Cantina Managed: Verified. It is now enforced that scores can't be larger than 255. As described above, this prevents the bit-shift overflow.

3.1.2 Old proposals can be unsupported to bypass the `canSupport()` check

Severity: High Risk

Context: CurrencyGovernance.sol#L640-L667

Description: To prevent a trustee from voting multiple times in a cycle, the `canSupport()` function checks against the trustee's `trusteeSupports` value. If a trustee decides to unsupport a proposal, their `trusteeSupports` is reset, allowing them to support in the current cycle again.

Currently, nothing is stopping a trustee from calling `unsupportProposal()` on a proposal they supported in a previous cycle. Since this will still reset their `trusteeSupports` value, a trustee can abuse this to support multiple times in one cycle.

Recommendation: Prevent calling `unsupportProposal()` on proposals from past cycles. This can be accomplished as follows:

```
  function unsupportProposal(
      bytes32 proposalId
  ) external onlyTrusted duringProposePhase {
      uint256 cycle = getCurrentCycle();

      MonetaryPolicy storage p = proposals[proposalId];
      uint256 support = p.support;

      if (support == 0) {
          revert NoSuchProposal();
      }
      if (!p.supporters[msg.sender]) {
          revert SupportNotGiven();
      }
+     if (p.cycle != cycle && proposalId != cycle) {
+         revert();
+     }

      // ...
  }
```

It's also recommended to update `supportProposal()` to disallow supporting proposals from previous cycles. This would be useful to prevent trustees from doing this by accident, because there would no longer be a way to fix this mistake with `unsupportProposal()`.

Eco: Fixed `unsupportProposal()` in PR 73. Fixed `supportProposal()` in PR 91.

Cantina Managed: Verified. The fix prevents to bypass the `canSupportCheck` by avoiding to unsupport old proposals with the suggested changes. Also, it updates `supportProposal` function to prevent possible mistakes from trustees.

3.1.3 Anyone can delete a leading default proposal

Severity: High Risk

Context: CurrencyGovernance.sol#L832-L861

Description: In the `CurrencyGovernance` contract, the `enact()` function is implemented as follows:

```
function enact(uint256 _cycle) external cycleComplete(_cycle) {
    if (participation < quorum) {
        revert QuorumNotMet();
    }
    bytes32 _leader = leader;

    // this ensures that this function can only be called maximum once per winning MP
    delete leader;

    // the default proposal doesn't do anything
    if (_leader == bytes32(_cycle)) {
        emit VoteResult(_cycle, _leader); // included for completionist's sake, will likely never be called
        return;
    }

    MonetaryPolicy storage _winner = proposals[_leader];

    if (_winner.cycle != _cycle) {
        revert EnactCycleNotCurrent();
    }

    enacter.enact(
        _leader,
        _winner.targets,
        _winner.signatures,
        _winner.calldatas
    );

    emit VoteResult(_cycle, _leader); // will not be emittable if enact cannot be called without reverting
↪   (downstream error)
}
```

Usually, this function will be called with the latest completed `_cycle`, and the logic will proceed to enact the proposal that received the most votes.

However, an edge case scenario exists, and this function can be called unexpectedly. Specifically, if the following conditions are true:

- The current cycle is in the reveal stage.
- There have been enough reveals to reach the minimum quorum.
- The current `leader` is the default proposal of the cycle.

Then calling `enact(0)` will unexpectedly succeed. This happens because:

- The `cycleComplete(0)` modifier succeeds, since the zeroth cycle has already passed.
- The `if (participation < quorum)` statement is not entered, since minimum quorum has been reached.
- The `if (_leader == bytes32(0))` statement is not entered, since the `_leader` is a non-zero cycle id.
- The `if (_winner.cycle != 0)` statement is not entered, since the `.cycle` value is uninitialized as zero for default proposals.
- The `enacter.enact()` call succeeds, because the `targets`, `signatures`, and `calldatas` are empty arrays for default proposals.

If someone were to call `enact(0)` like this, the `leader` storage variable would be deleted, and the remaining reveals would not behave correctly. In the worst case, an external attacker might strategically call `enact(0)` to pass a proposal that did not receive the most votes.

Recommendation: Change the `enact()` function to not take a `_cycle` as input. In normal circumstances, an `enact()` will happen in the proposal stage of the following cycle. So, `enact()` can instead ensure it is currently in the proposal stage, and can always attempt to enact cycle id `getCurrentCycle() - 1`. This will prevent the issue.

Also, if the changes to `commit()` described in "Committing to future cycles can bypass storage clean up" are taken, the `enact()` function could be allowed to execute in the voting stage in addition to the proposal stage.

Eco: Fixed in PR 81.

Cantina Managed: Verified. The Eco team has changed the `enact()` function to only be callable in the proposal stage, and to always use `getCurrentCycle() - 1`. Both of these changes mitigate the underlying issue.

3.2 Medium Risk

3.2.1 Committing to future cycles can bypass storage clean up

Severity: Medium Risk

Context: CurrencyGovernance.sol#L676-L684

Description: In a `CurrencyGovernance` cycle, the `leader` and `participation` values are deleted in the `commit()` function during the vote phase. This is so the `reveal()` function operates on cleared storage variables later on, and won't confuse values from across different cycles. This is especially important if the `enact()` function (which also deletes the `leader` storage variable) is never executed, which might be caused by a reverting proposal execution.

However, note that the `commit()` function does not prevent a trustee from committing to a *future* cycle and revealing their commitment once that cycle is reached. This implies it's possible to `reveal()` in a cycle where `commit()` was never called, meaning the `participation` and `leader` variables aren't necessarily cleared. While it's unlikely for all trustees to abstain in a cycle, one could leverage this scenario to pass a vote using the `participation` from a previous cycle, and this shouldn't be allowed.

Recommendation: Consider reworking the way that the `participation` and `leader` variables are cleared in each cycle. One implementation would be to:

1. Change `enact()` to delete the `participation` (in addition to how it currently deletes the `leader`).

2. Change `commit()` to not delete any storage variables.
3. Add a check in `reveal()` for the situation where a previous cycle never called `enact()`:

```
  function reveal(
       address _trustee,
       bytes32 _salt,
       Vote[] calldata _votes
   ) external duringRevealPhase {
       uint256 _cycle = getCurrentCycle();
       uint256 numVotes = _votes.length;
       if (numVotes == 0) {
           revert CannotVoteEmpty();
       }
       if (
           keccak256(abi.encode(_salt, _cycle, _trustee, _votes)) !=
           commitments[_trustee]
       ) {
           revert CommitMismatch();
       }

       // an easy way to prevent double counting votes
       delete commitments[_trustee];

-      participation += 1;
-      if (participation == quorum) {
-          emit QuorumReached();
-      }

       // use memory vars to store and track the changes of the leader
       bytes32 priorLeader = leader;
+      if (priorLeader != 0) {
+          // Check if this is a leader from a past cycle that was never enacted
+          if (priorLeader != _cycle && proposals[priorLeader].cycle != _cycle) {
+              delete participation;
+              delete leader;
+              priorLeader = 0;
+          }
+      }
       bytes32 leaderTracker = priorLeader;
       uint256 leaderRankTracker = 0;

+      participation += 1;
+      if (participation == quorum) {
+          emit QuorumReached();
+      }

      // ...
  }
```

This would solve any issues regarding cycles that bypass the `commit()` stage. Also, this change would extend the period when `enact()` can be called, since the `leader` and `participation` deletion no longer happen in the `commit()` function.

Eco: Fixed in PR 86.

Cantina Managed: Verified. The changes mentioned above have been made.

3.2.2 `setTrustedNodes` function should check trustees length is over or equal than quorum

Severity: Medium Risk

Context: CurrencyGovernance.sol#L419-L438

Description: The current implementation of the `setTrustedNodes` function of the `CurrencyGovernance.sol` governance allows to set the contract of `TrustedNodes` as global state variable. However this function does not check that the number of trusted nodes is bigger or equal than the quorum.

This can prevent to enact any proposal on the currency governance until this situation is changed from the policy contract through a proposal of the `CommunityGovernance`.

Recommendation: Consider adding a protection to prevent that `trustedNodes.numTrustees()` is not less than quorum unless quorum is zero, to ensure that the invariant of `quorum <= trustedNodes.numTrustees()` holds.

Eco: Fixed in PR 72.

Cantina Managed: Verified. The fix performs the suggested checks.

3.2.3 Approvals to the `Lockups` can be used unexpectedly

Severity: Medium Risk

Context: Lockups.sol#L177-L183

Description: In the `Lockups` contract, the `depositFor()` function allows a third party to deposit a beneficiary's funds that have been approved for the contract to spend. This is meant to be used alongside a `permit()` approval, so that users can be deposited into lockups without paying any gas.

However, even if a user has approved the `Lockups` contract and expects their funds to be used, they have no way of specifying *which* lockup id they want to deposit into. If there are multiple active lockup ids at a time, a griefer could abuse the `depositFor()` function to deposit people into lockups they aren't expecting. Since there are penalties associated with early withdrawals, this could be a major inconvenience.

Recommendation: Consider adding an EIP712 signature verification in the `depositFor()` function. This could be used to validate that the beneficiary intends to deposit into a specific lockup id, which would prevent griefing txs from succeeding.

Alternatively, consider removing the `depositFor()` function altogether, and force users to deposit for themselves.

Eco: Fixed in PR 90.

Cantina Managed: Verified. The `depositFor()` and `withdrawFor()` functions have been removed, so griefing is no longer possible.

3.3 Low Risk

3.3.1 Consider adding a minimum expected receivable amount on `exchange` function

Severity: Low Risk

Context: ECOxExchange.sol#L80-L84

Description: The current implementation of the `exchange` function of the `ECOxExchange` contract receives as parameter the amount of `Ecox` tokens to burn. In exchange the function computes the corresponding amount of `Eco` tokens and mints them for the msg.sender.

However the user can not specify a minimum of tokens to be received. Although under most scenarios this may not be relevant due to the design of the protocol it has been considered a reasonable measure to ensure users calculations are correctly satisfied.

Recommendation: Consider adding a `minAmountOut` variable that allows user to specify the amount of tokens they expect in return.

Eco: Fixed in PR 88 by adding additional documentation in the code. The two parameters that would affect the `exchange()` are the `ECO` total supply and the `initialSupply` of `ECOx` tokens stored in the `ECOxExchange` contract. The `initialSupply` is not planned to ever change, and a change in the `ECO` total supply will scale the `exchange()` output in a predictable manner.

Cantina Managed: Verified.

3.3.2 Create setters for non constant state variables on CommunityGovernance contract

Severity: Low Risk

Context: CommunityGovernance.sol#L77-L86l

Description: The next global state variables are not constant however can not be modified as they do not have setters:

- proposalFee
- feeRefund
- voteThresholdPercent

Recommendation: Consider creating a setter function only accessible with Policy role, for the afore mentioned variables. Nonetheless, if it is considered that their value may not require to change, consider declaring them as constants.

Eco: Fixed in PR 77.

Cantina Managed: Verified. All three variables have been changed to constants.

3.3.3 The global state variable supportThresholdPercent may require to be updated

Severity: Low Risk

Context: CommunityGovernance.sol#L83

Description: The supportThresholdPercent is used to measure the percentage of voting power required for a proposal to be elected for voting. If this threshold is not achieved the proposal can not reach the voting state transition.

Recommendation: Consider creating a setter for the global state variable supportThresholdPercent to allow voting state transitions for future proposals.

Eco: Fixed in PR 77.

Cantina Managed: Verified. The fix creates a setter function with Polciy role permission restriction access.

3.3.4 supportPartial should break from loop instead of reverting in certain cases

Severity: Low Risk

Context: CommunityGovernance.sol#L420-L438

Description: The current implementation of the supportPartial function from the CommunityGovernance.sol contract allows users to support on several proposals. However if any of the supported proposals achieves the minimum support threshold, the stage is automatically changed to Voting, reverting the transaction if any other proposal are left for support accounting.

Recommendation: Consider breaking the loop of the supported proposals if the state is changed after the execution of the internal function _changeSupport.

Eco: Fixed in PR 69.

Cantina Managed: Verified. The fix breaks from the loop avoiding to vote and revert on further proposals if Voting stage is reached.

3.3.5 `isOwnDelegate()` doesn't account for delegating an empty balance

Severity: Low Risk

Context: ERC20Delegated.sol#L135-L137, VoteCheckpoints.sol#L200-L205

Description: In both `ERC20Delegated` and `VoteCheckpoints`, the `isOwnDelegate()` function is used to determine if an account is currently not delegating to anyone. The implementation is as follows:

```
function isOwnDelegate(address account) public view returns (bool) {
    return _totalVoteAllowances[account] == 0;
}
```

Technically, it is possible that the account *is* delegating to someone, but their balance is zero. In this case, the `isOwnDelegate()` function will still return `true`.

This is unexpected, and can lead to undesired behavior in the `enableDelegationTo()` function. If an account is primary delegating but has zero balance, it will still be allowed to receive primary delegations from others, which is not supposed to be possible.

Fortunately, this unusual state doesn't lead to major problems, as all parties can exit their delegations normally.

Recommendation: To prevent confusion and prevent this unusual state, add an extra check in `isOwnDelegate()`:

```
function isOwnDelegate(address account) public view returns (bool) {
    return _totalVoteAllowances[account] == 0 && _primaryDelegates[account] == address(0);
}
```

With this code, an account that is primary delegating 0 balance will have `isOwnDelegate()` return `false`, which solves the issue.

Eco: Fixed in PR 68.

Cantina Managed: Verified. The fix implements the suggested extra check.

3.3.6 `updateStage()` should be called earlier in some functions

Severity: Low Risk

Context: CommunityGovernance.sol

Description: In the `CommunityGovernance` contract, `updateStage()` is a crucial helper function that advances the state of the contract. All important functions in this contract should call `updateStage()` as early as possible, to ensure that the logic is operating on the most up-to-date state.

Currently, the `support()`, `supportPartial()`, `unsupport()`, `vote()`, and `votePartial()` functions are calling `updateStage()` *after* calling the `votingPower()` function. This can be dangerous, as the later `updateStage()` *should* affect the `votingPower()` in certain situations (e.g. if the `updateStage()` initiates a snapshot), but it currently doesn't.

Fortunately, this doesn't seem exploitable, as any scenario where `updateStage()` affects the outcome of `votingPower()` would also advance the current cycle, which leaves no proposals for the voting power to actually interact with.

Recommendation: Update the five functions mentioned above to call `updateStage()` earlier. This can be accomplished by using an `updateStage()` modifier that's used in all the important functions.

Eco: Fixed in PR 80.

Cantina Managed: Verified. The fix creates the modifier `updateStage` and implements it on all the previous mentioned functions ensuring to call it in the appropriate time.

3.3.7 `Notifier` is not specifying the `gasCost` in external calls

Severity: Low Risk

Context: Notifier.sol#L82

Description: The `Notifier` contract can be used by the governance to attach non-atomic downstream calls to the actions taken by monetary policy levers. When these calls are queued, the governance specifies a `gasCost` value to be used in the call. However, this value is not being used during execution.

Recommendation: To make the `Notifier` more non-atomic, make use of this `gasCost` value as follows:

```
   function notify() public onlyLever {
       uint256 txCount = transactions.length;

       for (uint256 i = 0; i < txCount; i++) {
           Transaction memory t = transactions[i];
-          (bool success, ) = (t.target).call(t.data);
+          (bool success, ) = (t.target).call{gas: t.gasCost}(t.data);

           if (!success) {
               emit TransactionFailed(i, t.target, t.data);
           }
       }
   }
```

Eco: Fixed in PR 74.

Cantina Managed: Verified.

3.3.8 Threshold calculations can be more precise

Severity: Low Risk

Context: CommunityGovernance.sol#L479-L483, CommunityGovernance.sol#L572-L575

Description: In the `CommunityGovernance`, a proposal advances to the voting stage if it has reached the `supportThresholdPercent`:

```
if (
    prop.totalSupport >
    (totalVotingPower() * supportThresholdPercent) / 100
) {
    // ...
}
```

and a vote can be fast-tracked to execution if it has reached the `voteThresholdPercent`:

```
if (
    (totalEnactVotes) >
    (totalVotingPower() * voteThresholdPercent) / 100
) {
    // ...
}
```

In both code snippets, integer division is being used, which means the threshold values are being rounded down.

Recommendation: Consider making these calculations more precise by rearranging the equations as follows:

```
if (
    prop.totalSupport * 100 >
    (totalVotingPower() * supportThresholdPercent)
) {
    // ...
}
```

and

```
if (
    (totalEnactVotes) * 100 >
    (totalVotingPower() * voteThresholdPercent)
) {
    // ...
}
```

Eco: Fixed in PR 76.

Cantina Managed: Verified. The fix implements the recommended precision calculations.

3.4 Gas Optimization

3.4.1 Use `++i` operator instead of `i++` to save gas

Severity: Gas Optimization

Context: CommunityGovernance.sol#L431, CurrencyGovernance.sol#L766, Notifier.sol#L64, Notifier.sol#L80, TrustedNodes.sol#L120, TrustedNodes.sol#L198

Description: `++i` costs less gas compared to `i++` or `i += 1` for unsigned integer, as pre-increment is cheaper (about 5 gas per iteration). This statement is true even with the optimiser enabled.

Recommendation: Consider modifying the post-increment for pre-increment to save gas.

Eco: Fixed in PR 88.

Cantina Managed: Verified.

3.4.2 Consider changing strings for custom errors

Severity: Gas Optimization

Context: DelegatePermit.sol#L46C9-L50, ECOxExchange.sol#L116-L119, ERC20.sol#L186-L189, ERC20Delegated.sol#L124-L126, ERC20Pausable.sol#L31-L34, ERC20Permit.sol#L55, InflationSnapshots.sol#L212, TokenInit.sol#L34-L39, TotalSupplySnapshots.sol#L105-L108, VoteCheckpoints.sol#L150-L153, VoteSnapshots.sol#L85-L88, ForwardProxy.sol#L26, ForwardTarget.sol#L17-L20

Description: Since Solidity v0.8.4, the more gas-efficient custom-errors have been introduced. They allow passing dynamic data in the error and remove costly and repeated string error messages.

Recommendation: Consider replacing require statements with custom errors.

Eco: Acknowledged. We have decided not to make this switch since the project will be open source, and some existing tooling does not fully support custom errors.

Cantina Managed: Acknowledged.

3.5 Informational

3.5.1 `CommunityGovernance` constructor can add more input validation

Severity: Informational

Context: CommunityGovernance.sol#L235-L246

Description: In the constructor of the `CommunityGovernance` contract, the `currentStageEnd` storage variable is assigned using the `_cycleStart` input. It would be unexpected for this value to be set to a timestamp far in the past or far in the future, so extra input validation can potentially be added.

Recommendation: Consider adding additional input validation for the `_cycleStart` argument of the `CommunityGovernance` constructor.

Eco: Fixed in PR 75.

Cantina Managed: Verified.

3.5.2 Policy contract risk

Severity: Informational

Context: Policy.sol#L94-L107

Description: The `Policy` contract is designed to hold permission to modify and access core functionalities of the protocol. The way of triggering it is through a proposal with enough vote and support from the Community Governance. The `Policy` contract is declared as an immutable variable on the other protocol contracts.

If a malicious proposal triggers the `selfdestruct` operation code, the protocol may suffer a perpetual denial of service.

Recommendation: Ensure that proposals are detailed and explained properly to all the community. Consider hashing the code of the proposal when proposed and matching it with the code of the contract before the execution. Also ensure proposals are not proxied to avoid changes on the implementation contract before the execution.

It is important to notice that after **Cancun hard fork**, expected to be at the end of Q1 2024, this risk will nearly cease to exist due to EIP-4758.

Eco: Acknowledged. The timeline for submission and application of the changes in this repo to the ECO protocol are in line with EIP-4758 already being passed (end of Q1 2024). For this reason, we don't think we need to put in a mitigation to the issue and will be tracking both the timeline of the Cancun fork as well as the proposal submissions until that time.

Cantina Managed: Acknowledged.

3.5.3 `distrust` function does not check quorum before deleting a trustee

Severity: Informational

Context: TrustedNodes.sol#L175-L190

Description: The `distrust` function from the `TrustedNodes` contract deletes an specified address from the `trustee` array storage variable. The function does not check the quorum to ensure the proper function of the Currency Governance on that cycle.

However it is important to remark that this function can only be accessed by the Policy contract and thus, is expected to be used by a proposal that ensures the correct usage of the protocol. Moreover on the worst case scenario that the proposal may fail on appropriate usage, the only side effect is to disable Currency Governance for a Governance Cycle.

Recommendation: Consider enabling a mechanism that allows `distrust` function to comply with the requirements of Currency Governance to work as expected.

Eco: Fixed in PR 85 by allowing proposals to succeed if their participation is at least `trustedNodes.numTrustees()`. Additionally documented the importance of timing the quorum/node updates in PR 92.

Cantina Managed: Verified. This change allows proposals to succeed even if a `distrust()` reduces the number of trustees below the quorum. The importance of timing this functionality is indeed important and the added comments have helped to illustrate this.

3.5.4 Incorrect/unresolved comments

Severity: Informational

Context: VotingPower.sol#L50-L52, VotingPower.sol#L65-L69, ECOx.sol#L26-L29, IECO.sol#L7, CurrencyGovernance.sol#L624-L627, CurrencyGovernance.sol#L601-L602, Lockups.sol#L58-L60, Rebase.sol#L35, ECOxExchange.sol#L209, ECOxExchange.sol#L130-L132, ERC20Permit.sol#L35

Description: Throughout the codebase, there are some unresolved/old comments, or comments that are slightly incorrect. This includes:

1. The comments above `totalVotingPower()` indicate the function `"Calculates the total Voting Power by getting the total supply of ECO and adding total ECOX (multiplied by 10) and subtracting the excluded Voting Power"`. The last part about "excluded Voting Power" appears to be left over from a previous version and can be removed.
2. The comments above `votingPower()` have a typo that mentions `"vorting"` instead of `"voting"`.
3. The `ECOx` contract has a commented-out `PRECISION_BITS` constant that can be removed.
4. The `IECO` interface file contains a pending TODO comment saying `"TODO: make an interface for delegation"`.
5. The `supportProposal()` function has commented out logic for checking `p.supporters[msg.sender]`, which is logic that's indeed not necessary and can be removed.
6. The comments above `supportProposal()` mention `"need to link to borda count analysis by christian here"`.
7. The `Lockups` contract contains a pending TODO comment saying `"TODO: consider using an array for this after John solidifies gas optimization infra"`.
8. The `Rebase` contract has an old comment saying `"unclear how this works on the eco contract as of now, but ill shoot anyway"`.
9. The `generalExp()` function has a snippet of code that claims to `"divide by 33! and then add x^1 / 1! + x^0 / 0!"`. However, the addition of `x^0 / 0!` appears to have been deliberately removed from the code, since a subtraction of 1 is desired in the ECOx conversion formula.
10. The comments above `generalExp()` mention a `"maxExpArray"` that represents an upper bound value the function can be called with. However, this array does not actually exist in the code, and doesn't appear necessary. Indeed, the `generalExp()` function does not utilize the same `unchecked{}` block that's in the original `PrintFunctionGeneralExp` script, so overflows will be handled implicitly by Solidity v0.8. So, the mention of the `"maxExpArray"` can be removed.
11. The comments above the `ERC20Permit constructor()` mention that it is `"setting version to "1""`. However, the V1.5 codebase actually sets the version to "2" now, which is useful for preventing signature replay problems after the upgrade.

Recommendation: Consider adjusting or deleting each of these comments, so the documentation is more accurate.

Eco: Fixed point 8 in PR 70. Fixed all other points in PR 88.

Cantina Managed: Verified.

3.5.5 Prevent no-op lockup withdrawals

Severity: Informational

Context: Lockups.sol#L242-L283

Description: In the `Lockups` contract, it is possible to withdraw from non-existent lockup ids. This will not accomplish anything important, as zero ECO will be minted/transferred. However, this might be confusing for someone observing the `LockupWithdrawal()` event off-chain. This can also lead to wasted gas for users who do this by mistake.

Recommendation: Consider adding a revert if a withdrawal is for a non-existent id or for zero amount:

```
  function _withdraw(uint256 _lockupId, address _recipient) internal {
      Lockup storage lockup = lockups[_lockupId];
      uint256 gonsAmount = lockup.gonsBalances[_recipient];
      uint256 _currentInflationMultiplier = eco.inflationMultiplier();
      uint256 amount = gonsAmount / _currentInflationMultiplier;
      uint256 interest = lockup.interest[_recipient];
      address delegate = lockup.delegates[_recipient];

+     require(gonsAmount > 0, "non-existent/no prior deposit");

      // ...
  }
```

Eco: Fixed in PR 87.

Cantina Managed: Verified.

3.5.6 Add an extra revert in `votingPower()` for actions within the `snapshotBlock`

Severity: Informational

Context: VotingPower.sol#L71-L76

Description: In the `CommunityGovernance` contract, no supporting/voting actions should be allowed within the `snapshotBlock`, since the snapshot is not complete until the block is finished. This is enforced explicitly in the `totalVotingPower()` function, and implicitly in the `votingPower()` function due to a revert in `ecoXStaking.votingECOx()`. However, it could be worth the extra defense/clarity to add an explicit check within `votingPower()` as well.

Recommendation: Similar to `totalVotingPower()`, consider adding an explicit revert in `votingPower()` if the snapshot block is not complete:

```
  function votingPower(address _who) public view returns (uint256 total) {
+     if (block.number == snapshotBlock) {
+         revert NoAtomicActionsWithSnapshot();
+     }
      uint256 _power = ecoToken.voteBalanceSnapshot(_who);
      uint256 _powerX = ecoXStaking.votingECOx(_who, snapshotBlock);
      // ECOx has 10x the voting power of ECO per unit
      return _power + 10 * _powerX;
  }
```

Eco: Fixed in PR 79.

Cantina Managed: Verified.

3.5.7 Document the `VoteSnapshots` hook behavior

Severity: Informational

Context: VoteSnapshots.sol#L59-L72

Description: The `VoteSnapshots` contract implements the `_beforeVoteTokenTransfer()` hook to do any required snapshotting of voters that have an updated voting power. The implementation is as follows:

```
function _beforeVoteTokenTransfer(
    address from,
    address to,
    uint256 amount
) internal virtual override {
    super._beforeVoteTokenTransfer(from, to, amount);

    if (from != address(0) && voter[from]) {
        _updateAccountSnapshot(from);
    }
    if (to != address(0) && voter[to]) {
        _updateAccountSnapshot(to);
    }
}
```

Notice that the snapshot is only taken for addresses that have the `voter` mapping set. This implies that functions that enable/disable the `voter` mapping need to be careful with their ordering. The `enableVoting()` function needs to set the `voter` to `true` at the very start of the function, which it correctly does. A theoretical `disableVoting()` function (which doesn't exist currently) would have to set the `voter` to `false` at the very end of the function.

Recommendation: Consider documenting this behavior in a comment, so that it's clear for any potential future changes.

Eco: Fixed in PR 71.

Cantina Managed: Verified.

3.5.8 `_afterTokenTransfer()` hook logic can be simplified

Severity: Informational

Context: ERC20Delegated.sol#L363-L422, VoteCheckpoints.sol#L558-L603

Description: In the `ERC20Delegated` and `VoteCheckpoints` contracts, the `_afterTokenTransfer()` hook is used to transfer voting power whenever a mint/burn/transfer of the underlying token occurs. The logic that removes voting power from the `from` address is as follows:

```
bool fromVoter = voter[from];
// if the address has delegated, they might be transfering tokens allotted to someone else
if (fromVoter && !isOwnDelegate(from)) {
    uint256 _undelegatedAmount = _balances[from] +
        amount -
        _totalVoteAllowances[from];

    // check to see if tokens must be undelegated to transfer
    if (_undelegatedAmount < amount) {
        address _sourcePrimaryDelegate = getPrimaryDelegate(from);
        uint256 _sourcePrimaryDelegatement = voteAllowance(
            _sourcePrimaryDelegate,
            from
        );

        require(
            amount <= _undelegatedAmount + _sourcePrimaryDelegatement,
            "ERC20Delegated: delegation too complicated to transfer. Undelegate and simplify before trying
↪   again"
        );

        _undelegate(
            from,
            _sourcePrimaryDelegate,
            amount - _undelegatedAmount
        );
    }
}
```

While this logic is correct, a discussion with the Eco team led to the following conclusions:

- If the `from` address *is not* primary delegating, the `if (_undelegatedAmount < amount)` statement will necessarily revert. Indeed, this case means `_sourcePrimaryDelegate == from`, and thus `_sourcePrimaryDelegatement == 0`, so the require statement is guaranteed to fail.
- If the `from` address *is* primary delegating, then `_undelegatedAmount == 0` and `_sourcePrimaryDelegatement` is the `from` address' entire underlying balance. Since the actual transfer must have succeeded to reach the `_afterTokenTransfer()`, the underlying balance is already known to be sufficient, and all that needs to be done is undelegating `amount` from the primary delegate.

These conclusions imply the implementation can be simplified.

Recommendation: Consider simplifying this section of the code. The Eco team has already decided upon the following implementation:

```
// if the address has delegated, they might be transfering tokens allotted to someone else
if (fromVoter && !isOwnDelegate(from)) {
    address _sourcePrimaryDelegate = _primaryDelegates[from]; // cheaper than getPrimaryDelegate because we do
↪   the check to own delegate already
    if (_sourcePrimaryDelegate == address(0)) {
        // this combined with !isOwnDelegate(from) guarantees a partial delegate situation
        uint256 _undelegatedAmount = _balances[from] + // need to check if the transfer can be covered
            amount -
            _totalVoteAllowances[from];
        require(
            _undelegatedAmount >= amount, // can't undelegate in a partial delegate situation
            "ERC20Delegated: delegation too complicated to transfer. Undelegate and simplify before trying
↪   again"
        );
    } else {
        // the combination of !isOwnDelegate(from) and _sourcePrimaryDelegate != address(0) means that we're
↪   in a primary delegate situation where all funds are delegated
        // this means that we already know that amount < _sourcePrimaryDelegatement since
↪   _sourcePrimaryDelegatement == senderBalance
        _undelegate(from, _sourcePrimaryDelegate, amount);
    }
}
```

Eco: Changed `ERC20Delegated` in PR 71. The `VoteCheckpoints` contract was not changed, as `sECOx` is non-transferable and thus the optimization is not as important.

Cantina Managed: Verified.

3.5.9 Remove unused code, variable and events

Severity: Informational

Context: ECOx.sol#L19, Policed.sol#L34

Description: The global state variable `ecoXExchange` is not used as well as the setter function created to update this variable updateECOxExchange, and the event UpdatedECOxExchange.

Also, the event `NewPolicy` from the `Policed` contract is no longer required as policy is now immutable on all the protocol contracts.

Recommendation: Consider erasing the unused variable, events and code to reduce deployment cost and improve legibility and cleanliness.

Eco: Removed the `ecoXExchange` logic in PR 78. Removed the `NewPolicy` event in PR 90.

Cantina Managed: Verified.

3.5.10 Inflation snapshots can be taken too early

Severity: Informational

Context: InflationSnapshots.sol#L203-L217

Description: In the `TotalSupplySnapshots` and `VoteSnapshots` contracts, each snapshot is recorded on the first relevant action *after* the `currentSnapshotBlock` ends. On the other hand, the `InflationSnapshots` contract records its snapshot on the first relevant action *during* ***or*** *after* the `currentSnapshotBlock`.

This distinction is important, because a `snapshot()` and then a `rebase()` could happen in the same block. The intended behavior likely is that the `rebase()` affects the snapshot value (since the block hasn't ended yet), but it does not.

Recommendation: Change the implementation of `InflationSnapshots` to match the behavior of the two other snapshot contracts. This can be accomplished by adding an early return in `_updateInflationSnapshot()` if the snapshot block is not complete:

```
  function _updateInflationSnapshot() private {
-     // rebase function is guaranteed to have a new snapshot before manipulating the value so we don't need
↪   as strict checks as balances
+     // take no action during the snapshot block, only after it
      uint32 _currentSnapshotBlock = currentSnapshotBlock;
+     if (_currentSnapshotBlock == block.number) {
+         return;
+     }
      if (
          _inflationMultiplierSnapshot.snapshotBlock < _currentSnapshotBlock
      ) {
          uint256 currentValue = inflationMultiplier;
          require(
              currentValue <= type(uint224).max,
              "InflationSnapshots: new snapshot cannot be casted safely"
          );
          _inflationMultiplierSnapshot.snapshotBlock = _currentSnapshotBlock;
          _inflationMultiplierSnapshot.value = uint224(currentValue);
      }
  }
```

Eco: Fixed in PR 84. Added more accurate comments in PR 90.

Cantina Managed: Verified.

4 Appendix

4.1 Issues Raised by the Client

4.1.1 `getPastLinearInflation()` isn't backwards compatible

Severity: Medium

Context: InflationSnapshots.sol#L132-L136

Description: During the audit, the Eco team raised this issue. Added here for tracking purposes.

In parts of the old codebase, the `getPastLinearInflation()` function is called with `block.number` as its argument, and it's expected that this returns the current inflation multiplier. However, the new version of this function only returns the most recent *snapshot* of the multiplier, which may be out-of-date.

Recommendation: Return the current `inflationMultiplier` in `getPastLinearInflation()`:

```
  function getPastLinearInflation(
      uint256
  ) public view returns (uint256 pastLinearInflationMultiplier) {
-     return inflationMultiplierSnapshot();
+     return inflationMultiplier;
  }
```

Eco: Fixed in PR 83.

Cantina Managed: Verified.