Eco Contracts Audit

OPENZEPPELIN SECURITY | JANUARY 14, 2020

Security Audits

Get the title[uncode_info_box items="Date,Categories,Author|no_avatar|inline_avatar|display_prefix" text_font="font-202125" text_transform="uppercase" separator="pipe"]The Eco team asked us to review and audit the contracts for their open payment network. We looked at the code and have now published our results.

About Eco

Eco is building a transactional cryptocurrency and an open payment network, both designed to maximize user participation, transparency and consumer protection. The first version of this system is built on Ethereum.

The Eco cryptocurrency is intended to demonstrate lower volatility over time, not as a pegged stablecoin but as a floating currency with active governance by expert users. These elected experts monitor the state of the Eco economy, and periodically vote on adjustments to key economic variables (including supply changes, new supply distribution, internal interest rates, transaction fees, etc.).

Eco's system is extremely modular. Every component and contract can be upgraded or even replaced through community governance, in which every Eco token holder is able to vote.

Learn more about the Eco project on their Frequently Asked Questions site. Additional technical documentation and their whitepaper should be available soon after the publication of this audit

auditing the codebase, so you will find references to the old name in some of the original commits.

About the audit

It has been a year long journey since we met the Eco team and their project. This engagement started right at the time when we were restructuring our audits team. The objective of our new structure was to research best practices and security vulnerabilities of systems more complex and revolutionary than the ones we had audited before, so this was the best possible system to get started with. The engagement is ending now with our team bigger, stronger, and happier than ever before.

It feels like the Eco system was evolving at the same time that it was helping us with our evolution. This is what you will see in this huge consolidated report of the 5 phases of the audit: a plurality of voices, approaches, and technologies that our team used to help improve the quality and security of the projects that the Eco team was building.

In the first phase, we audited 3 components: Deploy, Proxy and Policy. The Deploy component takes care of bootstrapping the system and getting the smart contracts into the blockchain. It uses a clever idea proposed by Nick Johnson to deploy to all the test networks and to the mainnet using the same addresses. The Proxy component allows the smart contracts to be upgraded. The Policy component is a generic management framework that allows some contracts to be managed by others.

The second phase was for the Currency component, which is the implementation of the Eco token.

The third phase was for the Governance component, which is in charge of the voting for policy changes and the voting for inflation and deflation decisions.

In the fourth phase, we audited the implementation of a Verifiable Delay Function, that is used for the commit and reveal schema needed for the votes.

In the fifth phase, we reviewed all the changes that the Eco team has implemented to fix the vulnerabilities we have reported. We also spent some time analyzing the system as a whole and trying to attack the integration points of all the components.

By now, almost every single member of the OpenZeppelin team has participated in some way on this project. Some have tried very hard to break the system, some have joined discussions to come up with ideas to simplify very complicated operations, some have spent long nights obsessing about deciphering that last line of assembly code…

As we got deeper into the code, we got more engaged with the project. As the Eco team started fixing the problems we were finding, we got stricter and more demanding. By the end, we had collected **8 vulnerabilities of critical severity** and **9 vulnerabilities of high severity**, along with many less critical issues and multiple suggestions to make the code clearer. **Most of them have been successfully fixed by the Eco team by now**. As one of our teammates said, it was amazing to see the improvements in the code during this year. Now, they have some of the best contracts we have seen in terms of organization, documentation, and readability.

You can see the reports of the individual phases for extensive details of every vulnerability found and how they have been fixed or mitigated.

Recommendations before releasing to production

- Make all the audited repositories public. Parts of the code have been integrated into a single repository, so the old repositories should be deprecated.
- Increase the unit test coverage to 100%.
- Limit future pull requests to one single and independent change for improved traceability.
- Move the documentation of the contracts API from the `README` files to a documentation website, like the OpenZeppelin Contracts website (which uses `solidity-docgen`).
- Perform an audit to verify the cryptographic strength of the Verifiable Delay Function.
- Perform a game theoretic evaluation of the incentives of the system. In particular, note that votes that use a past snapshot of the balances could have perverse incentives. Users may sell their tokens before casting their votes, having nothing at stake when the results are executed. Game-theoretical analysis is a service we have recently started offering at OpenZeppelin, led by Austin Williams.
- Start building a community of users and contributors. We recommend the books and resources published by Jono Bacon.

during testing. We recommend the HackerOne platform.

- Publish guides explaining how to write new policy contracts.
- Write automated validators to ensure that new policy contracts implement the correct interfaces and will not break the system. You can follow a similar approach to OpenZeppelin's ERC20 verifier.
- If the Eco token implements the ERC777 standard, the hooks that accounts can execute introduce the risk of reentrancy attacks and can revert operations. Be extremely careful with the transfer of these tokens: Use the checks-effects-interactions pattern, pull payments, reentrancy guards, or execute the transfer without triggering the hooks.
- After the upcoming Istanbul fork, the reprice of EIP1884 will affect the implementation of proxy contracts with a payable fallback function. When adding upgrades or new components to the system, use pull payments or the `sendValue` helper.
- Move the `BigNumber` library to a separate repository. The OpenZeppelin Contracts maintainers might be interested in adding it to the project and co-maintaining it.
- Consider adding a Super User module during the first months of production. This will allow for emergency upgrades to the contracts that cannot wait for the Policy vote period. Then, it can be removed when the project has proved to be stable and ready for full decentralization.
- Consider using Meta Transactions for easier on-boarding of new users and to allow them to interact with Eco tokens, even if they do not have ether.

JTNDZGl2JTIwY2xhc3MlM0QlMjJidG4tY29udGFpbmVyJTIyJTNFJTBBJTBBJTNDYnV0dG9uJTIwb25jbGljayUzRCUyMmN1c3RvbXNjcm9sbCUyOCUyOSUyMiUzRSUzQ2ElMjBocmVmJTNEJTIyJTIzcGhhc2UtMSUyMiUyMGNsYXNzJTNEJTIyY3VzdG9tLWxpbmslMjIlM0VyZWFkJTIwZnVsbCUyMHJlcG9ydCUzQyUyRmElM0UlM0MlMkZidXR0b24lM0UlMEElMEElMkZkaXYlM0U=

Phase 1: Deploy, Policy and Proxy

The phase 1 audit was delivered to the Eco team on December 17th, 2018.

The locations and versions used for this report are:

Governance:

- Repository 2 [deploy and proxy components]: https://github.com/BeamNetwork/beam-bootstrap-chain—Commit: `96beab1c5cfd41310bfba733ab7216426caf4a38`
- Repository 3 [policy component]: https://github.com/BeamNetwork/policed-contracts—Commit: `f9299f43e2bf3629bee2f82bf637918ac9221d62`

Our biggest challenge during the audit was to understand how the three repositories for deployment, bootstrap, and proxy forwarding work together. We consider this process to be very complicated. Because the whole system will be built on top of this phase, it becomes one of the most vulnerable parts of the project.

We think the Eco team should consider simplifying the process by using and contributing to other projects that are focused on infrastructure so they can focus instead on building the payments network. For example, the OpenZeppelin SDK provides an upgrade proxy that is already tested and in production, so they can use it instead of having to test and maintain the risky assembly blocks on `beam-bootstrap-chain` and `policed-contracts`. By using OpenZeppelin, they will not need to deploy the contracts using Nick's method because the configuration files store the locations for every package on all the networks. Another example is Aragon, which provides voting solutions that can be used to govern when and how to modify the parameters of the system.

This is not to say that nobody should develop their own infrastructure if what is out there does not fit their needs. However, it should be taken into account that these are complex problems on their own and that this code will have to be a lot better documented and tested on isolation and end-to-end in order to ensure its safety.

Here are our audit assessment and recommendations, in order of importance.

Update: *The Eco team made some fixes based on our recommendations. We address below the fixes introduced up to commit* `af3428020545e3f3ae2f3567b94e1fbc5e5bdb4c` *of the currency repository, which now includes all the previously audited files of the deploy, proxy, and policy components.*

Critical Severity

[P1-C01] The bootstrap process can be hijacked

The `BeamBootstrap` contract stores its owner in the `deployAccount` variable, which also gets transferred to 20 `ForwardProxy` contracts through `BeamInitializable`.

The `BeamInitializable` contract also uses an ownership pattern, which allows only the owner to set the implementation that will be delegated from the proxy and to `selfdestruct` the contract.

The issue here is that the `initialize` function is `public`. It does not have a restriction that only the owner can call, and it can be called more than once.

For an attacker to hijack the entire Bootstrap process, it would only require them to deploy a malicious contract like this:

```
contract Malicious {
    address public owner;

    constructor(address _owner) public {
        owner = _owner;
    }
}
```

This sets an address in their control as the `owner`. Then, they can feed it to the 20 `FordwardProxy`s through a `BeamInitializable` interface, call `initialize` with the address of the `Malicious` contract, and give themselves ownership of all the `ForwardProxy`s.

They can also get ownership of the original `BeamInitializable` contract, but that does not seem to give them any gain.

Test case: https://gist.github.com/mattaereal/9a7fe9d20b3c3253b1effe049cb9211e

Consider requiring that only the `owner` can call the `initialize` function, thus making it possible to call it only once during initialization with the `onlyConstruction` modifier.

any account, not just by its current owner. See statement for this issue.

> The problem highlighted here has two parts to consider. First, the addresses reserved by `EcoBootstrap` respond to the `initialize` call described. These proxy contracts have their `implementation` slot set, so the `onlyConstruction` modifier will prevent calls. The modifier might not protect the underlying implementation contract though – since the underlying implementation isn't itself a proxy contract the `implementation` slot might not be set. Fortunately, the `ForwardTarget` constructor is run when the `EcoInitializable` contract is deployed, and it sets the `implementation` slot. This makes it impossible to call `initialize` on the underlying implementation contract.
>
> Per OpenZeppelin's audit, `initialize` can only be called once on any proxy, and it's always called immediately during proxy construction – ensured by the `ForwardProxy` constructor. Further, the `onlyConstruction` modifier checks the `implementation` slot, which is set in the `EcoInitializable` constructor preventing any calls at all on contracts that are deployed directly. OpenZeppelin's finding is correct, any account can make the one permissible call `initialize`, but there is no exposure here because either the `ForwardProxy` constructor or the `EcoInitializable` constructor always consumes the call.

[P1-C02] Storage collision on Policed contract

Critical

Component: `policy`

`Policy` contracts have the power to enforce actions over `Policed` contracts (i.e., the latter are managed by the former). This means that `Policy` contracts are allowed to execute arbitrary code in a `Policed` contract context, as long as the `Policy` contract is set as the manager during construction.

This enforcement of actions is done via the `policyCommand` function of the `Policed` contract, which can only be executed by the managing policy – this last restriction being enforced by the custom `onlyPolicy` modifier.

may be initially considered safe, since it is the `Policy` contract that is triggering it, the logic contract in charge of executing the code may unexpectedly overwrite the whole `Policed` contract storage due to the storage collision vulnerability. This vulnerability can be exploited as follows.

Given a Policy contract such as:

```
pragma solidity ^0.4.24;

import "../contracts/Policy.sol";
import "../contracts/Policed.sol";
contract BadPolicy is Policy {
    function execute(address policedAddr, address logicAddress,
        // This contract manages the contract at `policedAddr`,
        Policed(policedAddr).policyCommand(logicAddress, data);
    }
}
```

And the following contract with some logic that the Policy is going to enforce over the Policed contract:

```
pragma solidity ^0.4.24;

contract BadLogic {
    address public firstSlot = address(0x123);
    address public secondSlot = address(0x123);

    event Overwrite();

    /**
     * Instead of writing to this contract state variables,
     * when overwriter is called via delegatecall, it will end
     * writing to the caller's state variables at first and sec
```

```
        firstSlot = address(0);
        secondSlot = address(0);
    }

    // [... some more logic ...]
}
```

By looking at the `overwriter` function, one could initially think that it will just write over its `firstSlot` and `secondSlot` state variables, no matter how it is called. However, as the logic contract will be called via a `delegatecall`, the `overwriter` function will unexpectedly write over the first and second slot of the `Policed` contract (i.e., over the `implementation` variable in the first slot and the `policy` variable in the second slot, with the former inherited from `ForwardTarget`), without ever writing to its own state variables `firstSlot` and `secondSlot`.

Here is a snippet of the Policed contract showing its state variable `policy`:

```
pragma solidity ^0.4.25;
pragma experimental "v0.5.0";

import "erc820/contracts/ERC820ImplementerInterface.sol";
import "@beamnetwork/bootstrap-chain/contracts/ForwardTarget.so


/** @title Policed Contracts
 *
 * A policed contract is any contract managed by a policy.
 */
contract Policed is ForwardTarget, ERC820ImplementerInterface {
    address public policy;
// [...]
}
```

A proof of concept of this issue goes as follows:

```
const Policed = artifacts.require('Policed');
const assert = require('assert');

contract.only('Policed contract', () => {
  const NULL_ADDRESS = '0x0000000000000000000000000000000000000000

  beforeEach(async function () {
    this.badPolicy = await BadPolicy.new();

    // BadPolicy contract manages the Policed contract
    this.policed = await Policed.new(this.badPolicy.address);

    this.badlogic = await BadLogic.new();
  });

  describe('Storage collision', () => {
    it('overwrites Policed contract storage', async function ()
      // First, assert that initially the storage is not null

      // `implementation` is at slot 1, inherited from ForwardT
      assert.notEqual(await this.policed.implementation(), 0);
      // `policy` is at slot 2, defined at Policed contract
      assert.equal(await this.policed.policy(), this.badPolicy.

      // Make the managing policy execute the `overwriter` func
      const functionSignature = web3.utils.sha3('overwriter()')
      await this.badPolicy.execute(this.policed.address, this.b

      // Assert that the state variables at BadLogic were never
      assert.notEqual(await this.badlogic.firstSlot(), NULL_ADD
      assert.notEqual(await this.badlogic.secondSlot(), NULL_AD

      // Assert that finally the whole storage
      // was accidentally overwritten by BadLogic.sol#overwrite
      assert.equal(await this.policed.implementation(), 0);
      assert.equal(await this.policed.policy(), NULL_ADDRESS);
```

Low-level calls such as `delegatecall` are always to be implemented with utter care and be thoroughly tested. Strategies to avoid this vulnerability, along with further cases and explanations, can be found at OpenZeppelin SDK documentation on unstructured storage.

Update: *Partially fixed. The `implementation` is now stored in unstructured storage. The `policy` variable is still stored in the first slot of the delegator contract, making it vulnerable to storage collisions. Eco's statement for this issue:*

> The unstructured storage approach recommended by OpenZeppelin here does help prevent accidental storage collisions, but at the same time it significantly complicates maintenance and upgrades in the future. Eco's engineering team has adopted OpenZeppelin's unstructured storage approach for the core proxy functionality (the `implementation` slot), but prefers to adopt policies and develop tools to manage the risk for other parts of the system. Specifically, when dealing with the `policy` slot identified here, we plan to extend our linter rules and thoroughly test upgrades to ensure collisions do not cause problems.

High Severity

[P1-H01] Missing test coverage reports

High

Components: all

There are no test coverage reports on any of the repositories. Without these reports, it is impossible to know whether there are parts of the code which are never executed by the automated tests. For every change, a full manual test suite has to be executed to make sure that nothing is broken or misbehaving.

Consider adding the test coverage reports and making it reach at least 95% of the source code.

Update: *Partially fixed. The test coverage report was added to the currency repository, reporting 92% of coverage. Eco's statement for this issue:*

> us to iterate more quickly on the JavaScript code, while still covering the core functionality.

[P1-H02] Low unit test coverage

High

Components: all

In the `deployment-nicks-method` repository, there are no tests at all.

In `beam-bootstrap-chain`:

- Test case file named `compute-gas.js` only deploys a `BeamBootstrap` contract and nothing more. It does not compute gas as supposedly intended.
- The `ForwardTarget` and `BeamBootstrap` contracts have no unit tests.

In `policed-contracts`, only the `Policed` contract has tests.

Tests are the best way to specify the expected behavior of a system. Automating these tests and running them continuously pins down the behavior to make sure that it is not broken by future changes.

Consider adding unit tests for every code path. Consider running the tests on every pull request.

Update: *Partially fixed. Extensive tests have been added to the `currency` repository. However, it has 92% of test coverage. See Eco's statement for this issue in* ***[P1-H01] Missing test coverage reports****.*

[P1-H03] Outdated ERC820

High

Component: `policy`

The address used is `0x820c4597Fc3E4193282576750Ea4fcfe34DdF0a7`.

The correct address is `0x820b586C8C28125366C998641B09DCbE7d4cBF06`.

Also, the `npm` dependency `"erc820": "0.0.22"` is outdated. Moreover, it is set as fixed in the `package.json` file, so not even minor changes in the versions will be downloaded when users run `npm install`.

The address and version used correspond to a previous version while the standard was a work in progress. Now, the standard is on the final call, but there is still a chance that a new version will be released and/or that the address will change.

Consider changing the address of the `ERC820Registry` to point it to the actual one. Consider updating the version of the `erc820` package and allowing it to install newer minor versions.

For further reference: https://eips.ethereum.org/EIPS/eip-820

Before deploying to production, confirm that the EIP is final, that the address has not changed, and that the package version used is the latest.

Update: *Fixed. The EIP 820 is now final, but the Eco team found and reported an issue to the implementation. A new EIP 1820 was proposed to fix it, and it now supersedes EIP 820. The contracts are now using the upstream implementation of EIP 1820.*

Medium Severity

[P1-M01] There are multiple hard-coded constants

Medium

Components: all

Hard-coded booleans, numbers, and strings in the code are hard to understand, and they are prone to error, because after some time, their origin and context can be forgotten or mistaken.

For example:

In `beam-bootstrap-chain`:

- `BeamBootstrap.sol`:L17: hard-coded `20`
- `truffle.js`:L9: hard-coded address

In `policed-contracts`:

- `Policy.sol`:L59: hard-coded address
- `erc820.js`:L5: bytecode without comment
- `erc820.js`:L9: hard-coded address
- `erc820.js`:L10: hard-coded address

Consider defining a constant variable for every hard-coded value, giving it a clear and explanatory name. For the complex values, consider adding a comment explaining how they were calculated or why they were chosen.

Update: *Fixed. All the hard-coded values mentioned above have been moved to constants, commented, or removed.*

[P1-M02] Unit tests are not verifying a single condition

Medium

Components: all

Every unit test should verify a single code path to ensure isolation and to make it easier to blame a single line of code when one of the tests fails. The test structure should follow the four phase pattern, clearly separating the interactions with the system under test from the verification.

Consider splitting the tests to make them fully isolated and as short and focused as possible. Consider following the four phase pattern on each of them. Consider following a data-driven pattern for test cases with multiple scenarios like the abs tests.

Update: *Fixed. Many tests have been added with a nice and clean style.*

Medium

Components: `policy` and `beam-bootstrap-chain`

In `beam-bootstrap-chain/truffle.js` and `policed-contracts/truffle.js`, Solidity optimizations are enabled.

Solidity has some optimizations that are default and are always executed, and some others are optional. Enabling the optional ones increases the risk of unexpected behavior, since they are not as battle-tested as the default optimizations. Consider adding full test coverage without optimizations before enabling them to verify that the introduced optimizations preserve expected behavior.

Update: *Partially fixed. Test coverage has improved a lot, but it is still not 100%. A note has been added to the README to take this risk into consideration. Eco's statement for this issue:*

> Our Solidity test coverage is 98%. While this is slightly less than 100%, we're satisfied that it covers the risk introduced by using the Solidity optimizer.

[P1-M04] Overriding `initialize` is confusing

Medium

Component: `proxy`

The `ForwardTarget` contract defines and implements the `initialize` function. Then, the `BeamInitializable` contract inherits from `ForwardTarget` and overrides the `initialize` function.

It is not clear if this function is intended to be overridden, but if so, it is not clear why it has an implementation.

Consider documenting the intended use of `ForwardTarget` as a base class, explaining when and how to override the `initialize` function. Also, consider commenting on the overriding functions to make it clear that they are not shadowing members from the base class by mistake.

Medium

Components: `deploy` and `proxy`

The functions that take an address as an argument are not validating that the address is not `0`.

For example:

- `BeamBootstrap.sol`:L14
- `BeamInitializable.sol`:L11
- `ForwardProxy.sol`:L13
- `ForwardTarget.sol`:L21

In most cases, passing a `0` address is a mistake.

Consider adding a require statement to check that the address is different from `address(0)`.

Update: *Fixed only in EcoBootstrap.sol. Eco's statement for this issue:*

> Thorough testing prevents our system from mistakenly passing the 0 address when initializing a contract or configuring a proxy. Inserting code to prevent the 0 address from being passed at all also prevents us from doing so intentionally, so as a practice we avoid disallowing the 0 address in code.

[P1-M06] Unclear public cloning functionality

Medium

Component: `policy`

The `PolicedUtils` contract includes a **public** function `clone`, which according to the docstrings, "*Creates a clone of this contract by instantiating a proxy at a new address and initializing it based on the current contract […].*"

The single test that attempts to cover the cloning functionality is too vague and poorly documented to understand what is being tested and, more importantly, why. Furthermore, it strangely uses a mock to wrap the `clone` function inside another `cloneMe` function, the latter being the one called during the test.

Consider including further tests (while enhancing the existing one) and more thorough documentation regarding the cloning functionality. Moreover, analyze restricting the visibility of the function through custom modifiers if it is not supposed to be called publicly but only by certain privileged accounts.

Update: *Fixed. More documentation has been added to the clone function. The test has been improved, but note that the cloneMe function is still confusing.*

[P1-M07] Lack of input validation

Medium

Component: `policy`

In multiple functions, the values received as arguments are not validated.

In `PolicyInit.sol`:

- In the `fusedInit` function:
  - Not checking for empty arrays: `_setters`, `_keys`, `_values`, `_tokenResolvers`
  - Not checking for zero address: `_policyCode`
  - Not checking that `uint256(_policycode)` is different from the current `implementation`. Otherwise, `implementation` might not be actually changed, and `fusedInit` could be potentially called again.

In `Policy.sol`:

- In the `internalCommand` function, not checking for zero address: `_delegate`

- In the `initialize` function, not checking for zero address: `_self`

In `PolicedUtils.sol`:

- In the `initialize` function, not checking for zero address: `_self`
- In the `setExpectedInterfaceSet` function, not checking for zero address: `_addr`

Consider implementing `require` clauses where appropriate to validate all user-controlled input.

Update: *Only the check for a different implementation has been fixed. Eco's statement for this issue:*

> The `fusedInit` function of `PolicyInit` is only called by our own code during deployment and immediately disabled, so we're not concerned about invalid inputs. `Policy`, `Policed`, and `PolicedUtils` are library contracts that are built on top of our contracts. User-provided addresses are never passed to the functions described, so we're not concerned about accidental 0 address values.

Low Severity

[P1-L01] README empty or missing important information

Low

Components: all

The `README.md` on the root of the GitHub repositories are the first documents that most developers will read, so they should be complete, clear, concise, and accurate.

The `README.md` files of `deployment-nicks-method`, `beam-bootstrap-chain` and `policed-contracts`, have little or no information about what the purpose of the project is nor how to use it.

In `beam-bootstrap-chain`, it describes a set of instructions to create a bootstrap transaction without the needed context to understand it, including a command named `generate-`

Consider following Standard Readme to define the structure and contents for the `README.md` file. Consider including an explanation of the core concepts of each repository, the usage workflows, the public APIs, instructions to test and deploy them independently, and how they relate to the other repositories in the Beam project.

Update: *Fixed. All repositories and some sub-directories now have good README files.*

[P1-L02] Missing comments on the code and comments not following NatSpec

Low

Components: all

In the `beam-bootstrap-chain` and `policed-contracts` repositories, there are some contracts, struct fields, state variables, mapping keys, functions and parameters without comments, some with comments that do not follow the NatSpec format, and some that are missing important details on their comments.

In the `deployment-nicks-method` repository, there are no comments at all.

This makes it hard to understand the code and to review that the implemented functionality matches the intended behavior.

Consider adding docstrings for all contracts, struct fields, state variables, mapping keys, and functions, paying particular attention to the ones that represent complicated concepts. Consider following the NatSpec format thoroughly for docstrings to improve documentation and code legibility.

Update: *Fixed. Many comments have been added to all the contracts. Most of them exhaustively document the parameters and strictly follow the NatSpec format.*

[P1-L03] Not following consistent code style

Low

possible to combine contributions from wildly diverse people, as is the case in open source projects.

Consider making every file in the project follow the documented code style guide and enforce that every new contribution sticks to this code style by adding a linter check that runs on every pull request.

Note that `eslint` does not enable any rules by default, so at least, the recommended rules should be added to the configuration file.

Update: *Fixed. The two repositories now follow a consistent code style and have rules and tasks for running* `solhint` *and* `eslint`*.*

[P1-L04] Functions can be made external to clarify intended use

Low

Components: all

In `beam-boostrap-chain`, `BeamBootstrap`'s `cleanup`, `BeamInitializable`'s `cleanup`, and `setImplementation` are never called internally.

In `policed-contracts`, `policyFor` is never called internally.

Consider making the functions `external` to clarify intended use. If they should be `public`, consider documenting the reason and intended use from inside the contracts.

Update: *Fixed.*

[P1-L05] Undocumented assembly code

Low

Components: `policy`, `deploy`, and `proxy`

This is a low-level language that is difficult to understand.

Consider documenting the mentioned function in detail to explain the rationale behind the use of assembly and to clarify what every single assembly instruction is doing. This will make it easier for users to trust the code, for reviewers to verify it, and for developers to build on top of it or update it.

Note that the use of assembly discards several important safety features of Solidity, so this is a highly vulnerable part of the project. Consider implementing thorough tests to cover all potential use cases of the `ForwardProxy` contract to ensure it behaves as expected.

Update: *Fixed. Extensive documentation has been added to the assembly code blocks. Tests have been added to the policy and bootstrap contracts.*

[P1-L06] Unused imports

Low

Components: `policy`, `deploy`, and `proxy`

In `beam-bootstrap-chain`, `SampleForward.sol` imports `ForwardProxy` and never uses it.

In `policed-contracts`, `Policy` and `PolicedUtils` import `ERC820ImplementerInterface` and never use it.

Consider removing unused imports.

Update: *Fixed. The unused imports mentioned above have been removed.*

[P1-L07] Not using the latest OpenZeppelin Contracts library

Low

Components: `policy` and `deploy`

Consider updating the projects to the latest `openzeppelin-solidity` version.

Update: *Fixed. The openzeppelin-solidity dependency has been updated.*

[P1-L08] There are multiple typos

Low

Components: `policy`

There are multiple typos in the repository:

In `policed-contracts`:

- In `Policed.sol` and `PolicedUtils.sol`: pf instead of “of”

Consider running codespell on pull requests.

Update: *Fixed. The typos mentioned above have been fixed.*

[P1-L09] Reimplementing the Ownable pattern

Low

Component: `deploy`

`BeamInitializable` and `BeamBootstrap` implement the Owner pattern (with the latter calling it `deployAccount`).

Consider using the `Ownable` contract from OpenZeppelin instead of reimplementing it.

Update: A comment has been added explaining the reason for not using `Ownable` from OpenZeppelin.

[P1-L10] Unnecessary empty constructors defined

Both `Policy` and `PolicyInit` define empty constructors, which is not necessary and only hinders code readability. According to the Solidity docs on constructors, "*If there is no constructor, the contract will assume the default constructor:* `constructor() public {}`."

Consider removing all empty constructors from the codebase.

Update: *Fixed. Policy and PolicyInit no longer have empty constructors.*

[P1-L11] Proxy delegation logic can be simplified

Low

Component: `policy`

The fallback function in charge of the delegation process for the proxy can be simplified. It is not necessary to use the free memory pointer, and `returndatasize` can be saved to a variable instead of calling it twice. Also, the result of `delegatecall` can be checked after the `returndatacopy` to avoid the duplicated statement.

Because this code is hard to understand and Solidity does not check its safety, it is a good idea to make it as short and clear as possible.

Consider refactoring the assembly block. See the OpenZeppelin SDK delegate implementation as an example of a thoroughly tested alternative.

Update: *Partially fixed. The proxy implementation no longer uses the free memory pointer. Eco's statement for this issue:*

> The current implementation minimizes gas overhead, which was an important design consideration for us.

Notes & Additional Information

- All the contracts in the `beam-bootstrap-chain` and `policed-contracts` repositories use the Solidity `pragma experimental "v0.5.0";` statement. This is a

`experimental` to mainnet. Before releasing to mainnet, consider removing it and just use a stable Solidity version.

Update: *Fixed. Eco now uses Solidity 0.5.12 without the experimental pragma.*

- Good naming is one of the keys for readable code. Clean code facilitates future changes. To favor explicitness and readability, several parts may benefit from better naming. Our suggestions are:
  - `deployment-nicks-method` to `deployment-same-address`
  - In `beam-bootstrap-chain`:
    - `deployAccount` to `owner`
    - `deployed` to something like `placeholders` or `proxies`
    - `cleanup` (in `BeamBootstrap` and `BeamInitializable`) to `destruct`
    - `cycle.js` to `full_cycle_upgrade_demo.js`
  - In `policed-contracts`:
    - In `Policy.sol`:
      - `_id` to `interfaceHash`
      - `_policyCode` to `_policy`
    - In `test/policed.js`:
      - `dummy` to `testPoliced` or `samplePoliced`
      - `init` to `policyInit`
      - `proxy` to `forwardProxy`
      - `fake` to `fakePolicy`
      - `addr` to `proxyAddress`
      - `hash` to `dummy`
  - ***Update:*** *Some parts of the code have been renamed.*
- Several variables are not explicitly declared as `uint256` or `int256` but just as `uint` or `int`. Explicit is better than implicit. Consider changing all occurrences of this issue to improve code legibility and avoid any confusion.
  - In `beam-bootstrap-chain`:
    - `Migrations.sol`: L12, L25
    - `SampleForward.sol`: L14, L33, L40-L45, L58, L69, L79, L88
  - In `policed-contracts`:

- ***Update:*** *Fixed.*

- In many code segments of the `policed-contracts` repository, `this` is used instead of `address(this)` to refer to the contract's address. For example, in `PolicedUtils.sol`:L26,L50,L66,L67 and in `PolicyInit.sol`:L54,L58,L60, to favor explicitness and code readability, consider casting `this` to `address(this)` where appropriate.
  Update: *`PolicedUtils` and `PolicyInit` are now casting `this` to `address`.*
- In the test file `cycle.js` of `beam-bootstrap-chain`, some contract artifacts are required by name and some by path. Consider following a consistent style for all the required artifacts.
  Update: *All artifacts are now required by name.*

JTNDZGl2JTIwY2xhc3MlM0QlMjJidG4tY29udGFpbmVyJTIyJTNFJTBBJTBBJTNDYnV0dG9uJTIwb25jbGljayUzRCUyMmN1c3RvbXNjcm9sbCUyOCUyOSUyMiUzRSUzQ2ElMjBocmVmJTNEJTIyJTIzc3VtbWFyeSUyMiUyMGNsYXNzJTNEJTIyY3VzdG9tLWxpbmslMjIlM0UlM0MlMjBQcmV2aW91cyUzQyUyRiUyRmElM0UlM0MlMkZidXR0b24lM0UlMEElMEElM0NidXR0b24lMjBvbmNsaWNrJTNEJTIyY3VzdG9tc2Nyb2xsJTI4JTI5JTIyJTNFJTNDYSUyMGhyZWYlM0QlMjIlMjNwaGFzZS0yJTIyJTIwY2xhc3MlM0QlMjJidXN0b20tbGluayUyMiUzRW5leHQlMjAlM0UlM0MlMkZhJTNFJTNDJTJGYnV0dG9uJTNFJTBBJTBBJTNDJTJGZGl2JTNF

Phase 2: Currency

The phase 2 audit was delivered to the Eco team on April 12th, 2019.

The audited commit is `c0d8477866a8667a37f45c85378c5938c84569cd`, and the files included in the scope were BeamBalanceStore.sol, ERC223BeamToken.sol, ERC223TokensRecipient.sol, ERC777BeamToken.sol, ERC777Token.sol, ERC777TokensRecipient.sol, and ERC777TokensSender.sol.

Here are our audit assessment and recommendations, in order of importance.

Update: *The Eco team made some fixes based on our recommendations. We address below the fixes introduced up to commit `af3428020545e3f3ae2f3567b94e1fbc5e5bdb4c` of the*

No critical issues were found.

High Severity

[P2-H01] Broken token approval implementation

High

Contracts inheriting from OpenZeppelin's ERC20 implementation, such as `BeamBalanceStore` and `ERC223BeamToken`, reimplement several inherited ERC20 features (sometimes unnecessarily, as reported in **"Unnecessary reimplementation of ERC20 features in ERC223BeamToken"** and **"Unnecessary reimplementation of ERC20 features in BeamBalanceStore"**). This attempts to seamlessly integrate different token interfaces with the generational balance store in `BeamBalanceStore`. Unexpectedly, these contracts override functionalities related to token approvals, such as the inherited `approve` and `allowance` functions. Moreover, both `BeamBalanceStore` and `ERC223BeamToken` declare an internal `allowances` mapping (see `BeamBalanceStore` at line 94 and `ERC223BeamToken` at line 22) to independently track token allowances. Both `approve` and `allowance` were customized to work with this new `allowances` mapping.

Besides those already mentioned, two other relevant functions are inherited from the `ERC20` contract: `increaseAllowance` and `decreaseAllowance`, which are intended to increase and decrease allowances in a way that mitigates the known front-running vulnerability in ERC20 contracts. While both functions are inherited and therefore publicly available in the contracts' APIs, neither of these two functions are overridden in `BeamBalanceStore` nor `ERC223BeamToken` to work with the previously mentioned `allowances` mapping (as `approve` and `allowance` do). Instead, `increaseAllowance` and `decreaseAllowance` work with the private `_allowed` mapping defined in the parent `ERC20` contract. This leads to a severe mismatch in how allowances are tracked in the contracts, breaking the token approval feature. As regular `Approval` events are emitted by these calls, they will seem successful in the eyes of the caller, yet the modification in the allowance will not be reflected when querying the `allowance` public getter. If users notice this mismatch and thus

This issue stems from having the entire token approval mechanism reimplemented in both the `BeamBalanceStore` and `ERC223BeamToken` contracts. Consider reassessing the need to redefine such functionality when it is already inherited from a secure, audited, and battle-tested contract such as OpenZeppelin's ERC20. Should the development team still consider it necessary to adapt the inherited functions, all of them must be carefully and thoroughly tested to avoid severe bugs that can break entire contract features.

***Update:** Fixed. The `ERC223BeamToken` contract has been removed. The `EcoBalanceStore` contract no longer inherits from `ERC20`. The functions `approve` and `allowance` were removed. The `allowances` mapping was removed. Eco's statement for this issue:*

> When we upgraded the OpenZeppelin contract versions that we had been building on top of, we didn't notice significant changes in the layout of the OpenZeppelin Contracts package. Contracts that used to be interface definitions only were modified to be full implementations. This broke some functionality, and introduced duplicate implementation of others. Our tests have improved since then, and we're more careful about upgrading dependencies.

[P2-H02] Convoluted implementation of the updateTo function in the BeamBalanceStore contract

High

The function `updateTo` performs two fundamental tasks related to Beam's generational store:

- keeps the balances updated to the latest generation
- clean the balances for older generations

These two tasks are currently performed in two `for` loops (see lines 308 and 319 of `BeamBalanceStore.sol`) that modify token balances

Having two different loops in the function makes it unnecessarily complex, since the functions they perform could be easily encapsulated in helper functions. Moreover, the logic within these loops is tangled enough to render it hard to follow. The lack of inline comments and a detailed specification

“[P2-L06] Weak condition in cleaning mechanism”, and **“[P2-L05] Unclear and undocumented looping algorithm for balance updates”**.

Besides addressing these specific implementation issues, consider refactoring the `updateTo` function by splitting the responsibilities into several helper functions, making sure to include documentation explaining the update and cleaning mechanisms in detail.

Update: *Fixed. The `updateToCurrentGeneration` function has been refactored, cleaned, and commented. However, the function still has two `for` loops that, while short and properly commented, could be extracted to functions to make things cleaner.*

Medium Severity

[P2-M01] Token allowances are tracked in different contracts

Medium

The `BeamBalanceStore` contract implements a unified generational store for the token balances, which is independent of the interface used to interact with it. Despite currently only working with ERC20, ERC223, and ERC777, the Beam team aims to develop a balance store extensible and flexible enough to handle any other token interface in the future.

Token allowances are not tracked in a single contract as balances are. As in some interfaces, such as ERC777, it does not track allowances by design. The Beam team has avoided centrally tracking token allowances in the `BeamBalanceStore` contract and instead, has delegated the task to each interface that needs to work with them. While this course of action might be appropriate, some shortcomings must be addressed to effectively implement it.

First of all, the fact that the ERC20 interface is, for no clear reasons, entirely integrated into the `BeamBalanceStore` contract could confuse users and developers alike. After seeing that the contract’s code implements related functions such as `approve`, `allowance`, and an `allowances` mapping, they might be led to conclude that allowances are indeed centrally tracked in `BeamBalanceStore`. This would be a mistake, as these functions and data structure are only part of the `BeamBalanceStore` contract, because the ERC20 interface is integrated

Second of all, there is no documentation regarding how or why token allowances are tracked in separate contracts. As a consequence, it is difficult to understand what the system's intended behavior is, therefore hindering the assessment of the code's correctness. Simple and user-friendly documentation concerning how all users are expected to handle allowances should be included both in docstrings and in the repository. In it, it is of utter importance to highlight that allowances given through an interface are not going to be registered in other interfaces.

Lastly, even though the allowances mechanism is inherited from OpenZeppelin's secure implementation of the ERC20 standard, it is unnecessarily overridden both in `BeamBalanceStore` and `ERC223BeamToken`, which leads to a severe bug in the implementation reported in **"[P2-H01] Broken token approval implementation"**. Other issues related to these points are reported in **"[P2-M02] Unnecessary reimplementation of ERC20 features in BeamBalanceStore"** and **"[P2-M03] Unnecessary reimplementation of ERC20 features in ERC223BeamToken"**.

Design decisions always come with trade-offs. In this case, given the fact that some token interfaces do not handle allowances, the Eco development team decided to avoid centrally tracking them, and instead, they delegate that task to each particular interface. While this approach is feasible, it is not the one initially expected after reading about the generational store and how balances are centrally tracked. Consider, after fixing all particular issues in the implementation, raising more end-user awareness about how the system is expected to work by providing extensive documentation and guides on how users should interact with the contracts, in particular, in terms of token allowances.

***Update:** Fixed. The `ERC223BeamToken` contract has been removed. The `EcoBalanceStore` contract no longer inherits from `ERC20` and no longer tracks allowances.*

[P2-M02] Unnecessary reimplementation of the ERC20 features in BeamBalanceStore

Medium

The BeamBalanceStore contract inherits from OpenZeppelin's ERC20 implementation contract, unnecessarily redefining the functions approve and allowance. Considering that OpenZeppelin's contracts are already audited and considered secure implementations of the ERC20 standard, it is

Furthermore, the implementation of all ERC20 features within the `BeamBalanceStore` contract is not consistent with how other token interfaces (e.g., `ERC223BeamToken` and `ERC777BeamToken`) are treated, the latter being split into external contracts. Even though the documentation states, *"[The ERC20 interface] is implemented directly in the balance store for simplicity,"* we noted during the audit that the loss in consistency and modularization outweighs any potential gain in simplicity.

Hence, it is advisable to refactor the entire implementation of the ERC20 interface out of the `BeamBalanceStore` contract and implement it separately in a new `ERC20BeamToken` contract that interacts with `BeamBalanceStore`, similar to how `ERC223BeamToken` and `ERC777BeamToken` currently do. Should the development team not follow this last suggestion, it is advisable to at least remove the unnecessarily re-implemented ERC20 features (i.e., `approve` and `allowance` functions) to avoid code repetition and ease the project's maintainability.

Update: *Fixed. The `EcoBalanceStore` contract no longer inherits from `ERC20`. A new `ERC20EcoToken` contract was added to integrate the ERC20 standard with the `EcoBalanceStore`.*

[P2-M03] Unnecessary reimplementation of the ERC20 features in ERC223BeamToken

Medium

The `ERC223BeamToken` contract inherits from OpenZeppelin's ERC20 implementation contract, unnecessarily redefining the functions `approve` and `allowance`. If these features are not to be modified by the `ERC223BeamToken`, consider removing them from the contract to avoid code repetition, thus easing the project's maintainability.

Update: *Fixed. The `ERC223BeamToken` contract has been removed.*

[P2-M04] Off-by-one error in the generation cleaning mechanism

Medium

cleaning mechanism for state generations. According to the docs, "Old checkpoints (more than 12 generations ago) for the account are cleaned up during the new checkpoint creation to reduce storage requirements."

However, this statement does not hold true, as the current implementation is keeping an additional generation (i.e., 13) in the state. A unit test confirming the issue can be found in this secret gist.

The lack of additional context in the documentation regarding the purpose of the generational store prevents us from exactly determining the impact and severity of this issue. Consider then properly analyzing its risk and taking the necessary actions to fix the off-by-one error in the `updateTo` function. More extensive unit tests to increase this function's coverage are also suggested, which will avoid regression bugs in future changes to the codebase.

Update: *Fixed. The pruning loop now keeps only the number of generations declared in the* `GENERATIONS_TO_KEEP` *constant.*

[P2-M05] Lack of event emission in the updateTo function

Medium

The function `updateTo` in the `BeamBalanceStore` contract can be called by any user to update a given address to a specified generation. Under certain circumstances, the function also takes care of cleaning the address's balances in old generations that are no longer needed by the system. However, it never notifies users about old balances of their accounts being set to zero, which may be totally unexpected if an unaware user queries the balance at a cleaned generation and sees a balance set to zero where it should be positive (see **"[P2-M06] balanceAt function allows querying balances of cleaned generations"**). The lack of a notifying event further hinders the user's experience in tracking and understanding how and when the balance got changed to zero.

To improve off-chain client experience when interacting with the `BeamBalanceStore` contract, consider defining and emitting informative events every time:

- An address is updated to a more recent generation.

Update: *Partially fixed. The* `AccountBalanceGenerationUpdate` *event is now emitted when an address is updated to the current generation. No event is emitted when balances from past generations are set to 0. Eco's statement for this issue:*

> Emitting events increases gas costs, and the number of generations preserved is a constant relative to the current generation of each address. It's pretty easy to check which generations have data for a given address, both on-chain and off-chain. For analyzing when a generation was erased, the `AccountBalanceGenerationUpdate` event can be used as an indicator.

[P2-M06] balanceAt function allows for querying of balances of cleaned generations

Medium

The `balanceAt` function in the `BeamBalanceStore` contract allows any user to query the balance of a given account at a specified generation. However, there is no restriction regarding up to which past generation the balances can be queried. For instance, users can currently query their balance at generation 0, which is not even a valid generation. Moreover, after a generation has been cleaned, users not fully aware of the internal garbage collection mechanism of old balances might not expect their balance to be zero at a generation where they used to have a positive token balance. The user experience is further hindered considering that no notifying events are emitted when the balances are set to zero during the garbage collection (as seen in **"Lack of event emission in updateTo function"**).

As balances of old cleaned generations are no longer relevant for the system (they are set to zero), consider restricting up to which past generation a user can query the `balances` mapping. In this regard, it is advisable to include a `require` statement to revert the call to `balanceAt` function if the specified generation is 0 or has already been cleaned, which would be a more reasonable and user-friendly behavior for this function.

Update: *Partially fixed. The* `balanceAt` *function still allows querying balances of cleaned generations, but the docstrings have been updated to clearly state the function's behavior. Eco's statement for this issue:*

> staking, and doing an old generation fails in a convenient way — it reads the balance at 0 tokens, so participation in voting or staking is not possible. This has the same effect on the user's actions that a revert would, but does not cause iterative processes to break. Additionally, the implementation of balanceAt is simplified to avoid complex bounds checking that could be a significant source of bugs.

[P2-M07] No indexed parameters in the BeamBalanceStore events

Medium

None of the parameters of the events defined in the `BeamBalanceStore` contract are indexed. Consider indexing event parameters where appropriate to avoid hindering off-chain clients' ability to search and filter for specific events.

Update: *Fixed. All the events in the `EcoBalanceStore` contract have indexed parameters.*

[P2-M08] Implementation of EIP 777 does not fully match the specification

Medium

The following mismatches between the EIP 777 specification and the related implementation contracts (i.e., `ERC777Token` and `ERC777BeamToken`) were identified:

- The `ERC777BeamToken` contract does not register the interface with its own address via the ERC1820 registry deployed at an address specified here, which is strictly required by the EIP. According to the spec, this must be done by calling the `setInterfaceImplementer` function in the registry with the token contract address as both the address and the implementer and the `keccak256` hash of `ERC777Token` as the interface hash.
- The function `defaultOperators() external view returns (address[] memory)` is not defined in `ERC777Token` or in `ERC777BeamToken`, and it is considered a must by the specification. According to the EIP, if the token contract does not have any default operators, this function must still be included and return an empty list.
- Neither `ERC777Token` nor `ERC777BeamToken` define the `burn(uint256, bytes)` and `operatorBurn(address, uint256, bytes, bytes)` functions,

error message (Note, however, that the error message in line 168 of `ERC777BeamToken.sol` does refer to a burn feature, as reported in the issue **"Confusing error message in ERC777BeamToken contract"**.).

- The defined `Minted` event is missing one `bytes` argument. In other words, it is defined as `Minted(address indexed operator, address indexed to, uint256 amount, bytes operatorData);` but instead should be `Minted(address indexed operator, address indexed to, uint256 amount, bytes data, bytes operatorData)`. It is worth mentioning that this event is never actually used in the `ERC777BeamToken` contract.

While these particular noncompliances were detected during the audit, the development team must bear in mind that the EIP 777 is still under discussion and can therefore suffer breaking changes at any moment. Hence, it is highly advisable for the Beam team to closely follow the EIP's progress so as to implement any future changes that the proposed interface may suffer. As another reference, consider following the development of PR #1684 in the OpenZeppelin library.

Update: *Partially fixed. The `ERC777Token` contract was replaced with the `IERC777` interface from OpenZeppelin Contracts. The `ERC777EcoToken` contract is still not registering itself in the ERC1820 registry. The `defaultOperators` function was added, and it is returning an empty list. The functions `burn` and `operatorBurn` were added. The `Minted` event is now properly defined (imported from the OpenZeppelin Contracts `IERC777` interface). Eco's statement for this issue:*

> We register the ERC777 interface with the ERC1820 registry as part of our deployment process rather than in the ERC777 `EcoToken` implementation. See `fusedInit` in the `PolicyInit` contract, and its use in `tools/deploy.js`.

[P2-M09] Receiver contract is not notified in transferFrom function of the ERC223BeamToken contract

Medium

The `ERC223BeamToken` contract reimplements the ERC20 `transferFrom` function, adding the necessary logic to effectively transfer the tokens by calling the `BeamBalanceStore`

(following the EIP-223 specification).

Similar to what is already implemented in the `transfer` function, consider implementing the necessary logic to notify the receiver contract about the token transfer when using the `transferFrom` function. Even though this function is not currently part of the interface proposed in the EIP 223 (so there is no standard expected behavior), failing to notify the receiver contract goes against the EIP's original intention. Alternatively, consider removing the `transferFrom` function altogether to conform to the current state of the EIP.

***Update:** Fixed. The `ERC223BeamToken` contract has been removed.*

[P2-M10] BeamBalanceStore's initialize function lacks the onlyConstruction modifier

Medium

The `initialize` function of the `BeamBalanceStore` contract is missing the `onlyConstruction` modifier. While this issue does not pose a security risk, as the parent contract's `initialize` function does include this modifier (making any transaction to the child's function revert when called more than once), no unit tests covering this sensitive scenario were found in the test suite.

Consider including the `onlyConstruction` modifier in the mentioned function to favor explicitness. Additionally, given how critical this function is, thorough unit tests covering the behavior of the `initialize` function of the `BeamBalanceStore` contract should be included, making sure it is impossible for anyone to call it more than once.

***Update:** Fixed. The `initialize` function now has the `onlyConstruction` modifier. Unit tests covering calls to the `initialize` function of the `EcoBalanceStore` contract have been added.*

Low Severity

[P2-L01] Not following the Check-Effects-Interaction Pattern

Low

the storage to reduce the caller's allowance. Had the call been made to an unknown contract, this pattern would have led to a severe reentrancy vulnerability.

In this particular case, there is no actual risk of a reentrancy attack, as the external call is made to a trusted contract (i.e., `BeamBalanceStore`). Always consider following the Check-Effects-Interactions pattern, thus modifying the contract's state *before* making any external call to other contracts.

Update: *Fixed. The* `ERC223BeamToken` *contract has been removed.*

[P2-L02] Erroneous emission of the ERC20 Transfer event

Low

Authorized token contracts, such as `ERC777BeamToken` and `ERC223BeamToken`, are allowed to transfer tokens registered in the `BeamBalanceStore` contract by calling its `tokenTransfer` function, which in turn, calls the internal `tokenTransferHelper` function. After transferring the tokens, this last function emits two `Transfer` events. While the first `Transfer` event is defined in the `BeamBalanceStore` contract, the second one is the inherited ERC20 `Transfer` event. This means that for every single transfer triggered by any of the different interfaces allowed to interact with `BeamBalanceStore`, an ERC20 `Transfer` event will be emitted, but it will be erroneous if the original call was not made to an ERC20 function.

Docstrings for the `tokenTransfer` function further hinder discerning the intended behavior, as they state, *"A Transfer event must be emitted for any transfer made […]."* However, it is never specified which `Transfer` event in particular is expected to be emitted. Additionally, related unit tests only check for the emission of a single `Transfer` event.

As this unexpected behavior might be misleading for external off-chain clients tracking token transactions, consider only emitting the ERC20 `Transfer` event in the ERC20 `transfer` and `transferFrom` functions.

Update: *Fixed. The* `tokenTransfer` *function no longer emits* `Transfer` *events.*

Low

In the `BeamBalanceStore` contract, there are two public functions to return an address's token balance: `balance` and `balanceOf`. Given `balanceOf` internally calls `balance`, consider restricting `balance`'s visibility to `internal`, leaving `balanceOf` as the only public getter to retrieve the balance of an address. Should this suggestion be applied, functions `balanceOf` in `ERC223BeamToken` and `ERC777BeamToken` need to be updated accordingly, along with all related unit tests.

Alternatively, if the ERC20 interface is finally moved out of the `BeamBalanceStore` contract, as is strongly suggested in **"Unnecessary reimplementation of ERC20 features in BeamBalanceStore"**, then only the `balance` function should be kept in `BeamBalanceStore` (as a `public` or `external` function) to be used by the rest of the contracts to query the token balances registered in `BeamBalanceStore`.

Update: *Fixed. The* `balanceOf` *function was removed.*

[P2-L04] Repeated initialization code in the BeamBalanceStore contract

Low

The `BeamBalanceStore` contract is set up during construction with an initial configuration that defines which policies are allowed to interact with sensitive functions of the contract. This is done by pushing the IDs of three different policies to the `policies` array. As `BeamBalanceStore` is an upgradeable contract, it also features an `initialize` function that is to be called after construction, via a proxy, to effectively initialize the proxy's storage by copying the configuration set during the implementation contract's construction. The `initialize` function initializes policies in the exact same way as the `constructor`, repeating the code used to push the three initial policies.

Given how error-prone this approach is, consider refactoring the repeated code into a private function that encapsulates the initialization of all policies required during the initial setup of the `BeamBalanceStore` contract.

[P2-L05] Unclear and undocumented looping algorithm for the balance updates

Low

The function `updateTo` in the `BeamBalanceStore` contract aims at updating the balance of a given address to a specific generation. It also cleans the balances for older generations.

The documentation does not describe how the update mechanism is to be performed, but the goal seems to promote the balances of the last available balance for the `_owner` (referred by the variable `last`) to its future generations (up to `_targetGeneration`). This is done in the loop in lines 308-310 which is not documented, making it difficult to analyze. In particular, the loop ends up assigning the same value (i.e., the last available balance) to all subsequent balances, even if it takes that value from a different variable in each iteration.

Consider documenting how the `updateTo` function should update the balances and include inline comments to clearly describe the intended behavior. If this is indeed replicating the same value throughout the array, consider simplifying the logic to ease readability.

Update: *Fixed. The `updateToCurrentGeneration` function has been refactored, cleaned, and commented.*

[P2-L06] Weak condition in the cleaning mechanism

Low

The function `updateTo` in the `BeamBalanceStore` contract aims at updating the balance of a given address to a specific generation. It also intends to clean the balances for old generations, always keeping up to 12 past generations.

The cleaning process is guarded by an `if` statement that checks if the specified target generation is greater than `generationMinimumKeep` (set to 12 during construction). When the number of generations becomes greater than `generationMinimumKeep`, the `if` condition will always be true, meaning that the logic including the cleaning mechanism will be executed every time.

Update: *Fixed. The* `updateToCurrentGeneration` *function has been refactored and the conditions for cleaning improved.*

[P2-L07] Complex and potentially inefficient authorization mechanism

Low

The `BeamBalanceStore` contract restricts some of its functionality (e.g., the `tokenTransfer` function) to a set of authorized contracts. To do so, it relies on the `authorizedContracts` mapping which is built out of the `policies` array. In order to keep both variables synchronized, the `reAuthorize` function is intended to remove all elements from the mapping and recompute it from scratch by reading the information from the `policies` array. `reAuthorize` must be called each time a policy is authorized (via `authorize`) or revoked (via `revoke`). Each time one policy is added or removed, the whole authorization mapping is then entirely rebuilt.

Besides the fact that this approach could be quite inefficient if `authorize` and `revoke` are frequently called, the purposes of state variable `policies` and `authorizedContracts` seem to be too overlapped, which leads to the need to always keep them in sync. It appears, however, that the functions `authorize` and `revoke` only add and remove one policy at a time. In this setting, it may be possible to update the `authorizedContracts` mapping directly without completely rebuilding it. This could be done by resorting to the `policyFor` function, as in line 460 of the `reAuthorize` function. Given a policy `p`, the function `policyFor(p)` can be used to obtain the contract's address `c` and update `authorizedContracts[c]` with `true` or `false` accordingly. Using this approach, the array `authorizedContractsKeys` would become pointless.

Consider either making the authorization mechanism more efficient or documenting the rationale behind the current implementation (e.g., policies are rarely authorized or revoked once the contract is initialized).

Update: *Fixed. The authorization mechanism has been simplified and now only relies on the* `authorizedContracts` *and* `authorizedContractAddresses` *arrays.*

Low

The `revoke` function of the `BeamBalanceStore` implements all the necessary business logic to search for a given policy identifier as well as remove it from the `policies` array when found. Considering the entire operation is currently done in convoluted nested loops, without inline documentation explaining what each code segment is actually doing, the function may benefit from some refactoring and modularization into more decoupled private functions. For instance, functions for searching and deleting a policy could be implemented separately and called from `revoke`. This should improve the code's readability, thereby easing its documentation, testability, and maintenance.

Update: *Fixed. The `revoke` function has been refactored and is now more readable.*

[L2-09] The revoke function may not inform the user of a failed operation

Low

The `revoke` function of the `BeamBalanceStore` contract is in charge of finding and removing a given policy from the `policies` array. However, the function does not implement the necessary logic to handle a scenario where no policy is revoked. In such cases, the transaction will successfully be completed without explicitly notifying the sender that the policy identifier provided is not registered and was therefore not removed.

Consider adding the needed `require` statement to revert the transaction and explicitly notify the sender with an informative error message when it is impossible to revoke a given policy identifier.

Update: *Fixed. Now, when the policy is not revoked, the function reverts.*

[P2-L10] Redundant deletion of the array's element in the revoke function

Low

The `revoke` function of the `BeamBalanceStore` contract is in charge of finding and removing a given policy from the `policies` array. When the policy is found, a loop takes care of moving the tail of the array one step forward, thus successfully removing the target policy by

These last two instructions are redundant, since just decreasing the array's length will suffice. Therefore, consider removing the instruction that zeroes the last element in the array.

Update: *Fixed. The* `revoke` *function now only decreases the array's length.*

[P2-L11] A potentially inefficient mechanism for revoking policies

Low

The function `revoke` of the `BeamBalanceStore` contract removes all occurrences of policies matching a given `_policyIdentifier`. In order to do so, the function features two loops. The first compares the provided `_policyIdentifier` with the elements in the `policies` array. The second one, nested inside the first, overrides the matching element by moving all the following elements one position, thereby removing the target policy.

In a scenario where a `_policyIdentifier` appears several times in the `policies` array, the implemented algorithm for removing policies can be quite expensive in terms of gas costs, essentially due to the described nested loops. The worst scenario is in the order of `policy.length * R` where R is the number of times the element to be revoked appears.

If appropriate, consider either disallowing duplicate policies when registering them via the `authorize` function or providing a more efficient implementation for the `revoke` function. As a mere illustration of a plausible approach, consider this draft as an alternative version of `revoke`, which is linear on `policy.length`, even in the case of repeated policies (Note that this suggestion is only a proof of concept and should *not* be used in production under any circumstances).

Update: *Fixed. Now, the* `authorize` *function reverts if the policy has been authorized. The* `revoke` *function is now simpler, ending the loop after the first policy is found.*

[P2-L12] Potentially unnecessary extra iterations in the revoke function

Low

removed. Internally, the function loops through all registered policies in the policies array, reorganizing the elements left in the array after finding and removing the target policy. However, it currently does not break from the loop once the policy is found and deleted; instead, the function unnecessarily keeps iterating over the array.

As each iteration increases the gas cost of the transaction, consider adding a `break` statement right after the `revoked` local variable is set to `true`. This issue should be disregarded if the `policies` array can contain repeated policies. Consider documenting it in the function's docstrings if this was indeed the case.

Update: *Fixed. Now the `authorize` function reverts if the policy has been authorized. The `revoke` function is now simpler, ending the loop after the first policy is found.*

[P2-L13] The total token supply is not tracked across generations

Low

The generational balance store implemented in the `BeamBalanceStore` contract does not track the total token supply across generations. Instead, the token supply registered in the `tokenSupply` state variable always refers to the latest generation. If this is the intended behavior of the balance store, consider properly documenting it. Otherwise, consider implementing the necessary logic to track the total token supply across all generations.

Update: *Fixed. The behavior has been clearly documented.*

[P2-L14] The EIP 820 implementation should be updated to EIP 1820

Low

Considering that EIP 1820 has recently superseded EIP 820, the former being the latter's adaptation for Solidity 0.5, it is advisable to upgrade all references to EIP 820 in ERC777BeamToken and the README with the newer EIP 1820. Although entirely out of scope for this audit, it was observed that this work has already begun in https://github.com/BeamNetwork/currency/pull/39.

contract

Low

Docstrings for the `granularity` function of the `ERC777BeamToken` contract are phrased as a question, which may be confusing for users and developers alike. Consider rephrasing them as a clear and straightforward statement, such as `Return the smallest unit of this token that can be transferred`.

Update: *Fixed. The docstring of the `granularity` function is now clearer.*

[P2-L16] Confusing error message in the ERC777BeamToken contract

Low

In the `ERC777BeamToken` contract, the error message in line 168 should be clarified to better inform users about the specific failed condition that causes the transaction to revert (e.g., *"Recipient cannot be the zero address"*). Moreover, it points the user to use a burn feature not implemented in the `ERC777BeamToken` contract, which may be confusing.

Update: *Fixed. The revert message has been updated.*

[P2-L17] Incomplete API documentation

Low

The README.md file includes a section where the public API of the `BeamBalanceStore` contract is documented. However, public functions `tokenTransfer`, `balance`, `initialize`, `destruct`, and `reAuthorize` are not included. As these functions are part of the public API of the `BeamBalanceStore` contract, consider adding them to the documentation.

Update: *Fixed. The API documentation in the README file now includes the mentioned functions.*

[P2-L18] Outdated solc version in use

make sure that the latest version of the compiler is being used at the time of deployment (presently 0.5.7).

Update: *Fixed. Eco now uses Solidity 0.5.12.*

Notes

- The current implementation found in the `BeamBalanceStore` contract does not call the `reAuthorize` function to populate the `authorizedContracts` mapping after initial policies are set in the constructor and `initialize` functions. The Eco team confirmed this is the functions' intended behavior, as the `reAuthorize` function is called from off-chain scripts that handle the initial deployment. Nonetheless, it is advisable to include inline documentation in the functions that clearly states this, which will avoid confusion in developers and auditors.
  Update: *Fixed. Notes have been added to the docstrings of the `constructor` and `initialize` functions.*
- In line 57 of `BeamBalanceStore.sol`, there is a confusing reference to *"authorizedPolicies"* which likely refers to the `policies` array instead. Consider fixing it or clarifying its meaning.
  Update: *Fixed. The `authorizedContracts` mapping was removed.*
- As it may be difficult for regular users to understand the `balanceAt` function's intended behavior from its docstrings, they may greatly benefit from simpler and more straightforward phrasing.
  Update: *Fixed. The docstring of the `balanceAt` function is now clearer.*
- There is an unclear error message in line 270 of `BeamBalanceStore.sol` that reads, *"Source account must have at least enough tokens."* Consider rephrasing and clarifying it.
  Update: *Fixed. The error message is now clearer.*
- In line 329 of `BeamBalanceStore.sol`, consider calling the `isUpdated` function to avoid code repetition and improve readability.
  Update: *Fixed.*
- The function `isAuthorized` determines whether a contract address is authorized to operate with the `BeamBalanceStore` by looking at the `authorizedContracts` mapping. There is also a modifier `restricted` that checks for authorization by looking

- In the `BeamBalanceStore` contract, consider clearly documenting what the exact intended relationship between `authorizedContracts` and `policies` is, as both state variables are deeply related (e.g., for each `p` in `policies`, there is a contract `c` such that `authorizedContracts[c]` and `c == policies[policyFor(p)]`).
  Update: *Fixed. The `policies` variable was removed.*
- State variables `generationForAddress`, `cleanedForAddress`, and `balances` in the `BeamBalanceStore` contract are highly related. Therefore, it is of utter importance for the contract to keep consistency among its values. It is thus advisable to clearly describe and document some of the most relevant constraints governing these variables. For instance, `balances[_owner]` should be `0` for the generations ranging from `0` (or `1`) to `cleanedForAddress[_owner]`, `cleanedForAddress[_owner]` always be less than or equal to `generationForAddress[_owner]`, and the latter always less than or equal to `generation`.
  Update: *Fixed. Constraints are now clearly documented in docstrings (see lines 87 to 102 and 105 to 114).*
- There is a misleading inline comment describing how the value set in the state variable `generationDuration` is calculated. While it reads `1/12th of 365.25 days`, the actual value set is `1/12 * 365.25 * 24 * 3600`. Consider rephrasing this comment to avoid mismatches between documentation and implementation.
  Update: *Fixed. The comment was removed and now the magic value has a comment explaining where it comes from.*
- In the `BeamBalanceStore` contract, consider documenting the time units in which the duration of a generation is measured with an inline comment next to the `generationDuration` state variable .
  Update: *Fixed.*
- In the `BeamBalanceStore` contract, seeing as how the state variable `generationDuration` is only set during construction and is not intended to be modified, consider declaring it as `constant` to prevent functions from inadvertently modifying it in future changes to the codebase.
  Update: Fixed.
- The `BeamBalanceStore` contract currently has two different functions to update an account to a more recent generation: `update(address)` and

update an address even when updating it to the most recent generation, consider removing the `updateTo` function from the contract's public API, thereby restricting its visibility to `private`.

Update: *Fixed.*

- The function `incrementGeneration` in the `BeamBalanceStore` contract increments by one the value of the `generation` state variable and updates the `generationTimestamp` accordingly by adding the generation duration. Additionally, the function can only be called if the `generation` has finished (i.e., the current time is greater than the sum of the last generation timestamp and the generation duration). If the system is not updated for some time, several calls to the `incrementGeneration` function will be required until the timestamp for the generation is in sync with the block time. Should this be the expected behavior, consider documenting it.

  Update: *A comment was added.*
- In the `BeamBalanceStore` contract, consider restricting the `tokenSupply` public state variable to `private` and using the already defined `totalSupply` getter, so as to avoid having two public functions behaving in exactly the same way.

  Update: *Fixed. The `totalSupply` function was removed.*
- To favor explicitness and readability, the following renaming is suggested.
  - In `BeamBalanceStore`:
    - the `updateTo` function to `updateToGeneration`
    - the `update` function to `updateToCurrentGeneration`
    - the `generation` state variable to `currentGeneration`
    - the `generationTimestamp` state variable could be renamed to `currentGenerationTimestamp`
    - `generationMinimumKeep` to `generationsToKeep`

      Update: *Some of the variables and functions were renamed.*
- The function `isContract` appears both in `ERC223BeamToken` and `ERC777BeamToken` contracts. Both contracts import libraries from OpenZeppelin. Note that OpenZeppelin already provides the function `isContract` with the desired functionality. Therefore, consider using the function `isContract` from OpenZeppelin to avoid duplicates, making the code easier to read and maintain.

  Update: *Partially fixed. The `ERC223BeamToken` contract was removed. However, the `ERC777EcoToken` contract still has the `isContract` function.*

removing all unnecessary imports to favor readability and avoid confusions.

Update: *Partially fixed. The* `SafeMath` *import was removed, but the* `ERC1820ImplementerInterface` *is still imported and unused.*

- Several docstrings do not follow the Ethereum Natural Specification Format (e.g., all docstrings of the functions in the ERC777BeamToken contract). Consider following this specification across the entire codebase on everything that is part of the contracts' public APIs.

  Update: *Not fixed.*

- Consider prepending all `private` and `internal` functions with an underscore (see `tokenTransferHelper`, `doSend`, `callSender`, `callRecipient`, `isContract`, and `getStore`), so as to explicitly denote their restricted visibility and favor readability.

  Update: *Not fixed. In Eco's words: "We develop against our own internal style guide, which does not prefix private or internal functions with underscores."*

- To keep a consistent code style across the entire codebase, consider removing the underscore from the event parameter `_contract` in events `Authorized` and `Revoked` of the `BeamBalanceStore` contract.

  Update: *Fixed. The parameters in* `Authorized` *and* `Revoked` *no longer have the underscore.*

- To favor explicitness, consider declaring the visibility of all contract state variables (see `policies`, `authorizedContracts`, `authorizedContractKeys`, `balances`, `generationForAddress`, `cleanedForAddress`, and `allowances`).

  Update: *Fixed. All the state variables in the* `EcoBalanceStore` *contract now have the visibility declared.*

- The contract `ERC777Token` in `ERC777Token.sol` should be refactored into an interface, while changing the contract's name to `IERC777Token` to better denote it is simply an interface and not the actual implementation. Similarly, the interfaces defined in `ERC777TokensRecipient.sol`, `ERC777TokensSender.sol`, and `ERC233TokenRecipient.sol` should be renamed to `IERC777TokensRecipient`, `IERC777TokensSender`, and `IERC223TokensRecipient`, respectively, making sure these suggested changes are also reflected in the files' names.

arguments is line breaking after the opening parenthesis but not breaking before the closing one. Although this style was consistently seen throughout the entire codebase, it embodies a minor deviation from the Solidity style guide. Consider following Solidity's coding style guide, which can be enforced across the entire project by means of a linter tool, such as Solhint.

Update: *Fixed. Now the code style breaks after the closing parenthesis.*

- Diagrams included in the `README.md` file do not exactly match the contracts' code. In particular, while diagrams state that all contracts include a `kill` public function, this function was not found in any of them. Consider fixing this documentation mismatch, renaming the `kill` function to `destruct` (which is the implemented function that the documentation presumably refers to).

  Update: *Fixed*.

- Consider fixing the grammar in the "Usage" section of the README file, replacing `interfaces are providing conforming to ERC223 and ERC777` with `interfaces conforming to ERC223 and ERC777 are provided`.

  Update: *Fixed. Grammar in the currency README has been corrected.*

JTNDZGI2JTIwY2xhc3MlM0QlMjJidG4tY29udGFpbmVyJTIyJTNFJTBBJTBBJTNDYnV0dG9uJTIwb25jbGljayUzRCUyMmN1c3RvbXNjcm9sbCUyOCUyOSUyMiUzRSUzQ2ElMjBocmVmJTNEJTIyJTIzcGhhc2UtMSUyMiUyMGNsYXNzJTNEJTIyY3VzdG9tLWxpbmslMjIlM0UlM0MlMjBQcmV2aW91cyUzQyUyRmElM0UlM0MlMkZidXR0b24lM0UlMEElMEElM0NidXR0b24lMjBvbmNsaWNrJTNEJTIyY3VzdG9tc2Nyb2xsJTI4JTI5JTIyJTNFJTNDYSUyMGhyZWYlM0QlMjIlMjNwaGFzZS0zJTIyJTIwY2xhc3MlM0QlMjJjdXN0b20tbGluayUyMiUzRW5leHQlM0MlMkZhJTNFJTNDJTJGYnV0dG9uJTNFJTNDJTJGZGI2JTNF

Phase 3: Governance

The phase 3 audit was delivered to the Eco team on June 14th, 2019.

The audited commit is `51b65c5c7af455391af2bc30d13d761566d8c236`, and the files included in the scope were CurrencyGovernance.sol, DepositCertificates.sol, Inflation.sol, PolicyProposals.sol, PolicyVotes.sol, Proposal.sol, SimplePolicySetter.sol, TimedPolicies.sol, and TrustedNodes.sol.

fixes introduced up to commit `af3428020545e3f3ae2f3567b94e1fbc5e5bdb4c` *of the currency repository.*

Critical Severity

[P3-C01] Funds can be lost

Critical

Any token holder can call the `registerProposal` function of the `PolicyProposals` contract to register a proposal address. There is a token cost per registration, which can be partially refunded with the `refund` function if the proposal does not make the ballot.

However, the guard conditions intended to avoid duplicate proposal registrations or prevent refund requests for ineligible proposals do not apply to the zero address. This has a number of consequences.

Each time the zero address is registered as a proposal, it will overwrite the proposer address, add a new item to the `allProposals` array, increment `totalproposals`, and emit a new `ProposalAdded` event. This will cause `allProposals` and `totalproposals` to return incorrect values.

Interestingly, this would also prevent `totalproposals` from being decremented to zero, which would prevent the contract from destructing, as long as the `refund` function did not have the same bug.

If the zero address proposal is registered, it can be refunded like any other proposal that did not make the ballot. Subsequently (or if it was never registered), calls to `refund` the zero address will decrement the `totalproposals` variable and send `REFUND_IF_LOST` tokens to the zero address.

This will cause the `totalproposals` variable to return the wrong value when queried. Additionally, each call to `refund` reduces the contract balance, and it may not have enough funds to honor all refunds, causing some legitimate refund requests to fail. Alternatively, if `totalproposals` is decremented to zero, anyone can `destruct` the contract before

Consider preventing `registerProposal` and `refund` from being called with the zero address.

Update: *Fixed. The `registerProposal` and the `refund` functions now require that the address passed as an argument is not 0.*

[P3-C02] Any token holder can prevent currency governance

Critical

The `CurrencyGovernance` contract oversees a governance process that may culminate in the deployment of an `Inflation` contract or a `DepositCertificates` contract.

In the first case, after inflation tokens are minted and assigned to the `Inflation` contract, `startInflation` confirms the contract token balance exactly matches the expected value. Similarly, in the second case, after interest tokens are minted and assigned to the `DepositCertificates` contract, `startSale` confirms the contract token balance exactly matches the expected value.

However, either contract would be deployed to a predictable address. This makes it possible for any token holder to pre-compute the contract address and transfer tokens to the contract before it is deployed. Consequently, when it is initialized after deployment, it will have an unexpectedly high balance, and the transaction will be reverted. This effectively means that any token holder can prevent the `CurrencyGovernance` contract from deploying new `Inflation` or `DepositCertificates` contracts.

Consider changing the guard condition to set a minimum (but no maximum) bound on the contract balance.

Update: *Fixed. The `startInflation` and `startSale` functions now check for a balance greater than or equal to the required.*

[P3-C03] Inflation parameters are miscalculated

Critical

inflation votes. It does this by requiring the user to sort the array of voter addresses by their inflation votes (implicitly creating a sorted array of inflation votes) and separately by their prize votes (implicitly creating a sorted array of prize votes) and then, by selecting the middle of the non-zero elements in the implicit array.

If there is an even number of non-zero elements, it is supposed to average the middle two. However, the middle index is miscalculated. Specifically, it is set to half the number of non-zero elements, but it is not offset into the non-zero elements section of the array.

This means that inflation and prize will be the average of two elements that are too low (and possibly zero) and will not reflect the results of the vote. Consider moving the offset operation from `CurrencyGovernance` (line 360) out of the `if` statement (line 358) to correctly calculate the middle index.

Update: *Fixed. The* `initiateInflationCycle` *function now correctly offsets the middle index.*

[P3-C04] Miscalculation of `claimed interest`

Critical

The `DepositCertificates` contract maintains a public `claimedInterest` variable that tracks the total amount of interest that has been issued by the contract.

Whenever a user invokes the `deposit` function to purchase a deposit certificate, only the newly issued interest should be added to `claimedInterest`. Instead, the total interest issued to the user by the contract is added, which effectively counts previously issued interest multiple times. This means that whenever a user buys certificates over multiple transactions, the public `claimedInterest` value will be incorrect.

In addition, `claimedInterest` is used in the `destruct` function to determine whether all issued and locked funds have been withdrawn. Since it will be too high if the bug is triggered, the contract will be unable to burn its tokens and will self destruct.

Update: *Fixed. The* `deposit` *function now adds only the unclaimed interest.*

High Severity

[P3-H01] Incorrect currency vote result

High

The `computeVote` function in the `CurrencyGovernance` contract contains a loop to calculate the number of zero inflation votes and zero certificate votes. These values are later compared with the total number of revealed votes in order to calculate the number of non-zero inflation and certificate votes.

It does this by requiring the user to sort the array of voter addresses by their inflation votes (implicitly creating a sorted array of inflation votes) and by their certificate votes (implicitly creating a sorted array of certificate votes). It then cycles through both implicit lists in order, updating the number of seen zero votes in each iteration, stopping at the second to last element. This means that if either implicit list contains only zero votes, the count will be one less than it should be.

In most cases, this has no effect, because inflation votes are anti-correlated with deflation votes. Additionally, when there are exclusively zero votes on one side, only the other side is used in subsequent calculations. However, relying on these facts about the current code base makes it fragile to changes.

More importantly, it is possible for a voter to select zero for both inflation and certificates. Consider a vote with exactly one non-zero ballot for one of the parameters and at least one zero vote for both parameters (and no other votes). This should resolve in favor of the non-zero ballot; in fact, it resolves as a tie.

Consider updating the algorithm to correctly count the number of zero votes in this scenario.

Update: *Fixed. A patch was added to cover the cases when the last value of the array is 0. Note, however, that the function is still big and hard to read.*

[P3-H02] Executed policy proposals have full control over the system

proposals can take place. The voting has two staking phases. At the end of the voting, 3 proposals can be enacted if they have more stake than the no-confidence vote and more than 3/4 of the total staked. When a proposal is enacted, it will immediately replace one of the policy contracts, and it can only be removed following the same voting procedure.

This means that a new contract can take full control over a part of the system functionality. For example, the new enacted proposal could add or remove trusted nodes for the monetary vote, it could change the rules or frequency of the votes, or it could completely replace the token.

It is expected that the token holders will not put their stake on proposals that will break the system. But humans are known for making bad decisions collectively once in a while. And even if the proposals are extensively discussed, votes carefully thought out, and many people voting for what appears to be a good decision out of the wisdom of the crowd, it is possible for the proposal to have issues that will have unintended consequences that nobody noted until they were enacted.

Registering a proposal for voting requires spending a lot of tokens, but anybody who can get that amount of tokens can register one. In addition, all the token holders will be able to stake in favor of proposals. This is a very interesting and radical approach for transparent and decentralized governance. However, there is a lot of risk in allowing the token holders to decide directly over the execution of all the functions of the system.

Consider adding the role of trusted auditors and adding an extra phase to the policy proposal process to allow the auditors to review the proposed contracts. The auditors would not have any control over the results of the vote. They would just review that the contracts do exactly what the proposers intend them to do — that there are no bugs and that they will not break the Eco system. After a successful audit, the proposal can be whitelisted, and only whitelisted proposals should be enacted.

Update: *Not fixed. Eco's statement for this issue:*

> In order to reduce the risks inherent in the policy proposals process, we've reduced the number of proposals that can be enacted in a given voting cycle from 3 to 1. This helps ensure that proposals do not conflict with each other – if multiple changes should be made at the same time, they need to be combined into a single proposal covering all of the changes to make in the voting

> We considered the creation of a trusted auditor role as suggested in the report, but came to the conclusion that this sort of arrangement is better handled off-chain. By enforcing the auditor role on-chain, we would introduce the challenges of managing which auditors are trusted using the governance process itself. We would have to rely on the auditor to approve a proposal that replaces the auditor, which is more than a small conflict of interest.
> Instead, we rely on our community to demand an audit, and vote down any proposal that has not been thoroughly audited by someone they trust.

Medium Severity

[P3-M01] The proposal registration fee is staked

Medium

When the `registerProposal` function of the `PolicyProposals` contract is called, a registration fee is transferred to the contract.

The fee is also added to the total stake supporting the proposal. This is confusing in two ways. Firstly, the proposal's `staked` mapping is not updated, thereby creating a situation where there are staked tokens not assigned to anybody in particular. Secondly, if the proposer decides to `support` their proposal, they will stake the tokens from their balance at the contract's `generation`, which may include the tokens they already staked as part of the registration fee.

Whether or not this is considered a bias in the result depends on if the registration fee is intended to be taken from the staked amount. If so, some proposers are double staking the tokens used to pay the registration fee, while others are staking tokens that they did not have at the contract's `generation`. On the other hand, if the registration fee is intended simply as a deterrent against ill willed or spam proposals, it should not be treated as part of the stake.

In the first case, consider confirming the proposer had enough tokens in the contract's `generation` before accepting the fee in the current generation. Also ensure that the `support` function does not add tokens that were part of the registration fee.

In the second case, consider setting the proposal's initial stake to the proposer's balance at the contract's `generation` and updating the proposal's `staked` mapping accordingly.

Update: *Fixed. The `registerProposal` function now leaves the initial stake at zero.*

[P3-M02] The non-refunded tokens are not transferred

Medium

In the `PolicyProposals` contract, registering proposals requires transferring tokens to this contract. Then, up to 3 proposals are selected to go to a second staking phase. The proposers of the unselected ones can get a partial refund.

It is not clear what should happen to the tokens that cannot be refunded (i.e., the ones corresponding to the registration of the selected proposals and what remains after the refunds are processed). They are just held by the `PolicyProposals` contract, until it is destroyed, and then, they are locked forever.

Consider transferring these tokens after the top proposals are selected. If they should be locked forever, consider explicitly signalling this by transferring them to the 0 address.

Update: *Fixed. The `destruct` function now transfers the tokens to the `policy` contract. Note, however, that this behavior is not documented. It is not clear what happens to the tokens owned by the policy: Are they now owned by the Eco team? How can the `policy` contract transfer or use those tokens? Because these details might be unexpected or sensitive for some users, consider documenting them as explicitly as possible. Also, note that the transfer function could fail, and the contract would still be destroyed.*

[P3-M03] Lack of input validation

Medium

In the `SimplePolicySetter` contract, the `set` function is used to set the ERC20 key and address value that will update the system to use a new implementation contract.

This function does not verify that the key and address arguments passed are not 0, which in almost every case will be wrong values. Also, the docstring of the function says that it can only be called once, but without validation it can be called with arguments set to zero multiple times.

However, the `_value` argument can still be 0. Eco's statement for this issue:

> Allowing 0 as a `_value` argument here is necessary to allow us to unset the value. We use this throughout our code to grant temporary privileges. User-provided values should always be sanitized before passing them to a `SimplePolicySetter` call to prevent significant security issues.

[P3-M04] The function that computes votes is too long

Medium

The `computeVote` function in the `CurrencyGovernance` contract is almost 200 lines long.

There are too many statements in this function covering many different tasks, making it very hard to read and very prone to errors if it is changed in the future. It is the source of multiple issues found in this audit.

Consider splitting this function into many smaller private functions with intent-revealing names. This would increase the cost of calling this already expensive function, but it is important in a function so sensitive as this one.

If the cost increases are considered prohibitive, try to find other ways to reduce the size of the function like separating it into different transactions, or at the very least, document the limitations that make the function so long.

Update: *Not fixed. Eco's statement for this issue:*

> We'll be rewriting this before launch – we plan to move to a new vote tabulation algorithm.

[P3-M05] Failed vote ends with no error

Medium

In the `computeVote` function of the `CurrencyGovernance` contract, there are two cases when a failed vote ends with no error and instead emits a `VoteResults` event:

This is misleading, and it does not follow the principle of failing as early and loudly as possible. It also produces the same behavior as a successful vote that ends in a tie, which may lead to confusion.

Consider emitting a `VoteFailed` event on these cases and ensuring that `computeVote` cannot be called again.

Update: *Not fixed. Eco's statement for this issue:*

> In both of the cases covered the result of the vote is that 0 inflation is printed, 0 tokens are distributed to each prize winner, 0 tokens are accepted for deposit certificates, and 0 interest is paid on deposit certificates. We concluded that this is accurately represented by the `VoteResult(0,0,0,0)` event that the system emits, and the regular vote tally process already ensures that `computeVote` cannot be called a second time.

[P3-M06] Rejected votes can affect currency governance

Medium

The `reveal` function of the `CurrencyGovernance` contract ensures that the voter did not choose a non-zero value for both inflation and the number of certificates to issue.

However, there are other combinations of parameters that can affect the vote unexpectedly. In particular, it is possible to vote for inflation (or neither inflation nor deflation) and still choose a high interest. If the deflation vote succeeds, the interest chosen by the inflation voter is still used when calculating the median, provided it is higher than the lowest value chosen by a deflation voter. In this case, the deflation voter's choice is discarded. Similarly, it is possible to vote for deflation (or neither) and still choose a high prize, which will be considered in the same way if the inflation vote succeeds.

It may seem counterproductive to vote in this way but there are potential game-theoretic reasons (e.g., a pre-commitment to an outrageous result on one side of the vote). Regardless, it is preferable to prevent undesirable results where possible rather than rely on assumptions about how users will behave.

Consider requiring all voters to choose either inflation or deflation and to prevent or disregard non-zero values for `interest` and `prize` when they are inconsistent with the vote.

Update: *Fixed. The `reveal` function now calls the `invalidVote` function, which requires votes to be consistent.*

[P3-M07] Ticket Registration emits the incorrect ticket number

Medium

In the `registerFor` function of the `Inflation` contract, the registrant is assigned a ticket immediately after the contract emits the corresponding event. However, the event uses 0-up indexing, whereas the `holders` mapping, which can be used to look up the ticket associated with an address, uses 1-up indexing.

This means the two mechanisms for determining the ticket associated with an address are inconsistent with each other. Consider using 1-up indexing in the `TicketsRegistered` event.

Update: *Fixed*.

[P3-M08] Duplicable votes

Medium

In the `commit` function of the `PolicyVotes` contract, a token holder can commit to a vote that supports a subset of the proposals on the ballot. The commitment is a `keccak256` hash of the approved proposals with a random salt to provide entropy. Later, the token holder can `reveal` their vote and salt so it can be compared to the commitment and then processed if valid.

However, the commitment is not cryptographically tied to the token holder address. This means it is possible for a token holder to copy and submit another voter's commitment without knowing the contents. When the original voter reveals their vote and salt, the copycat can reveal the same values.

Consider including the voter address within the hash commitment to prevent duplication.

process the vote.

Once again, the encrypted vote is not cryptographically tied to the voter's address, which means anyone can reuse the commitment and corresponding VDF seed without knowing the contents.

Consider including and verifying a hash of the voter address and their chosen parameters within the encrypted vote to prevent duplication. It is important to note that encrypting the voter address directly will not prevent vote duplication—it should be hashed, along with the vote parameters first.

Although simply encrypting the voter address will prevent direct duplication, it does not mitigate targeted modification of the ciphertext. When using a stream cipher for encryption (as the `CurrencyGovernance` contract does), a change in a particular location of the ciphertext will produce a corresponding change in the same location in the plaintext.

Note that we can segment the key stream created by `keyExpand` and the ciphertext using the corresponding plaintext section:

```
plaintext  = p_inflation || p_prize || p_certificates || p_inte
keystream  = k_inflation || k_prize || k_certificates || k_inte
ciphertext = c_inflation || c_prize || c_certificates || c_inte
```

Then,

```
ciphertext  =  plaintext  ^  keystream
```

becomes:

```
c_inflation  =  p_inflation  ^  k_inflation
c_prize  =  p_prize  ^  k_prize
c_certificates  =  p_certificates ^  k_certificates
c_interest  =  p_interest  ^  k_interest
```

```
c_voter = p_voter ^ k_voter
```

There are two important observations:

1. Flipping bits in **c_voter** will produce corresponding bit flips in the decrypted **p_voter**.
2. This will not affect the decryption of any of the other parts of the ciphertext.

Notably, the copycat could construct a ciphertext that would replace the voter address in the corresponding plaintext with their own:

```
c'_voter = c_voter ^ p_voter ^ [copycat address]
```

This will now decrypt to the copycat address:

```
p'_voter = c'_voter ^ k_voter
         = c_voter ^ p_voter ^ [copycat address] ^ k_vo
         = p_voter ^ ( c_voter ^ k_voter ) ^ [copycat add
         = p_voter ^ p_voter ^ [copycat address]
         = 0 ^ [copycat address]
         = [copycat address]
```

Since the rest of the plaintext will be unchanged, this allows the copycat to use another vote as their own.

In fact, using this method, they could make targeted modifications to another vote provided the modifications can be described as a bitwise change. For example, they could toggle the high bit of the prize chosen by a target voter without knowing the original parameters.

This attack is possible whenever the copycat wants to change a predictable part of the underlying plaintext. Including and verifying a hash of all parameters and the voter address ensures that any modification will be detected.

Update: *Not fixed. Eco's statement for this issue:*

Medium

The inflation module of the Eco system is characterized by having clearly defined phases. For example, there is a commit phase, a reveal phase, a challenge phase, and a refund phase. Usually, at the end of the process, there is a destruct phase in which the contract is no longer useful. It is removed from the policy network and destructed.

These phases are critical for the success of the governance of the monetary processes and the policy network. However, the transitions between the phases are not so clear. Most of the functions will start with a set of `require` statements that ensure that they are executed only when the time-based and state-based constraints are satisfied. There are so many constraints of different sorts that it is hard to think about the possible phases, the things that trigger a transition, and which correct phase follows.

Consider modeling the system as a finite-state machine and refactoring the code to make this model clearly visible. The `require` statements could then be grouped into state-checking functions that return `true` if the system is in a specified state. Then, every action could be guarded by a single check to verify that the system is in the correct state and could end up updating the system to the next state.

The code would become more readable, it would be easier to analyze the system as a whole, and it would open the door to interesting new options like formally verifying all the state transitions.

Update: *Partially fixed. The `atStage` modifier was added to make the transitions of the `CurrencyGovernance` contract clearer. The other contracts are still using the unclear stage transitions. Eco's statement for this issue:*

> We agree that the code here is complicated and could be cleaner. We're anticipating changes in this part of the system before we launch, and will take OpenZeppelin's advice into consideration as we adapt the code to those changes.

[P3-M10] Contracts can be destructed

contract has successfully finished all its responsibilities and is no longer needed.

The contracts are cloned just for a one-time usage, so there is no risk of other contracts internal to the Eco system that are depending on them. If they hold a policy role in the network, they remove themselves before calling `selfdestruct`; if they hold ether, it is transferred to the policy contract.

The `selfdestruct` happens in a very controlled way. However, it still is a very extreme action with no way to rollback. There could be a bug in the system that allows a call to `selfdestruct` before it is supposed to happen. This could leave a very important vote without the code that should execute it. And because the functions are not always verifying the sender, other kinds of bugs could let the wrong people destruct a contract. Not all of the contracts transfer the Eco tokens they might hold, so destructing them could lock the tokens transferred by mistake forever. Finally, even if the contracts are no longer useful for the Eco system, they could be useful for historical purposes or for other systems building on top of it.

Consider the more conservative approach of disabling the contracts instead of destructing them. Alternatively, consider allowing only trusted destroyers to call `selfdestruct`. Consider adding functions to recover tokens transferred by mistake.

Update: *Not fixed.*

[P3-M11] Missing incentives to complete the voting process

Medium

In the `CurrencyGovernance` contract, in order to complete the voting process, the trusted nodes have to `commit` votes, and then, somebody has to `reveal` them. Finally, somebody has to call the `computeVote` function. Similarly, in the `PolicyVotes` contract, the stakeholders have to `commit` votes and then `reveal` them, and finally somebody, has to call the `execute` function. These functions require work and money to pay for the gas.

There are no direct incentives for any of the prospective callers. There are going to be external incentives, the Eco team can take care of calling some of these functions, and the intrinsic incentive of everybody who is part of the system to make it work can be very powerful. However, a

Consider adding incentives to the voting protocol. An idea could be to give priority to the voters when disbursing the inflation prize or to purchase deposit certificates. Another could be to set up a gas station network that will be used to pay for the functions that finalize the voting, and that could be financed through the same minting process that ends a vote. At the very least, the external incentives should be described in the contracts, so voters know that they can trust the system to continue and not worry about their money being poorly managed due to neglect.

Update: *Not fixed. Eco's statement for this issue:*

> In the launch version of the system, trustees will be compensated. Compensation will be conditional on revealing a valid vote.

[P3-M12] Events not emitted for important state changes

Medium

In Solidity, events are used to log important state changes on the contracts. Users of the contracts can subscribe and be notified when these events happen. They are also useful to analyze the historic transactions and to find issues or potential problems.

Some of the important state changes of the Eco contracts do not emit events.

Consider emitting events at the end of the following functions:

- In `PolicyProposals.sol`: `support` and `refund`
- In `PolicyVotes.sol`: `reveal`, `commit`, `challenge`, and `execute`
- In `TrustedNodes.sol`: `trust` and `distrust`

Update: *Fixed. The events were added and are now emitted by the corresponding functions:* *`ProposalSupported`*, *`ProposalRefunded`*, *`PolicyVoteRevealed`*, *`PolicyVoteCommitmentRecorded`*, *`PolicyVoteChallenged`*, *`VoteCompleted`*, *`TrustedNodeAdded`*, *and* *`TrustedNodeRemoved`*.

[P3-M13] Unbounded loops

- In `CurrencyGovernance`, the loops in L266 and L313 are bounded by the number of trusted nodes.
- In `Inflation`, the loop in L111 is bounded by the number of `winners` resulting from a vote.
- In `PolicyVotes`, the loops in L125, L138, L160, L210, and L277 are bounded by the number of candidate proposals.

These variables are supposed to be small. For example, according to the Eco team, the initial number of trusted nodes is going to be around 20. In order to add more trusted nodes, a proposal needs to be voted on. The number of winners is selected after a vote in which only trusted nodes can participate. Because they are trusted, this is likely to be a reasonable number that does not break the system. The maximum number of proposals that go into `PolicyVote` is set to 3 by the `PolicyProposals` contract.

However, there is nothing in the contracts of those loops that enforces a maximum value. Actors and proposals can be malicious or make mistakes, and the loops could get too big. Therefore, the system is susceptible to hit the maximum `gas` limit, which would prevent critical functions to execute.

Consider defining and documenting a safe maximum value for the upper bound on every loop, to guarantee the correct functionality of the contracts.

***Update:** Partially fixed. One of the loops in the `CurrencyGovernance` contract and all of the loops in the `PolicyVotes` contract have been removed. Comments about the bounds were added to all the other loops. However, note that the bounds are still not enforced on-chain.*

[P4-M14] Contracts not using safe math

Medium

The contracts `CurrencyGovernance` and `Inflation` do not use `SafeMath` to prevent overflows and underflows in arithmetic operations. They have a comment at the top that says, “Humongous Safety Warning: This does not check for math overflow anywhere.” This comment is more alarmist than informative.

`SafeMath` nor have safety warning comments. `PolicyProposals` contains a decrement that may underflow in some scenarios.

The only valid reason for not using `SafeMath` is that the design of the system makes it impossible for overflows or underflows to occur. But even in that case, the only way to prove it is with extensive formal verifications, and a mistake in any place of the system could be catastrophic.

Consider being extra safe by adding `SafeMath` operations to all the arithmetic operations in all the contracts.

If that becomes significantly more expensive, then consider adding a comment to every statement with an arithmetic operation explaining why it is safe. Ideally, these claims would be accompanied by a formal verification.

Update: *Fixed. All the contracts are now using* `SafeMath` *for all their arithmetic operations that can overflow or underflow.*

Low Severity

[P3-L01] Top proposals are recalculated every time one is supported

Low

In the `PolicyProposals` contract, the `best` array stores the top 3 most supported proposals. Every time somebody supports a proposal, the top 3 are recalculated. This increases the gas costs of supporting a proposal, and in many cases, it could be wasted work.

Consider moving the calculation of the top 3 proposals after the staking phase ends, doing it only once in the `compute` function.

Note: The Eco team is planning to select only one proposal out of this voting phase. In that case, there is no need to sort the proposals every time `support` is called. The most staked proposal can be kept in a simple variable updated in the `support` function, or it can be found at the end with a simple loop in the `compute` function.

[P3-L02] Confusing operation to update the no-confidence stake

Low

In the `PolicyVotes` contract, it is possible to `challenge` all the proposals. When this happens, the token balance of the challenger set at a specific historical time when the contract was `configured` is added to the `noConfidenceStake`.

This function is confusing, because it subtracts the amount previously staked by the sender and adds the new amount. Also, it checks that the new stake is higher than the previous value. However, the amount is always taken from the same historic balance, so the sender can only go from 0 stake to everything staked.

Consider making this function clearer by requiring that the sender has not challenged the proposals yet and that it held tokens at the configured time. In that case, the balance can just be added to the `noConfidenceStake` without the comparison and the subtraction.

Update: *Fixed. The `challenge` function now has the suggested checks and is simpler and clearer.*

[P3-L03] Missing time requirement to challenge

Low

In the `PolicyVotes` contract, there are state variables that define the time intervals for the different phases of the policy decision process. Those variables are `holdEnds`, `voteEnds`, `revealEnds`, and `challengeEnds`.

The comment of the `challengeEnds` variable says that it is used to define when the veto period ends, but it is never used in the `challenge` function. Furthermore, this function does not have any type of time requirement. Based on how the other functions work, it should be possible to `challenge` proposals only during the `CHALLENGE_TIME`.

Consider adding a time requirement in the `challenge` function that allows challenging proposals before the `challengeEnds` timestamp. It could make sense to also require that the

passed. However, note that the state transitions in this contract are still confusing and prone to error, as explained in **"[M3-09] Confusing phase transitions"**.

[P3-L04] Confusing reorder function

Low

Once a vote in the `PolicyVotes` contract has finished and has been executed, the proposals with enough staked tokens are enacted. In this contract, there is also a `reorder` function that can alter the order in which the proposals are enacted.

However, it is not clear why a new order is needed, when the proposals should be reordered, or how the sender decides the new order.

Consider adding comments to explain the purpose of this function in further detail.

Note: The Eco team is planning to stake on only one proposal at a time and to remove the `reorder` function.

Update: *Fixed. Now only one proposal at a time is staked, and the* `reorder` *function was removed.*

[P3-L05] No easy way to get the number of committed or discarded votes

Low

In the `CurrencyGovernance` contract, when a new vote is committed, the number of `totalVoters` is increased by 1. Then, during the reveal phase, when an invalid vote is discovered, this same `totalVoters` variable is decreased by 1.

Reusing this variable makes it harder after the reveal phase to get the number of committed and discarded votes. These values can be calculated by getting the event logs or by looking at the state of the contract in the past, but this is not straightforward. Getting these values quickly could be important for checking the health and security of the system or for other contracts building on top of it.

Update: *Not fixed. Eco's statement for this issue:*

> We will likely be replacing this entire mechanism before launch. Subsequent feedback on the currency governance system has led us to refine certain processes which may represent larger changes, so we want to hold off on the smaller fixes for the time being.

[P3-L06] SafeMath used for uint128 variables

Low

The `SafeMath` contract of OpenZeppelin protects against overflows and underflows but only for operations on `uint256` variables.

In line 28 of the `DepositCertificates` contract, it is declared that `SafeMath` will be used for `uint128` operations, which would be problematic. However, in this contract, no `SafeMath` operations on `uint128` are executed.

Consider removing the statement that declares the use of `SafeMath` for `uint128` variables.

Update: *Fixed. Now `SafeMath` is used for `uint256` variables.*

[P3-L07] Wrong variable type in an event

Low

The `Sale` event of the `DepositCertificates` contract is declared with a `uint256` as the second parameter.

This event is only used in the `deposit` function, which passes a `uint128` variable as the second argument.

Consider changing the parameter of the event to be `uint128`.

Update: *Fixed.*

[P3-L08] Unnecessary require statement

`require` statements that check if a withdraw can be executed.

The second `require` statement checks if the current time is greater than or equal to the end of the sale. The fourth `require` statement checks if the current time is greater than or equal to the end of the sale plus one interest period. Therefore, the second `require` statement is unnecessary.

Consider deleting the unnecessary `require` statement.

Update: *Fixed.*

[P3-L09] Slightly biased inflation lottery

Low

The `Inflation` contract achieves inflation by pseudo-randomly distributing the funds among token holders in proportion to their holdings (at the previous generation in the balance store).

To achieve this, it first allows anyone to register a lottery ticket on behalf of any token holder. Each ticket is assigned a range of unique winning numbers, where the size of the range is equal to the number of tokens held. Subsequently, it divides the inflation funds evenly among a series of lotteries and allocates each jackpot to the holder of the ticket containing the pseudo-randomly chosen winning number.

However, the mapping from pseudo-random number to lottery ticket is slightly biased towards lower winning numbers, corresponding to a slight bias in favor of early registrants. Specifically, if we define the total number of winning numbers as `N`, the lottery chooses a uniformly distributed 256-bit pseudo-random number `R` and reduces it modulo `N`. Whenever `N` does not evenly divide `2**256`, the largest possible 256-bit number will map to some value between `0` and `N`. Numbers less than this value can be reached from one more possible `R` than numbers above this value.

This effect is very small for large values of `2**256 / N` and is unlikely to be important in practice. Nevertheless, it is a provable bias in the lottery. Consider discarding the range of `R`

Update: *Not fixed. Eco's statement for this issue:*

> We agree that this is a problem. Since we know that the inflation lottery is going to change to reflect new thinking on economic governance, we're deferring fixing this until our second audit.

[P3-L10] Important state variables are not public

Low

In the `Inflation` contract, the `generation` state variable is used to get a historical token balance from the address registering for inflation payouts. Similarly, the `generation` state variable in the `PolicyProposals` contract and the analogous one in the `PolicyVotes` contract are used to get a historical token balance from the address supporting a proposal.

These variables are not public, which makes them hard to read for users and contracts building on top of them. They are important to understand the state of the contracts and for the stakeholders to make informed decisions before making their transactions.

Consider giving easier access to the `generation` variables by making them public.

Update: *The `generation` state variables are now public in the `Inflation`, `PolicyProposals`, and `PolicyVotes` contracts.*

[P3-L11] Redundant addresses in structs

Low

In the `PolicyVotes` contract, there is a `Prop` struct and a `proposals` mapping. The key of the mapping is the address of a proposal, and the value of the mapping is the `Prop` struct which stores the same address and the amount staked. Something similar happens in the `PolicyProposals` contract where its own `Props` struct and `proposals` mapping duplicate and store the same address.

This redundancy is needed because mappings "are virtually initialised such that every possible key exists and is mapped to a value whose byte-representation is all zeros". Therefore, it is necessary

boolean named `exists` in the struct instead of the address. This will make the pattern clearer and more readable, while avoiding duplicating the address.

Update: *Partially fixed. The proposals mapping was removed from the `PolicyVotes` contract. The `PolicyProposals` contract still has the `proposals` mapping duplicating the address.*

[P3-L12] Units of variables are not clear

Low

When writing code for Ethereum, there are usually at least two units at play: wei and the basic unit of a token specific to the system. Often, percentages and the count of other items are also used.

In many places of the Eco system it is not clear what the units of the variables and parameters are, such as in `inflation`, `prize`, `certificatesTotal`, and `interest`.

Consider clarifying which unit they are using in the comments of the variables and parameters of the system .

Update: *Fixed. The variables now have comments indicating their units.*

[P3-L13] Inconsistent use of the onlyClone modifier

Low

In the Eco system, most of the contracts that are executed are clones, instantiated for a one-time use only and then destructed. There is an `onlyClone` modifier that prevents a function from being executed on the original contract. This modifier is correctly used on the cloned contracts to prevent the original from self-destructing. However, it is also used on a couple of other functions that will not destruct the original, like configure.

This inconsistent use is confusing. It makes no sense to execute most of the functions on the original contracts, so consider adding the `onlyClone` modifier to all the functions that are intended for clones only.

> every call would increase the gas cost for those calls. Since there's no negative impact on the system resulting from functions being called on original contracts, we prefer to save the gas on clone calls at the expense of not reverting for invalid ones to the original contracts.

[P3-L14] Duplicated statements

Low

In the `withdrawalAvailableFor` function of the `DepositCertificates` contract, there are two different `if` statements that check the same condition—whether or not the certificate term has expired. If so, `_availableBalance` is set to the total interest earned and then later, incremented by the principal, indicating that the user can withdraw all locked funds.

Consider merging the two operations into a single code block under one conditional.

In the `reveal` function of the `PolicyVotes` contract, there is a duplicated loop — first in L125 and then in L138. This wastes gas and makes the code harder to understand.

Consider removing the second loop by moving the statement in L139 to the end of the first loop.

In the `computeVote` function of the `CurrencyGovernance` contract, there are duplicated calculations:

- The number of non-zero inflation votes is independently calculated in L357 and L359.
- The number of non-zero certificate votes is independently calculated in L386 and L388.
- The shift of the `mid` variable to the relevant section of the `_certificatesTotalOrder` array is independently calculated in L389 and L393.

Consider assigning the calculations to a variable in a common code block instead of duplicating the code.

Update: *Partially fixed. The loops were removed from the `reveal` function of the `PolicyVotes` contract. The other duplicated statements are still present. Eco's statement for this issue:*

[P3-L15] Unused variables

Low

In the `CurrencyGovernance` contract, there are two unused variables. Firstly, the `REVEAL_TIME` constant is declared but never used. Secondly, the state variable `randomVDFDifficulty` is instantiated in L435, but other than that, it is never used.

Similarly, in the `PolicyVotes` contract, the `LOCKUP_TIME` constant is declared but never used.

Consider removing all the unused code to reduce the size of the contracts and to make them clearer.

The `winners` variable of the `CurrencyGovernance` contract is a slightly different situation. It is declared as an internal state variable, but it is only used inside the `computeVote` function.

Consider declaring the `winners` variable inside the `computeVote` function.

Update: *Fixed. The `REVEAL_TIME` constant and the `randomVDFDifficulty` state variable were removed from the `CurrencyGovernance` contract. The `LOCKUP_TIME` constant was removed from the `PolicyVotes` contract. The `winners` variable is now declared in the initiateInflationCycle function.*

[P3-L16] Multiple wrong comments

Low

In the `computeVote` function of the `CurrencyGovernance` contract, when the number of revealed valid votes is 0, the comment says, “[…] some votes could not be revealed […].” A more accurate comment would be, “All the committed votes were invalid.” It also says, “Alternately, the next inflation cycle has started, and this contract is invalid,” but that check happens in a higher block in L232.

which is then removed from the end of the array.

The comments in the functions `trust`, `distrust`, and `trustedNodesLength` are wrong, suggesting that the size of the array is never decreased and that new trusted nodes are inserted into empty slots.

In the comment of the `support` function of the `PolicyProposals` contract, it says that if the stake cannot be increased, the function will do nothing. However, the actual behavior is to revert when the call fails to increase the stake.

The comment from the `DepositCertificates` contract states that the contract instance is cloned by the `Inflation` contract when a deflationary action is the winner. However, the `CurrencyGovernance` contract is the one that does that during the `computeVote` function.

The `certificatesTotal` variable in the `DepositCertificates` contract is the maximum deposit amount the contract will accept, whereas the `totalDeposit` variable is the amount currently deposited. The variable comments indicate the opposite.

Consider updating the comments to describe the actual behavior.

Update: *All the comments mentioned above were fixed.*

[P3-L17] Wrong error messages in require statements

Low

In the `computeVote` function of the `CurrencyGovernance` contract, there are multiple `require` statements in L269, L275, L281, and L287 that state "`... but one is not`", when actually, it should say "`... but at least one is not`".

Consider changing the messages to describe the actual behavior.

Update: *Fixed. Error messages have been modified following our suggestion.*

[P3-L18] Use of magic constants

initialized with magic string constants `"CurrencyGovernance"`, `"PolicyProposals"`, and `"PolicyVotes"`.

This makes the code harder to understand and maintain.

Consider defining constant variables with inline comments explaining the relationship between the magic strings and the corresponding constants in the `PolicedUtils` contract.

Update: *Fixed. The string constants were replaced with named constants from the* `PolicedUtils` *contract.*

Notes

- The `configure` function in `PolicyVotes` initializes the contract variables. It is called immediately after being cloned in `PolicyProposals`, as it is supposed to be. However, this is not enforced by the `PolicyVotes` contract and could lead to vulnerabilities if it is not called immediately. For example, if the `execute` function is called before `challengeEnds` and `proposalOrder` are initialized, the contract will self-destruct without performing the vote or executing proposals.
  Consider initializing `holdEnds`, `voteEnds`, `revealEnds`, `challengeEnds`, and `generation` in the `initialize` function. This is possible since they do not depend on the argument to `configure`. For the other variables (`proposalOrder` and `proposals`), consider adding a guard condition to all functions that use them to ensure they were initialized.
  Update: *Not fixed.*
- In the `PolicyVotes` contract, the synonyms `veto`, `challenge`, and `noConfidence` are used interchangeably to refer to the same concept. This makes the readers of the code wonder if the three words are actually synonyms in the contract.
  Consider choosing only one of them and using it in all comments, variables, and functions.
  Update: *Fixed. The* `PolicyVotes` *contract now uses* `veto` *only, never* `challenge` *nor* `noConfidence`
- The cast to address in L205 of the `PolicyProposals` contract is unnecessary because the variable is already an address. Consider removing it.
  Update: *Not fixed.*

make it clear that some proposals with the same amount of stake as the selected ones can be left out of the ballot.

Update: *Fixed. The `best` variable now holds only one address.*

- In the `PolicyProposals` contract, when a proposal is registered, a `ProposalAdded` event is emitted. This event only logs the address of the proposal. Consider also logging the address of the proposer.

  Update: *Not fixed.*

- In the `computeVote` function of the `CurrencyGovernance` contract, when the result of the vote is in favor of inflation, an `InflationStarted` event is emitted. When the result is in favor of deposit certificates, a `DepositCertificatesOffered` event is emitted. These events only log the address of the contract that will handle the inflation or the deposit certificates. Consider adding the resulting values of the vote as parameters to the events:
  - `InflationStarted` log `winners` and `prize`
  - `DepositCertificatesOffered` log `certificatesTotal` and `interest`

  Update: *Not fixed. Eco's statement for this note: "The `InflationStarted` and `DepositCertificatesOffered` events don't contain the values of the votes as those values are already available through the `VoteResults` event."*

- The `CurrencyGovernance` contract contains a configurable `votingVDFDifficulty` parameter that enforces a computational delay between when a vote is committed and when it can be unilaterally revealed by a third party.
  To ensure votes remain secret during the voting phase, the difficulty should be chosen so that optimized hardware requires longer than the `VOTING_TIME` parameter to compute the VDF result.
  This implies voters will need to pre-compute their own VDF result in preparation for the vote. The Eco team have indicated that voters will be aware of this requirement and will likely prepare the VDF result well in advance. Consider documenting this requirement, along with the rationale in the contract's comments.
  Update: *Not fixed. Eco's statement for this note: "VDFs will no longer be used in our currency governance process once it has been updated based on economic model feedback from our advisors.".*

of code after the `else` block, the `return` statement is not necessary. Consider removing the `return` statement.

Update: *Not fixed.*

- The docstrings of the contracts and functions are partially following the Ethereum Natural Specification Format (NatSpec). They use the `@title` and `@param` tags, but they do not use `@return`, `@notice`, or `@dev`. Consider adding the missing tags.
- The following state variables are using the default visibility:
  - In `CurrencyGovernance.sol`: `inflation`, `prize`, `winners`, `certificatesTotal`, `interest`, `validInflation`, `validPrize`, `validCertificatesTotal`, `validInterest`
  - In `Inflation.sol`: `generation`
  - In `PolicyProposals.sol`: `generation`
  - In `PolicyVotes.sol`: `generation`.
  - In `TrustedNodes.sol`: `trustedNodeIndex`

    For readability, consider explicitly declaring the visibility of all state variables. Consider declaring as `private` all the state variables that do not require access from other contracts.

    Update: *Fixed.*
- Variables are declared as `uint` in `TrustedNodes.sol`: L27, L64, and L65. To favor explicitness, consider changing all instances of `uint` to `uint256`.

  Update: *Fixed.*
- The `uint128` type is used in the `DepositCertificates` contract for the `_amount` and `_interest` variables. The reason for not using the more common `uint256` type is not clear, and this is not explained in the comments of the functions. Consider documenting the reason for using `uint128` or updating all variables to be `uint256`.

  Update: *Not fixed.*
- To favor explicitness and readability, several parts of the contracts may benefit from better naming. Our suggestions are:
  - In `CurrencyGovernance`:
    - `VOTING_TIME` to `COMMIT_PHASE_DURATION`
    - `REVEAL_TIME` to `REVEAL_PHASE_DURATION`
    - `votingEnds` to `commitPhaseDeadline`

- `prize` to `inflationPrize`
- `winners` to `inflationWinnersCount`
- `certificatesTotal` to `depositCertificatesAmount`
- `interest` to `depositCertificatesInterest`
- `inflationVotes` to `votedInflationAmount`
- `prizeVotes` to `votedInflationPrize`
- `certificatesTotalVotes` to `votedDepositCertificatesAmount`
- `interestVotes` to `votedDepositCertificatesInterest`
- `validInflation` to `validInflationAmountVotes`
- `validPrize` to `validInflationPrizeVotes`
- `validCertificatesTotal` to `validDepositCertificatesAmountVotes`
- `validInterest` to `validDepositCertificatesInterestVotes`

In `DepositCertificates`:

- `SALE_TIME` to `SALE_PHASE_DURATION`
- `LOCKUP_TIME` to `LOCKUP_PHASE_DURATION`
- `saleEnds` to `salePhaseDeadline`
- `lockupEnds` to `lockupPhaseDeadline`
- `INTEREST_PERIOD` to `INTEREST_PERIOD_DURATION`
- `totalDeposit` to `depositedAmount`
- `withdrawnInterest` to `withdrawnAmount`
- `certificatesTotal` to `maximumDepositsAmount`

In `Inflation`:

- `PAYOUT_PERIOD` to `PAYOUT_PHASE_DURATION`
- `REGISTRATION_TIME` to `REGISTRATION_PHASE_DURATION`
- `registrationEnds` to `registrationPhaseDeadline`
- `holders` to `ticketHolders`
- `claimed` to `claimedSequenceNumbers`
- `Claimed` to `TicketsClaimed`
- `registerFor` to `registerTicketsFor`

In `PolicyProposals`:

- `Props` to `Proposal`

- `staked` to `stakers`
- Split `totalproposals` into `proposalsReceived` and `proposalsToRefund` (Note here that the variable is not using camel case.)
- `allProposals` to `proposalContracts`
- `best` to `mostStakedProposals`
- `PROPOSAL_TIME` to `PROPOSAL_PHASE_DURATION`
- `STAKE_TIME` to `STAKE_PHASE_DURATION`
- `COST_REGISTER` to `PROPOSAL_REGISTRATION_COST`
- `REFUND_IF_LOST` to `UNSELECTED_PROPOSAL_REFUND_AMOUNT`
- `MAX_PROPOSALS` to `MAX_SELECTED_PROPOSALS_NUMBER`
- `proposalEnds` to `proposalPhaseDeadline`
- `stakeEnds` to `stakePhaseDeadline`.
- `_policyvotes` to `_policyVotesImplementation` (Note here that the variable is not using camel case.)
- `_simplepolicy` to `_simplyPolicyImplementation` (Note here that the variable is not using camel case.)

- In `PolicyVotes`:
  - `Prop` to `Proposal`
  - `proposal` to `contract`
  - `LOCKUP_TIME` to `LOCKUP_PHASE_DURATION`
  - `HOLD_TIME` to `REVIEW_PHASE_DURATION`
  - `VOTE_TIME` to `COMMIT_PHASE_DURATION`
  - `REVEAL_TIME` to `REVEAL_PHASE_DURATION`
  - `CHALLENGE_TIME` to `CHALLENGE_PHASE_DURATION`
  - `holdEnds` to `reviewPhaseDeadline`
  - `voteEnds` to `commitPhaseDeadline`
  - `revealEnds` to `revealPhaseDeadline`
  - `challengeEnds` to `challengePhaseDeadline`
- In `TimedPolicies`:
  - `CURRENCY_TIME` to `MINIMUM_MONETARY_VOTE_INTERVAL`
  - `POLICY_TIME` to `MINIMUM_NETWORK_VOTE_INTERVAL`
  - `CurrencyGovernanceDecisionStarted` to `MonetaryVoteStarted`

- ***Update:*** *Not fixed.*

JTNDZGl2JTIwY2xhc3MlM0QlMjJidG4tY29udGFpbmVyJTIyJTNFJTBBJTBBJTNDYnV0dG9uJTIwb25jbGljayUzRCUyMmN1c3RvbXNjcm9sbCUyOCUyOSUyMiUzRSUzQ2ElMjBocmVmJTNEJTIyJTIzcGhhc2UtMiUyMiUyMGNsYXNzJTNEJTIyY3VzdG9tLWxpbmslMjIlM0UlM0MlMjBQcmV2aW91cyUzQyUyRmElM0UlM0MlMkZidXR0b24lM0UlMEElMEElM0NidXR0b24lMjBjbGFzcyUzRCUyMmJvbmNsaWNrJTNFJTIyY3VzdG9tc2Nyb2xsJTI4JTI5JTIyJTNFJTNDYSUyMGhyZWYlM0QlMjIlMjNwaGFzZS00JTIyJTIwY2xhc3MlM0QlMjJidXN0b20tbGluayUyMiUzRW5leHQlMjAlM0UlM0MlMkZhJTNFJTNDJTJGYnV0dG9uJTNFJTBBJTBBJTNDJTJGZGl2JTNF

Phase 4: Verifiable Delay Function

The phase 4 audit was delivered to the Eco team on July 17th, 2019.

The audited commit is `de81d9bad4195e03f07aedd2a6817f0cb04a8c8d`, and the files included in the scope were VDFVerifier.sol and BigNumber.sol.

Note that the audit was to verify the correct functioning of the contracts with respect to their intended behavior and specification. The cryptographic strength of the design was not explored and should be verified by cryptographic experts.

Here are our audit assessment and recommendations, in order of importance.

Update: *The Eco team applied a number of fixes based on our recommendations. We address below the fixes introduced up to commit* `af3428020545e3f3ae2f3567b94e1fbc5e5bdb4c` *of the currency repository.*

Contract Summary

VDFVerifier.sol

A Verifiable Delay Function (VDF) is a mathematical technique to produce a provably unbiased random number. This is achieved by constructing a function that requires a known minimum time to compute, even with parallel hardware, but can be verified quickly.

without the cooperation of the original voter.

The `VDFVerifier` contract verifies claimed solutions to VDF inputs based on the Pietrzak specification.

BigNumber.sol

The `BigNumber` library provides a mechanism to perform mathematical operations on unsigned integers that are larger than `uint256`. It achieves this by storing the numbers as `bytes` arrays, operating on one 256-bit word at a time and then combining the results back into the `bytes` arrays.

It is used in the `VDFVerifier` contract as part of the verification procedure but it is also designed to perform generic mathematical operations.

Ongoing Research

The number of sequential operations in the VDF is a configurable parameter. It must be large enough that no plausible attacker can compute the VDF solution faster than the minimum time bound. However, it must also be small enough for legitimate users to compute VDF solutions within reasonable times.

This is a balancing act, and the ideal range of values is still an area of ongoing research, which includes estimating the effectiveness of the best designs executed in any plausible hardware, as well as attempting to reduce a well-resourced attacker's advantage over consumer grade hardware.

The Eco team intends to track this research and configure the VDF difficulty parameter accordingly.

Open Questions

Two of the questions that were raised during the audit have been taken under consideration by the Eco team to confirm with their cryptographic experts.

Firstly, a direct reading of the VDF specification suggests that all values should be in the positive signed quadratic residue group with modulus `N`. This means they should be squares (mod `N`),

perform validity bounds checking of transformations into values less than [illegible]. The purpose of using the signed quadratic residue group is that membership in it can be efficiently tested. Eco has sidestepped this issue by explicitly squaring all values before they are used in calculations. This makes it plausible that the additional mapping is unnecessary. Nevertheless, this should be confirmed with a cryptographer. The Eco team have indicated that the algorithm was originally designed with a cryptographer and they are in the process of double-checking its validity.

Secondly, in one use case, the Eco contracts use an Ethereum block hash, which is a 256-bit pseudo-random value, such as a VDF input. However, if an adversary precomputes the VDF solution for a large number of prime values and the block hash happens to have factors exclusively drawn from that list, the adversary will be able to use their precomputed solutions to immediately calculate the required solution. This raises a question: What is the required security strength S, and what is the chance that a uniformly random 256-bit value would be S-smooth? The Eco team have indicated that they are in the process of resolving that question.

Critical Severity

[P4-C01] Unknown attacker advantage

Critical

The VDF difficulty parameter must be chosen to ensure any plausible attacker cannot compute the VDF solution within the minimum time bound. Correctly setting this value requires estimating the capability of the attacker's hardware. Unfortunately, this is an area of active research and therefore is currently unknown. It is worth noting that the Ethereum Foundation, Protocol Labs, and other contributors are actively involved in this research and intend to significantly raise the standard of widely available hardware in order to minimize the plausible attacker advantage.

The Eco security documentation provides time estimates for consumer hardware, but it does not contrast this with the advantage an attacker may have from using an optimized ASIC.

Consider documenting the maximum attacker advantage implied by the choice of time windows and difficulty functions for all use cases. However, it is our understanding that the field is not mature enough to confidently estimate the true upper bound, and the procedure will remain vulnerable to potential attackers with a greater advantage than the selected one.

> We've done internal research on high end consumer hardware and extrapolated beyond that to attempt to account for custom hardware. However, we anticipate more information about the power of custom hardware before we launch. We'll need to review the Ethereum Foundation's work on hardware VDF implementations and factor those gains in to the security parameters we use in our VDFs at launch.

[P4-C02] Calculation error in the BigNumber addition

Critical

The `BigNumber` library has an `innerAdd` function that adds two numbers, starting from their least significant 256-bit words.

This function needs to take the carry into account, when the addition of two words overflows. To do this, in L283, it checks if the word of the first number exceeds the minimum size that can be added without overflow. This value should be calculated as follows:

```
word1 + word2 + carry > MAX_256_BIT_INTEGER [ overflow condition ]
=>    word1 > MAX_256_BIT_INTEGER - word2 - carry
=>    word1 > MAX_256_BIT_INTEGER - ( word2 + carry )
```

However, it is calculated as:

```
word1 > MAX_256_BIT_INTEGER - ( word2 - carry )
```

In addition to this incorrect calculation, the last term underflows when `word2` is 0 and `carry` is 1.

Consider fixing the carry calculation. Note that the correct calculation obsoletes the `switch` statement in L288, which should handle the case where `word2 + carry` overflows instead.

Update: *Fixed. First, the conditional was updated to the suggested calculation. And afterwards, the switch statement was updated to handle the case where* `word2 + carry` *overflows.*

High Severity

[P4-H01] Undocumented cryptographic algorithm changes

High

The verification procedure in the `VDFVerifier` contract is based on the The Simple Verifiable Delay Functions specification. However, the implementation has some differences with the specification:

- The implementation uses the Fiat-Shamir heuristic to make the verification non-interactive. This needs to be done very carefully to ensure the prover, who now has some control over the random number, cannot leverage their additional power to generate a false proof.
- The specification requires that all input variables are tested to ensure they are in the Positive Signed Quadratic Residue group mod `N`. Instead, the input values are squared to produce a corresponding value in that group.
- The commitment is for the input `x` value before it is squared, which is a value that has no analogue in the specification.
- In the specification, the difficulty of computing the VDF is parameterized by the number of squaring operations, which can be any positive integer. In the implementation, it is parameterized by the number of steps in the proof, thereby implicitly fixing the number of squaring operations to be a power of 2.
- The last step in the specification occurs when the function has been reduced to `y = x^2`. The implementation stops at the previous step when `y = x^4`.

There are practical reasons for these deviations, and they mostly restrict and simplify the procedure, but as a general principle, modifications to cryptographic algorithms have a high probability of introducing vulnerabilities. They may also mislead users who are unaware of the changes to misuse the algorithm.

Consider documenting any discrepancies between the specification and the implementation, along with an explanation for the change and a security justification.

Update: *Not fixed. Eco's statement for this issue:*

because it's applied as described in the source paper "Simple Verifiable Delay Functions" (Pietrzak) and in the similar paper "Efficient verifiable delay functions" (Wesolowski).

Ben Fisch, one of our early team members and cryptographic advisors, drafted the following responses to the points raised:

- Our implementation follows the protocol for the original version of Simple Verifiable Delay Functions, which can be retrieved from ePrint. Our implementation squares the input x to produce z = x^2, and then runs the VDF on z as an input. This guarantees that z is in the Quadratic Residue subgroup mod N. The VDF verifier program receives x, which enables it to verify that z is in the Quadratic Residue subgroup. Thus, the VDF protocol we implemented, which proves that y is the result of the repeated squaring of z a specified number of times, has been proven secure in both the original version of the paper, as well as in the survey article https://crypto.stanford.edu/~dabo/pubs/papers/VDFsurvey.pdf. It is statistically secure as long as the input z lies in a group with no low order elements, which is indeed true of the Quadratic Residue subgroup mod N, where N is the product of strong primes.
- On page 6 of the latest version of "Simple Verifiable Delay Functions", referenced here as the "specification", it is explained that the original version of the paper used the subgroup of Quadratic Residues, while the newer version uses the Positive Signed Quadratic Residues. The motivation for this change is irrelevant to our setting, where it is completely fine to fix the VDF input challenge to be x^2. In fact, the same technique is also used in the recent paper Continuous Verifiable Delay Functions. https://eprint.iacr.org/2019/619.pdf.
- A restriction on the number of squaring operations does not affect security, and is good enough for our application.
- Stopping at `y = x^2` does not affect the security of the protocol.

[P4-H02] Excessive use of assembly

High

The `BigNumber` library uses inline assembly to manage buffers and manipulate individual words in memory. However, in many functions, it also uses assembly to perform the calculations and

Consider extracting low-level memory management operations (like storing a word at a particular location or clearing the zero words from the start of a buffer) into separate functions that use inline assembly and using Solidity for all other logic and processing.

Update: *Not fixed. Eco's statement for this issue:*

> The `BigNumber` library is external code that we've imported so we could modify it. We'd prefer to participate in a rewrite in collaboration with someone who could be a permanent owner for the library.

[P4-H03] Missing unit tests

High

Tests are the best way to specify the expected behavior of a system. When the code is written test-driven, the test coverage is automatically 100%, many issues are prevented, and the design of the code can be improved. Additionally, when bugs are found, it is easier to fix them without introducing regression errors.

There are no unit tests for the functions of the `BigNumber` library. Instead, they are tested through the unit tests of the `VDFVerifier` contract. The correct functionality of this library is critical for the success of the system. Even if all the features of `BigNumber` used by the `VDFVerifier` are currently correct, it is possible that, in the future, a change in the verifier assumes that an untested `BigNumber` code path is correct and starts breaking in surprising ways.

In this case, it is particularly important because a large fraction of the contract is written in assembly, which is harder to parse and reason for. Note that the use of assembly discards several important safety features of Solidity, which may render the code less safe and more error-prone.

Consider test-driving a rewrite of the `BigNumber` library.

Update: *Not fixed. See Eco's statement for this issue in* ***[P4-H02] Excessive use of assembly****.*

[P4-H04] BigNumber internal values can be manipulated

`uint256` or a `bytes` buffer with a length not divisible by 32, a new `bytes` buffer is allocated for the `BigNumber` object.

However, when it is initialized with a `bytes` buffer with a length divisible by 32, it uses that buffer directly in its internal representation. This means that if the caller subsequently clears or modifies the input buffer, it will corrupt the `BigNumber` value.

Similarly, the `asBytes` functions return a new buffer when no output size is specified or the output needs to be padded However, they return a reference to the internal `bytes` buffer when the requested size matches the `BigNumber` size. Once again, if the caller modifies the returned buffer, it will corrupt the `BigNumber` value.

Consider creating a new buffer for every `BigNumber` instance and returning a new buffer whenever `asBytes` is called.

Update: *Not fixed. Eco's statement for this issue:*

> A user of the library could definitely modify the buffer as described, but our system uses the library internally only so there's no opportunity to use this maliciously. The problem should be fixed before this `BigNumber` code is released as a library, but it doesn't represent an attack against our system.

Medium Severity

[P4-M01] Undocumented security assumptions

Medium

The `VDFVerifier` contract uses a constant modulus `N`. This is the 2048-bit number from the (elapsed) RSA challenge. The security of the VDF depends on various assumptions about that value:

- It is the product of two primes.
- The factors are safe primes.
- No attacker knows or can obtain the factorization.

Consider adding the reason for the choice of `N` as well as the security assumptions and implications to the documentation.

Update: *Fixed. A comment has been added to explain the security assumptions of the chosen* `N`.

[P4-M02] VDF inputs may be precomputable

Medium

The security of the VDF relies on potential attackers having a known maximum time window to complete the attack. This assumption can be undermined if:

- The attacker has precomputed the VDF solution for the input `x`.
- The attacker has precomputed the VDF solution for the factors of `x`.

In the current design:

- The `Inflation` contract uses a recent Ethereum block hash as input to the VDF. This is believed to be probabilistically immune to precomputation (see the Open Questions section).
- The individual trusted nodes use a VDF to mask their votes during currency governance. The VDF inputs will be chosen by the Eco-provided software to be immune to precomputation.

However, the `VDFVerifier` contract is designed to be usable in other scenarios and should enforce secure inputs wherever possible. Potential modifications:

- Enforce a minimum `x` value outside the range of plausibly precomputed values. It is worth noting that the Eco VDF documentation states that `x` should be at least 256 bits, but this is not enforced by the contract.
- Hash the input `x` value to probabilistically ensure a large factor.
- Use the Miller-Rabin primality test to ensure `x` is probably prime and is therefore immune to having the VDF solution of its factors precomputed. This idea was suggested by the Eco team.

Consider restricting or modifying the input `x` to mitigate against precomputation and documenting the method chosen, along with its security assumptions.

> The next iteration on our code will reduce the use of VDFs to just a few cases, where the input will always be a block hash. We believe that this, combined with the Miller-Rabin primality testing, will be sufficient to ensure the security of the processes it will be used in.

[P4-M03] Returning the incorrect buffer

Medium

The `innerModExp` function in the `BigNumber` contract has an assembly loop that removes leading zero words in the result and decrements the buffer length accordingly. However, if the base value is zero, the result will be zero. This will cause the loop to continue past the end of the result buffer until it reaches the first non-zero word. Then the length field will underflow and become huge, causing the function to return an arbitrary junk value. Alternatively, if there are no non-zero bytes after the return buffer, the loop will never terminate and exhaust all the gas.

Similarly, the `privateRightShift` also has an equivalent loop. If the right shift causes the result to be zero, it will also either return an arbitrary junk value or enter an infinite loop.

Consider adding bounds checking to ensure the loop does not pass the length of the result buffer.

Update: *Fixed. Now both `privateRightShift` and `innerModExp` implement bound checks.*

[P4-M04] The identity precompile return value is ignored

Medium

In the `innerModExp` function of the `BigNumber` library, the identity precompiled contract stored at address `0x4` is called in L543, L554, and L565 to copy the arguments to memory. Only the last call checks the return value for errors during the call.

Consider adding checks to all the calls to the identity precompiled contract.

Update: *Fixed. Now, the results of all the calls to the identity precompiled contract are combined with an `and` operation, and afterwards, they are checked that none of the calls failed.*

When starting a proof in the `VDFVerifier` contract, the minimum `t` value is 1. However, the `update` function that is used to complete the proof has a minimum `_nextProgress` value of 1, which is assumed to be less than `t` (and `t - 1` in the last step of the proof).

The contract should be able to directly verify the relationship, but it requires the prover to provide the intermediate `u` value, which does not exist. Consequently, the prover cannot complete the proof.

Consider requiring `t` to exceed 1, or provide a mechanism for the contract to verify the function when `t` is 1.

Update: *Fixed. Now `t` must be at least 2.*

Low Severity

[P4-L01] Inconsistent representations of zero

Low

The `BigNumber` library makes a new instance by either passing a dynamically-sized `bytes` array or a `uint256` value.

When the empty `bytes` array is used to make a new instance, it is internally represented as an array of length 0. This is consistent with the manually-constructed values produced by other functions. However, when the 0 `uint256` is used to make a new instance, it is internally represented as an array of 32 zero bytes. This inconsistency makes the `cmp` function fail when the two different representations are compared. Consequently, the functions that use `cmp` (like `privateDiff`) will behave in unexpected ways when operating on the two different representations of 0.

Consider using a single internal representation for an instance of value 0.

Update: *Fixed. Now, when the `BigNumber` is instantiated by passing a 0 `uint256` value, is represented as an array of length 0.*

Low

In the function `opAndSquare` of the `BigNumber` library, the square is calculated by using the modular exponentiation precompile with a modulus value that is big enough to return the full exponentiation value.

The value of modulus is calculated in L487 and L496 with the following formula:

```
((base.length * 16 / 256) + 1) * 32
```

This is very hard to understand and unnecessary given the rest of the function assumes that `base.length` is a multiple of 32. Consider simplifying it to:

```
(base.length * 2) + 32
```

This shows in a clearer way that it has to be twice the length plus one extra word.

Update: *Fixed*.

[P4-L03] No mechanism to make a BigNumber instance from a zero byte

Low

The `BigNumber` library makes a new instance by calling one of the two `_new` functions. The first `_new` function receives a dynamically sized `bytes` array. When the array size is not a multiple of 32 bytes, it confirms that the first element of the array is not 0.

This means that it is not possible to make a new instance from a `bytes` array with a single zero byte.

However, it is possible to make a new instance with value 0 by calling the first `_new` function with an empty `bytes` array or by calling the second `_new` function with the 0 `uint256`. Nevertheless, for completeness, consider supporting the `bytes` array `0x0` as a valid argument to make a new `BigNumber` instance.

> published library.

[P4-L04] Lack of input validation

Low

In the `BigNumber` library, the function `privateRightShift` has a `uint256` parameter for the number of bits to shift. The maximum value for that parameter should be 256, but this is never validated. If this function is called by mistake with a higher value, there will be an underflow in L632, and the results returned will be unexpected.

This is not currently vulnerable, because the `privateRightShift` function is only used internally to shift 2 bits. However, it will be hard to find the problem if the code is changed in the future and the wrong value is mistakenly passed to the function.

Consider adding this validation to fail as early and loudly as possible.

Update: *Fixed. The `privateRightShift` function now has a requirement to only shift by two positions.*

[P4-L05] Confusing loop boundary

Low

In line 639 of the `BigNumber` library, the variable `max` is intentionally set (unnecessarily with assembly) to `uint256(-32)`. It is then used to bound a loop that starts counting from a multiple of 32 down to 0, subtracting 32 each time until it terminates when it reaches `max`. This is very confusing. Presumably, this pattern was chosen because the condition `i >= 0` would always succeed for a `uint256` value. Consider using a signed loop counter and stopping the loop when it becomes negative.

Update: *Fixed. Note, however, that this change reduces the maximum size of a `BigNumber`. The length of the array is stored in a `uint256` variable. Now, if the length is bigger than 2^128, it will overflow the cast to `int256`, giving unexpected results. This has been reported as an issue in the Phase 5.*

Low

In the `innerAdd` function of the `BigNumber` library, the length of the two parameter arrays are repeatedly loaded with `mload(_max)` and `mload(_min)`. Consider assigning the length of the arrays to two variables named `maxLength` and `minLength`, which will make the code more readable.

Also, note that in L277, the difference between `mload(_max)` and `mload(_min)` is repeatedly calculated inside a `for` loop. This difference is a constant, so consider calculating it once and assigning it to a variable.

Update: *Fixed in 68adc904.*

[P4-L07] Maximum uint256 value calculated with underflow

Low

In L261 and L370 of the `BigNumber` library, the maximum `uint256` value is calculated by forcing an underflow.

This is hard to read. Consider using `not(0)` instead.

Update: *Fixed.*

[P4-L08] Inconsistent use of hexadecimal and decimal numbers

Low

In the `BigNumber` library, hexadecimal and decimal numbers are used inconsistently. For example, see `0x20` in L58 and `32` in L88.

To make the code easier to read, consider using number bases consistently.

Update: *Fixed. The `BigNumber` library now uses hexadecimal numbers.*

[P4-L09] Use of magic constants

L545, L583, and L584). These values make the code harder to understand and maintain.

Consider defining a constant variable for every magic constant, giving it a clear and self-explanatory name. For complex values, consider adding an inline comment explaining how they were calculated or why they were chosen. All of this will allow for added readability, easing the code's maintenance.

Update: *Not fixed. Eco's statement for this issue:*

> We didn't refactor all of the code when we imported the library. We'd prefer not to include external libraries this way at all, and are interested in helping to build a gas-efficient general big number library for Solidity. Once one is available we'll migrate the code over to use it.

[P4-L10] Misleading comments

Low

In lines 600 and 606 of the `BigNumber` library, the comments refer to the variable `length_ptr` as `start_ptr`. Consider updating the comments to reflect the correct variable name.

In line 616, the comment is only valid if there are no leading zero words. Otherwise, the offset needs to be adjusted. Consider updating the comment to include the case with at least one leading zero word.

Update: *Fixed.*

[P4-L11] Missing explicit return statement

Low

Solidity allows the naming of the return variables on function declarations. When this feature is used, it is optional to use the `return` statement. If `return` is not explicitly called by the time a function ends, it returns any value assigned to the named return variable.

For explicitness and readability, consider always using the `return` statement.

Update: *Fixed.*

[P4-L12] Not using mixed case for variables

Low

In the assembly blocks of the `BigNumber` library, variables are named using lowercase with underscores.

This does not follow the Solidity style guide and is inconsistent with the rest of the variables declared outside of assembly blocks.

Consider using mixed case for all variables.

Update: *Not fixed. See Eco's statement for this issue in* ***[P4-L09] Use of magic constants***

Notes

- In the `BigNumber` library, the `cmp`, `innerAdd`, `innerDiff`, and `opAndSquare` functions all implicitly assume the `BigNumber` instances have a length divisible by 32 bytes. To assist code comprehension and to catch errors early, consider adding an `assert` statement in each function to validate this condition.
  Update: *Fixed.*
- Lines 481 and 516 of the `BigNumber` library, initialize a `BigNumber` instance by casting from a constant value with a long string of leading zeros. For clarity, consider passing a `uint256` to the `_new` function. Similarly, line 494 allocates a word of memory by assigning a long hex string of zeros to a variable. Consider using the `new bytes` constructor.
  Update: *Fixed.*
- Some variables are declared as `uint` and `int` in the `BigNumber` library (i.e., L126, L136, L191, L238, and L344). To favor explicitness, consider changing all instances of `uint` to `uint256` and `int` to `int256`.
  Update: *Fixed.*

- In lines 272 and 383 of the `BigNumber` library, there is a condition that uses two `eq` operators to check if a value is non-zero. In lines 318, 412, 429, 603, 647, and 668, the `eq` operator is used to check if a value is zero. For clarity, consider using the `iszero` operator.
  Update: *Fixed.*
- In lines 574 and 591 of the `BigNumber` library, there is a switch statement with only one case. For clarity, consider using an `if` statement.
  Update: *Fixed.*
- In line 603 of the `BigNumber` library, a boolean value is compared with 1. Consider removing this redundant check.
  Update: *Fixed.*
- In the first `_new` function of the `BigNumber` library, `_val.length` is assigned to a variable after it has been used repeatedly. Consider defining the variable at the top of the function and using it throughout.
  Update: *Fixed.*
- In lines 111–113 of the `BigNumber` library, fill a buffer from the back by incrementing the loop counter and subtracting it from the buffer length. For clarity, consider counting down.
  Update: *Not fixed.*
- The `opAndSquare` function of the `BigNumber` library has a boolean parameter to select between the `privateAdd` and `privateDiff` operations. Consider adding two functions named `addAndSquare` and `absDiffAndSquare` to make it clearer which operation will be executed. These two functions should be the only direct callers of `opAndSquare`.
  Update: *Not fixed.*
- To favor explicitness and readability, several parts of the contracts may benefit from better naming. Our suggestions are:
  - In `BigNumber.sol`:
    - `val` to `value`
    - `_new` (L48, L85) to `new_`, following the Solidity style guide
    - `_val` (L48, L85) to `_value`
    - `r` (L48, L85) to `instance`
    - `word` to `mostSignificantWord`
    - `len` to `length`

- `privateAdd` to `add`
- `privateDiff` to `absdiff`
- `privateMod` to `modulo` or `mod`
- `size` to `argBufferSize`
- `length_ptr` to `resultPtr`
- `mask` to `precedingWord`

◦ ***Update:*** *Most of the renames were applied.*

JTNDZGl2JTIwY2xhc3MlM0QlMjJidG4tY29udGFpbmVyJTIyJTNFJTBBJTBBJTNDYnV0dG9uJTIw
b25jbGljayUzRCUyMmN1c3RvbXNjcm9sbCUyOCUyOSUyMiUzRSUzQ2ElMjBocmVmJTNEJTIyJ
TIzcGhhc2UtMyUyMiUyMGNsYXNzJTNEJTIyY3VzdG9tLWxpbmslMjIlM0UlM0MlMjBQcmV2aW91
cyUzQyUyRmElM0UlM0MlMkZidXR0b24lM0UlMEElMEElM0NidXR0b24lMjBvbmNsaWNrJTNEJTI
yY3VzdG9tc2Nyb2xsJTI4JTI5JTIyJTNFJTNDYSUyMGhyZWYlM0QlMjIlMjNwaGFzZS01JTIyJTIw
Y2xhc3MlM0QlMjJjdXN0b20tbGluayUyMiUzRW5leHQlMjAlM0UlM0MlMkZhJTNFJTNDJTJGYnV0
dG9uJTNFJTBBJTBBJTNDJTJGZGl2JTNF

Phase 5: Integration

The phase 5 audit was delivered to the Eco team on November 8th, 2019.

To finish the audit engagement with the Eco team, we wanted to spend some additional time reviewing the changes they had done to the contracts since we audited them and to evaluate the system as a whole.

In this phase, we reviewed the integration of the 4 previous phases. We worked on commit `af3428020545e3f3ae2f3567b94e1fbc5e5bdb4c` of the currency repository.

Here are our audit assessment and recommendations, in order of importance.

Update*: the Eco team made some fixes based on our recommendations. We address below the fixes introduced up to commit* `d1b4bfe01310c709e54f9b4bf3fc123fba55af2a` *of the* `currency` *repository.*

Critical Severity

No new high severity issues were found.

Medium Severity

[P5-M01] Policy self-removal not failing loudly

Medium

Component: `policy`

In the `Policy` contract, the `removeSelf` function checks if the sender is the current interface implementer. If it is not, the function just does nothing, failing to explicitly inform the caller of a failed transaction.

Following the Fail Loudly principle, and to avoid hiding errors from the caller, consider reverting the transaction when the sender is not required.

***Update**: Not fixed. The code remains the same. Nevertheless, now the comment has changed and it states, "[…] if another contract has taken the role, this does nothing". Eco's statement for this issue:*

> The `removeSelf` function fails silently to allow its use in places where another contract may have taken ownership of the role. For example, when a contract is being destructed it can first `removeSelf` from its usual role without worrying that another contract might have been granted the role.

Low Severity

[P5-L01] Use of hash constants without comment

Low

Component: `policy`

where the hashes come from.

Consider adding a comment before every constant with the string used to generate the hash.

***Update**: Fixed. Now all 11 constants have comments with the strings used to generate the hashes.*

[P5-L02] The AccountBalanceGenerationUpdate event is missing relevant data

Low

Component: `currency`

In the `EcoBalanceStore` contract, the `AccountBalanceGenerationUpdate` event does not log the generation to which the address has been updated. This may impede reconstructing the history of generation updates for an account through emitted events.

Consider adding the generation as a parameter to the event.

***Update**: Fixed. The event now logs the `generation` too.*

[P5-L03] Outdated README file in the currency repository

Low

Component: `currency`

The `README` file of the `currency` repository is outdated. For example, it refers to the functions `burn` and `updateTo`, which have been renamed. It also mentions that the `EcoBalanceStore` contract implements the `ERC20` interface, which is no longer true.

Consider updating all documentation in the `README` file to reflect the current state of the code base.

***Update**: Fixed. All documentation related to the `EcoBalanceStore` contract API has been removed.*

Low

Component: `currency`

In the `EcoBalanceStore` contract, the `tokenTransfer` and `tokenBurn` functions are do not follow the fail early pattern. These functions first update the balances, and afterwards check if the operation is authorized, reverting the transaction if not.

Consider first validating whether the sender is an authorized account, and then apply balance updates.

***Update**: Fixed. Now, in the `tokenTransfer` function the authorization check comes before the balance updates. The same happens with the `tokenBurn` function: the authorization check comes before the updates.*

[P5-L05] Undocumented authorization in the tokenBurn function

Low

Component: `currency`

In the `EcoBalanceStore` contract, the `tokenBurn` function allows the `policy` contract to burn tokens from any account. However, such sensitive behavior is not documented.

Consider explicitly stating this in the function's docstrings.

***Update**: Fixed. Now, the function's docstrings state that the root `policy` contract is authorized to call the function. Yet, it doesn't explicitly say that it can burn tokens from any account.*

[P5-L06] Unused Transfer event

Low

Component: `currency`

In the `EcoBalanceStore` contract, there is a declared `Transfer` event that is never used.

[P5-L07] Wrong comment in the AccountBalanceGenerationUpdate event

Low

Component: `currency`

The comment of the `AccountBalanceGenerationUpdate` event in the `EcoBalanceStore` contract mentions a value parameter that does not exist.

Consider removing this line from the comment.

***Update**: Fixed. The wrong comment in the `AccountBalanceGenerationUpdate` event has been removed.*

[P5-L08] Wrong comment in the tokenTransfer and tokenBurn functions

Low

Component: `currency`

In the `EcoBalanceStore` contract, the functions `tokenTransfer` and `tokenBurn` say in their docstrings, “This function is restricted and can only be accessed by authorized addresses — specifically those in the authorizedContracts set.”

However, the `authorizedContracts` array does not contain addresses. The array actually used in these functions is `authorizedContractAddresses`.

Consider updating the functions’ docstrings to better reflect which data structure holds the authorized addresses allowed to call them.

***Update**: Fixed. Now the `authorizedContractAddresses` array is mentioned in the docstrings of the `tokenTransfer` function and in the ones of the `tokenBurn` function.*

[P5-L09] Undocumented assembly block in the IsPrime contract

The `IsPrime` contract contains an assembly block without extensive documentation.

As this is a low-level language that is harder to parse by readers, consider including extensive documentation regarding the rationale behind its use, clearly explaining what every single assembly instruction does. This will make it easier for users to trust the code, for reviewers to verify it, and for developers to build on top of it or update it.

***Update**: Fixed. The ECO team has added inline comments to the assembly code.*

[P5-L10] Wrong comments for the calculation of maximum uint256

Low

Component: `VDF`

In lines 289 and 403 of `BigNumber.sol`, the maximum possible `uint256` is calculated with `not(0x0)`. However, the comments next to both statements say it is calculated using an underflow.

Consider updating the inline comments to avoid mismatches with the actual implementation.

***Update**: Fixed. Both comments in lines 289 and 403 have been updated to describe the actual implementation.*

[P5-L11] Maximum BigNumber is not enforced or documented

Low

Component: `VDF`

In the `privateRightShift` function of the `BigNumber` library, there is a cast of the array length from `uint256` to `int256`. This means that if the length of the array is bigger than 2^128, it will overflow the cast and become negative, giving unexpected results. We do not expect users of the library to require numbers this big. We do not even know what would happen to the

Consider adding a require statement to the `from` function or before the cast to make sure that it will never overflow.

Consider experimenting with the upper limits of the library to document the maximum possible `BigNumber` that it can handle.
***Update**: Fixed. Now, the length of the array is checked before the cast to `int256`. Note that the `length` local variable could be used to avoid reading the array's length again. Moreover, the error message should better denote the actual condition checked, such as "Length of the dividend's value must be equal or less than 1024 bytes".*

Notes

- In the `generateTx` function of the `nicks.js` file, there is an in-line statement to check if the `paramdata` argument is hexadecimal. Multiple in-line operations in the same line of code are hard to read. Consider extracting the hexadecimal check to a function.
  ***Update**: Partially fixed. The readability of the code that checks if `paramdata` is hexadecimal has been improved. Nevertheless, the check is still done in the `Transaction` generation and not in a separated function.*
- In the `ForwardTarget` contract, the string identifier to generate the `IMPLEMENTATION_SLOT` makes reference to the old name `"io.beam.ForwardProxy.target"`. Consider updating the string to `"com.eco.ForwardProxy.target"`. Note that if this value is changed, the value of the `IMPLEMENTATION_SLOT` state variable (which is currently the `keccak256` of `"io.beam.ForwardProxy.target"`), must be updated too.
  ***Update**: Fixed. The name string and the `IMPLEMENTATION_SLOT` have been updated to the new project's name. Also, it has been updated in other places where it is used like in the `ForwardProxy` contract.*
- In the `revoke` function of the `EcoBalanceStore` contract, there is a `require` statement hard-coded to fail passing a `false` argument. Consider replacing it with a `revert` statement.
  ***Update**: Fixed. The `require(false)` statement has been replaced with `revert`.*

cyUzQyUyRmElM0UlM0MlMkZidXR0b24lM0UlMEElMEElM0MlMkZkaXYlM0U=

Related Posts

Beefy Zap Audit

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits

Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits

OpenZeppelin

Company

About us
Jobs
Blog

Contracts Library

Docs

Privacy | Terms of Use