Celo Contracts Audit

OPENZEPPELIN SECURITY | FEBRUARY 5, 2021

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 cLabs team working on the Celo platform asked us to review and audit the smart contracts for the protocol. We examined the code, and here we publish our findings.

About Celo

The Celo platform implemented a new blockchain protocol and a node software forked from Geth. It has a proof-of-stake based rewards system, built-in stablecoins, governance procedures, and a system for linking accounts and phone numbers. The main goal of Celo is to make the blockchain easily accessible for anyone with a mobile phone.

The contracts are divided into four main groups: `common`, `governance`, `identity`, and `stability`. Each have their own folder of the same name within the `contracts` folder.

The `governance` folder handles the election of validators, which verify transactions and earn rewards for participating in the Celo network. Governance also provides a means to make proposals, which are transactions that are voted on and then executed by the Celo protocol. It also contains mechanisms for reward distribution and enforcing penalties for validator misbehavior.

The `stability` folder manages the stable token (Celo Dollars) and the reserve intended to keep this stable token collateralized. These files also contain the means for on-chain exchanges of both the native asset of the protocol (Celo Gold) and Celo Dollars.

Finally, the `common` folder contains resources used in multiple places throughout the protocol. A registry of all contracts and addresses is found here, which is used by most contracts in the protocol to find and access other contracts. Utilities like `FractionUtil` and `FixidityLib` exist here, as well as frequently needed contracts like `GoldToken`, which manages the Celo Gold.

Elections and Validators

The protocol encourages the creation of groups of validators, which are then voted for during every epoch to validate transactions on the Celo blockchain. To register as a validator, a user must lock up a certain amount of Celo Gold. Validators cannot be elected without being part of a validator group. This was chosen as a design feature in order to minimize the information gap between voters and validators. It also incentivizes good behavior of validators, since they risk being ejected from an otherwise good group if they misbehave.

To vote, Celo users lock up Celo Gold, and then they can apply this gold as votes to different validator groups. Roughly once a day, an election is held to determine the validators for the next day. Validator groups are elected, and then validator seats are distributed to them via the d'Hondt method. For every seat that a group receives, 1 validator account from the group is officially elected as a validator for the following epoch.

Validators and/or validator groups can be slashed if they act maliciously or are found to have too little up-time. This mechanism takes away a portion of their locked gold, and can forcibly remove validators from groups.

Importantly, users and validators receive rewards when voting for and participating in winning validator groups. Validators and groups that win elections receive rewards, as well as the voters for those groups. However, those who vote for losing groups receive nothing.

Proposal and Hotfixes

The governance system also can execute hotfixes, transactions to fix errors that have been detected in the network. This process requires off-chain and on-chain coordination. First, the cLabs team will need to publicly disclose the hash of the transaction that will be executed. Secondly, two thirds of the Celo community will need to whitelist this hash for deployment. The hotfix can only be triggered after it is whitelisted by the community and approved by the approver.

Attestations and Identity

Celo implements a mapping which ties accounts to cell phone numbers, as well as an on-and-off chain system to verify that account owners are who they say they are. This is done via the attestations system, which basically involves validators sending secret SMS messages to cell numbers, and then account owners proving they have received them.

Privileged Roles

There are two roles with far-reaching power. The first is the aforementioned virtual machine (VM), identified by the `address(0)`. The VM has the ability to make protocol-level changes. The VM is a powerful account, having the ability to do things like increase the total supply of Celo Gold or modify its oracles. The VM should only perform calls in an automated and predictable fashion, as dictated by the protocol.

The other role is the `approver`, which must approve all proposals before they can be executed. Initially, the `approver` is planned to be a multi-sig contract. The `approver` account does not receive rewards for participation.

The cLabs team plans to progressively decentralize the control of the `approver` and to protect control of the VM such that it is strictly limited to its intended functionality.

About the audit

The audit was divided in two phases. The first phase started in January 2020 and included a detailed review of all the smart contracts in the `celo-monorepo` up to that moment. The

All the smart contracts within the `contracts` folder of the `celo-monorepo` are included in the audit scope. All other code within the repository was assumed to work correctly, as stated by the cLabs team and the Celo Docs. In particular, the specific details on how the virtual machine is implemented in the node software was out of scope of this audit, and the report was made assuming that it works as intended by the developers. Moreover, oracles were not a part of the current codebase and were not reviewed by OpenZeppelin.

3 critical and 4 high-severity issues were found. You can see the reports of phase 1 and phase 2 for extensive details of every vulnerability found and how they have been fixed or mitigated.

We are impressed by the handling of this complex monorepo by the cLabs team. They are following very good processes to keep their project healthy. Of special interest for us during this phase are the small and focused pull requests, with proper descriptions and tags, links to reported issues, peer reviews and extensive automated testing.

Looking at how the cLabs team has carefully implemented fixes and new features during the time we have been auditing the smart contracts gives us confidence in that they are ready to publish the system, to constantly help monitor and update it, and to grow the repository in the direction required by the ecosystem.

Before starting the deep dive into the smart contracts, we performed a general health check of the project which we are now including in this full report.

Project Overview

(as of January 9th, 2020)

Summary

The Celo project has a healthy number of active contributors and dedicated maintainers for every package.

The documentation inside and outside of the repository is good and the community has plenty of ways to get news about the project and to communicate with the cLabs team.

- It has code and configuration files written in many different languages, which increases the learning curve to collaborate in some of the packages.
- The unit test coverage report does not cover the whole project.
- It has no documented security policy or instructions on how to communicate with the Celo developers to communicate a security issue.

Below we present our overview of the project.

We reported some health concerns and other security vulnerabilities and quality issues during the phase 1 of the audit.

Repository

The repository is hosted in GitHub, as part of the Celo organization.

It has 25 watchers, 77 stars and 48 forks.

The repository is a monorepo with 24 packages managed by lerna. This audit is limited to the smart contracts in the `protocol` package.

License

The project is free software.

The `protocol` package is released under the GNU Lesser General Public License version 3.

The rest of the project is released under the Apache License v2.0.

Languages

1345 files are written in TypeScript.
This is the majority of the source code in the repository.

There are also 116 files written in Solidity.
This audit is limited to the Solidity code.

Code Contributors

Total code contributors: 53

Maintainers

(According to the `CODEOWNERS` file)

- Nam Chu Hoai (nambrot), with 99 commits.
- J M Rossy (jmrossy), with 94 commits.
- Aaron DeRuvo (aaronmgdr), with 77 commits.
- Trevor Porter (tkporter), with 64 commits.
- Ashish Bhatia (ashishb), with 64 commits. Inactive since november 2019.
- Asa Oines (asaj), with 63 commits.
- Mariano Cortesi (mcortesi, with 42 commits.
- Connor McEwen (cmcewen), with 42 commits.
- Anna K (annakaz), with 40 commits.
- Tim Moreton (timmoreton, with 25 commits.
- Audrey Penven (yerdua), with 17 commits.
- Martin (m-chrzan), with 17 commits.

Frequent Contributors (during the last year)

- Jean Regisser (jeanregisser), with 57 commits.
- Victor “Nate” Graf (nategraf), with 30 commits.
- Sami Mäkelä (mrsmkl), with 25 commits.
- Pedro Gutiérrez (Pedro-vk), with 16 commits.
- Yorke Rhodes (yorhodes), with 14 commits.

Past Frequent Contributors (now inactive)

- Martín Volpe (martinvol), with 25 commits.
- sallyjyl, with 20 commits.
- Derrick Lee (drklee3, with 17 commits.

Issues

Closed: 871.

Open:

- Total: 362.
- Without activity for 3 months: 45.
- Without a reply nor a tag: 9.
- Tagged as "security": 1.
- Tagged as "audit": 17.
- Tagged as "bug": 82.
- Tagged as "good first issue": 11.
- Tagged as "help wanted": 1.
- Tagged as "1 hour tasks": 18.

Pull Requests

Closed: 1107.

Open:

- Total: 55.
- Without activity for a week: 22.
- Without a reply: 12.
- With more than 10 comments: 10.

Releases

Releases are tagged with Semantic Versioning.

Number of releases so far: 10.

First release on github: wallet v1.3.1 on July 23, 2019.

Release cadence: No stable release cadence. No release in the last month.

- Source code (tar.gz)

Projects

Celo uses GitHub projects to manage the pending tasks.

The relevant projects for this audit are Betanet 2019 and Betanet 2020.

Together, these projects have 199 tasks in the To do, 10 In progress, 11 with Review in progress, and 56 Done.

Branches

There are 144 branches in the organization repository. 122 are active and 21 are stale.

Development happens in the master branch.

Contributing

The project has documented guidelines for code contributors. There are also instructions to build the project.

The `protocol` package uses `solhint` and `tslint` to follow a consistent code style.

The project requires contributors to respect the Celo Community Code of Conduct.

Testing

Unit tests for the contracts of the `protocol` package are defined in the `test` directory. They are run by Circle CI on Github pull requests.

There is a test coverage report, but it only includes the `mobile` package.

There are two test networks: Alfajores and Baklava.

Documentation

There is a main documentation website aimed at developers.

Installation

There mobile wallet application can be installed from the Android and iOS app stores.

No installers or binary files have been published for the nodes.

Docker images are available to easily connect to the Baklava test network.

Communication

The people from the project have multiple ways to communicate with its users:

- Medium Blog, with 25 posts.
- Youtube channel, with 10 videos and 81 subscribers.
- Forum, with 67 topics.
- Twitter, with 73 tweets and 321 followers.
- Subreddit, with 51 members.

Security

The project does not have a Security Policy document.
There are no instructions for responsible disclosure of security vulnerabilities. There is no Bug Bounty program.

Audits

The project has not published any audit reports yet.

The commit for the first phase of the audit is `7be22605e172ca536c028e408b147aab83202e4a`. All code within the `contracts` folder is included in the audit scope.

find that the cLabs team had thoroughly considered some of the more complex nuances of the incentive and governance structure of their contracts, including how to manage ownership of privileged roles. We valued that the Celo Docs were up-to-date and had detailed descriptions of the protocol's intended functionality. We found that in some places, the code could have a bit more consistency and input validation, but generally we were impressed with the technical depth of the contracts.

Vulnerabilities

Below, we list all vulnerabilities identified in the Celo contracts.

Update: *The cLabs team made some fixes and comments based on our recommendations. We address below the fixes introduced in individual pull requests. Our analysis of the mitigations disregards any other changes to the code base. Note that at the time of this writing, not all pull requests have been merged.*

Critical

[C01] Funds of the `Escrow` contract can be locked by anyone

The `Escrow` contract allows users to send payments to other users who do not yet have a public and private key pair or an address.
Given two different actors, Alice and Bob, with Alice being a current participant of the Celo protocol and Bob a person who does not possess Celo Gold, Alice could use the `transfer` function to send a payment to Bob to an address specified in the `paymentId` parameter. This `paymentId` is used to save in storage the state of the whole process in the `escrowedPayments` mapping.
To withdraw the amount of money Alice has sent him, Bob uses the `withdraw` function, which is in charge of recovering from storage the state of the payment into the `payment` variable and after validating that the transaction is correct, it sends the funds to Bob's address.

Nevertheless, as there is not any validation on whether the `paymentId` has been used before to generate another transfer, it is possible for anyone to call a second time the `transfer` function with the same `paymentId` as the initial one and overwrite the storage of that transaction before retrieved state of `paymentId` will only correspond to the second transfer.

Consider modifying the logic of the contract to be able to handle multiple escrows for a specific `paymentId`. Alternatively, consider preventing the creation of an escrow that will overwrite an existing escrow.

Update: *Fixed in pull request #2797. The transfer now reverts if the `paymentId` has been used before. A regression unit test was added in pull request #3351.*

[C02] Any owner of the Multisig can execute unconfirmed transactions

The Celo project has a custom implementation of a MultiSig wallet. This contract gives allowance to a group of accounts, known as owners, to submit, confirm, and execute transactions to other smart contracts, or to itself. It also defines a maximum number of 50 owners.

For a transaction to be executed, there must be a minimum number of owners that have to confirm it after it is submitted. This minimum is defined by the `required` variable.

The contract provides a group of functions for adding new owners, removing and replacing already existent owners. When a new owner is added, it will be pushed to the `owners` array, and will be mapped in `isOwner` mapping to `true`. Some restrictions defined in these functions are:

- An already existent owner cannot be added twice
- An owner cannot be removed if they are not part of the multisig
- An existent owner cannot be replaced by another existent owner
- A non-existent owner cannot be replaced by another owner

The `isOwner` and `owners` variables can get out of sync when the last added owner is replaced. In this case it is possible to set the `newOwner` as `true` and the `owner` as `false` in the `isOwner` mapping, but not replace it in the `owners` array.

When this happens, the whole logic of the contract gets broken as these two variables are used throughout the contract interchangeably to check whether a specific account possess ownership over the wallet, and whether they have confirmed transactions to be executed. The Multisig wallet will:

- Not be able to fully remove an owner from the Multisig
- Change the number of required confirmations without enough confirmers
- Show an unconfirmed transaction as confirmed
- Show an unreal number of confirmations per transaction
- Show an incorrect list of owners
- Not be able to reach the `MAX_OWNER_COUNT` limit

Step-by-step proof-of-concept exploits for some of the scenarios listed above can be found in this private gist.

Consider reimplementing the `replaceOwner` function so it can handle the replacement of the last added owner, by removing the `-1` in the break condition of the `for loop`.

Update: *Fixed in pull request #2808. The `replaceOwner` function now handles the replacement of the last added owner. A regression unit test was added in pull request #3351. cLabs' statement for this issue:*

> *The team working on Celo chose to use the Gnosis multisig implementation, rather than build a new one, in order to use proven, audited code. But as mentioned on N18, there's no actual process to verify this on a regular basis, which will be addressed as stated in the N18 response.*

High

[H01] The `approver` role can prevent their own removal

The `setApprover` function allows Celo governance to set the approver for proposals. However, it has an `onlyOwner` modifier on it, meaning that the only way to access it is via a proposal. Such a proposal cannot be voted on if the proposal has not been approved by the approver. Thus, if the approver wishes, they can block all proposals with calls to `setApprover`, preventing themselves from being removed from power.

Consider implementing a special type of proposal for changing the approver. This type of proposal should not require approver approval, and should only be able to externally call `setApprover`.

> represents a set of individuals authorized to perform a "committer" like role. As per most open source projects, the approvers themselves self-govern and handle the addition and removal of new members. Since the approver role itself is a pointer to this contract, changing the approver role itself is more like removing all approvers, and as such the design intentionally requires approval on any proposal that does this.

[H02] Commissions can deceive members of a validator group

The `updateCommission` function of the `Validators` contract can be called at any time by the account under which a group is registered. This will update the fraction of rewards paid to the group owner versus the members of the group. Since this can be called at any time, it is possible for the group owner to call this immediately before rewards are distributed, setting the fraction paid to themselves to `1.0`, which pays the group owner the total rewards and pays validators in the group nothing. The validators and voters supporting this group may not realize a change like this has been made until it is too late.

Consider implementing a time-locking mechanism or a commission history, such that changes to the commission only take effect after the epoch in which they are initiated.

Update: *Fixed in pull request #2913. The cLabs team has implemented functions to queue commission updates and then apply them after a specified number of blocks.*

[H03] Elected validators of a group can be chosen by the validator group registrant

The `electValidatorSigners` function of the `Election` contract is responsible for the election of validators within validator groups. After calculating the amount of seats each validator group receives, it uses the `getTopGroupValidators` function defined in the `Validators` contract to get the elected members of the group. In this specific case, the `headN` function is internally used by `getTopGroupValidators` to get the first `numMembersElected[i]` elements of the `electionGroups[i]` validator group.

As a result, this return value will be saved in the `electedGroupValidators` variable, thus assigning the validators to the `electedValidators` mapping and establishing them as validators of a certain epoch.

`registerValidatorGroup` function, to benefit whichever validator they want by ordering them in the head of the `member` list.

Two general issues result from this:

- Users naively joining validator groups assuming they are secure or fair will be subject to the registrant decision on whether they will be able to be picked as validators, which could lower the user's earnings over time.
- A registrant or multiple colluding registrants can instantaneously reorder their group's validators to give those validators monetary rewards, and, with enough colluding validators, control over the protocol's consensus mechanism. This action would be publicly visible after the fact, so groups that have a reputation for doing this would likely lose voter support over time.

Consider modifying the `members` list's type to match the `AddressSortedLinkedList` type in which the validators in a group are assigned by a specific value, which could be for example, the amount of locked gold of the validators. Moreover, if the ability to reorder validators in a group is important for specific use cases, such as organizations that want to change the specific machines or keys under which they validate in the case of hardware or connectivity failure, consider modifying the codebase so that this freely reorderable functionality is specifically opt-in, and not the default configuration.

Update: *Not Fixed. cLabs' statement for this issue:*

> A single entity that controls 2/3rds of validator groups can indeed take over the network. Such an attack would require 2/3rds of the votes to achieve, and the primary defense against this attack is the high vote threshold required. One needs to assume that a compromised validator group compromises all of the members of that validator group. The remedy for a malicious validator group is voting, and the add/remove/reorder operations of a validator group are not intended to be resilient against a malicious registrant.
> The critical question here is, how does the mechanism of validator groups affect the overall security of the system? To explore this, let us consider a spectrum where the maximum validator group size (currently set to 5) is varied between the extremes of 1 and unlimited.

> to allow swapping out down nodes or doing system upgrades.
> At the other extreme, if there were no limits on validator group size, the system would become similar to parliamentary democracy. A single political party (validator group) could get a large amount of control, which would be less decentralized. However, larger validator groups have some advantages. A larger validator group is better able to do thorough vetting and security audits of validators than individual voters. It can also respond more quickly to outages or attacks. Changing significant numbers of votes is a slow process, and validator groups add an element of agility to the network. Voters might also find it easier to research a larger validator group than an individual validator. The security of the system is fundamentally derived from voting, so making voting easier increases security.
> It is open to debate what the correct limit should be. The election of 5 errs close to the direct election extreme while hopefully maintaining some of the benefits larger validator groups.

Medium

[M01] Expired attestations lock funds

The `Attestation` contract allows users to request an attestation that they own a specific telephone number. To do so, a user must call the `request` function, transferring funds from their ERC20 account to the `Attestation` contract.

For the flow to be completed, the user must call the `selectIssuers` function and, after receiving the attestation via SMS, call the `complete` function before the `attestationExpiryBlocks` amount of blocks have passed.

If the user was not able to complete this request, or if the issuers have any problem sending the attestation to the user's SMS within the time it took the network to produce the `attestationExpiryBlocks`, users' funds will be locked forever in the `Attestation` contract. The `validateAttestationCode` function, which is called by the `complete` function, will revert on line 514 once the attestation has expired.

Consider implementing specific functionality to let users withdraw their money in case the attestation could not be completed.

[illegible]

> The loss of the attestation fee in the event of an incomplete attestation is a necessary requirement for the security of the protocol. There are two negative consequences that would result from the ability for users to withdraw their funds:
> – Most importantly, it would allow malicious validators to spam other validators and cause them to send text messages for which they would not be able to be compensated through the completion of the attestation at virtually no cost to the attacker. Only without the fee being recoverable can the protocol incur a cost to the attacker.
> – It would reduce the cost for attackers to just request a lot of attestations, but only complete the ones that they can control validators for. For the remaining ones, they would just refund themselves with the attestation fee.
> Thus M01 is not a bug, but a deliberate protocol design decision.

[M02] Functions in StableToken do not emit Transfer event

Both `debitFrom` and `creditTo` functions of the `StableToken` contract work in a very similar way to the `mint` and `burn` functions of an `ERC20` token in the sense that when those are called by the `VM`, the total supply of the token will increase or decrease.

The ERC20 specification states that on the `transfer` function *“A token contract which creates new tokens should trigger a Transfer event with the `_from` address set to `0x0` when tokens are created”*, while the StableToken does not trigger such an event in the `creditTo` function.

Similarly, the ERC20 specification does not mention anything about events when burning tokens, but emitting a `Transfer` event with the `_to` address set to `0x0` has become a de-facto practice found in widely used ERC20 tokens, such as the OpenZeppelin’s ERC20 implementation.

When tokens are burned, the `StableToken`‘s `burn` and `debitFrom` functions are not triggering the `Transfer` event and thus the amount of burned tokens is not being properly logged.

Clients attempting to reconstruct the entire history of transferred tokens by parsing `Transfer` event logs should take this issue into account to avoid mismatches in expected token balances.

Update: *Fixed in pull request #2805. The `burn` function now emits the `Transfer` event. The `creditTo` function was renamed to `creditGasFees`. It now calls the `_creditGas` internal function which emits the `Transfer` event.*
The `debitFrom` function was renamed to `debitGasFees`. In pull request #3438 comments were added to explain why the `debitGasFees` and the `creditGasFees` functions only emit the `Transfer` events for the net gas payments.

[M03] Users can avoid some slashing penalties by front-running

The `slash` function of the `LockedGold` contract is called whenever any type of slashing occurs. It will decrement the account's non-voting locked gold balance, as well as the accounts' active and pending votes if needed. However, an attacker can take advantage of the protocol in specific circumstances, when slashed accounts have more than the locked gold requirement and the slashing `penalty` is greater than the locked gold requirement.

In order to avoid some slashing penalties, a user can simply front-run the call to `slash` with calls to the functions `revokePending` and `revokeActive` of the `Election` contract, and the `unlock` function of the `LockedGold` contract. These first two functions will turn pending or active votes into nonvoting account balance, while the `unlock` function will turn the nonvoting account balance into `PendingWithdrawal` objects.

Aside from locked gold requirements for validators, a user can `unlock` all of their other votes and account balances, turning them into `PendingWithdrawal`s. The `slash` function does not attempt to decrement from these, so a user can avoid some slashing penalties by doing so. If a user sees that they are being slashed before the slasher's transaction is mined, the user can construct a transaction that moves their locked gold to a `PendingWithdrawal` and send it with a higher gas cost, front-running the slashing transaction.

It should be noted that there is little incentive for a validator or validator group to have any amount of locked gold above the required amount, so if they plan to be malicious, they will likely rid their accounts of anything other than the minimum amount of locked gold ahead of time, rather than taking a chance on trying to front-run.

Consider adding functionality to `slash` that will also decrement from `pendingWithdrawals`.

> `accountTotalLockedGold(validator))` so an account will never be slashed more than an account's locked gold balance.
> The intent is for penalties to never exceed the `lockedGoldRequirement` for any account. For non-validators or validators/groups slashed via governance, they may still front-run their slashing by withdrawing. This is being tracked in celo-org/celo-monorepo#2887.

[M04] Undocumented assembly blocks

The `Proxy.sol` contract includes a couple of assembly blocks. In particular, the `fallback function` uses a large block of assembly for delegating calls from the `Proxy` contract to the implementation contract that lives in the `IMPLEMENTATION_SLOT`.

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.

While this does not pose a security risk per se, it is one of the most critical parts of the system, as it is in charge of the communication between the proxy and the implementation using the proxy's storage for preserving it between contract upgrades.

Note that the use of assembly discards several important safety features of Solidity, which may render the code less safe and more error-prone. Hence, consider implementing thorough tests which cover all potential use cases of these functions to ensure they behave as expected.

Update: *Fixed in pull request #2895. Inline comments have been added to the assembly blocks.*

[M05] Missing counter addition

In the `Attestations` contract, the `addIncompleteAttestations` function creates a `currentIndex` variable before entering a `while` condition.

If the random validator has already been selected as an issuer for a specific request, then the `attestation.status` will be different from `AttestationStatus.None` and the `while` loop will restart, attempting to find new random validators until the transaction either runs

The while loop condition will always be `true` if the `attestationsRequested` parameter of the `request` is bigger than the current amount of validators.

There is a specific attack vector that can trigger this behavior: An attacker uses the `Escrow` contract to generate a `transfer` for a victim with the `minAttestations` parameter's value higher than the current amount of validators. With the intention of generating the minimum amount of attestations to withdraw the money, the victim will make use of the `Attestations` contract to generate a `request` with the `attestationsRequested` parameter equal to the number of `minAttestations` used by the attacker. Finally, when the victim executes the `selectIssuers` function with the corresponding `identifier`, the transaction will fail with an `out of gas` error.

Consider picking validators, which is now done via the `seed` variable, from a list of unselected validators, which can only include validators that have not already provided an attestation. Additionally, consider including a check either in `transfer` or `selectIssuers` that requires that the number of attestations does not exceed the total number of validators. Alternatively, consider modifying the `currentIndex` variable, or some other variable inside the `if` condition in such a way that the `while` loop can eventually finish its execution. If the latter is chosen, consider the effects of successfully finishing execution of `addIncompleteAttestations`, such as the deletion of the relevant request afterwards.

Update: *Fixed in pull request #2794. Now there is a maximum number of attestations, there is a new check to require that the number of attestations does not exceed the number of issuers, and the list of issuers is reduced after every iteration.*

[M06] Lack of input validations

Across the repository, there have been multiple situations where the input has not been checked before using it in a function. In particular:

- The `_setOwner` function in the `Proxy` contract does not check that the new owner of the proxy is not the zero address. This would result in the permanent loss of control over the proxy ownership if the zero address is passed by mistake. Also note that some Ethereum clients may default to sending null parameters if none are specified.

- The `increaseAllowance`, `decreaseAllowance` and `approve` functions in the `StableToken` and the `GoldToken` contracts do not check that the spender is not the zero address, and the `transferFrom` function in both contracts does not check that the zero address, which in Celo represents the VM, will not transfer tokens on other's behalf.
- The `Governance` contract is in charge of important parameters of the Celo blockchain. In particular, the `minDeposit` variable holds the value of the minimum amount of Gold to submit a proposal. Although in the `initialize` function the `minDeposit` variable is checked not to be 0, the same condition is not checked in the `setMinDeposit` function. If the `minDeposit` is set to `0` by the administrator, it could lead to an insecure blockchain state where an attacker can send a lot of proposals to the network without having to spend Celo Gold.

Consider requiring that these addresses are not the zero address, restricting integer values when appropriate, and consider adding more checks along the code for proper input validation.

***Update:** Fixed in pull request #2807. The cLabs team has implemented input validation for functions `increaseAllowance()` and `approve()`, which together prevent setting an allowance greater than 0 for `address(0)`. `decreaseAllowance()` and `transferFrom()` can then no longer be used by `address(0)` to spend any amount. The cLabs team has also implemented the recommended fixes for `Proxy.sol` and `Governance.sol`. They have added a comment in `setWalletAddress` explaining what it means for a wallet address to be set to 0.*

[M07] Elements with equal value within a Sorted List can be arbitrary ordered

When using the `SortedLinkedList` library, the `update` function makes use of the `remove` and the `insert` function. After some validations , the `insert` function uses the `isValueBetween` function, which will return true if the `lesserKey` is 0 or if the value associated with `lesserKey` is less or equal than `value` or if the `greaterKey` equals 0 or the value associated with `greaterKey` is greater than or equal to `value`

This will allow any `update` operation to move a specific item in the list to the spot directly before or after an item if the `value` is the same for both.

type. The first one saves the proposals in the order they will be dequeued and the second one holds the eligible validator groups ordered by number of votes, establishing the winners of an election. By using the `upvote` or the `revokeUpvote` functions in the first case, or using the `vote`, `revokePending` or `revokeActive` functions in the second case, a participant of the Celo protocol can order themselves at any position as a selected proposal or validator by matching the amount of votes of the selected element in the list and by using specially crafted `lesser` and `greater` parameters.

Consider reviewing if this is the intended case of the library and documenting it, explaining the risk and attack vectors that may arise from this behavior.

Update: *Not Fixed. cLabs' statement for this issue:*

> There are four such examples of sorted lists:
> 1. The list of election eligible validator groups, ordered by vote total. Here, ties are very unlikely, as votes are denominated in wei. Furthermore, ties would only have an effect if the groups that are tied spanned the boundary defined by maxElectableValidators. Otherwise, the ordering of groups that are tied in the number of votes does not affect election results, even if those groups happen to elect validators.
> 2. The list of governance proposals, ordered by upvote total. Here, ties are very unlikely, as votes are denominated in wei. Furthermore, ties would only have an effect if the tied proposals spanned the boundary defined by concurrentProposals. Even if this was the case, the ramifications would be minor, as the tied proposal(s) that didn't get dequeued would become the proposal(s) with the highest number of upvotes, making it extremely likely the proposal would get dequeued next.
> 3. The list of oracle reports for a given token, ordered by price. Here, the unstable sorting doesn't matter, as the same median value will be returned regardless of how ties are ordered.
> 4. The list of oracle reports for a given token, ordered by timestamp. Here, the unstable sorting doesn't matter, as the same median value will be returned regardless of how ties are ordered.

[M08] The reveal process can get stuck for a proposer

mapping, and at the beginning of a new block, this `commitment` is revealed.

The `revealAndCommit` function is in charge of this process. It is only callable by the VM, and redirects the functionality to the internal `_revealAndCommit` function. There, it checks if the proposer had submitted a commitment in the past, and if that condition is `true`, it will check if the revealed randomness was zero or not. If the revealed randomness is zero, the transaction will revert.

The problem might occur if the function call from the VM passes a `commitment` derived from a zero randomness as a parameter. As the hash of `bytes32(0)` is not zero, if in the first `revealAndCommit` call the `newCommitment` parameter was `0x290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563` (the `bytes32(0)` hash), then in the following call, when the randomness is revealed, the requirement in L70 will revert although the stored commitment corresponds to the input randomness.

These calls are made by the node client twice: when the client checks that the randomness is correct and when the proposer reveals the randomness. In both cases, when the transaction reverts, both the `process` and `commitNewWork` functions from the blockchain client will return an error.

Update: *Fixed in pull request #2795. Now a commitment of 0 randomness will revert.*

[M09] Contracts storage layout can be corrupted on upgradeable contracts

The Celo protocol has an unstructured storage proxy pattern system implemented, very similar to the one proposed in the OpenZeppelin SDK.

This upgradeability system consists of a Proxy contract which users interact with directly and that is in charge of forwarding transactions to and from a second contract. This second contract contains the logic, commonly known as the implementation contract. This approach has two main considerations to be aware of:

for the logic contract. This is well managed by Celo, by reserving a pseudo-random slot for both the logic contracts' implementation and owner address. This way, they are guaranteed to not clash with state variables allocated by the compiler, since they depend on the hash of a string that does not start with a storage index.

- Storage collisions between different implementation versions: There can be storage collisions between different versions of the same implementation contract. Some possible scenarios are:
- When changing the variables order in the contract
- When removing the non-latest variable defined in the contract
- When changing the type of a variable
- When introducing a new variable before any existing one
- In some cases, when adding a new field to a struct in the contract

The unstructured storage proxy mechanism does not safeguard against this situation by itself, and there are no validations being done by the script that is in charge of upgrading contracts , thus there is no certainty that storage layout will remain safe after an upgrade. Violating any of these storage layout restrictions will cause the upgraded version of the contract to have its storage values mixed up, and can lead to critical errors in Celo protocol's contracts.

Consider checking whether there were changes in the storage layout before upgrading a contract by saving the storage layout of the implementation contract's previous version and comparing it with the storage layout of the new one.

In addition to this, consider using the `OpenZeppelin SDK`, which already covers these scenarios among others, such as multilevel inheritance using a more complete version of the `Initializable contract` and the transparent proxy pattern.

***Update:** Not Fixed. cLabs' statement for this issue:*

> As stated in the report, there's a risk of performing not compatible upgrades that use a different storage layout. The actual upgrade script is a development one; on mainnet upgrade will be performed by governance proposal; and when this happens we'll make sure there are programmatic tools for verifying contract upgrades compatibility. The TransparentProxy pattern is

> performing upgrade, and also makes other calls to the underlying implementation.

[M10] Not using OpenZeppelin contracts

OpenZeppelin maintains a library of standard, audited, community-reviewed, and battle-tested smart contracts.
Instead of always importing these contracts, the Celo project reimplements them in some cases, while in other cases it just copies portions of them.
This increases the amount of code that the cLabs team will have to maintain and misses all the improvements and bug fixes that the OpenZeppelin team is constantly implementing with the help of the community.
In particular, the following contracts and libraries are being reimplemented or copied:

- In the `AddressHelper` library, an old version of the `isContract` function is used. The most recent version of this function in the OpenZeppelin contracts project uses `extcodehash` instead of `extcodesize` to give better guarantees and work with more cases.
- `Signatures` library can be replaced with the OpenZeppelin `ECDSA` library.
- `SafeCast` library can be replaced with the OpenZeppelin `SafeCast` contract which was included in the `v2.5.0` release.
- `StableToken` contract and `GoldToken` contract can inherit from the `IERC20` interface and the `ERC20Detailed` contract among others where appropriate.

Consider importing the OpenZeppelin contracts instead of reimplementing or copying them, and consider updating the library to its latest version.

Update: *Partially fixed in pull request #2877. Now Celo is using version 2.5.0 of OpenZeppelin instead of copying the contracts to their repository. Except for the `ReentrancyGuard` contract. This `ReentrancyGuard` version was copied into the Celo repository because the most recent version of `ReentrancyGuard` in OpenZeppelin requires its constructor to be called to work correctly. This does not happen when it is used through a proxy. However, note that the `ReentrancyGuard` version now in use by Celo worked with proxies just by coincidence, because the `uint256 _guardCounter` variable would work even if it is not initialized. The*

[M11] Users are not able to verify releases

Celo does not publish the checksums of its releases. Moreover, while some releases are signed (for example, `alfajores-v0.3`), some of them are left unsigned (for example, `wallet-v1.4.2`).

Publishing the checksum allows users to verify the integrity of the downloaded files, which may have been modified by attackers.
On the other hand, signing the releases allows users to verify that they come from a trusted source. It is an important security measure in case the GitHub account of one of the maintainers gets compromised.

Even though we understand that the smart contracts in the scope of the engagement will not be executed in a user's context, we decided to include this issue in the report as the `celo-monorepo` does not only contain the smart contracts, but also other applications, such as mobile clients.

Consider signing all the releases and publishing their corresponding SHA256 checksum. The article Managing commit signature verification explains more details about signatures in GitHub.

Update: *Not fixed. cLabs' statement for this issue:*

> All production releases of the project will include release notes detailed on GitHub. For each downloadable binary (currently only Docker images), a checksum and a signature generated via a securely custodied offline GPG key will be provided.

[M12] There is no security policy

The `celo-monorepo` project has no security policy.

The security policy includes instructions for the responsible disclosure of any security vulnerabilities found in the project.

Consider adding a method for secure and encrypted communication with the team, like an email address with its GPG key.

Update: *Fixed in pull request #2906. The security policy document has been added.*

[M13] Missing test coverage report

There is no automated test coverage report for the `protocol` package. Without this report it is impossible to know whether there are parts of the code never executed by the automated tests; so 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 report and making it reach at least 95% of the source code.

Update: *Not Fixed. cLabs' statement for this issue:*

> The `celo-monorepo` configuration includes a coverage that runs once a day. It's not included on every PR/commit since for the moment it takes ~3 hours to run all tests with coverage enabled. This will be revisited before mainnet so as to improve the coverage tooling.

[M14] The relations between loops and their boundaries are not clear

There are several occurrences in the codebase where `for` loops are used to iterate an entire array.

This is a widely known problem in the Ethereum ecosystem, as it is common that loops in Smart Contracts result in an `out of gas` error when the number of iterations that the loop needs to execute is too big.

Some examples of this behavior are:

- In the `Accounts` contract: L353
- In the `Attestations` contract: L133, L358, L360, L420, L430, L543
- In the `DowntimeSlasher` contract: L82
- In the `Election` contract: L224, L397, L848, L854, L878, L902, L944
- In the `Governance` contract: L917, L1026

- In the [illegible] contract: [illegible]
- In the `StableToken` contract: L124
- In the `Validators` contract: L341, L380, L804, L912, L929, L974

Also, the family of LinkedLists libraries makes heavy use of for loops for iterating the lists, as seen in the `popN`, `headN`, `getElements` and `numElementsGreaterThan` functions. This could also generate problems in other parts of the protocol that are using these functions under the hood.

Some examples of this behavior are:

- In the `Election` contract: L805, L822, L826
- In the `Governance` contract: L879, L965
- In the `SortedOracles` contract: L201, L232
- In the `Validators` contract: L910

Some examples of the implications arising from this issue are:

- In the `Governance` contract, if the `concurrentProposals` is set as a big enough number, the `propose`, `execute`, `upvote`, `revokeUpvote`, `approve` and `vote` functions will be unusable.
- In the `Proposals` and `Governance` contracts, a proposal can be made with a transaction array which is determined by user input and it is not being validated in the Smart Contract layer. Later on, when these transactions are being validated and gas consumptions of the functions called are higher, the proposal may not be able to execute.
- In the `Elections` contract, it could result in the impossibility of further voting for a user if the amount of groups it has voted on is too big.
- In the `Reserve` contract, `getOrComputeTobinTax` can run out of gas if the `_tokens` array is big enough, as its length is used as the break condition in a for loop.
- In the `DowntimeSlasher` contract, if the `slashableDowntime` variable is big enough, the `isDown` function will be rendered unusable and it will not be able to slash byzantine validators.

that these properties are invariant. Finally, consider thoroughly reviewing the usage of loops in the protocol and closely analyzing the effects of failures that may arise from the lack of limits in these loops.

Update: *Not Fixed. Issue #3420 was created to keep track of this. cLabs' statement for this issue:*

> All of the listed "unbounded" loops are either bounded by input arguments, are external view calls intended to be called via RPC, or are bounded governable parameters.
> In the first case, the caller is responsible for setting the arguments appropriately. For example, the "unbounded" loop in batchGetMetadataURL is bounded by the length of the input array. Clients that want to query metadata URLs (e.g. the Celo Wallet mobile application) can reduce the size of the batches being fetched if larger batches consume too much gas.
> Examples:
> * In the `Accounts` contract: L353
> * In the `Attestations` contract: L133, L358, L360, L543
> * In the `Governance` contract: L1026 (transaction array length set by proposer)
> * In the `Proposals` library: L82, L121 ((transaction array length set by proposer)
> * In the `Reserve` contract: L116
> * In the `SortedOracles` contract: L112
> * In the `StableToken` contract: L124
> * In the `Validators` contract L380, L912, L929
> In the second case, the function exposes smart contract storage state via a view function. All of these functions were written to be called via RPC by a trusted client, in which gas provided can be made to exceed the block gas limit.
> Examples:
> * In the `Accounts` contract: L353
> * In the `Attestations` contract: L420, L430
> * In the `Election` contract: L805
> * In the `Governance` contract: L879
> * In the `SortedOracles` contract: L201, L232
> * In the `Validators` contract: L341, L380, L804, L910, L912, L974, L929
> In the third case, the length of the loop is bounded by a storage variable set via on-chain governance. Should the gas consumed by the loop grow too large, it can be lowered via on-chain

> * In the Election contract: L224, L397, L848, L854, L878, L902, L944
>
> * In the Governance contract: L917
>
> * In the Reserve contract: L116, L275, L294
>
> * In the Validators contract: L341, L380 (where needed, limited by the max group size), L804, L910 (where needed, limited by the max group size), L912 (where needed, limited by the max group size), L929 (where needed, limited by electableValidators.max)
>
> * In the Election contract: L822, L826
>
> * In the Governance contract: L965

Low

[L01] `Abi.encodePacked` function receives dynamic-sized parameters

The `checkProofOfPossession` function of the `UsingPrecompiles` contract uses both dynamic-sized variables `blsKey` and `blsPop` as the parameters of the `encodePacked` function.

As stated in the Solidity documentation webpage it is easy to craft collisions if more than one parameter of the `encodePacked` function is dynamically-sized.

Although no attack vector has been identified from this behavior, consider using `abi.encode` instead.

***Update:** Not fixed. cLabs' statement for this issue:*

> `Abi.encodePacked` is used here to concatenate bytes before they are passed on to a precompile. The length of these arrays are checked in `Validator._updateBlsPublicKey`. Perhaps it would be helpful to check that here or change the parameter types. On the other hand, it's possible to call the precompile using any arguments without using this method.

[L02] Inconsistent quorum calculation

In the `byzantineQuorumValidatorsInCurrentSet` function of the `Governance` contract, the quorum is calculated by doubling the total amount of validators, then dividing the obtained value by 3 and finally adding 1. On the other hand, in the `minQuorumSize` function of the `DoubleSigningSlasher` contract, the quorum is calculated by doubling the value of the total amount of validators, then adding 2 and dividing that value by 3. The `return` values of these functions differ when `numberValidatorsInSet` (for the `minQuorumSize` function) and `numberValidatorsInCurrentSet` (for the `byzantineQuorumValidatorsInCurrentSet` function) are equal to any integer multiple of 3. For example, with 21 validators, `byzantineQuorumValidatorsInCurrentSet` will return `15`, while `minQuorumSize` will return `14`.

Also worthy of note is that `byzantineQuorumValidatorsInCurrentSet` uses `SafeMath` functions while `minQuorumSize` does not.

Consider being consistent in the calculation of the quorum and thoroughly comment in the source code an explanation regarding how the quorum is being calculated. Consider also using `SafeMath` operations in `minQuorumSize` for better code consistency and security.

Update: *Fixed in pull request #2796. The cLabs team has implemented a single quorum calculation.*

[L03] Missing revert messages in require statements

There are several `require` statements in the project without revert messages. This behavior results in an increased difficulty of debugging the system.

The list of affected `require` statements is:

- In the `Freezable` contract: L9, L16
- In the `FixidityLib` library: L122, L151-L153, L192, L202, L231, L236, L240, L243, L248, L267, L280, L282
- In the `FractionUtil` library: L67, L102, L113
- In the `Initializable` contract: L7
- In the `UsingPrecompiles` contract: L62, L93, L111, L123, L136, L168, L181, L194, L209

cases in `Freezable.sol`, `Initializable.sol`, `UsingPrecompiles.sol`, and `FixidityLib.sol`. `FractionUtil.sol` has been deleted from the repository.

[L04] Not using the registry EIP

The Ethereum Improvement Proposal (EIP) 1820 defines a standard registry.

Instead of using it, the cLabs team implemented its own simplified version of the Registry.

Deviating from the community accepted standards limits the opportunities to interoperate with other systems and to use shared implementations that are collectively maintained and already tested in mainnet by other projects.

Consider using the `OpenZeppelin` implementation of the EIP 1820 registry.

If there are important reasons to not use the EIP 1820 registry, consider documenting them.

***Update:** Not fixed. cLabs' statement for this issue:*

> EIP 1820 serves a different purpose than the Celo Registry.
> The Celo Registry is a central "address book" controlled by Governance that 1-1 maps component names to their implementations.
> EIP 1820 allows the owner of any address to declare that that address implements a certain interface. There is only a "check if address A implements interface X" functionality.
> There is no "lookup contract that is the implementation of X" functionality, which is what the Celo Registry implements.

[L05] Incumbency Bias possible in validator elections

The Celo protocol determines which validators are elected each epoch via a voting scheme. Validator seats are given to the top-voted groups every election. Rewards are distributed later on to every voter who voted for a winning validator group. Rewards are notably not given to voters for losing validator groups. The concept of incumbency bias refers to users choosing to only vote for the top groups, and never for the lower-ranked groups, in order to ensure they receive rewards. This is exacerbated by the existence of transaction fees, making voting for a losing group a costly

This vulnerability is listed as low severity because the cLabs team is aware of it and it is a necessary design feature to enable rewards based on validator performance.

This issue is difficult to solve with a single simple fix, as solutions could create even more issues. Although we cannot make a firm recommendation as a full analysis of incentives is not within the scope of this audit, consider the potential solutions of implementing losing-group base rewards, or a vote change pledge contract as discussed in OpenZeppelin/Celo communications. Note that while incumbency bias currently does tend slightly towards the centralization of the protocol, it does not necessarily break the protocol or go against its intended functionality.

Update: *Not Fixed. cLabs' statement for this issue:*

> This is a known issue, which has two potential mitigants. First, a vote change pledge contract can be built that allows users to commit their votes to a group provided that group meets a minimum threshold of commitments. Second, in the early stages of the network larger groups of gold holders (of more than 1% of staked gold) can play a catalyst role by using their voting power to lift promising new validators groups past the initial threshold to help them establish a track record to attract other voters.

[L06] Lack of validation may produce unexpected state in the distribution of epoch rewards

The `_distributeEpochPaymentsFromSigner` function of the `Validators` contract is used to distribute the rewards for the last epoch to a specific signer.

Any validator will be assigned to the group on `address(0)` when it is registered or when it is deregistered from its current validator group.

However, when the epoch rewards are distributed the case where the `group` variable is assigned to `address(0)` is not handled.

Consider strictly checking with a `require` or `assert` statement if a validator has been part of a proper validator group during the epoch reward distribution.

[L07] The minimum gas price calculation differs from documentation

The documentation defines the gas price minimum as `gas_price_minimum' = gas_price_minimum * (1 + ((total_gas_used / block_gas_limit) − target_density) * adjustment_speed)`.

However, in the code the gas price minimum is defined as `oldGasPriceMinimum * (1 + (adjustmentSpeed * (blockDensity - targetDensity))) + 1`.

It is not clear what the last unit added is nor why it is being used.

Consider keeping your documentation updated to your current codebase and clearly stating how the gas price minimum is defined. For instance, consider adding the `+1` that exists in the code to the documentation version. Having accurate documentation will improve developer experience and project auditability.

***Update:** Fixed in pull request #2889. The documentation has been updated.*

[L08] Unused events

Line 120 of `Validators.sol` declares a `ValidatorEpochPaymentSet` event.

As it is never emitted, consider removing the declaration or emitting the event appropriately.

***Update:** Fixed in pull request #2811. The cLabs team has removed the unused event.*

[L09] Lack of indexed parameters in events

Throughout the codebase, there are events defined without any indexed parameters. Some examples are:

- Lines `44-52` in the `Reserve` contract.
- Lines `121-123` in the `Validators` contract.
- Line `43` in the `SortedOracles` contract.
- Line `104-106` in the `Election` contract.
- Line `20` in the `BlockchainParameters` contract.

Update: *Fixed in pull requests #2907 and #3125.*

Now the events have indexed parameters, except for the ones that just emit integer values.

[L10] Inconsistent validation of variables

The setter functions for important variables of the system are rigorously validated such as in the `Validators` contract , and not validated at all in some others such as in the `SortedOracles` contract.

Consider defining a consistent strategy for validation across the whole protocol, rather than validating only some important variables within the smart contract layer.

Update: *Fixed in pull request #2813. Validations are now consistent. They have been moved to internal helpers and to setter functions.*

[L11] Voting cap can be manipulated

The `canReceiveVotes` function of the `Election` contract determines whether more votes can be cast for a specific group or not. The result will be `true`, meaning the group can receive votes, if the proportion of votes they will have after the votes are cast is less than or equal to `(numGroupMembers + 1) / min(maxElectableValidators, numRegisteredValidators) * getTotalLockedGold`.

Since this calculation is based on the total locked Celo Gold, it is plausible that an attack could increase the locked gold tremendously, thus also significantly impacting the maximum number of votes that a group can have. One group increasing the voting cap by locking gold, then casting as many votes as possible for their group, could use this to win an election immediately before it ends. When there are many groups at the same voting cap, increasing the cap dramatically and then voting for your group could give you an advantage, with other groups having little time to do anything about it.

This is a vulnerability of low severity because the cost to perform it (amount of locked Celo Gold) has to be high. Additionally, even if the voting cap is changed, for other groups to get back to the voting cap (if they were at it before it was changed) it would only take a small fraction of the deposited attack amount. Finally, the voting cap is a mechanism intended to increase fairness in

Consider making it clear to voters how the voting cap is determined, so that groups are not surprised by voting caps changing, especially near the end of elections.

Update: *Not Fixed. cLabs' statement for this issue:*

> This is possible, though there is little benefit for increasing a validator group's votes past that vote cap because a group's number of validator slots is also capped. Those votes past the cap would be wasted votes in terms of gaining more control of the system. The cost is also high because locking gold would distribute cap increases uniformly to all validator groups.So the impacts are low and the costs are high here. Making vote caps clearer will be a better experience for voters. One way to fix this would be to base vote caps on the total locked gold at the end of the prior election, rather than the current time. This will make last minute changes in the vote cap impossible and provide a better voting experience.

[L12] `increaseSupply` in GoldToken does not emit Transfer event

When the total supply of Celo Gold tokens increases, the `increaseSupply` function is called to update the `totalSupply_` to keep it synchronized with the total amount of gold in the system. This function is not triggering the `Transfer` event, and thus the increase in the quantity of tokens is not being logged.

Clients attempting to reconstruct the entire history of transferred gold tokens by parsing `Transfer` event logs should take this issue into account to avoid mismatches in expected token balances. Additionally, emitting a `Transfer` event upon token creation is encouraged by the ERC20 spec.

Consider modifying the `increaseSupply` function so it receives the `to` address, and consider and emitting the `Transfer` event from `address(0)` to that address.

Update: *Not Fixed. cLabs' statement for this issue:*

> Because native transfers are not tracked through events, only those made through the ERC20 contracts, it should not be expected that a user could track all gold in existence by replaying event logs.

When operations with fractions are done, the result `Fraction` *might* have a greatest common divisor (GCD) greater than one. In this scenario, that `Fraction` can be reduced. This is called after finishing operations such as in the `add`, `sub`, and `mul` methods.

However, the `div` method does not reduce the `Fraction` once the operation ends, creating the possibility of having denominators and numerators that have a greater than one GCD.

Consider reducing the result of the `div` method after the operation is completed.

***Update**: Fixed in pull request #2890. The `FractionUtil` contract was removed. Now the only function that used it returns a tuple of (numerator, denominator).*

[L14] Account's name field may allow phishing scenarios

The `Accounts` contract allows any user to set a `name` for their account with the `setName` function. This function does not limit in any way the name an account can set, which could result in a lot of accounts using the name of famous people to deceive naive users.

Clients from the Celo network should not use this name as an unique identifier of the account or as a trusted piece of information. Moreover, this feature could be misused by attackers if there is no proper documentation on the risks this feature introduces to the network, such as phishing scenarios which are so common in the Blockchain space that there are published scientific papers on how to recognize addresses associated with phishing.

Consider analyzing and publicly documenting the risks associated with the ability to set a name for the accounts, not only for developers generating clients but also for end-users of the Celo network.

***Update:** Not Fixed. cLabs' statement for this issue:*

> It is correct that an account's name is fundamentally impossible to verify. Therefore, the name should be considered untrusted by users and developers. This will be clarified explicitly in the documentation and consider renaming the field to `untrustedName`. In anticipation of this problem, verifiable identity attributes have been added to the Metadata feature (https://docs.celo.org/celo-codebase/protocol/identity/metadata). With this, users can verify

[L15] Failure of `ecrecover` function does not revert

The `ecrecover` function is called within the `getSignerOfMessageHash` function, which is in turn called by the `getSignerOfAddress` function. The `ecrecover` function returns the signer of a message, given the correct parameters `v`, `r`, and `s`. In the event of an invalid combination of parameters, the `ecrecover` function will return `address(0)`. If misapplied, the results of `getSignerOfAddress` and `getSignerOfMessageHash` could make it appear that a valid signature of the `address(0)` was produced.

Currently, if a user calls the `transfer` function of the `Escrow` contract to lock some payment with `paymentId` set to `address(0)`, any user can then call `withdraw` with `paymentId` set to `address(0)` and an invalid signature, and they will receive that escrowed payment.

If not used carefully, `ecrecover` could potentially give users some appearance of controlling `address(0)`. Considering that the VM is assigned to `address(0)`, a signature that appears to come from this address could have significant consequences. On the other hand, we should be able to safely assume that the private key for `address(0)` is never knowable, and therefore a valid signature from it is never producible.

Consider adding a `require` statement that ensures that the result of `ecrecover != 0`, so that in the future, any uses of it do not mistakenly give the impression that a valid signature of `address(0)` was produced.

***Update:** Fixed in pull request #2892. `ecrecover` now reverts if the signer address is 0.*

[L16] Not using the SafeMath library

Several arithmetic operations in the codebase are not using the `SafeMath` library that would prevent arithmetic overflow and underflow issues.

As an example, we detailed some of the occurrences below:

- In the `AddressLinkedList` contract: L80
- In the `AddressSortedLinkedList` contract: L95, L115, L136
- In the `AddressSortedLinkedListWithMedian` contract: L153

- In the [illegible] contract: [illegible]
- In the `Election` contract: L902
- In the `Exchange` contract: L161, L182, L289
- In the `Governance` contract: L917
- In the `IntegerSortedList` contract: L72, L108
- In the `LinkedList` contract: L146
- In the `LockedGold` contract: L238
- In the `MultiSig` contract: L113, L142, L147, L162, L279, L280, L305, L316, L317, L325, L327, L347, L350, L353, L373, L376
- In the `Random` contract: L101, L102, L103, L106, L109
- In the `Reserve` contract: L116, L162, L202, L275, L294
- In the `SortedLinkedList` contract: L94
- In the `SortedLinkedListWithMedian` contract: L144
- In the `SortedOracles` contract: L112
- In the `UsingRegistry` contract: L65

While this issue does not pose a security risk, and it is very unlikely that these operations could cause undesired outcomes as no exploitable overflows or underflows were detected in the current implementation, this may not hold true in future changes to the codebase.

Consider using the `SafeMath` library in these and all other places where an arithmetic operation is involved.

Update: *Fixed in pull request #2893. The cLabs team is now using the `SafeMath` library extensively.*

[L17] Excessive indirection

`LinkedList`, `SortedLinkedList`, and `SortedLinkedListWithMedian` are part of a group of libraries for managing double linked lists throughout the project. For instance, they are being used for saving oracles' reports in the `SortedOracles` contract to keep them in order and therefore, be able to calculate the rates' median.

Every time a `SortedLinkedListWithMedian` variable is defined, the list structure within it has to wrap a `SortedLinkedList` for saving and managing `integer values` and thus

Therefore, to access a particular element of the list from `SortedLinkedListWithMedian`, going through the `SortedLinkedList` and `LinkedList` is imperative, and this leads to confusing blocks of code such as in line 142 to pop an element, or line 242 to update the median.

While this does not pose a security risk per se, it introduces a lot of complexity to an important section of the code, is error prone and difficult to maintain in the long term.

Consider redesigning these libraries to remove excessive indirection, and consider documenting how these libraries work and what their purposes and responsibilities are.

[L18] Lack of checks for denominator

The `FractionUtil` library adds the functionality to use `Fractions`.

These Fractions consist of two numbers, an integer numerator and an integer denominator, where the denominator should not be zero or otherwise it could create a division by zero, which may cause a `VM error: invalid opcode` error during execution.

Nevertheless, the functions inside the library do not have checks for this issue and the output may not be correct in almost all functions. For instance, the `reduce` function would return the same Fraction when reducing multiples of `(1,0)`, the `equals` function would return `true` when comparing `(0,1) == (0,0)`, and the `div` function would return `(y.denominator,0)` for the pair `(1,0)` and `(y.numerator, y.denominator)`.

This library is only used in the `getOracleExchangeRate` function from the `Exchange` contract. There, a new `Fraction` variable is created with the output from the `medianRate` function of the `SortedOracles`. In particular, the `rateDenominator` could be zero if the `numRates(token) == 0` in the `SortedOracle` contract and no further checks will be done for this variable causing a division by zero in the `getOracleExchangeRate` function of the `Exchange` contract . If this `Fraction` is then used in any of the previously mentioned methods from the `FractionUtil` library, then the actual result will not be the expected value.

Consider adding checks to eliminate the possibility of haveing zero as denominator in all the places where the methods from the `FractionUtil` library are used, or adding those checks in the library methods themselves.

[L19] Signers cannot change association once set

In the `Accounts` contract, it is possible to cause a mismatch between the `authorizedBy` and `accounts[].signer.validator` data structures.

If an account `A` calls `authorizeValidatorSigner()` successfully with a signature from address `B`, and then again with a signature from address `C`, address `B` will no longer be able to be a signer for any account. The second call to `authorizeValidatorSigner()` effectively removes `B` as a signer and replaces them with `C`.

Now, if account `B` attempts many actions in the code where a call checking `validatorSignerToAccount(B)` is made, execution will revert in the require on L242 of the `Accounts` contract, since `authorizedBy[B] == A` but `accounts[A].signers.validator != B`. There is no way for a fourth account, `D`, to authorize `B` as their signer, as `authorizeValidatorSigner(B, vb, rb, sb)` will fail upon reaching the call to `isNotAuthorizedSigner` called from line 436.

The most important thing to note here is that once a signer has been authorized, that address can never be authorized as a signer for any account again. Additionally, every account can have only 1 authorized validator signer.

Consider adding more information about signers and accounts to the documentation, including a specific mention of these properties, so that users are not confused.

Update: *Fixed in pull request #2959. The cLabs team added documentation to explain that a deauthorized key cannot be reauthorized.*

[L20] TODO in reward calculations can cause reverts

In line 244 of `EpochRewards.sol` there is a TODO comment along with a `require` that will always throw. The result of this is that any calls to `getTargetGoldTotalSupply`, or functions that call it, like `getRewardsMultiplier` and `calculateTargetEpochPaymentAndRewards`, will cause a revert after

Consider implementing the intended calculation and removing the `TODO` comment.

***Update:** Not fixed. cLabs' statement for this issue:*

> The EpochRewards contract is upgradeable via on-chain governance. So long as a fix is deployed within the next 15 years, this should not be an issue.

[L21] Lack of `assert` statement may produce unexpected slash

In line 78 of `SlasherUtil.sol`, execution will `return` prematurely if the group of the validator being slashed has `address(0)`. This should be impossible, and represents a user who was not a member of any group at the time in question. The comment on the same line corroborates this by stating this `Should never be true`.

Consider changing this line to `assert(group != address(0))`. This way, unexpected errors will revert, rather than making state changes to the code, and any errors in this regard will not result in slashing a potentially innocent victim.

***Update:** Fixed in pull request #2891. The statement was changed to an `assert`.*

[L22] Not following the fail-loudly principle

The `executeTransaction` function from the `MultiSig` contract allows an owner to try to execute the desired transaction.

The first problem happens when the transaction does not have enough confirmations, in which the case call to `executeTransaction` will succeed but it will not emit any event nor change anything in storage.

The second problem comes when there are enough confirmations but there is an issue in the use of the `external_call` function. Because the returned value is inside an `if` conditional statement instead of a require or a revert statement, in case of an unsuccessful external call, the main transaction will succeed and an `ExecutionFailure` event will be emitted.

For instance, because the `external_call` function does not have a balance check, if the transaction being executed by `external_call` was supposed to send a value greater than the

A problem in the entire execution of a transaction should be as loud as possible by ending the transaction unsuccessfully, reverting it, and marking why it failed by the usage of a revert message.

To address the first problem, consider changing the `if` statement on line 231 to a `require` statement, including an error message indicating that the transaction had too few confirmations. For the second problem, consider changing the `if` statement on line 234 to a `require` statement to fail loudly in case something wrong happens with the external call, instead of emitting a failure event and continuing the execution.

***Update:** Fixed in pull request #2894. The `executeTransaction` function now reverts if the `transactionId` is not confirmed.*

[L23] There is no stable release cadence

The last release of the `celo-monorepo` project was on October, 2019.
Before that, there have been from 1 to 3 releases per month, without any stable cadence.

Predictability is a very good feature for a software project.
It makes it possible to tell your users when to expect the features or fixes they are waiting for, it removes a lot of the uncertainty that can become problematic for maintainers, and over time it will allow to make estimates about the velocity of the team.

Consider setting a hard date for releases, once or twice per month, on the same days every month.

***Update:** Not fixed. Refer to the cLabs statement for **[M11] Users are not able to verify releases**.*

[L24] Missing changelog

The `celo-monorepo` project has no changelog.

Keeping a comprehensive chronological log of the important changes of the project helps maintainers, contributors, and users to keep track of the progress of the project and to take action when breaking changes, important bug fixes, or new features are implemented.

changelog entry for further details. As an example, see the changelog of the `openzeppelin-contracts` and its releases.

Update: *Not fixed. Refer to the cLabs statement for* ***[M11] Users are not able to verify releases****.*

[L25] There are many stale branches

There are 21 stale branches in the `celo-monorepo` repository.

When there are many open branches in the team repository, it is not clear which contain work in progress, which ones are experimental and which ones were discarded.

Consider creating the branches in the personal fork of the contributor instead of in the team repository, and deleting all the branches that were discarded or have already rot.

Consider splitting the work as much as possible so smaller branches can be landed into the development repository instead of remaining open for a long time. For work that requires a branch to be open for a long time and involves more than one contributor, consider documenting it on an issue or in the project board.

Update: *Not Fixed. cLabs' statement regarding this issue:*

> This will be cleaned up prior mainnet.

Notes

[N01] Unnecessary variable assignment

The `Random` contract defines and assigns the `randomnessBlockRetentionWindow` variable which will be used to set the amount of past randomness values to keep in the contract's storage.

Since this variable is already set in the `initialize` function, the assignment in L18 is unnecessary.

Consider removing the unnecessary assignment to benefit conciseness and save gas.

Deviations from the Solidity Style Guide were identified throughout the code base. Some examples are:

- In the `Exchange`, `Escrow`, `GasPriceMinimum`, and `MultiSig` contracts among others, events are defined before state variables, and therefore the recommended layout order is not being followed.
- In several contracts, the recommended order of functions by its visibility is not being followed.
- In the `MultiSig` contract, specifically in lines 142 and 162 some deviations from the control structures syntax recommendations were found.

Taking into consideration how much value a consistent coding style adds to the project's readability, enforcing a standard coding style is recommended.

[N03] Transparent proxy pattern

In the proxy pattern used in the Celo protocol, the Proxy contract is in charge of forwarding calls to the implementation contract. However, it will not be able to delegate calls to functions with the same function signature as one of its own functions.

Consider using the the transparent proxy pattern for increased generality.

Update: *Not fixed. cLabs' statement for this issue:*

> As mentioned in the response to M09, it's an interesting idea but not of immediate implementation.
> This will be analyzed as a future change.

[N04] Misleading and incomplete comments

Across the repository, multiple occurrences of comments that are not accurate or incomplete, had been found:

In the `Governance` contract:

In the `FixidityLib` library:

- L213 does not show the expected answer for that test case.
- L261-L264 discuss test cases for the `reciprocal` function. The comments test the extreme cases of inputting `0`, `fixed1()` and the two “max” input values, but they fail to mention a test case for the input `1`, which should result in the highest output value of the function.

In the `Signatures` library:

- The docstring of the `getSignerOfMessageHash` function says “signed address” but it should say “message hash” instead.

Consider updating the comments to more accurately describe the purpose and effect of the codebase.

Update: *Fixed in pull request #2862. The comments have been updated.*

[N05] Typos

There are multiple typos across the repository. Some examples are:

In the `Accounts` contract:

- L424 says “of of” instead of “of”.

In the `Election` contract:

- L120 says “acconut” instead of “account”.

In the `SortedLinkedList` library:

- L169 says “greaerKey” instead of “greaterKey”.

In the `Validators` contract:

- L322 says “goven” instead of “govern”.

In the `UsingPrecompiles` contract:

- L188 says "correspoinding" instead of "corresponding".

In the `SortedOracles` contract:

- L249 says "alwas" instead of "always."

Consider running codespell on pull requests.

Update: *Fixed in pull request #2896. However,* `codespell` *is not being run in pull requests.*

[N06] Lack of consistency

In the `Accounts` contract there are functions such as `setName`, `setWalletAddress`, or `setAccountDataEncryptionKey` that do not create a pointer to the `Account` struct element in the `accounts` mapping although other functions such as `createAccount` use a pointer to then assign the values.

To improve auditability and reduce the potential surface for errors, consider modifying these places to use a consistent development style in the code.

Update: *Fixed in pull request #2897. Now the functions consistently assign the account to a variable.*

[N07] Attestations may be vulnerable to SIM swapping attacks

The documentation for the identity part of the project explains the process to link a phone number to an account address, generating an association that the software that interacts with the Celo blockchain will consider as certified. This documentation also states that `Attestations can theoretically be attacked, whether by a service provider or collusion with a validator`.

However, it should be noted that SIM swapping attacks are not `theoretical` and that the risk of being able to associate a user with a telephone number is not minor, as it could generate theft of stating how these attacks can generate phishing scenarios or theft of funds, and how this kind of attack should be mitigated.

Update: *Fixed in pull request #2956. The documentation has been corrected and detailed.*

[N08] Trust assumptions on hotfixes

The `Governance` contract allows for specific transactions to be executed without a normal "proposal" taking place.

This functionality is labeled as "Hotfix" and is built upon a mixture of off-chain and on-chain activities. If a problem in the protocol is discovered by the cLabs team, they will determine what transactions to execute in order to fix it. Then, they will publicly disclose a hash for users to whitelist it. Notably, the full details of the transaction will not be published, which is by design. If this hash is whitelisted by 2/3rds of the current set of validators and approved by the `approver` role, then anyone will be able to execute it.

This gives the Celo community, with the help of the validators, a way to address bugs and vulnerabilities. However, it can also be abused to execute transactions on behalf of the Governance contract. The general public should know that there is no way to understand what is contained in each transaction given just its hash, and that there is no guarantee of which specific transactions will be executed in a Hotfix. So they will need to trust the Hotfix proposer and its intentions.

Consider informing users of the mechanics of the hotfix system, highlighting the importance of the trust relationship between community and the proposers.

Update: *Fixed in pull request #2898.*

[N09] Lack of explicit return case

The `isConfirmed` function from the `MultiSig` contract checks if a submitted transaction has enough confirmations to be considered confirmed.

The issue is that if the total number of confirmations is not enough to reach `required`, then the function will be returning by default the value `false`, although there is not an explicit `return` statement.

To improve readability of the code, consider adding the explicit return case when the total number of confirmations is not enough to change the status of the transaction to confirmed, instead of relying on the default value that will be returned by the function.

Update: *Fixed in pull request #2863.*

[N10] Some ERC20 functions are susceptible to frontrunning

The `GoldToken` contract and `StableToken` contract are ERC20 tokens and are based on the OpenZeppelin IERC20 implementation.

All the standard ERC20 functions are completely implemented in the `GoldToken` contract. Among them, the `approve` function is susceptible to front-running attacks. For example, assume that Alice approves Bob to spend 100 tokens. After the transaction is mined, Alice wants to reduce the approved value for Bob so she again calls the `approve` function with 50 tokens as a parameter. Bob sees the transaction in the mempool and, before this second transaction is mined, he spends the initial amount of 100 tokens. This can accomplished by him by submitting the "spend" transaction with a higher gasPrice than the second call to `approve`, which should get the spending transaction processed first. After both transactions are mined, Bob can now spend another 50 tokens on behalf of Alice.

Another case in which a front-running scenario happens is in the `decreaseAllowance` function, although its severity is less than in the `approve` function. In this case, an attacker could only front-run a transaction that Alice makes in `decreaseAllowance` to spend more than the new allowed value.

It should be noted that `approve` double-spending issue is a well-known problem with ERC20 tokens and that the `decreaseAllowance` front-running problem is not solvable. However, consider making sure users are informed of the risks of using `approve` and

[N11] Unnecessary modifier in `mint` function

Both `mint` and `_mint` functions in the `StableToken` contract implement the `updateInflationFactor` modifier.

Given that `mint` only validates that the sender of the function call is the `Exchange` or the `Validators` contract before calling `_mint`, and given that some extra operations will be executed, consider removing the `updateInflationFactor` modifier from the `mint` function.

Update: *Fixed in pull request #2864.*

[N12] Incomplete docstrings

Some `public` and `external` functions are not properly documented using docstrings. Some examples are:

- `addToken`, `removeToken`, `addOtherReserveAddress`, and `transferGold` functions in the `Reserve.sol` contract do not specify the return parameter.
- All public and external functions in the `AddressLinkedList`, `AddressSortedLinkedList`, `AddressSortedLinkedListWithMedian`, `IntegerSortedLinkedList`, `LinkedList`, `SortedLinkedList`, and `SortedLinkedListWithMedian` contracts do not specify the `list` parameter when appropriate.
- The `removeOracle` function of the `SortedOracles` contract lacks the `token` and `index` parameters docstrings.
- The `initialize` functions in the `SortedOracles`, `Escrow`, `Attestations`, and `Accounts` contracts do not have any documentation.
- The `getUnselectedRequest` function in the `Attestation` contract does not properly list the return parameters.

Consider checking that all the external and public functions are properly documented.

Update: *Partially fixed in pull request #2778.*

A few parameters like `list` in `AddressSortedLinkedList.getElements()` or

To favor explicitness and readability, several parts of the source code may benefit from better naming. Our suggestions are:

In the `Accounts` contract:

- `vote` to `voter`.
- `attestation` to `attestor`.
- `isAccount` to `accountExists`.
- `isNotAccount` to `accountDoesNotExist`.
- In the `Registry` contract:
- `getAddressForOrDie` to `getAddressForHashOrDie`.
- `getAddressFor` to `getAddressForHash`.

In the `LockedGold` contract:
* `totalNonvoting` to `totalNonVoting`.
* `nonvoting` to `nonVoting`.
* `incrementNonvotingAccountBalance` to `incrementNonVotingAccountBalance`.
* `_incrementNonvotingAccountBalance` to `_incrementNonVotingAccountBalance`.
* `decrementNonvotingAccountBalance` to `decrementNonVotingAccountBalance`.
* `_decrementNonvotingAccountBalance` to `_decrementNonVotingAccountBalance`.

In the `Multisig` contract:
* `external_call` to `externalCall`.

[N14] Broken links in comments

The comment on line 407 of the `Governance` contract contains this broken link, and the comment on line 414 of the `StableToken` contract contains this other broken link.

Consider changing these links to the correct ones, or consider removing these comments.

[N15] Redundant `require` Statements in `vote()`

Within the `Governance` contract, in the function `vote`, there are `require` statements that check for the same conditions. In particular, on lines 603 – 608, there are two `require` statements checking that `value != Proposals.VoteValue.None` and `weight > 0`.

Consider removing one of each redundant check so that it is only performed once. Consider keeping the code to one check per `require` to maximize the effectiveness of error messages from `require`s.

Update: *Fixed in pull request #2614. The cLabs team has removed the redundant checks.*

[N16] TODOs in code

There are "TODO" comments in the code base that should be removed and instead tracked in the project's issues backlog. See for example:

- line 5 of the `FractionUtil` library.
- line 27 of the `Proposals` library.
- line 324, and line 723 of the `Election` contract.
- line 10 of the `SortedOracles` contract.
- line 335 of the `Election` contract.
- line 26, and line 71 of the `UsingRegistry` contract.
- line 17, line 407, line 464, line 531, line 551, line 552, line 640, and line 982 of the `Governance` contract.
- line 71, line 73, and line 151 of the `SortedLinkedList` library.
- line 121, and line 123 of the `SortedLinkedListWithMedian` library.
- line 413 of the `StableToken` contract.
- line 335 of the `Exchange` contract.
- line 59 of the `Validators` contract.
- line 244 of the `EpochRewards` contract.

Having well described `TODO` comments will make the process of tracking and solving them easier. Without that information, these comments might tend to rot and important information for link to the corresponding issue in the project repository.

Consider updating the `TODO` comments, adding at least a brief description of the task pending to do, and a link to the corresponding issue in the project repository. In addition to this, for completeness, a signature and a timestamp can be added. For example:

```
// TODO: Move to uint128 if gas savings are significant enough.
// https://github.com/celo-org/celo-monorepo/issues/1338
// --asaj - 20200117
```

[N17] Different address nomenclature for precompiled contracts

The project uses precompiled contracts to perform intensive calculations outside the VM native language, Solidity.

The contracts that make use of these precompiled contracts are the `GoldToken` contract and the `UsingPrecompiles` contract. Although they both use the `Transfer` precompile, they both define its address in a different way.

Consider using the same value to make usage of these precompiled contracts within the project consistent, and reduce the surface for error.

***Update:** Fixed in pull request #2901. The `GoldToken` contract has been updated to be consistent with the other usages of the precompiled contracts address.*

[N18] Check that imports are up-to-date

The Celo protocol builds off of several contracts not written by the cLabs team. Over time, it is likely that bugs will be found in these contracts by other teams, or improved functionality will be implemented. Before deployment, it is a good idea to check any contracts written by outside groups to ensure that any updates are at least considered. Some contracts written by other groups include the `FixidityLib` and the `Multisig` contracts, as well as various OpenZeppelin contracts.

***Update**: Partially fixed. The cLabs team has acknowledged this and begun creating a process for checking for 3rd-party updates.*

[N19] Docs incorrect regarding proposal deposits

In the Celo docs it states:

> Any user may submit a “Proposal” to the Governance smart contract, along with a commitment of Locked Celo Gold. This commitment is required to avoid spam proposals.

However, to create a proposal a user must send in regular Celo Gold, analogous to Ether, rather than Locked Gold, to create a proposal.

Consider changing the docs to state that creating proposals requires a transfer of regular Celo Gold.

***Update:** Fixed in pull request #2902. The documentation has been corrected and expanded.*

[N20] Unnecessary checks

The `FixidityLib` library provides fixed point arithmetic with protection against overflow operations to smart contracts.

To represent these fixed point decimals there are maximum values that a variable can take.

One of the methods, `newFixedFraction`, allows to convert a two-`uint256`-parameters fraction representation into a fixed point variable. This method checks if the `numerator` and `denominator` are less than the `maxNewFixed` value and checks if the denominator is now zero. After that, both parameters are converted into `Fraction`s by calling the `newFixed` method and then the division is returned by calling the `divide` method.

The issue comes from the fact that the mentioned checks are already done in the `newFixed` method and in the `divide` method, so implementing them again in the `newFixedFraction` method is unnecessary and a waste of gas.

Update: *Fixed in pull request #2865. The unnecessary checks have been removed.*

[N21] Inconsistent use of `onlyVm` modifier

Throughout the code, many functions are designed to be only called by the “VM”, or `address(0)`. In `Validators.sol`, there exists an `onlyVm` modifier. However, at different spots in the code, the same functionality is implemented with an inline `require` statement. The following places have been identified for this issue:

- Line 529 of the `Election` contract.
- Line 379 of the `EpochRewards` contract.
- Line 57 of the `Random` contract.

For better auditability and for the sake of code re-use, consider implementing an `onlyVm` modifier in place of such inline `require` statements.

Update: *Fixed in pull request #2903. The cLabs team has added a contract containing an onlyVM modifier that is now used throughout the code.*

[N22] Not using EIP 1967: Standard Proxy Storage Slots

In the `Proxy` contract, the `implementation` and `owner` storage slots positions are being defined without following the EIP 1967.

Even though EIP 1967 is still a draft, it is being used by Etherscan for distinguishing whether a contract is a proxy or not, and to read the implementation contract directly from the proxy.
In addition to this, the EIP 1967 calculates the storage slots in a way that the preimage of the hash cannot be known, reducing the chances of a possible attack.

Consider changing the implementation and owner slot positions to match the standard.

Update: *Fixed in pull request #2904.*

[N23] Unused code

not being used.

- Lines 61-112 of the `FixidityLib` library declare maximum values for the different functions of the library. Nevertheless, `maxUint256`, `maxFixedAdd`, `maxFixedMul`, and `maxFixedDividend` are not used along the code of the project.

This increases the code complexity and the size of the bytecode to be deployed.

Consider removing all the unused functions, and consider using the maximum values to restrict the input along the code.

***Update**: Fixed in pull request #2905. The unused code has been removed, and relevant information has been moved to the docstrings.*
However, note that some docstrings now refer to the names of the removed functions. Consider replacing these stale references with the information that was returned by the corresponding functions.

Conclusion

3 critical and 3 high-severity issues were found. Some changes were proposed to encourage code re-use and consistency. A number of issues were identified which relate to incentive design, some of which may simply be inherent to the protocol. The changes proposed in these cases should be considered carefully, and may not be necessary to implement. However, we report them so users and developers are aware of potential future issues and can keep themselves and their funds safe.

After a first phase of auditing, the cLabs team asked us to review and audit the recent changes in the smart contracts for their protocol.

The audited commit is `db2e561e1f13220a98743a7999818e48a5089d13`. All the changes to the `contracts` folder since commit `7be22605e172ca536c028e408b147aab83202e4a` are included in the scope for this phase.

Vulnerabilities

Below, we list all vulnerabilities identified in the Celo contracts.

Critical

None.

High

[H01] `removeSlasher` can cause mismatch between `slashingWhitelist` and `slashingMap`

It is possible to successfully call the `removeSlasher` function within `LockedGold.sol` with a mismatch between the input parameters `slasherIdentifier` and `index` (for example, with `index` corresponding to the index of a different identifier than `slasherIdentifier` within `slashingWhitelist`). Upon doing so, `slashingWhitelist[index]` will be deleted and `slashingMap`'s entry corresponding to `identifier` will be changed to `false`.

This leads to two problems. The first is that the `slasherWhitelist` and `slashingMap` will be mismatched. The second is that the `slashingMap[keyBytes]` will now equal `false`, so future calls to this function with the same `slasherIdentifier` (from which `keyBytes` is derived) will revert, and the slasher may be able to be added to the whitelist twice.

This issue's severity is minimized by the onlyOwner modifier on `removeSlasher`. As long as the owner can be trusted, this issue is less likely to cause problems. Additionally, the owner can perform certain combinations of calls to `removeSlasher` and `addSlasher` to fix any mismatch problems. Still, to avoid this entirely, consider adding a `require` that checks that `slashingWhitelist[index] == keyBytes` within the `removeSlasher` function.

Update: *Fixed in pull request #3122.*

Medium

the value `0xffffffffffffffffffffffffffffffff`, which is 32 hex digits or 128 bits. Since default `uint` types in solidity have 256 bits, the `MAX_UINT` value should be `0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff` (with 64 hexidecimal `f`'s).

When the `initialize` function is called with an `initialDistributionRatio` of `1000` and the `totalGrant` is greater than this erroneous value of `MAX_INT`, it will result in some gold not being able to be withdrawn. In the same way, an erroneous distribution could happen in the `setMaxDistribution` function.

Consider changing `MAX_UINT` to the correct value. Furthermore, to be extra careful, consider setting `maxDistribution` to `totalGrant` and `totalBalance`, instead of using the maximum possible value which is usually going to be much larger than the total withdrawable amount.

Update: *Fixed in pull request #3274. It is now forcing an underflow to get the maximum possible value for a `uint256`. Note however that this is not clear, and it might cause problems in the future if Solidity decides to change the default behavior for underflows.*

[M02] Inputs are not checked in `setReserveFraction`

The `setReserveFraction` function of the `Exchange` contract has no checks on its input, which means it can set `reserveFraction` to values corresponding to a `Fraction` of greater than or less than `1.0`.

In the function `getUpdatedGoldBucket`, the `updatedGoldBucket` is set to `reserveFraction` multiplied by the unfrozen reserve gold balance.

According to the documentation, "`the Celo Gold bucket must remain smaller than the total reserve gold balance`". So, it follows that the `reserveFraction` should never be equal to or greater than `1.0`.

Consider adding a `require` such that `reserveFraction`'s value is restricted to being strictly less than `1.0` when being set.

[L01] Duplicated `onlyWhenFrozen` modifier

The functions `transfer` and `transferWithComment` of the `GoldToken` contract have the `onlyWhenNotFrozen` modifier. They both call the internal `_transfer function`, which also has the `onlyWhenNotFrozen` modifier. This causes the modifier to be called twice during the execution of transfers.

Consider removing the `onlyWhenNotFrozen` modifier from the `transfer` and `transferWithComment` function, or removing it from the `_transfer` internal function, so it is called only once per transfer.

***Update:** Fixed in pull request #3123. The modifier was removed from the `transfer` and `transferWithComment` functions.*

[L02] Interfaces are missing functions present in their implementations

Some interfaces are not fully matching their implementation. For example:

- The `IAccounts` interface is implemented by the `Accounts` contract. The implementation includes extra functions not declared in `IAccounts`: `removeVoteSigner`, `removeValidatorSigner`, and `removeAttestationSigner`.
- The `IStableToken` interface is implemented by the `StableToken` contract. In a recent change to the `StableToken` implementation, the function `debitFrom` was renamed to `debitGasFee` and the function `creditTo` was renamed to `creditGasFee`.
  However, in the interface these functions were just removed, not renamed.
- The `IFreezer` interface is implemented by the `Freezer` contract. The interface is missing the declaration of the `freeze` and `unfreeze` functions.
- The `IReleaseGold` interface is implemented by the `ReleaseGold` contract. The interface is missing the declaration of the `setLiquidityProvision`, `setMaxDistribution`, `setBeneficiary`, `authorizeValidationSigner`, and `authorizeAttestationSigner` functions.

Consider adding the missing functions to the corresponding interface, so there is one basic implementation that directly matches the interface specification. Alternatively, if there are reasons for these functions to be only in the implementations, for each case consider to add a new contract that inherits from the basic implementation, to move the extra functions to this contract, and to add to its docstrings the reason for extending the interface.

[L03] The `initialize` function is inconsistently declared in interfaces

The `initialize` function is declared in some interfaces (like in `IExchange`), but not in others (like in `IAttestations`).

The `initialize` function plays a role similar to the constructor, so consider not declaring it in the interfaces. If it is important to add it for some reason, consider consistently adding it to all interfaces.

***Update:** Fixed in pull request #3124. The interfaces no longer declare the `initialize` functions.*

[L04] Lack of indexed parameters in events

Throughout the codebase, there are events defined without any indexed parameters. Some examples are:

- Lines 17-18 in the `TransferWhitelist` contract.
- Lines 34-36 in the `StableToken` contract.

Consider indexing event parameters when appropriate, to avoid hindering the task of off-chain services searching and filtering for specific events.

[L05] Lack of input validation in the Heap library

Neither the `heapifyDown` nor the `siftDown` functions of the `Heap` library are validating that both array parameters have the same length.

However, this could introduce bugs and vulnerabilities in the future, if the library is used in a different part of the code or if the `keys` and `votesForNextMember` array have a different length.

Consider introducing a `require` statement validating the parameters of any of these functions so that the code will remain secure in the future.

Update: *Fixed in pull request #3121.*

[L06] The relations between loops and their boundaries are not clear

There are some occurrences in the codebase where `for` loops are used to iterate an entire array.

This is a widely known problem in the Ethereum ecosystem, as it is common that loops in Smart Contracts result in an out of gas error when the number of iterations that the loop needs to execute is too big.

- In the `getWhitelist` function of the `TransferWhitelist` contract there are two loops iterating over the `whitelist` and the `registeredContracts` arrays correspondingly.
- In the `getAssetAllocationWeights` function of the `Reserve` contract, there is a `for` loop iterating over the `assetAllocationSymbols` array.

Consider setting a hard cap on the number of iterations that the smart contracts should execute. Likewise, when the iteration limit is bounded by invariant properties in the code, consider enforcing that those properties are invariant. Finally, consider thoroughly reviewing the usage of `for` and `while` loops in the protocol and closely analyzing the effects of failures that may arise from the lack of limits in these loops.

Update: *Not Fixed. Issue #3420 was created to keep track of this.*

[L07] Unnecessary loop

loops goes from 0 to the size of the [illegible], so having them separately wastes gas and duplicates code unnecessarily.

Consider refactoring the code to fill these arrays in a single `for` loop.

Update: *Fixed in pull request #3126.*

[L08] There are two different whitelist getters

The `TransferWhitelist` contract is used to whitelist addresses.
Addresses can be added directly to the whitelist, or they can be added through the registry. Then, the `getWhitelist` function will query the registry to get the addresses of the registered contracts, will append them to the directly whitelisted address, and thus return all the whitelisted addresses.

Note that the directly added addresses are stored in the `whitelist` public state variable.
Because this variable is public, the compiler will create its getter function `whitelist`. This automatic getter will return a different list than the one returned by the `getWhitelist` function.

To avoid confusion, consider making the `whitelist` state variable private.

Update: *Fixed in pull request #3127.*

[L09] Whitelist setters do not emit events

The `TransferWhitelist` contract is used to whitelist addresses.
Addresses and contract identifiers can be added individually by calling the `addAddress` and the `addRegisteredContract` functions. Or lists of addresses and contract identifiers can be set by calling the `setWhitelist` and the `setRegisteredContracts` functions.

When the items are added individually to the whitelist, the events `WhitelistedAddress` and `WhitelistedRegistryId` are emitted. However, when the lists are set, these events are not emitted. This could cause the event log to be misleading.

Consider also emitting events when lists are added to the whitelist. Ideally, when a list is set overwriting the previously whitelisted items, a removal event should be emitted for each of the can be ambiguous because looking at the log it will not be clear if the previous whitelisted items were kept or overwritten. In this case, consider thoroughly documenting the events and the functions to make the behavior clear.

***Update:** Fixed in pull request #3134](https://github.com/celo-org/celo-monorepo/pull/3134). `setWhitelist` now emits the `WhitelistedAddressRemoved` event for all the previous addresses. It calls the `addAddress` function to add all the new ones, which emits the `WhitelistedAddress` event.*

[L10] Lack of validation consistency

The functions `addRegisteredContract` and `setRegisteredContracts` of the `TransferWhitelist` contract are in charge of assigning the values of `registeredContracts`.

While the `addRegisteredContract` function validates that the parameter being added to `registeredContracts` is not `address(0)`, the `setRegisteredContracts` function does not perform any validation.

Consider validating that none of the elements passed into `setRegisteredContracts` are `address(0)`.

[L11] Removed account signers cannot be reauthorized

The functions `removeVoteSigner`, `removeValidatorSigner` and `removeAttestationSigner` of the `Accounts` contract remove authorized signers from the sender account, but the signers cannot be authorized again.

When signers are authorized through any of these functions, a call is made to the `authorize` function and they are added into the `authorizedBy` mapping. When signers are removed, they are removed from the account's `signer` structure, but they are left in the `authorizedBy` mapping. Because during authorization the contract checks that authorized addresses are not in the `authorizedBy` mapping, this means that removed accounts cannot be authorized again.

If the remove functions are intended to be the inverse of the authorize functions (which is what might be expected just by reading the signatures and the docstrings), consider updating the

Update: *Fixed in pull request #3128. The functions now have clear comments about signers not able to be reauthorized.*

[L12] Comments still refer to removed functions

In the `FixidityLib` library, the functions `maxFixedMul` and `maxFixedDividend` have been removed. However, they are still mentioned in the docstrings in lines 13, 100, 101, 105, 166, 236, 238, 239, 240, 242, and 243.

Consider removing these comments or changing them so they reflect what still exists in the file.

Update: *Fixed in pull request #3143.*

[L13] Duplicated code in `updatePublicKeys`

The `updatePublicKeys` function of the `Validators` contract updates the ECDSA and BLS public keys of a validator account atomically by calling the private functions `_updateEcdsaPublicKey` and `_updateBlsPublicKey`.

This `updatePublicKeys` function is duplicating some code from the external functions `updateEcdsaPublicKey` and `updateEcdsaPublicKey`.

Consider refactoring the `external` and `private` key update functions to avoid code duplication as much as possible.

Update: *Fixed in pull request #3282.*

[L14] `maxDistribution` can be set before the factory sends the gold

In the `initialize` function of the `ReleaseGold` contract, the scheduled release is used when setting the `maxDistribution` value because at this moment the factory has not sent the gold.

However, when the `setMaxDistribution` function is called, the `getTotalBalance` function is used despite the fact that there is nothing in this contract ensuring that the factory has `getTotalBalance` can return the correct value.

If this makes the design too complicated, consider at least adding a comment explaining that the `setMaxDistribution` function should not be called before the factory has sent the gold.

Update: *Fixed in pull request #3276. Now* `setMaxDistribution` *reverts if `totalBalance is 0.*

[L15] Missing revert messages in `require` statements

In line 30 of `ReentrancyGuard.sol` there is a `require` statement without revert messages. This behavior results in an increased difficulty of debugging the system.

Consider including specific and informative revert messages in all require statements.

Update: *Fixed in pull request #3277.*

[L16] Unclear overwrite of commission update queue

In the `queueCommissionUpdate` function of the `Validators` contract the `nextCommission` and `nextCommissionBlock` for a group are updated. When the `nextCommissionBlock` has passed, the `updateCommission` function can be called to update the group commission.

It is not clear what should happen when `queueCommissionUpdate` is called twice before `updateCommission` succeeds. Currently, the `nextCommission` and `nextCommissionBlock` are just overwritten. However, by calling it a queue it gives the impression that the values of the second call will be qeued to be applied after the values of the first call.

Consider documenting this case, explaining that the pending commission values will be overwritten. Since this is a queue of one element, consider renaming the function to something like `setNextCommission`.

Update: *Fixed in pull request #3278.*

Notes

- Import of `IValidators.sol` of `Attestations.sol`.
- Import of `IStableToken.sol` of `Reserve.sol`

Consider removing these unused imports for improved code auditability.

Update: *Fixed in pull request #3136.*

[N02] Renaming suggestions

Good naming is one of the keys for readable code, and to make the intention of the code clear for future changes. There are some names in the code that make it confusing or hard to understand.

Consider the following suggestions:

In the `TransferWhitelist` contract:
* `whitelist` to `directlyWhitelistedAddresses`.
* `registeredContracts` to `whitelistedContractIdentifiers`.
* `WhitelistedRegistryId` to `WhitelistedContractIdentifier`.
* `addAddress` to `whitelistAddress`.
* `addRegisteredContract` to `whitelistRegisteredContract`.
* `registryId` to `contractIdentifier`.
* `getRegisteredContractsLength` to `getNumberOfWhitelistedContractIdentifiers`.
* `setWhitelist` to `setDirectlyWhitelistedAddresses`.
* `setRegisteredContracts` to `setWhitelistedContractIdentifiers`.

Update: *The suggested renames were applied in pull request #3184.*

[N03] Misleading comment

In the Reserve contract:
* L435 implies that the `ratio` needs to be `greater than 2` in order for the protocol to transfer tax on Celo Gold but in the code the `ratio` can be greater than *or equal to* 2.

Update: *Fixed in pull request #3185.*

[N04] MockReserve.sol has `mintToken` function

The MockReserve contract has a `mintToken` function, while the contract it is based on, Reserve, does not.

Consider removing `mintToken` from the `MockReserve` contract so it works better as a test double for the production contract.

Update: *Fixed in pull request #3281.*

Conclusion

One high severity vulnerability was identified within the changes to the code since our previous audit. Several lower severity vulnerabilities were found relating to style, efficiency, and consistency throughout the codebase. Overall, the OpenZeppelin team was quite pleased with the quality of the code and the receptiveness of the cLabs team to our previous audit phase.

After a third phase of auditing, the cLabs team asked us to review and audit the recent changes in the smart contracts from their protocol.

Scope

The original commit in scope is `b2f0a58fcc7667d41585123aae5b24c47aa894f6`. To ensure that our audit process is complete and no unaudited code was added to the codebase, OpenZeppelin audited the difference between the commit in scope and the commits in scope for previous phases of our engagement with the cLabs team, such as commit `db2e561e1f13220a98743a7999818e48a5089d13` from the second phase, and commits `c224b1f5fd86745a67acfef67f5ef08e2d3c0e6a`, `2155ef0208af542d4e1301a78534c2656072dc91`, and `bd51ef21ca50962e6bb81f742d0b6fa666c438b9`, from the previous PR3030, PR3033, and PR3172 diff-audit.

commit. This commit further implemented a versioning system and a proxy system for some of the contracts, which were also in scope. As a result of this audit round, an additional issue was reported to the cLabs team. This issue was fixed in commit da72e7e85ca95f3a17269e0fe905ffee4ff30822, making it the newest release candidate commit.

The changes in scope were the ones to the following files between the mentioned commits:

`packages/protocol/contracts/common/Accounts.sol`
`packages/protocol/contracts/common/ExternalCall.sol`
`packages/protocol/contracts/common/FixidityLib.sol`
`packages/protocol/contracts/common/Freezer.sol`
`packages/protocol/contracts/common/GasPriceMinimum.sol`
`packages/protocol/contracts/common/GoldToken.sol`
`packages/protocol/contracts/common/MetaTransactionWallet.sol`
`packages/protocol/contracts/common/MultiSig.sol`
`packages/protocol/contracts/common/Registry.sol`
`packages/protocol/contracts/common/TransferWhitelist.sol`
`packages/protocol/contracts/common/interfaces/IAccounts.sol`
`packages/protocol/contracts/common/interfaces/ICeloVersionedContract.sol`
`packages/protocol/contracts/common/interfaces/IFeeCurrencyWhitelist.sol`
`packages/protocol/contracts/common/interfaces/IFreezer.sol`
`packages/protocol/contracts/common/interfaces/IMetaTransactionWallet.sol`
`packages/protocol/contracts/common/interfaces/IRegistry.sol`
`packages/protocol/contracts/common/interfaces/IValidators.sol`
`packages/protocol/contracts/common/libraries/Heap.sol`
`packages/protocol/contracts/common/libraries/ReentrancyGuard.sol`
`packages/protocol/contracts/governance/DoubleSigningSlasher.sol`
`packages/protocol/contracts/governance/DowntimeSlasher.sol`
`packages/protocol/contracts/governance/Election.sol`

`packages/protocol/contracts/governance/LockedGold.sol`

`packages/protocol/contracts/governance/Proposals.sol`

`packages/protocol/contracts/governance/ReleaseGold.sol`

`packages/protocol/contracts/governance/Validators.sol`

`packages/protocol/contracts/governance/interfaces/IElection.sol`

`packages/protocol/contracts/governance/interfaces/ILockedGold.sol`

`packages/protocol/contracts/governance/interfaces/IReleaseGold.sol`

`packages/protocol/contracts/governance/interfaces/IValidators.sol`

`packages/protocol/contracts/identity/Attestations.sol`

`packages/protocol/contracts/identity/interfaces/IAttestations.sol`

`packages/protocol/contracts/identity/interfaces/IEscrow.sol`

`packages/protocol/contracts/stability/Exchange.sol`

`packages/protocol/contracts/stability/Reserve.sol`

`packages/protocol/contracts/stability/SortedOracles.sol`

`packages/protocol/contracts/stability/StableToken.sol`

`packages/protocol/contracts/stability/interfaces/IExchange.sol`

`packages/protocol/contracts/stability/interfaces/IReserve.sol`

Outside of the scope for this phase are the unchanged parts of the contracts that we audited in previous phases. Also, tests and mock contracts were not inside the scope.

Overview of the changes

The Celo protocol not only has improved the codebase by following several suggestions given in previous phases, but it also added more significant changes in the functionality. Some of the main changes are: the `Governance` contract can modify the `tobinTax` and `tobinTaxReserveRatio` variables, the incorporation of the Carbon Offseting Fund concept to improve and achieve a carbon-neutral platform by sending a percentage of the epoch rewards to a partner account, the introduction of two contracts called `MetaTransactionWallet` and `ExternalCall`, a new versioning system of the whole codebase, changes to the `Exchange` contract so it is possible to buy a certain amount of tokens with the maximum price of other token, changes in the `DowntimeSlasher` contract to send intervals across different epochs, the ability

Update

All issues listed below have been reviewed by the cLab's team, who applied changes to the code base to address them. These were implemented in several pull requests applied to the branch `oz-prerelease`. The pull requests addressed are mentioned in the corresponding issue. In addition, the PR 5194 has been audited during the fourth-phase-fixes review in anticipation of Celo's upcoming release.

Vulnerabilities

Below, we list all vulnerabilities identified in the Celo contracts.

which

Critical

None.

High

None.

Medium

[M01] Multiple iterations of the same transfer can occur

The `TransferWhitelist` contract implements the functionality to store "a whitelist of addresses for which transfers should not be frozen so that network initialization can take place".

Nevertheless, the `whitelistAddress` function does not check if the `newAddress` to be pushed into the `directlyWhitelistedAddresses` array was already added to it, making it possible to whitelist multiple times the same address.

Because these addresses can be used to perform transfers, if contracts that use this array do not check if there are multiple intances of the same address, then it could use the array from the `getWhitelist` outcome and re-do multiple same transactions during the execution.

Consider validating the input parameters to check that no-other element in the arrays match with the `newAddress` parameter. Furthermore, consider complementing the array with a mapping to keep track of the whitelisted addresses and restrict the possibility for those that are already flagged up.

***Update**: Acknowledged, and will not fix. Affected contract was only used once and it is not going to be used again. The cLabs team's response was:*

> The `TransferWhitelist` contract has been deprecated. It was used to bootstrap the network and enable features in phases but will not be used again.

[M02] Data lengths alteration could lead to malicious transaction executions

The `MetaTransactionWallet` contract allows a signer to execute multiple transactions under a single function call.

To do that, the signer must package all the data from each transaction under one `bytes` variable and then send it along with an array that has the length of each transaction's data in the respective element.

Nevertheless, there are no validations to check if the respective slice of data corresponds to the specific transaction. In case the signer uses a malicious or buggy client to generate the transaction, this vulnerability may allow the execution of undesired transactions resulting in unexpected results.

Consider dividing the data from each transaction in the `data` parameter with a slot that carries the size of the following data's transaction, to then be able to compare it with the elements of the `dataLengths` before the transaction is executed.

***Update**: Acknowledged, and will not fix. The cLabs team's response was:*

> cLabs has decided to leave the functionality as is. If the client puts the data length in two different parameters it becomes redundant, and a malicious client could easily provide matching lengths but bad data. If the transaction lengths are inaccurate in the worst case the execution will fail.

off-line transactions and then submit them into the blockchain.

It also allows to execute multiple transaction on behalf of the signer when calling the `executeTransactions` function. To do so, the signer has to send arrays with the destination addresses, the CELO values to send to those addresses, and the lengths of the data for each transaction, while packing all the data under one `bytes` parameter.

To split the respective data from each transaction, the contract uses the external `slice` method from the `BytesLib` library. That method, has only a requirement that checks if the passed `bytes` parameter has *at least* the needed length to slice it.

However, because there are no checks for the data parameter, and only the lengths of all three arrays are checked, the data could be way bigger than it should.

This behavior could allow the malfunction of the smart contract, which should restrict and validate that the data that wants to be executed matches in length with the one sent to the contract.

Consider requiring that the `data` parameter's length equals to the sum of all the lengths passed in the `dataLengths` parameter.

***Update**: Fixed on pull request 5107.*

Low

[L01] Frozen function can be bypassed

The `Election` contract inherits from the `Freezable` contract the functionality to freeze certain parts of the code.

When the `Election` contract is frozen through the `Freezer` contract, the `electValidatorSigners` function cannot be called.

Nevertheless, as the `electValidatorSigners` function is only a wrapper for the `electNValidatorSigners` function and, because `electNValidatorSigners` is also a

Although these two functions do not change storage variables, consider either freezing both functions at the same time or documenting the decision of only freezing the wrapper function.

***Update**: Fixed on pull request 4910. The `Election` contract no longer inherits the freezable functionalities from the `Freezable` contract, and the `onlyWhenNotFrozen` modifier was removed from the `electValidatorSigners` function.*

[L02] Parameter and input variable type mismatch

The `isAttestationExpired` function of the `Attesatations` contract receives a `uint128` parameter as input.

Nonetheless, the variable used as input when calling that function is `blockNumber`, which has a `uint32` variable type.

Consider being consistent with the type used in the `isAttestationExpired` function, either by modifying the `isAttestationExpired`'s parameter type to `uint32`, by modifying the `blockNumber` variable's type to `uint128`, or by casting the `blockNumber` variable to `uint128`.

***Update**: Fixed on pull request 4881.*

[L03] Downtime slash is frontrunnable

The `DowntimeSlasher` contract holds the logic for the offline-validators protocol's slashing process. The `bitmaps` mapping is a complex data structure which is written in the `setBitmapForInterval` function, and maps the bitmap information to the address of the reporter of the downtime. The `bitmaps` mapping is then used to validate if a certain validator was down in a specific interval in the `wasDownForInterval` function. This last function is the core component that will eventually be used by the `slash` function to perform the slash.

The `slash` function is frontrunnable; any new reporter that sees other `slash` function call in the mempool of the node can write the `bitmaps` mapping by calling the `setBitmapForInterval` function and then calling the `slash` function with the same parameters as the original transaction in the mempool.

Celo network is more mature and in high demand.

In the future, consider penalizing reporters that send false downtime proofs so that frontrunning bots need to calculate whether the `slash` transactions in the mempool are correct instead of spamming the network.

***Update**: Acknowledged, and will not fix. The cLabs team's response was:*

> The reward received for slashing a validator is unlikely to warrant intense competition, and is simply meant to subsidize the cost of the slash transaction itself. The relative cost of the failed clone transactions is likely to make these frontrunning attempts unprofitable, and the network can govern the penalty/reward ratio if this ever becomes a throughput bottleneck.

[L04] `BitmapSetForInterval` event does not broadcast caller information

The `BitmapSetForInterval` event in the `DowntimeSlasher` contract does not broadcast the `msg.sender` information.

Because the `bitmaps` mapping requires the `msg.sender` to save the setted bitmap, when two or more reporters are competing for slashing a certain validator the `BitmapSetForInterval` event will broadcast the same information without the possibility to know which event emittion corresponds to which reporter of the downtime.

Consider adding the `msg.sender` information in the `BitmapSetForInterval` event to add clarity to these events.

***Update**: Fixed on pull request 4914.*

[L05] Interfaces are missing functions present in their implementations

Interfaces in the codebase are not fully matching their implementation, defining only a part of the functionalities implemented in their respectives contracts. Some of them are:

- The `IMetaTransactionWallet` interface is implemented by the `MetaTransactionWallet` contract. The interface does not include all function such as

- The `IReleaseGold` interface is implemented by the `ReleaseGold` contract. Not all functions implemented in that contract are included as definitions in the interface, such as the `setCanExpire` function.
- The `IValidators` interface is implemented by the `Validators` contract. The interface does not include all functions implemented by the contract, such as the `getCommissionUpdateDelay` function.
- The `IAttestations` interface is implemented by the `Attestations` contract. Not all the implemented functions are defined in the interface, such as the `getAttestationIssuers` function.

These mismatches between the interface and its implementation are confusing. It can mean that there are problems with the modeling of the system or that the updates to the implemenations and interfaces got out of sync.

Consider adding these, and all other, missing functions to the corresponding interface, so there is one basic implementation that directly matches the interface specification. Alternatively, if there are reasons for these functions to be only in the implementations, for each case consider to add a new contract that inherits from the basic implementation, to move the extra functions to this contract, and to add to its docstrings the reason for extending the interface.

Update: *Partially fixed on pull request 4958. There are still some interfaces that do not represent their corresponding implementation, such as the `IExchange` interface.*

[L06] Lack of validation

In several parts of the codebase, there were seen operations that lack a validation of its inputs. Some of those places are the followings:

- The `signer` variable from the `MetaTransactionWallet` is a crucial asset of the contract, because it is the one authorized to execute transactions for the wallet. Nevertheless, the `setSigner` function does not check input validation after the former `signer` has executed a signer's address change by calling the `executeTransaction` function. The new `signer`'s address could be even the zero address, making imposible to recover its control.

- The `approveTransfer` function from the `Attestations` contract allows to approve and transfer an identifier from one address to another. However, if either the `to` or `from` addresses call twice the `approveTransfer` function, the implementation does not check if that same action was performed already, emitting multiple times the `TransferApproval` event. Furthermore, `to` and `from` can be the same address, resulting in a unuseful but possible transfer.
- Some functions from the `LockedGold` contract, such as the `relock` function, do not check if the `value` to withdrew or lock is zero. In that scenario, calls to those functions will trigger events even though there are not changes in the balances.
- The `setCarbonOffsettingFund` function from the `EpochRewards` contract does not check if the address used to write the `carbonOffsettingPartner` variable is the `address(0)` value, which may result in loss of funds.

Even though this issue does not pose a security risk, the lack of validation on user-controlled parameters may result in erroneous transactions considering that some clients may default to sending null parameters if none are specified. Consider adding validation checks in both input parameters and outcomes from operations.

Update*: Partially fixed on pull request 5035. The cLabs team's response was:*

> We have decided to take a stance against checking for no-op contract calls when the worst case scenario is a user spending gas on a null operation. Examples include: transferring 0 tokens, locking 0 tokens, reassigning variables to the same state. For this reason, instead of adding more checks to `LockedGold` against locking/relocking/unlocking 0 tokens, we've taken the existing check out. We've also made the same decision for Attestations.approveTransfer() therefore omitting the check against the user reassigning their status to its current state.
> The protections against address(0), which represents the special VM sender used throughout Celo's Proof of Stake protocol, are considered useful and were added to the flagged locations.

### [L07] Lack of event emission

In the `TransferWhitelist` contract, there are two functions that modify the `whitelistedContractIdentifiers` variable: the `whitelistRegisteredContract`

`WhitelistedContractIdentifier` event.

Consider adding events to make it easier to track important contract storage changes.

Update: *Acknowledged, and will not fix. Affected contract was only used once and it is not going to be used again.*

[L08] Lack of indexed parameters in events

In the `MetaTransactionsWallet` contract, the following events are defined without any indexed parameters:

- `SignerSet`.
- `TransactionExecution`.
- `MetaTransactionExecution`.

Consider indexing the events' parameters when appropriate to avoid hindering the task of searching and filtering specific events to off-chain services.

Update: *Fixed on pull request 5107.*

[L09] Misleading comments and typographical errors

Several docstrings and inline comments throughout the code base were found to be erroneous and / or incomplete and should be fixed. In particular:

- In line 37 from the `TransferWhitelist.sol` file it states "removedAddress The address to add" instead of "removedAddress The address to remove".
- In line 185 of the `Attestations.sol` file it states "attestion" instead of "attestation".
- In lines 919 to 920 of `Validators.sol` file, the comment states `Add a bit of "wiggle room" to accommodate the fact that vote activation can result in a 1 // wei rounding error`. However, in line 921 the value added as "wiggle room" is 10 wei instead of the 1 stated in the comment.

Clear docstrings are fundamental to outline the intentions of the code. Mismatches between them and the implementation can lead to serious misconceptions about how the system is expected to

Update: *Fixed on pull request 4929.*

[L10] Missing docstrings

Some of the contracts and functions in Celo's codebase lack documentation. This hinders reviewers' understanding of the code's intention, which is fundamental to correctly assess not only security, but also correctness. Additionally, docstrings improve readability and ease maintenance. They should explicitly explain the purpose or intention of the functions, the scenarios under which they can fail, the roles allowed to call them, the values returned, and the events emitted. Some examples of incomplete docstrings are:

- The `inflationFactor` parameter of the `_valueToUnits` function in the `StableToken` contract.
- The `index` parameter of the `approveTransfer` function in the `Attestations` contract.
- The events located in the `Attestations` contract.
- The new `getTotalPendingWithdrawals` function in the `LockedGold` contract.

Consider thoroughly documenting all functions (and their parameters) that are part of the contracts' public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Partially fixed on pull request 5041. Events of the `Attestations` contract still do not have documentation.*

[L11] Missing revert messages in require statements

There are several `require` statements in the project without revert messages. This behavior results in an increased difficulty of debugging the system.

The list of affected require statements is:

- In the `EpochRewards` contract lines: 161, 182, and 183.
- In the `Attestations` contract line: 709.

***Update**: Fixed on pull request 4913.*

[L12] Transactions' outputs are not being returned

The `MetaTransactionWallet` contract allows the `signer` address to execute multiple transaction in a batch by calling the `executeTransactions` function.

This function takes the wrapped parameters from the batch and splits them into single transaction, to then call the `executeTransaction` function where the transaction will be executed.

However, although the `executeTransaction` function returns the data from the transaction's execution, the `executeTransactions` function does not handle the returned data from each one of the transactions in the batch, meaning that any relevant data that the transaction would output will not be returned to the caller.

Although it does not impose any security risk per se, consider handling, packaging, and returning the data output from each one of the transactions in the batch so the `signer` is able to make use of it if necessary.

***Update**: Fixed on pull request 5107. As an additional comment, always consider saving both return values of this function. If not, it will be impossible to decode the information retrieved.*

[L13] Not using the SafeMath library

The `SafeMath` library is not being used in the increment operation in line 215 of the `MetaTransactionWallet.sol` file.

The usage of this library prevents arithmetic overflow and underflow issues.

While this issue does not pose a security risk, and it is very unlikely that these operations could cause undesired outcomes as no exploitable overflows or underflows were detected in the current implementation, this may not be the case in future changes to the codebase.

Consider using the `SafeMath` library in these and all other places where an arithmetic operation is involved.

[L14] Solidity compiler version is not pinned and it is not the same for all contracts

The version of Solidity used throughout the codebase is not pinned to the same nor the latest stable version.
Consider pinning the version of the Solidity compiler to its same and latest stable version to prevent introducing unexpected bugs due to incompatible future releases. To choose a specific version, developers should consider both the compiler's features needed by the project and the list of known bugs associated with each Solidity compiler version.

Update*: Partially fixed on pull request 5032. The Solidity compiler is not pinned in the code base and some new contracts added on pull request 5194 do not have the same version of Solidity.*

[L15] Stable token may not be backed up in the reserve

The `Reserve` contract implements the functionality that ensures the price-stability of the stable token with respect to their pegs.

To do that, a tax called Tobin tax allows the protocol to transfer CELO assets into the reserve to preserve the amount of collateral needed for the peg.

If the CELO reserve ratio, defined as the ratio of the current CELO reserve balance to total stable token valuation, is less than the `tobinTaxReserveRatio` variable, then it will be applied a `tobinTax` value to raise the reserve's balance of CELO assets.

Nevertheless, the `tobinTaxReserveRatio` variable's value can be changed by the protocol when calling the `setTobinTaxReserveRatio` function, where no further validation checks are made to the input parameter. Due to that, if the input parameter is less than 100%, then the collateral that the protocol will seek to ensure the peg with the stable token will not cover the totality of the token's supply value.

Although it is unlikely that a proposal could be passed, succeeded, and set a problematic value, the human component will allways be present, allowing possible mistaken values in the protocol.

***Update**: Acknowledged, and will not fix. The cLabs team's response was:*

> It is true that the `TobinTax` represents the on-chain collateralization ratio, which is a problem in some scenarios:
> – Not all CELO in the reserve is in the reserve contract, some could be held in custody, see https://celoreserve.org/
>
> - Assuming all CELO in the reserve is in the smart contract, even if we know the target reserve ratio on-chain, there's no guarantee that the target ratio is met, due to market volatility
>
> Thus, it is difficult to fully automate when the tobin tax should kick in. Even if all the reserve assets are on chain (like using some wrapped version of BTC, ETH and DAI on Celo), we'd need oracles providing the price for all of them.

[L16] Transfer of CELO may unexpectedly fail

There are several parts of the code which use of the `transfer` Solidity built-in function to transfer CELO from the protocol:

- The `refundAndFinalize`, `withdraw`, and `fundSigner` functions of the `ReleaseGold` contract.
- The `_transferGold` function of the `Reserve` contract.
- The `withdraw` and `slash` functions of the `LockedGold` contract.

This function forwards a limited amount of gas to the receiver. Should the receiver is a contract with a fallback function that needs more than 2300 units of gas to execute, the transfer would inevitably fail. After the Ethereum's Istanbul hard fork, it has become a recommended practice not to use the transfer method to avoid hard dependencies on specific gas costs.

To avoid unexpected failures in the withdrawal of CELO, consider replacing the `transfer` method with the `sendValue` function available in the OpenZeppelin's Contracts library.

***Update**: Fixed on pull request 4981.*

Throughout the contracts, there are several unnecessary import statements:

- The `UsingRegistry.sol` file in the `SortedOracles` contract.
- The `IRandom.sol` file in the `Attestations` contract.
- The `IValidators.sol` file in the `ReleaseGold` contract.
- The `UsingRegistry.sol` file in the `Election` contract, as it is already being imported in the `Freeezable` contract.

Consider removing the unnecessary statements to clean the codebase from unused elements.

***Update**: Partially fixed on pull request 5047. The `Election` contract is still inheriting twice the `UsingRegistry` contract.*

[N02] Duplicated imports

Throughout the codebase, duplicated imports have been found:

- Lines 1 to 7 of the `SortedOracles.sol` file declare the Solidity's version used in the `SortedOracles` contract and the dependencies for that contract. However, the same Solidity's version declaration and imports are also being declared again right after that in lines 8 to 21 of the same file.
- Line 17 of the `Attestations.sol` file imports the `ICeloVersionedContract` which was already imported in line 11.

Consider removing duplicated lines of code to improve readability.

***Update**: Fixed on pull requests 5033 and 5047.*

[N03] Non-descriptive function name

In the `Attestations` contract, when someone who has been already identified wants to transfer its `identifier` to a different address, both addresses have to call the `approveTransfer` function after passing the parameters involved between those accounts and a `true` status.

confuse developers interacting with the contract.

Consider encapsulating the approval operation into a separate function and wrapping both functions in a new one with a name describing its behavior in a better way. By doing this, it will be clearer to review that the operations are complete, consistent, and complementary.

Update: *Acknowledged, and will not fix. The cLabs team's response was:*

> If we changed this, it would break backwards compatibility with clients and we feel comfortable with the current name with proper documentation.

[N04] Gas optimization possible

The `ReentrancyGuard` contract can be refactored to be more gas-efficient by triggering gas refunds with the usage of the changes incorporated in the EIP2200.

Consider modifying the current implementation to trigger gas refunds. Although it cannot be imported, consider using the OpenZeppelin's `ReentrancyGuard` contract as a base for the refactor.

Update: *Acknowledged, and will not fix. The cLabs team's response was:*

> Proxy contract storage layout backwards incompatibilities prevent us from making this change for now.

[N05] Lack of outcome validation

Thorought the codebase there are cases in which the outcome of a function call is not being validated:

The `_updateEcdsaPublicKey` and the `_updateBlsPublicKey` functions of the `Validators` contract, which return `true` upon success, are wrapped up inside a `require` statement in the `updateEcdsaPublicKey`, `updateBlsPublicKey`, and the `updatePublicKeys` functions.

these functions have been executed successfully or not.

The same behavior can also be found when calling the `mint` function of the `StableToken` contract. When this function is called from the `_exchange` function of the `Exchange` contract, it is being wrapped in a `require` statement, but when it is called from the `_distributeEpochPaymentsFromSigner` function of the `Validators` contract, the result of this function call is not being checked to be successful.

Consider modifying these function calls so that the result value is being checked througout the entire system.

Update: *Fixed on pull request 5036.*

[N06] Inconsistency in event emission

In the project's codebase, there are two smart contract wallets: the `Multisig` and the `MetaTransactionWallet` contracts.

In the `Multisig` contract, transferring the intrinsic asset to the `fallback` function will emit a `Deposit` event with the amount of assets involved.

However, when a similar transfer is done to the `MetaTransactionWallet` contract using its `fallback` function, the operation will not emit an event notifying how much has been transferred.

Because these two contracts share some similar functionalities with each other, consider emitting a `Deposit` event when transferring assets to the `MetaTransactionWallet` wallet to make it more consistent project wide and reflect state changes in the contract's balance, with the additional effect of allowing applications to subscribe to such changes.

Update: *Fixed on pull request 5250.*

[N07] Inconsistent coding style

The codebase does not always follow a consistent coding style. Some cases identified are the following ones:

`TransferWhitelist` contract has a named return variable, although other functions such as the `getWhitelist` function does not.

- Revert messages do not follow the same style in several parts of the codebase, such as in line 307 and line 326 of the `Attestations.sol` file.
- The `IAttestations` interface has blank lines between some of the function definitions.

Consider fixing these inconsistencies to improve the project's readability. As reference, consider always following the style proposed in Solidity's Style Guide. Taking into consideration how much value a consistent coding style adds to the project's readability, enforcing a standard coding style with help of linter tools such as Solhint is recommended.

***Update**: Partially fixed on pull requests 4958 and 5032. Revert messages do not reflect a consistent use of capital letters.*

[N08] FixidityLib created as an external library

The FixidityLib library is meant to be used as an internal library by the cLabs team.

Even though this contract is not intended to be used externally nor with a proxy, the external `getVersionNumber` function that keeps a record of the contract's version was added to it and a `Proxy` contract was created for it.

Consider deleting the `getVersionNumber` function from the `FixidityLib` library and removing the `FixidityLibProxy` contract from the code base.

[N09] Function's parameter is not easy to obtain

The `removeSlasher` function from the `LockedGold` contract allows the owner of the contract to remove a slasher from the whitelist of approved slashing addresses.

To do so, the function asks for the `slasherIdentifier` and the `index` of the same identifier in the `slashingWhiteList` array. However, that index is not provided as an output along with the identifier when a slasher is added with the `addSlasher` function, so the way to get it is by retrieving the entire array and find the identifier among all elements.

***Update**: Acknowledged, and will not fix. The team's response was:*

> We typically opt for gas efficiency and prefer to defer arguments to client code where computation is significantly less expensive. The `removeSlasher` function shuffles indices, making a log of the index less useful.

[N10] Indistinguishable functionality of the fallback function

In the codebase, there are multiple times in which the fallback function is used to receive CELO into the contract, such as in the `MetaTransactionWallet`, the `Governance`, or the `ReleaseGold` contracts.

However, it is usually recommended to define a `receive` function as well to distinguish asset transfers from interface confusions. As the Celo protocol is mostly written in Solidity version 0.5.3, this function should be created with the `function` keyword instead of using the available one in newer versions of Solidity.

Consider adding an extra function that receives the CELO asset when possible to make the distinction between correct and erroneous transactions.

***Update**: Acknowledged, and will not fix. The cLabs team's response was:*

> We will wait until we upgrade to solc 0.6.0 to start using the receive functionality. Making it a regular function would mean it would have to behave differently than this new feature, so we prefer to integrate this on a version upgrade.

[N11] TODOs in code

There are `TODO` comments in the code base that should be removed and instead tracked in the project's issues backlog. See for example:

- line 349 of the `Proposals.sol` file.
- line 750 of the `Attestations.sol` file.

Having well described `TODO` comments will make the process of tracking and solving them easier. Without that detailed information, these comments might tend to diminish important

Consider either removing the `TODO` comments from the codebase, or updating the `TODO` comments, adding a brief description of the pending task to perform, and a link to the corresponding issue in the project's backlog. In addition to this latest suggestion, consider adding a signature and a timestamp for completeness, for example:

```
// TODO: Move to uint128 if gas savings are significant enough.
// https://github.com/celo-org/celo-monorepo/issues/1338
// --asaj - 20200117
```

Update*: Partially fixed on pull request 5034. There are still places with `TODO` comments in the code.*

[N12] Undocumented protocol changes

There are two specific changes that were not documented and could generate misalignment on how the cLab team thinks the protocol works and the current functioning of the system.

- The `initialize` function of the `ReleaseGold` contract is no longer validating if the release schedule end time is in the future, because of the removal of this `require` statement. This could allow the smart contract to be initialized with a release schedule that ends in the past.
- The `setSlashableDowntime` function is no longer validating whether the `interval` is shorter than the epoch size. This could cause that the incentives, for the correct functioning of the blockchain, may fail as it could allow validators to be offline for a full epoch and still gain rewards, rewards that may be bigger than the penalty for being offline during the same epoch.

Consider analyzing if these changes could introduce unexpected results and documenting the rationale of removing these validations, explaining how the risks associated with this changes will be mitigated.

Update*: Acknowledged, and will not fix. The cLabs team's response was:*

> ReleaseGold changes are not being applied to already deployed contracts, so there is little risk here.

> slashing multiplier which is halved each time they are slashed that also contributes directly to the reward calculation. See the Validators contract payment calculation for details.

[N13] Unused code

There are two `IValidators.sol` contracts throughout the codebase, one in the `common` folder and the other one in the `governance` folder.

Consider deleting the one in the `common` folder as it is not being used.

***Update**: Fixed on pull request 4958.*

No critical or high severity issues were found. Lower severity recommendations were made to improve the project's overall quality and robustness. Overall we found the code to be very clean and well-organized, and it is notable that the new code takes into consideration recommendations from previous phases.

After a fourth phase of auditing, the cLabs team asked us to review and audit the recent changes in the smart contracts from their protocol.

Scope

The original commit in scope is `af49272ef57a077ee6e704aad76d506c039798d3`. To ensure that our audit process is complete and no unaudited code was added to the codebase, OpenZeppelin audited the difference between the commit in scope and the commit in scope for the previous phase of our engagement with the cLabs team, commit `db2e561e1f13220a98743a7999818e48a5089d13`.

The changes in scope were the ones to the following files between the mentioned commits:

```
packages/protocol/contracts/common/FixidityLib.sol
packages/protocol/contracts/common/MetaTransactionWallet.sol
packages/protocol/contracts/common/MetaTransactionWalletDeployer.sol
```

`Deployer.sol`

`packages/protocol/contracts/common/proxies/MetaTransactionWalletDeployerProxy.sol`

`packages/protocol/contracts/common/proxies/MetaTransactionWalletProxy.sol`

`packages/protocol/contracts/governance/SlasherUtil.sol`

`packages/protocol/contracts/stability/Exchange.sol`

Outside of the scope for this phase are the unchanged parts of the contracts that we audited in previous phases. Also, tests and mock contracts were not part of the scope.

Overview of the changes

For this audit round, the cLabs team made changes to their versioning system, removing it from non-standalone libraries. A new contract called `MetaTransactionWalletDeployer` was added, which is in charge of deploying `MetaTransactionWalletProxy` instances by trusted parties. The `Signatures` library was improved to handle specific use cases related to EIP712, and different proxy contracts were added for the `MetaTransactionWallet` and `MetaTransactionWalletDeployer` contracts.

Vulnerabilities

Below, we list all vulnerabilities identified in the Celo contracts.

Critical

None.

High

None.

Medium

[M01] Deployment can be frontrun

`MetaTransactionWalletProxy` instance and assign it to the same owner as the original transaction.

This has two different outcomes:

- The `owner` address can no longer be set as the owner of another address due to this `require` statement
- The attacker can call the proxy's `_setAndInitializeImplementation` function with any calldata they want by using the `initCallData` parameter, which will lead to the execution of a `delegatecall` from the `MetaTransactionWalletProxy` context, resulting in the possibility of dynamically loading code from a malicious wallet implementation at runtime.

We can conclude that using a MetaTransactionWallet deployed by a malicious third party via the `MetaTransactionWalletDeployer` is not safe.

Consider modifying the `deploy` function to use an implementation address saved in storage by the owner of the `MetaTransactionWalletDeployer` contract and not using this value from untrusted sources such as the parameters of the function.

Update: *Fixed in PR#5683. The `MetaTransationWalletDeployer` contract has been greatly simplified. Now the deployment of `MetaTransactionWalletProxy` proxy contracts is permissionless and does not depend on the deployer contract's state.*

Low

[L01] Duplicated transfer of ownership

In the `initialize` function of the `MetaTransactionWallet` contract the internal `_transferOwnership` function is first called with the `msg.sender` as its parameter and then it is called to assign the ownership to itself. This complexity makes the transaction less gas-efficient by writing two times to storage.

the amount of gas used by this function.

Update: *Fixed in PR#5943. The `initialize` function no longer transfers ownership twice.*

[L02] Function visibility

For certain functions that are only called from outside the smart contract, using function visibility `public` can make the gas costs greater than using the visibility `external`. This is due to the fact that Solidity copies arguments to memory in `public` functions, whereas arguments are accessed directly in `calldata` in `external` functions. A number of functions in the `MetaTransactionWallet` contract have `public` visibility whilst only being called externally. Some examples of this can be found below:

- The `getMetaTransactionDigest` function.
- The `getMetaTransactionSigner` function.

Consider changing the visibility of `public` functions to `external` if they are only accessed externally.

Update: *Fixed in PR#5944. The development team correctly pointed out that the `getMetaTransactionSigner` function is being used internally by the contract, so it remains `public`.*

[L03] Interfaces don’t match with their implementations

Interfaces in the codebase are not fully matching their implementation, defining only a part of the functionalities implemented in their respectives contracts some of the cases and presenting differences in other cases. Some of them are:

- The `IMetaTransactionWalletDeployer` interface is missing the `initialize` and the `getVersionNumber` functions.
- The `IMetaTransactionWalletDeployer` interface declares a `wallets` function which is not implemented in the implementation contract.
- The `IMetaTransactionWallet` interface declares the third parameter of the `executeTransaction` function, the third parameter of the `getMetaTransactionDigest` function and the third parameter of the

`getMetaTransactionSigner` use the `memory` location for these parameters.

These mismatches between the interface and its implementation are confusing. It can mean that there are problems with the modeling of the system or that the updates to the implementations and interfaces got out of sync.

Consider modifying these, and all other mismatches between interfaces and implementations, so there is one basic implementation that directly matches the interface specification. Alternatively, if there are reasons for these functions to be only in the implementations, for each case consider to add a new contract that inherits from the basic implementation, to move the extra functions to this contract, and to add to its docstrings the reason for extending the interface.

Update: *For the first point, the development decided not to implement our suggestion following their internal conventions. For the second point, the `wallets` function has been removed from the contract and corresponding interface in PR#5683. For the third point, the development team correctly pointed out that functions in interfaces with `external` visiblity cannot include parameters declared as `memory`.*

Notes

[N01] Unnecessary imports

In the `MetaTransactionWalletDeployer` contract, there are several unnecessary import statements:

- The `SafeMath.sol` file library is being imported but none of its functions are being used throughout the contract.
- The `BytesLib.sol` file library is being imported but none of its functions are being used throughout the contract.
- The `MetaTransactionWallet.sol file` is being imported and not used.

Consider removing the unnecessary statements to clean the codebase from unused elements.

Update: *Fixed in PR#5966.*

notable that the new code takes into consideration recommendations from previous phases.

After a fifth phase of auditing, the cLabs team asked us to review and audit the recent changes in the smart contracts and verification scripts of their protocol.

Scope

The audited commit for this phase was `0e876e5cad0e93f5dd1fff853f4be3c9a0e5c2a7` of the Celo Monorepo. This commit has associated tag `celo-core-contracts-v2.pre-audit`. To ensure that the audit process was complete and no unaudited code was added to the contracts, OpenZeppelin audited the difference between this commit and the previous phase's audited commit, `af49272ef57a077ee6e704aad76d506c039798d3`.

In addition to the changes to the contracts, audits were performed for entire files within the `packages/protocol/scripts` and `packages/protocol/lib` directories. These files are outlined below.

As a result of this audit round, the cLabs team has fixed or acknowledged the issues reported in individual pull requests. These pull requests have been merged to produce the final auditted commit `1ffaa2863c39485a7cfa0a2ddf7f43a62b0f1661`, making it the newest release candidate commit. This newest release candidate commit has associated tags `celo-core-contracts-v2.rc0` and `celo-core-contracts-v2.alfajores`.

The changed contracts, in which the diff between commit `af49272ef57a077ee6e704aad76d506c039798d3` and `0e876e5cad0e93f5dd1fff853f4be3c9a0e5c2a7` was audited, were:

`packages/protocol/contracts/stability/SortedOracles.sol`
`packages/protocol/contracts/common/Accounts.sol`
`packages/protocol/contracts/common/MetaTransactionWalletDeployer.sol`
`packages/protocol/contracts/common/MetaTransactionWallet.sol`
`packages/protocol/contracts/common/interfaces/IMetaTransactionWallet`

`packages/protocol/contracts/governance/proxies/ProposalsProxy.sol`

`packages/protocol/contracts/identity/Attestations.sol`

`packages/protocol/contracts/identity/Escrow.sol`

The new scripts which were audited in full were:

`packages/protocol/scripts/bash/verify-release.sh`

`packages/protocol/scripts/bash/verify-deployed.sh`

`packages/protocol/scripts/bash/check-versions.sh`

`packages/protocol/scripts/bash/make-release.sh`

`packages/protocol/scripts/bash/release-on-devchain.sh`

`packages/protocol/scripts/truffle/make-release.ts`

`packages/protocol/scripts/truffle/verify-bytecode.ts`

`packages/protocol/lib/compatibility/ast-code.ts`

`packages/protocol/lib/compatibility/verify-bytecode.ts`

`packages/protocol/lib/compatibility/ast-layout.ts`

`packages/protocol/lib/compatibility/version.ts`

`packages/protocol/lib/compatibility/report.ts`

`packages/protocol/lib/compatibility/change.ts`

`packages/protocol/lib/compatibility/ast-version.ts`

`packages/protocol/lib/compatibility/utils.ts`

`packages/protocol/lib/compatibility/internal.ts`

`packages/protocol/lib/compatibility/library-linking.ts`

`packages/protocol/lib/compatibility/categorizer.ts`

`packages/protocol/lib/contract-dependencies.ts`

`packages/protocol/lib/proxy-utils.ts`

`packages/protocol/lib/registry-utils.ts`

Anything not listed above was considered outside of the scope for this audit. Note that while there may be some references to out-of-scope files in this report, these files should not be considered as audited.

Overview of the changes

functionality or were previously verified in the last auditing round. Additionally, many contracts, such as `AddressLinkedListProxy.sol` and `AddressSortedLinkedListProxy.sol` were deleted.

The scripts which were audited allow users to verify the contents of protocol upgrades, as well as which version of Celo's core contracts is currently deployed. Users can follow the Release Process outlined in the celo docs to verify deployed core contract versions, verify release candidates, and deploy release candidates to various networks.

Vulnerabilities

Below, we list all vulnerabilities found in this audit phase of the Celo codebase.

Update: *The cLabs team has fixed in individual pull requests the issues we reported. We refer to these updates in their corresponding issues. Our analysis of the mitigatations acknowledges that all these pull requests have been merged to produce the final audited commit for this phase,* `1ffaa2863c39485a7cfa0a2ddf7f43a62b0f1661`.

Critical

None.

High

None.

Medium

[M01] Metadata stripping regex may mismatch

Within `bytecode.ts`, the regex patterns on lines 12 and 14 are used to match contract metadata and strip it out. Contract metadata exists at the end of a contract. However, since these expressions match to a string of any length and at any location within the bytecode, they could potentially match to some string which is not the metadata.

matches and only assume the final one is the contract metadata.

Update: *Fixed in PR#6382.*

Low

[L01] Celo registry address defined multiple times

In the codebase, the celo registry address (`0x000000000000000000000000000000000000ce10`) is used multiple times and defined multiple times. For instance, it is defined as a `const` in `registry-utils.ts`. The value is also used in `deploy_release_contracts.ts` and defined as a new `const` in `make-release.ts`.

If a change is ever made to one instance of this value, it may not be reflected across the entire codebase. Consider referencing only a single instance of the value across all files. This will make the codebase more cohesive and reduce the surface for developer error.

Update: *Fixed in PR#6381.*

[L02] Logs are incomplete and overwritten with each script invocation

Many bash scripts in the `scripts/bash` folder use a specific logging implementation that overwrites each script's corresponding log with incomplete logging. For example, the `verify-deployed.sh` script writes to the `$LOGFILE` with both `stderr` and `stdout` from many steps in the script. But, by using the `2>` operator directly with `$LOGFILE`, this particular logging implementation overwrites the `$LOGFILE` with each step logged in the script. Elsewhere, in `check-versions.sh`, lines 47 and 51 write `stdout` directly to `$LOGFILE`, overwriting its contents.

Consider combining `tee` and the append operator (`>>`) to form complete logs of `stderr` and `stdout`.

Update: *Fixed in PR#6379.*

spans around 100 lines and has four levels of nested conditional blocks. This affects readability and maintainability of the code.

Consider refactoring long and complex functions. Blocks of code that require comments can be extracted as internal helper functions with descriptive names. The same can be done with blocks of code inside conditionals. Consider following the top-to-bottom definition of functions and the stepdown rule from the Clean Code book to define every public API function followed by their lower-level helpers, which allows reviewers to easily dig into the implementation details that they care about. Consider measuring the cyclomatic complexity of the code and keeping it low.

Update: *Fixed in PR#6408.*

[L04] Remove commented out code

On lines 29 and 30 of `proxy-utils.ts`, there exists a comment suggesting that line 30 should be un-commented for future use.

The `verifyProxyStorageProof` function checks that the storage trie of `proxy` only contains `owner` in the correct position. If lines 29 and 30 are uncommented, the function will then check that the storage trie contains `owner` and `implementation`, and will return `false` if not.

When `verifyProxyStorageProof` is called within `verify-bytecode.ts`, it is with the understanding that the proxy should be deployed (`owner` is set) but the proxy should NOT have an implementation set. If the mentioned lines are uncommented, it will break the current usage of `verifyProxyStorageProof`, and the code will need significant changes to function correctly again.

If it is desired to have a function which includes the code on line 30 of `proxy-utils.ts`, consider implementing it as a separate function with a new name. Alternatively, consider removing the lines, or adding comments to specify the conditions under which to implement the mentioned code.

Update: *Fixed in PR#6491*

converts each version component from `Buffer` to a hex `string` representation to return a `ContractVersion` object using its `fromString` method. The `fromString` method throws within the call to `isValid` if the conversion of `versionComponent` using `Number` is `NaN`. The `Number` wrapper object returns `NaN` on string representations of integers containing non-decimal characters.

This means that checking backward compatibility will throw for most versions having a component greater than 9, since non-decimal hexadecimal digits such as `a` or `f` may be present in the hex representation of a version component.

Consider forming a string of joined decimal representations of the components to be input of the `fromString` method.

Update: *Fixed in PR#6487*

Notes

[N01] Commented out code

In the `release-on-devchain.sh` bash script, there is a commented out test. However, there is not enough context on why this line has been discarded, thus providing developers with little to no value at all.

As the purpose of this line is unclear and may confuse future developers and external contributors, consider removing it from the codebase. If it is to provide alternate implementation options, consider extracting it to a separate document where a deeper and more thorough explanation could be included.

Update: *Fixed in PR#6375.*

[N02] Naming issues

To favor explicitness and readability, some variables may benefit from better naming. Our suggestions are:

- `OWNER_POSITION` to `ADMIN_SLOT`.

`openzeppelin-upgrades` module.

Update: *Acknowledged, and will not fix. cLabs' statement for this issue:*

> The Celo Proxy was developed before EIP-1967, and, for better or worse, utilizes different naming conventions (e.g. "owner" instead of "admin", "position" instead of "slot"). Changing these now would require changing the names of some functions (e.g. `_getOwner` to `_getAdmin`) which would change the bytecode generated by compiling `Proxy.sol`, making offchain tooling for verifying a contracts system deployment more complex. Additionally, it is now infeasible to replace some of the already deployed Proxies (e.g. StableToken, GoldToken), so the convention between the currently deployed Proxies and any newly deployed ones with the suggested changes would differ, increasing complexity and developer confusion.

[N03] TODOs in code

There are "TODO" comments in the code base that should be tracked in the project's issues backlog.

On line 38 of `verify-deployed.sh`, there is a "TODO" comment. Additionally, on line 30 and line 37 of `registry-utils.ts`, there are "TODO" comments.

During development, having well described "TODO" comments will make the process of tracking and solving them easier. Without that information, these comments might tend to rot and important information for the security of the system might be forgotten by the time it is released to production.

These "TODO" comments should at least have a brief description of the task pending to do, and a link to the corresponding issue in the project repository. Adding information about the assigned developer or the user which created the "TODO", and a timestamp, are similarly helpful.

Consider updating these "TODO"s to add detail, or moving the "TODO"s to the project's issues backlog.

Update: *Fixed in PR#6409.*

[N04] Typos

Consider correcting the typos mentioned above.

Update: *Fixed in PR#6377*

[N05] Unused `OPTARGS`

In both the `release-on-devchain.sh` and `verify-deployed.sh` bash scripts, the `r` `OPTARG` is never used and should therefore be removed.

Consider either removing the `r` `OPTARG` or implementing a use for it to improve the code's readability and simplify the codebase.

Update: *Fixed in PR#6375.*

[N06] Unused variables

In the `release-on-devchain.sh` bash script, the variables `NETWORK`, and `FORNO` are never used and should therefore be removed.

Consider implementing a use for these variables, or removing them, to increase code readability and simplify the codebase.

Update: *Fixed in PR#6375.*

[N07] Reliance on unmaintained Python version

Python 2 is a requirement for packages installed during the `yarn` build process of this project.

The Python 2.X end of life was January 1, 2020, meaning it is no longer officially maintained.

Consider changing dependency packages to versions that rely solely on Python3.

Update: *Acknowledged, and will fix later. cLabs' statement for this issue:*

> `node-gyp` is an NPM package that is depended on by other packages in the celo-monorepo. `node-gyp` is known to rely on Python 2.7, there is an issue tracking it. `nodejs/node-`

> bearing on the [illegible] package, we won't be addressing this for this release.

Conclusion

No critical issues and one medium severity issue were found. Suggestions were made to improve the readability and overall health of the codebase, as well as address uncommon but possible operating conditions which could result in error.

Update: *The medium severity issue, and many of the lesser issues were fixed or acknowledged by the cLabs team.*

Introduction

After a sixth phase of auditing, the cLabs team asked us to review and audit the recent changes in the smart contracts and scripts of their protocol.

Scope

The audited commit for this phase was `6b5143a142f4715a9b8bc428b1cf391eec414ec8` of the Celo Monorepo. This commit has associated tag `celo-core-contracts-v3.pre-audit`. To ensure that the audit process was complete and no unaudited code was added to the contracts, we audited the difference between this commit and the previous phase's audited commit, `1ffaa2863c39485a7cfa0a2ddf7f43a62b0f1661` (tagged `celo-core-contracts-v2.rc0` and `celo-core-contracts-v2.alfajores`), that also contains fixes from the previous audit round.

In addition to the changes to the contracts, changes to previously audited scripts have been taken into account.

The changed files, in which the diff between commit `6b5143a142f4715a9b8bc428b1cf391eec414ec8` and `1ffaa2863c39485a7cfa0a2ddf7f43a62b0f1661` was audited, were:

`packages/protocol/contracts/governance/test/BlockchainParametersTest.sol`
`packages/protocol/contracts/stability/ExchangeEUR.sol`
`packages/protocol/contracts/stability/Reserve.sol`
`packages/protocol/contracts/stability/StableToken.sol`
`packages/protocol/contracts/stability/StableTokenEUR.sol`
`packages/protocol/contracts/stability/interfaces/IReserve.sol`
`packages/protocol/contracts/stability/proxies/ExchangeEURProxy.sol`
`packages/protocol/contracts/stability/proxies/StableTokenEURProxy.sol`

`packages/protocol/lib/registry-utils.ts`

`packages/protocol/scripts/bash/check-versions.sh`
`packages/protocol/scripts/bash/make-release.sh`
`packages/protocol/scripts/bash/release-on-devchain.sh`
`packages/protocol/scripts/bash/verify-deployed.sh`
`packages/protocol/scripts/bash/verify-release.sh`
`packages/protocol/scripts/truffle/make-release.ts`

Anything not listed above was considered outside of the scope for this audit. Note that while there may be some references to out-of-scope files in this report, these files should not be considered as audited.

Overview of the changes

For this phase of auditing, the cLabs team introduced changes to the calculation of validator's uptime scores, adding a grace period to cover possible downtimes due to maintenance and restart. It also converted the uptime lookback window (the number of past blocks to look at when searching for a validator signature) into a governance controlled variable which can be now dynamically set.

`Reserve` contract has been updated to support several exchanges at the same time.

Vulnerabilities

Below, we list all vulnerabilities found in this audit phase of the Celo codebase.

Update: *The cLabs team promptly addressed the issues highlighted in this report and individual pull requests have been submitted with the proper fixes. The reference commit that includes all fixes, among other minor changes, is* `64e618b9b856073305dd5748fc04fc772ff72714`.

Critical

None.

High

None.

Medium

[M01] Mint and burn functionalities can break

The `mint` and `burn` functions of the `StableToken` contract make use of the `getAddressForOrDie` of the `Registry` contract. This function will revert if the passed parameter is associated with a null address in the `registry` mapping.
Having said that, one of the changes introduced in the `StableToken` contract is the ability to interact with an `Exchange` contract that is different from the default one. This is represented by the `exchangeRegistryId` state variable, which is set in the `initialize` function.

The `initialize` function doesn't check whether the `exchangeIdentifier` parameter passed as input parameter is a non-null value or if it corresponds to a valid address in the `registry` mapping of the `Registry` contract.

If an incorrect or null value is set:

modifier.

To avoid initializing the `StableToken` contract with incorrect values that could potentially break some functionalities, consider implementing the necessary checks in the `exchangeIdentifier` parameters before setting it in the `exchangeRegistryId` variable.

Update: *The cLabs team decided to not fix this for the time being. In their words: "The risk here is only on `StableTokenEUR.sol` and `ExchangeEUR.sol` not working after CR3, and these contracts won't be active until subsequent proposals regardless, at which point we will make fixes. In fact, we are **intentionally** providing an invalid address via parameters".*

Low

[L01] Grace period is hardcoded

The `grace_period` variable of the `calculateEpochScore` function in the `Validators` contract is hardcoded and it's not possible to set it again to another value once the contract is deployed.

To improve flexibility and efficiency of the code base, consider implementing it in such a way that the governance can change its value without re-deploying the contract again.

Update: *Fixed in PR#6987.*

[L02] Sequential calls to `removeExchangeSpender` can revert

When sequential calls to the `removeExchangeSpender` function of the `Reserve` contract are made with `spender` and `index` valid in regard to their respective function calls, the execution of one transaction can shift the target `index` of the remaining transaction. This way, the remaining transaction can revert either at line 337 or at line 338 where the `index` can either be an invalid index of the `exchangeSpenderAddresses` array or the `spender` can be shifted to another index by the first function call. This can result in wasted gas, and increased complexity in developing a client that would safeguard against this conflict of otherwise valid function calls.

Update: *Not fixed. In the words of the cLabs team: "If its always restricted to Governance then the worst that can happen is that the proposal construction isn't aware of how the indices work, the proposal miraculously passes, and the proposal fails to execute."*

Notes

[N01] Lack of input validation

The `addExchangeSpender` function in the `Reserve` contract does not check that the `spender` is not the zero address.

Even though this issue does not pose a security risk, the lack of validation on parameters may result in erroneous transactions considering that some clients may default to sending null parameters if none are specified.

Consider adding a require statement to validate the `spender` input.

Update: *Fixed in PR#6989.*

[N02] Function behaves as `modifier`

The `isAllowedToSpendExchange` function in the `Reserve` contract behaves as a `modifier` in its intended use. This can be seen where it restricts who can make transfers in the `transferExchangeGold` function.

Even though this issue does not pose a security risk, the function can better realize its intended use as a `modifier` which adds to the project's readability. Consider refactoring or wrapping `isAllowedToSpendExchange` so it can be used as a `modifier`.

Update: *Fixed in PR#6988.*

[N03] Inconsistent style

In the `removeSpender` and `removeExchangeSpender` functions of the `Reserve` contract, an inconsistent style is used to remove the key-value pair `spender`:`true` from their respective `*Spender` mappings (`isSpender` and `isExchangeSpender`). For in the

Taking into consideration how much value a consistent coding style adds to the project's readability, enforcing a standard coding style is recommended.

Update: *Fixed in PR#7049.*

[N04] Missing docstrings

The `isAllowedToSpendExchange` function of the `Reserve` contract is missing information about its purposes, parameters and return values.

Consider thoroughly documenting all functions (and their parameters) that are part of the contracts' public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Fixed in PR#7051.*

Conclusion

One medium and two low severity issues were found. Additional comments were made with respect to code quality as well as other minor notes. Overall we found that the release within this scope has focused and clean changes addressing well-documented improvements to the protocol.

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