PUBLIC

Code Assessment

of the Liquity V2 Governance Smart Contracts

January 22, 2025

Produced for

Liquity

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Liquity Team,

Thank you for trusting us to help Liquity with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Liquity V2 Governance according to Scope to support you in forming an opinion on their security risks.

Liquity implements a governance contract that distributes the incoming revenues based on the votes from users that have a stake in the system. A set of contracts are also provided to simplify the development of smart contracts that serve as proposals in the voting, known as initiatives.

The most critical subjects covered in our audit are precision of arithmetic operations, asset solvency, invariant preservation, functional correctness, and front-running. Several issues of high and critical severity issues were identified in the first two iterations of the codebase, see Resolved Findings. The Governance contract was refactored in Version 3 to mitigate the reported issues by changing the core accounting and placing new restrictions on user operations (always reset all votes before new allocations).

In summary, we find that the codebase provides a satisfactory level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	1
• Code Corrected	1
High-Severity Findings	6
• Code Corrected	5
• Specification Changed	1
Medium-Severity Findings	12
• Code Corrected	7
• Specification Changed	4
• Risk Accepted	1
Low-Severity Findings	24
• Code Corrected	18
• Specification Changed	1
• Code Partially Corrected	2
• Acknowledged	3

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Liquity V2 Governance repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	25 Sep 2024	1add9a44d62faa8d7ab382d5d00ffcf8f522faee	Initial Version
2	21 Oct 2024	168760d750d7051c51f9804b136e1032f7e72f65	Version with Fixes
3	16 Dec 2024	d11e15a11ebdb26c7b297572d9674a7801f50922	Refactored Version
4	16 Jan 2025	e7ed5341f2f54fb9bf89497a7be294c61f21ebe3	Final Version

For the solidity smart contracts, the compiler version `0.8.24` and EVM version `cancun` were chosen.

The following files were in the scope for this review:

```
src/interfaces
src/utils/DoubleLinkedList.sol
src/utils/Math.sol
src/utils/Types.sol
src/BribeInitiative.sol
src/CurveV2GaugeRewards.sol
src/Governance.sol
src/UserProxy.sol
src/UserProxyFactory.sol
```

The following files were added to the scope in **Version 2**:

```
src/utils/EncodingDecodingLib.sol (removed in version 3)
src/utils/UniqueArray.sol
```

The following file was added to the scope in **Version 3**:

```
src/utils/Ownable.sol
```

The following file was added to the scope in **Version 4**:

```
src/utils/VotingPower.sol
```

2.1.1 Excluded from scope

Any file not listed explicitly above is excluded from the scope. Tests, third-party libraries, ERC20 tokens, and other contracts from Liquity, such as LQTY token, LUSD, Staking V1, etc., were not in scope of this review. The external integrations of the system (CurveV2 gauges, bribe tokens, etc.) are out of scope for this review, and are expected to always work correctly and according to specification. Finally, the following files were explicitly excluded from the scope:

```
src/utils/BaseHook.sol (removed in version 4)
src/UniV4Donations.sol (removed in version 4)
src/utils/Multicall.sol (replaced with MultiDelegateCall.sol in version 3)
```

The following files were added in Version 2 and explicitly excluded from the scope:

```
src/utils/SafeCallMinGas.sol
src/ForwardBribe.sol (removed in version 3)
```

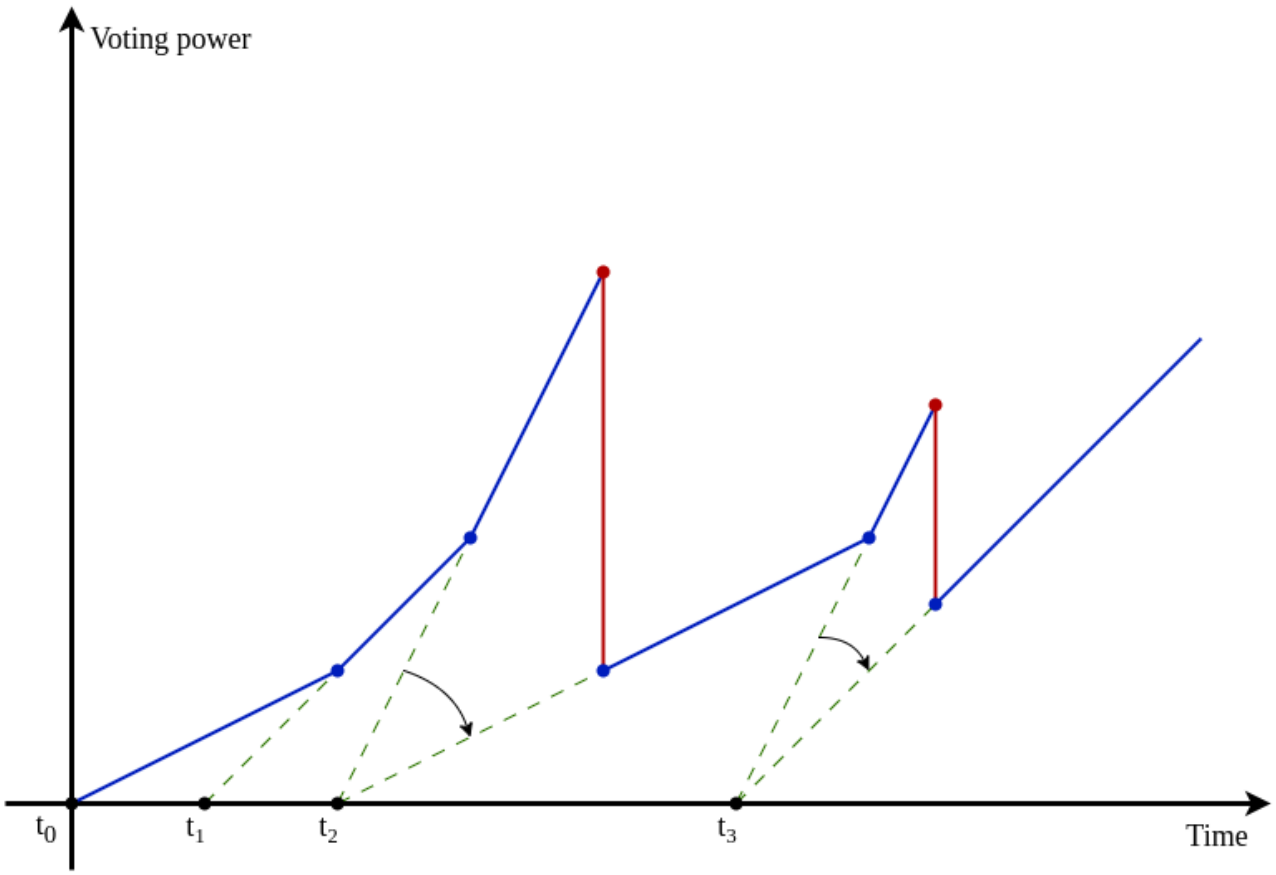
2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

At the end of this report section, we have added subsections for each of the changes according to the versions.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Liquity implements the Governance module of the wider Liquity V2 system. The Governance serves mainly as a voting-based distribution platform of the incoming revenues. One-fourth (known as the "incentive portion") of the total revenue earned in fees by the core protocol, denominated in BOLD (the native stablecoin of LiquityV2), are sent to the Governance contract.
Proposals (known as Initiatives, in LiquityV2 parlance) are encoded as arbitrary addresses, which may implement specific hooks called by the `Governance` contract upon relevant user actions. They are voted upon by the users and, in case they pass (see next sections for more details), they are awarded an appropriate fraction of the total BOLD accrued, based on the votes received.
Users can only vote "YES" or "NO" (referred to as "votes" and "vetoes", respectively) to initiatives. To gain some voting power in this platform, users must be holders of LQTY (the secondary token of Liquity V1), and must stake in LiquityV1 *through the Governance contract of Liquity V2*. The `Governance` contract deploys and manages a `UserProxy` contract for every user. A user's voting power is updated iff he directs his staking/unstaking requests to the `Governance` contract, which performs the due bookkeeping and then relays them to the correct `UserProxy`.
A user's voting power increases with time, following a linear growth, with the staked LQTY amount being the slope. A new amount added to a pre-existing stake starts off with a voting power of 0 (to prevent flash-loan-like abuses): therefore, right after staking, the total voting power stays the same (the line is continuous), but the slope increases. Conversely, when unstaking, there is an immediate drop (discontinuity) in the user's voting power, besides the obvious decrease of the slope. The overall trend followed by the user's voting power over time, as the user stakes and unstakes, is depicted in Figure 1.



A stake behaves like a line $v(t) = m \cdot (t - t_0)$, with the slope m being the LQTY amount, and the intercept t_0 being the "virtual" or "average" staking timestamp: the voting power behaves as if all the LQTY had been staked at once, at this ideal point in time. When a user stakes m_s additional LQTY at time t_s, he is "adding the line" $v_s(t) = m_s \cdot (t - t_s)$ (not shown in the picture) to his current stake, which then becomes $v(t) = m_1 \cdot (t - t_1)$, with $m_1 = m + m_s$ being the total stake, and $t_1 = \frac{m \cdot t_0 + m_s \cdot t_s}{m + m_s}$ being the updated average staking timestamp of the user. The picture shows another staking event, after which the line becomes $v(t) = m_2 \cdot (t - t_2)$. Then, when the user unstakes m_u LQTY at time t_u, he "erodes" part of his pre-existing stake, with its pre-existing "average timestamp" t_2. This means that he "subtracts the line" $v_u(t) = m_u \cdot (t - t_2)$ from his stake, resulting in the new line $v(t) = (m_2 - m_u) \cdot (t - t_2)$: the average staking timestamp remains the same, and the voting power immediately drops by the appropriate fraction m_u/m_2. Such "line arithmetic" patterns are ubiquitous throughout the system.

When voting on an initiative, users allocate part of *their staked LQTY* - rather than their current voting power - for the "YES" or for the "NO". Therefore each initiative has also two associated "stakes", one for the "YES" and one for the "NO", each growing in voting power linearly over time, with different slopes and intercepts. When a user allocates m_a LQTY from his stake that has an average timestamp of t_0, he adds the line $v_a(t) = m_a \cdot (t - t_0)$ to the relevant stake ("YES" or "NO") of the initiative; when de-allocating, he subtracts the same line. For proper accounting, this does *not* modify the "staking line" of the user; instead, an additional per-user variable is tracked, called `allocatedLQTY`, which simply limits the maximum LQTY amount that can be unstaked.

As noted, the "YES" and "NO" votes for an initiative increase over time, and it is these vote counts that decide whether an initiative passes, and how much its payout should be. To keep the outcomes race-free, and to make the system as a whole more "predictable", the operations are clocked in sequential *epochs* of equal one-week duration. The system state is snapshot at epoch boundaries, providing an unambiguous reference point to decide on initiative results.

Given the full system state at the end of epoch *x* (total BOLD accrued for payout + number of "YES" and "NO" votes for every initiative), an initiative can be in one of three "states", during all of epoch *x+1*:

1. Claimable. The initiative has more "YES" than "NO" votes, and the "YES" votes are above a threshold, which is the max of

    1. a fraction of 3% (lowered to 2% in Version 3) of the total number of "YES" votes
    2. the number of votes required to achieve a minimum payout of 500 BOLD (lowered to 0 in Version 3)

    The initiative can be "claimed" permissionlessly, which causes the appropriate fraction of the total BOLD accrued to be transferred to it, and a hook (`onClaimForInitiative()`) to be called. The fraction of the total BOLD to be awarded is computed as the proportion of "YES" votes among all Claimable initiatives: this means that "NO" votes only decide whether an initiative passes, but are then inconsequential for the payout in case it does.
2. Deregisterable. The initiative has more "NO" than "YES" votes, and the "NO" votes are above a threshold, equal to the "YES" threshold. Alternatively, the initiative can also just be "stale" (see next point). The initiative can be de-registered permissionlessly, requiring re-registration to be voted on again.
3. "Limbo". The winning option ("YES" or "NO") does not reach the threshold. If an initiative stays in this state for 4 consecutive epochs then it becomes "Deregisterable".

This "state machine" has been made more explicit and rigorous (and more granular), in Version 2 of the codebase.

Initiatives can be registered permissionlessly. This requires a flat payment of 100 BOLD (raised to 1000 in Version 3), and a voting power above 0.1% of the total "YES" votes (lowered to 0.01% in Version 3), in order to avoid spam.
Users can vote freely on initiatives with part of their staked LQTY at any point during an epoch. A noteworthy observation is that the users' vote allocations are not reset at epoch boundaries; instead, they persist until they are explicitly de-allocated. In particular, an initiative can be claimed more than once, across several epochs, if it keeps meeting the abovementioned requirements.

Almost all of the system's functionality is implemented in the `Governance` contract (which also acts as the `UserProxyFactory`); the integration with Liquity V1's staking contract is handled by the `UserProxy`. Liquity also offers two example initiative implementations - one integrating with CurveV2 and one with UniswapV4 - both deriving from a base `BribeInitiative` contract.
In what follows, we detail the workings of the main contract functionalities.

2.2.1 UserProxy

The `UserProxy` contract is the address effectively holding a LQTY stake in Liquity V1, in lieu of the user; all of its state-modifying functions can only be called by the `Governance` contract, which invokes it in response to user actions.
This will be deployed as a per-user minimal clone of a "master" implementation (see next section). The clone does not have any state variables, nor immutables of its own: in particular, it does not "know" which user address it is acting on behalf of, so the `Governance` needs to pass this address every time as a function argument.
Its functions are:

1. `stake()`: Pulls LQTY funds from the user and stakes them on Liquity V1. This action causes the pending Liquity V1 rewards (possibly accrued through a pre-existing stake) to be sent to the `UserProxy`. These rewards, denominated in ETH and LUSD, are not relayed to the user immediately; instead, they are kept in the `UserProxy` contract, and can be later retrieved via a call to `unstake()`.
2. `stakeViaPermit()`: Same as the previous one, but the LQTY funds are pulled after a call to `permit()` to the LQTY contract; this saves the need for an explicit approval in a separate transaction.
3. `unstake()`: Unstakes the specified LQTY amount from Liquity V1. This action again causes pending Liquity V1 rewards to be pushed to the `UserProxy`: in fact, one can decide to

`unstake()` 0 LQTY in order to simply collect the rewards. The unstaked LQTY amount, together with the entire ETH and LUSD balance of the `UserProxy`, is pushed to the specified recipients. This allows to correctly account for the previously-uncollected rewards from calls to `stake()`.

4. `staked()`: View function (callable by anyone) that returns the current LQTY amount staked by this `UserProxy`.

2.2.2 UserProxyFactory

The `UserProxyFactory` deploys a `UserProxy` "master" implementation contract only once, in its constructor; then, when needed, it deploys minimal per-user Clones of it: these clones are compliant with ERC-1167, and are deployed with the CREATE2 opcode, for deterministic correspondence between user and `UserProxy`; the salt provided to CREATE2 is the user address, left padded with zeros.
There are two functions:

1. `deriveUserProxyAddress()`: A view function that pre-computes the deterministic address of the `UserProxy` clone corresponding to the specified `user` address.
2. `deployUserProxy()`: Deploys the `UserProxy` Clone associated to `msg.sender`. This can be called permissionlessly by anyone to pre-deploy their own `UserProxy`. Still, the `Governance` is the only address that can then interact with that proxy.

2.2.3 Governance

The main contract of the system. Inherits from `UserProxyFactory`, therefore it exposes all the foregoing proxy-deployment functions. It defines functions to interact with the `UserProxy` clones, in order to intercept the user's staking/unstaking requests, and update its own internal bookkeeping before relaying them. Finally, it implements the full voting functionality, with methods to propose initiatives, vote for them (or against them), claim them, and unregister them.
The staking-related functions are:

1. `depositLQTY()`: Deploys the `UserProxy` associated with `msg.sender`, if it does not exist already, then calls `stake()` on it, after updating the user's `averageStakingTimestamp` as described before. The user's total stake (i.e. the slope of the line) does not need to be tracked into an explicit per-user variable, as it is readily available by calling `UserProxy.staked()`.
2. `depositLQTYViaPermit()`: Same as before, but calls into `UserProxy.stakeViaPermit()`. This variant removes the need for a separate LQTY approval but, remarkably, it cannot be used for sponsored transactions: the signer of the Permit struct is checked to be `msg.sender`, who therefore has to pay for the gas fees.
3. `withdrawLQTY()`: Forwards the call to `UserProxy.unstake()`, specifying `msg.sender` as the recipient of the unstaked LQTY and the accrued ETH and LUSD. As was mentioned, a per-user variable `allocatedLQTY` is kept that counts how many LQTY the user has allocated in total: this variable is read here to assert that the user does not unstake too much, touching the LQTY that have been allocated to initiatives.
4. `claimFromStakingV1()`: Calls `UserProxy.unstake()` with a 0 amount, in order to simply collect the rewards. A caller-supplied `_rewardRecipient`, instead of `msg.sender`, is specified as the recipient of ETH and LUSD rewards.

The voting system is more complex. Before diving into its functions, we briefly describe the most important accounting-related state variables that are tracked, and their intended semantics.
As was noted, each initiative has two "stakes" (slope + intercept) associated with it, one for the "YES" and one for the "NO". These are tracked into dedicated per-initiative variables: they are updated every time someone votes on an initiative, allocating LQTY either for the "YES" or for the "NO". We call these "running" variables (as opposed to "snapshot") because they are updated at every user vote.
Again as noted before, not all initiatives end up being Claimable in the next epoch; to compute the payout for those that do, the proportion of "YES" votes should only be taken among Claimable initiatives, in order

to properly distribute the accrued BOLD. To this end, a global "counted YES" stake is tracked, combining all the "YES" stakes of the initiatives that will be Claimable in the next epoch. This is also a running variable, as it is updated "live" at every user vote. This consideration also motivates the introduction of a further per-initiative running variable, a `counted` flag telling whether that initiative is currently contributing towards the global "YES" sum. **This behavior has changed in Version 2 of the codebase: now all initiatives (except those that were already deregistered) count towards the global state. The `counted` flag has been removed.**
At epoch boundaries (the first time any action is taken in an epoch), the global running "YES" stake gets snapshot into a separate global state variable, which will remain unchanged as a firm reference point for all of this new epoch. This state variable actually counts the number of *votes* (not LQTY) that the global running "YES" line evaluates to, at the beginning of this new epoch: we call this variable the `voteSnapshot`. This will be used as the denominator in the claiming function, when computing the fraction of accrued BOLD to be paid out.
The total BOLD balance of the contract is also snapshot into a `boldAccrued` state variable, which will be the total payout available for repartition among initiatives, in this new epoch. Should a Claimable initiative not be claimed, this mechanism allows the unclaimed funds to simply be carried over to the next epoch.
The procedure is similarly repeated for the per-initiative stakes. The first time in an epoch any action is taken on an initiative, its "YES" staking line is evaluated at the epoch's initial timestamp, and gets snapshot into a separate per-initiative variable. If the initiative is not `counted`, though, this `initiativeVoteSnapshot` is instead set to 0 (**starting from Version 2, this is no longer applicable**).
Note that no "iteration over all initiatives" is ever performed in the snapshotting logic: an initiative's snapshot is only taken the first time in an epoch that *that initiative* is touched.

All of this having been clarified, we can now describe the four main voting-related functions.

1. `registerInitiative()`: Allows anyone with sufficient voting power to register an arbitrary address as an initiative. First, it updates the global snapshot, in case it is stale. Then, it asserts that the initiative does not exist already, and that the caller's voting power is large enough. After pulling a flat BOLD payment from `msg.sender`, the initiative is registered in the system, and an optional `onRegisterInitiative()` hook is called on it.
2. `unregisterInitiative()`: Allows anyone to de-register an existing initiative that is in the Deregisterable state. First, it updates the global snapshot, and the initiative's snapshot, in case they are stale. Then, it asserts that the initiative indeed exists and is Deregisterable; as a special case not mentioned before, no initiative can be unregistered if it is younger than 4 epochs. Once these checks pass, it is erased from the system, and an optional `onUnregisterInitiative()` hook is called on it.
3. `allocateLQTY()`: Allows LQTY stakers to batch-vote on several initiatives at once. First, it updates the global snapshot, and the initiatives' snapshots, in case they are stale. Then, for every specified initiative, it adjusts the user's "YES" and "NO" allocation: this modifies the initiative's running "YES" and "NO" stakes, and, depending on the `counted` flag, the global running "YES" stake (**this is now done unconditionally, starting from Version 2**). Through an additional, per-user per-initiative data structure, the function asserts that the user never ends up with positive "YES" and "NO" stakes for the same initiative. Also, an additional check is performed, not specified before: additional votes can only be allocated during the first 6 days of an epoch, so only withdrawals are possible past this cutoff. Finally, the per-user `allocatedLQTY` is updated, and an optional `onAfterAllocateLQTY()` hook is called on the affected initiatives.
4. `claimForInitiative()`: Allows anyone to claim BOLD for an initiative that is in the Claimable state. First, it updates the global snapshot, and the initiative's snapshot, in case they are stale. These snapshots are used to compute the payout: 0 if the initiative is not Claimable, the appropriate fraction of the total BOLD accrued otherwise. The sum is transferred to the initiative, and an optional `onClaimForInitiative()` hook is called on it.

2.2.4 BribeInitiative

An example implementation for a basic bribe-based initiative, provided to developers to be inherited and extended.
Bribes are denominated in two tokens, BOLD and a different, arbitrary ERC20. Anyone with an interest in the initiative can deposit any amount of the two tokens, as bribes for the voters. The bribes are proportionally shared among the users who vote for it: this signal is intercepted by implementing the `onAfterAllocateLQTY()` hook, which allows to have a full picture of everyone's "YES" votes across the epochs.
Bribes are strictly linked to an epoch, to be shared among those who voted for the initiative in that epoch; they never expire, nor are they carried over: a user can come at any point and claim his fair share of the bribes for an epoch arbitrarily far back in the past.
Since the voting platform is such that a user's vote persists without any user action, the initiative's internal data structures should allow to detect the most recent epoch at which the user voted. To suit sparse allocations (across epochs), the initiative stores a per-user linked list, with each item containing an epoch number and a user's final allocation in that epoch. When a user claims for a past epoch *x*, the initiative must determine what is the largest epoch *y* <= *x* that is present in the list. Instead of computing it, it explicitly asks *y* as a hint from the user, which is then verified.
A similar, global linked list is stored to keep track of the total "YES" allocation, across the epochs. As for the per-user list, retrieving the most recent epoch *y* <= *x* for a given epoch *x* requires an explicit hint by the user.
The contract's main functions are:

1. `depositBribe()`: Allows anyone to deposit any amount of BOLD and the second `bribeToken`, to be aggregated to the bribes for an arbitrary *future* epoch.
2. `claimBribes()`: Allows a user to batch-claim for several epochs at once. For every individual claim, the user provides the epoch *x* and the aforementioned hints *y1* <= *x* and *y2* <= *x*, for its own list and for the global list. After verifying the hints (checking that they are indeed the *largest y* <= *x* present in the respective list), the contract simply computes the relevant proportion of the bribes, marks the epoch as claimed by the user, and transfers the bribes.
3. `onAfterAllocateLQTY()` Hook called by (and only callable by) the `Governance`, upon user votes. Updates the user's and the global allocation linked lists.

This implementation does not specify what to do with the BOLD awarded as claims by the `Governance`, and leaves the `onClaimForInitiative()` hook to be implemented by derived contracts.

2.2.5 Changes in Version 2

- The snapshotting logic has been revised and is now cleaner: the functions `snapshotVotes()` and `snapshotVotesForInitiatives()` now simply carry the running state into the relevant snapshot variable, in case it is outdated, without recomputing anything. In particular, the function `snapshotVotesForInitiatives()` no longer computes whether an initiative should be `counted`; the `counted` flag has actually been completely suppressed: all initiatives now contribute to the running `GlobalState`, whether or not they are `CLAIMABLE`.
- An explicit `InitiativeStatus` has been introduced as an enum to represent the 7 states in which an initiative can be, across its lifetime. This effectively models initiatives as finite-state machines; state transitions mostly happen at epoch boundaries, because most of the states are only dependent on snapshot values: the only two exceptions are the transition from `CLAIMABLE` to `CLAIMED` (when an initiative is claimed for, it goes into the `CLAIMED` state for the rest of the epoch) and from `UNREGISTERABLE` to `DISABLED` (when an initiative is unregistered, it goes into the `DISABLED` state permanently). These states are now used as a firm reference point to decide on outcomes of user actions (register/unregister/vote/claim).
- The function `allocateLQTY()` now requires users to reset their allocations before setting the new ones.

- The function `UserProxy.unstake()` now sends all tokens to a single recipient address.
- The `BribeInitiative` contract now distributes bribes based on the user's voting power, rather than their allocated LQTY.

2.2.6 Changes in Version 3

- A new role `owner` with limited privileges is introduced in Governance: `owner` can register at any point after deployment initiatives for epoch `1`. This function can be called only once as the `owner` role is immediately renounced in `registerInitialInitiatives()`. If the initial initiatives are passed as `_initiatives` in constructor, then `owner` should be the deployer and the role is automaticalle renounced.
- The internal representation of staking lines has been modified: instead of tracking slope (LQTY amount) and x-intercept (average timestamp), the code now tracks slope and y-intercept (called `offset`). The previous representation was inconvenient, as the x-intercept (which is ideally a fractional number) was rounded to an integer: this rounding error applied every time two lines were added or subtracted, leading to severe accounting issues. The new representation is more precise, as now no precision is lost upon line arithmetic operations: some minor rounding errors are still present upon allocation and unstaking, but they do not impair the coherence of the system by violating its core invariants.
- The bookkeeping has been made more granular: the per-user, per-initiative allocation struct now tracks all of the line (slope and offset) that was allocated by the user to the initiative. This allows the allocation to be perfectly undone upon resets. Furthermore, the user's state now consists of an "allocated line" and an "unallocated line". Together, they sum to the user's total line, which is no longer tracked explicitly (although the slope can be retrieved by a call to `stakingV1`). The sum over all initiatives of a user's allocations equals his allocated line.
- Unstaking now requires the user to have no allocation on initiatives. This is because there is no easy way to proportionally scale down all of his allocations.
- The functions to stake/stakeViaPermit/unstake now all accept optional parameters signalling whether the pending LiquityV1 rewards should be paid out, and to which address.
- An initiative is considered now in the `WARM-UP` state only during the epoch that it is registered.

2.2.7 Changes in Version 4

- The requirement imposed on users, to have a zero allocation when staking or unstaking, has been lifted.
- The Governance no longer calls an initiative's `onAllocateLQTY()` hook upon NO votes.

2.3 Roles and Trust Model

The following roles can be identified in the system:

1. Deployer. Expected to parametrise the system meaningfully and correctly upon deployment, but otherwise untrusted for the ordinary operations of the system.
2. Users. Completely untrusted.
3. Initiatives. Trusted by the users who vote for them or interact with them in any way. Untrusted by the rest of the system.
4. BribeToken. Trusted by the voters of a `BribeInitiative`. Tokens that transfer less than the specified amount (such as `cUSDCv3`) are not supported. Should the BribeToken be malicious, it could arbitrarily re-interpret the `transfer` functionality to block the claiming of bribes. Still, it

cannot steal the BOLD bribes, alter the recorded allocations, or tamper with the `Governance` in any way.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1

- Vetoes Can Be Removed Past the Epoch Voting Cutoff Risk Accepted

Low-Severity Findings	5

- Excessive Gas Buffer in safeCallWithMinGas() Acknowledged
- Function safeCallWithMinGas() Always Forwards MIN_GAS_TO_HOOK Acknowledged
- Hard Requirements on claimBribes Might Harm User Experience Acknowledged
- Inconsistent Timestamp Used for Registration Threshold Code Partially Corrected Risk Accepted
- Possible Stuck Tokens in BribeInitiative Code Partially Corrected Acknowledged

5.1 Vetoes Can Be Removed Past the Epoch Voting Cutoff

Correctness Medium Version 1 Risk Accepted

CS-V2Gov-012

The function `Governance.allocateLQTY()` asserts that no "YES" votes can be cast past the six-day cutoff. However, the implemented check is not sufficient to cover the case where someone de-allocates some "NO" votes at the last moment.
This effectively nullifies the intended protection for users, who should be given the last 24 hours to block "malicious" proposals. The proposer can allocate some "NO" votes on his own initiative, lulling the users into a false sense of security that the initiative will not pass, only to de-allocate them at the last moment.

Risk accepted:

Liquity is aware of this behavior and replied as follows:

> We believe that the Veto is a first class citizen, so much so that removing a vote is a necessary functionality to make Vetos work. In lack of the ability to remove votes, Vetos would lose their first class position and would make it +EV to vote on something instead of Veto.

5.2 Excessive Gas Buffer in `safeCallWithMinGas()`

Correctness Low Version 2 Acknowledged

CS-V2Gov-037

The function `safeCallWithMinGas()` calls `hasMinGas()`, which deducts a gas buffer from the total gas left, before verifying whether that is enough to forward the specified amount of gas to a call. The buffer is computed to account for the worst-case cost of the `CALL` opcode itself; however, the Governance always triggers this call with `value` set to 0, so `positive_value_cost` and `value_to_empty_account_cost` are zero.

Acknowledged:

Liquity acknowledged the issue and decided not to fix it

5.3 Function `safeCallWithMinGas()` Always Forwards `MIN_GAS_TO_HOOK`

Design Low Version 2 Acknowledged

CS-V2Gov-038

The low-level `call()` in the function `safeCallWithMinGas()` always forwards `_gas` (set to `MIN_GAS_TO_HOOK` by the Governance) to the hook, regardless of whether the user supplied more gas in the tx, in case some complex operations need to be performed in the initiative.

Acknowledged:

Liquity is aware of this behavior and has decided to keep the code unchanged.

5.4 Hard Requirements on claimBribes Might Harm User Experience

Design Low Version 1 Acknowledged

CS-V2Gov-013

The function `claimBribes()` takes as input an array of epochs, with the respective hints, and iterates through them to claim the respective bribes for a user. The internal function `_claimBribe()` implements a set of checks that cause reverts in case the claimable bribe for a user in an epoch is zero. This might harm the user experience, as passing one epoch in which the user has zero bribes causes the whole transaction to revert.

Acknowledged:

Liquity has acknowledged the issue but has decided to keep the code unchanged.

5.5 Inconsistent Timestamp Used for Registration Threshold

Correctness | Low | Version 1 | Code Partially Corrected | Risk Accepted

CS-V2Gov-019

The function `Governance.registerInitiative()` requires that the proposer has a sufficient voting power, measured in terms of the snapshot total number of "YES" votes. However, the user's current voting power is gauged at the current timestamp, rather than the start of the epoch.

Code partially corrected:

The threshold check for user voting power compared to the overall YES voting power has been changed to use `epochStart` as a reference for user's voting power.

Risk Accepted:

Computing user's voting power at `epochStart` is not always correct as user's staking line may change (due to deposits or withdrawals) between the `epochStart` and the time `registerInitiative()` is executed, leading to an underevaluation of the user's voting power. Liquity is aware of this inconsistency and accepts the associated risks.

5.6 Possible Stuck Tokens in BribeInitiative

Design | Low | Version 1 | Code Partially Corrected | Acknowledged

CS-V2Gov-018

The contract `BribeInitiative` implements the function `depositBribe()` that allows anyone to deposit BOLD and another ERC20 token as bribes. These tokens are transferred out only when there are eligible user to claim bribes. However, it is possible that an initiative might not receive any YES vote in an epoch although there is a bribe. In such scenarios, the deposited tokens for that epoch would be stuck and cannot be recovered.

Furthermore, `_claimBribe()` rounds down the bribe amounts, therefore there is some dust that accumulates over time and is not recovered.

Code partially corrected:

As of Version 4, there is no more dust left behind after everyone has claimed: the accounting now keeps track of the "remaining" amounts to be paid out, and the votes "already" used to claim.

Acknowledged:

The possibility of the stuck token if no YES votes are recorded is still present. Liquity replied:

```
We believe the change could create more issues.
In such a scenario it would be in the briber's interest to vote with 1 wei and
get all the bribes back.
```

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	1

- Initiative Votes Can Be Manipulated Code Corrected

High-Severity Findings	6

- Deallocating From DISABLED Initiatives Makes GlobalState Incorrect Code Corrected
- Votes in BribeInitiative Are Evaluated at Current Timestamp Code Corrected
- Initiative Hooks Are Susceptible to Low Gas Attacks Code Corrected
- Typo in the Code Causes Incorrect Calculation Code Corrected
- User's Average Timestamp Can Change Between Allocation and Deallocation of LQTY Specification Changed
- Vote Tallying Is Incorrect Code Corrected

Medium-Severity Findings	11

- Bribes Are Capped at Total Balance Code Corrected
- Conflicting Functionalities for ForwardBribe Specification Changed
- Governance Does Not Enforce Warm-Up Period for Initiatives Specification Changed
- Initiative's Timestamp Can Be Manipulated Specification Changed
- Bribe Accounting Is Based on LQTY Amounts Instead of Votes Code Corrected
- EOAs Are Not Supported as Initiatives Code Corrected
- Initiative Deregisterability Is Inconsistent Code Corrected
- Initiative Is Deregisterable Right After Claiming Code Corrected
- Possible Frontrun in CurveV2GaugeRewards Specification Changed
- Stuck Allocations Code Corrected
- Unsafe Casting on Math Functions Code Corrected

Low-Severity Findings	19

- Possible Underflow in withdrawLQTY Code Corrected
- Rounding Error on Offset Calculation Code Corrected
- Allocation of Zero LQTY Triggers State Updates and Emit Events Code Corrected
- CurveV2GaugeRewards Does Not Check remainder Code Corrected
- Incomplete Events Code Corrected
- Meaningless Checks in BribeInitiative._claimBribe() Specification Changed
- Mismatch of Code With the Error Message Code Corrected
- Remaining ToDos in the Codebase Code Corrected
- Unsafe Casting in lqtyToVotes Code Corrected

- Inconsistent Behavior for Retrieval of Tokens From Staking V1 Code Corrected
- Incorrect Specifications Code Corrected
- Initiative Snapshot Is Never Deleted Code Corrected
- Initiatives Can Be Registered With No Staking in First Weeks Code Corrected
- Possible Simplification of onAfterAllocateLQTY Code Corrected
- Return Value of Ether Transfer Code Corrected
- Special Check in onAfterAllocateLQTY Code Corrected
- Unnecessary Approval Given to LQTYStaking Code Corrected
- Votes for UNREGISTERABLE Initiatives Code Corrected
- _calculateAverageTimestamp() Could Be off by One Second Code Corrected

Informational Findings	7

- Confusing Name for Initiative Status Code Corrected
- Encoding Functionalities Are Unused Code Corrected
- Governance Interface Is Incomplete Code Corrected
- Inconsistent Use of safeTransfer Code Corrected
- Misleading Function Name Code Corrected
- Signed Integers Are Needlessly Wide Code Corrected
- Unused Function in Math Code Corrected

6.1 Initiative Votes Can Be Manipulated

Security Critical Version 1 Code Corrected

CS-V2Gov-001

The Governance contract uses the type `uint88` for accounting the LQTY tokens allocated to an initiative:

```
struct InitiativeState {
    uint88 voteLQTY; // LQTY allocated vouching for the initiative
    uint88 vetoLQTY; // LQTY allocated vetoing the initiative
    ...
}
```

The total supply of LQTY token is capped at 100 million (18 decimals), hence it fits in `uint88`. The function `allocateLQTY()` uses the type `int176` when a user changes its vote regarding an initiative:

```
function allocateLQTY(
    address[] calldata _initiatives,
    int176[] calldata _deltaLQTYVotes,
    int176[] calldata _deltaLQTYVetos
) external;
```

A positive value signifies an additional allocation of either votes or vetoes to an initiative, while a negative value signifies a removal of previously allocated LQTY either as votes, or vetoes.

The function `allocateLQTY()` passes mostly the user input values in `_deltaLQTYVotes` and `_deltaLQTYVetos` to function `add()` which performs an unsafe casting from `int192` to `uint88`, see

Unsafe casting on Math functions. Due to unsafe casting in `add()` any value larger than `2^88-1` or smaller than `-2^88-1` would be truncated, hence having higher bits ignored. For instance, both `2^88` and `-2^88` would be casted to `0`.

The only occurrence where the function `allocateLQTY()` accesses the full value of the input is in the following code snippet:

```
userState.allocatedLQTY = add(userState.allocatedLQTY, deltaLQTYVotes + deltaLQTYVetos);
```

Note that both `deltaLQTYVotes` and `deltaLQTYVetos` are of type `int176`, hence the addition is performed on the full width of the input parameters. Having a value of `2^88` or `-2^88` in either of the inputs, would break the accounting as it would be treated as `0` by the rest of the function.

To exploit the vulnerability, an attacker can perform the following steps:

- Allocate `X` LQTY as votes to `initiative A`.
- Allocate `1` LQTY as vetoes to `initiative B`.
- Allocate `2^88 - X - 1` LQTY as votes to `initiative C`, and simultaneously allocate `-2^88` LQTY as vetoes to `initiative C`.

Attacker can choose the value of `X` depending on its preference for `initiative A` or `initiative C`. After the steps above, the allocation of the user would be accounted as being zero.

Although the attack works by allocating `2^88` LQTY tokens, which is larger than the market cap of LQTY, the attack is feasible because the checks on attacker's balance are enforced after all the allocations are settled, by which point the attacker has already manipulated its allocation to be zero. Therefore, the following check passes as the first condition is satisfied:

```
require(
    userState.allocatedLQTY == 0
        || userState.allocatedLQTY <= uint88(stakingV1.stakes(deriveUserProxyAddress(msg.sender))),
    "Governance: insufficient-or-unallocated-lqty"
);
```

Attacker gets the maximum voting power if it does not have any staked LQTY, hence its `averageStakingTimestamp` is 0, which is treated by the function as having staked since the genesis time.

Code corrected:

The codebase has been revised to avoid the possibility of the unsafe casting which was the root cause for enabling the issue. The following improvements were implemented:

1. The input type of function `allocateLQTY()` for LQTY amounts is changed to `int88` from `int176`.
2. The functions in `Math.sol` are refactored to avoid performing unsafe casting.

6.2 Deallocating From DISABLED Initiatives Makes GlobalState Incorrect

Correctness | High | Version 2 | Code Corrected

CS-V2Gov-030

The `GlobalState` is now the sum of *all* active initiative lines, whether they are `CLAIMABLE` or not; an initiative is only removed from the `GlobalState` upon a call to `unregisterInitiative()`. At that

point, however, users might still have LQTY allocated on that initiative: the function `allocateLQTY()`, therefore, allows users to deallocate LQTY from a `DISABLED` initiative.
However, the function still updates the `GlobalState` in this case, even though the whole `initiativeState` was already discounted when the initiative was unregistered. Specifically, the subtraction of the `prevInitiativeState` does not happen, but the addition of the new `initiativeState` does. This effectively adds the other users' voting lines back into the `GlobalState`, making it incorrect.

Code corrected:

The vulnerability has been mitigated in Version 3. When an initiative is unregistered, the global state is updated to discount the initiative that is removed:

```
function unregisterInitiative(address _initiative) external nonReentrant {
    ...
    state.countedVoteLQTY -= initiativeState.voteLQTY;
    state.countedVoteOffset -= initiativeState.voteOffset;
    ...
}
```

The internal function `_allocateLQTY()` has been updated to avoid updates of `state` when deallocating from a disabled initiative:

```
function _allocateLQTY(...) internal {
    ...
     if (status != InitiativeStatus.DISABLED) {
        ...
        // Remove old initative LQTY and offset from global count
        vars.state.countedVoteLQTY -= vars.prevInitiativeState.voteLQTY;
        vars.state.countedVoteOffset -= vars.prevInitiativeState.voteOffset;
        // Add new initative LQTY and offset to global count
        vars.state.countedVoteLQTY += vars.initiativeState.voteLQTY;
        vars.state.countedVoteOffset += vars.initiativeState.voteOffset;
    }
}
```

6.3 Votes in BribeInitiative Are Evaluated at Current Timestamp

Security High Version 2 Code Corrected

CS-V2Gov-031

The BribeInitiative contract now tracks both the allocated LQTY and the user's timestamp, to distribute bribes according to the effective voting power. However, the voting power is evaluated at `block.timestamp`, instead of the start of the reference epoch, which leads to a wrong proportion being calculated for the user. This can lead to stuck bribes (if the proportions sum to less than 1), or insolvency/stolen funds (if the proportions sum to more than 1).
If, for example, a user has an average age lower than the average age of an initiative that he votes for, he can wait a long time to claim for that epoch, so as to "inflate" his voting power compared to the other voters, and thus his proportion. This could lead to insolvency; however, the (flawed) mitigation described in Bribes are capped at total balance could make things worse: since the total payout is capped at the

BOLD balance of the contract, rather than the total bribe for that epoch, the user could effectively steal BOLD from the contract that were allocated for other purposes (e.g. bribes for other epochs).

Code corrected:

In Version 3 the voting power is now evaluated at the end of the reference epoch.

6.4 Initiative Hooks Are Susceptible to Low Gas Attacks

Security High Version 1 Code Corrected

CS-V2Gov-002

The Governance contract triggers calls into an initiative when a user action changes state that is relevant for an initiative. An initiative can implement the following callback hooks:

```
function onRegisterInitiative(uint16 _atEpoch) external;
function onUnregisterInitiative(uint16 _atEpoch) external;
function onAfterAllocateLQTY(uint16 _currentEpoch, address _user,
    uint88 _voteLQTY, uint88 _vetoLQTY) external;
function onClaimForInitiative(uint16 _claimEpoch, uint256 _bold) external;
```

The initiative can implement custom logic on these callback functions to update its internal accounting for important state updates on Governance, such as vote tracking on `BribeInitiative`.

The Governance contract always triggers the callback hooks inside a `try/catch` block, therefore a failure in the initiative's functions does not bubble up to revert the whole transaction. Given that the external functionalities in Governance are permissionless, an attacker can initiate a call to Governance and pass a low amount of gas which is not sufficient to execute the code in the initiative's callback hooks. After the failed call, 1/64 of the gas will still be available, which will be sufficient to return from the Governance function.

For example, an attacker can trigger `Governance.claimForInitiative()` and pass a limited amount of gas such that the call to `onClaimForInitiative()` reverts due to out-of-gas exception, hence no state gets updated in the initiative.

Code corrected:

In Version 2, hooks are called via the helper function `safeCallWithMinGas()`, which ensures that a minimum gas stipend (fixed to a constant of 350'000) is forwarded to the hook.

6.5 Typo in the Code Causes Incorrect Calculation

Correctness High Version 1 Code Corrected

CS-V2Gov-003

The function `allocateLQTY()`, when discounting the `prevInitiativeState` from the running `GlobalState` (in case it is `counted`), computes the new timestamp based on that of the new `initiativeState`, rather than the `prevInitiativeState`:

```
if (prevInitiativeState.counted == 1) {
    state.countedVoteLQTYAverageTimestamp = _calculateAverageTimestamp(
        state.countedVoteLQTYAverageTimestamp,
        initiativeState.averageStakingTimestampVoteLQTY, // wrong timestamp
        state.countedVoteLQTY,
        state.countedVoteLQTY - prevInitiativeState.voteLQTY
    );
    state.countedVoteLQTY -= prevInitiativeState.voteLQTY;
}
```

This leads to `state.countedVoteLQTYAverageTimestamp` being calculated incorrectly, hence breaking the accounting of the global votes.

Code corrected:

Function `allocateLQTY()` has been refactored in Version 2 and the code snipped shown above has been moved to the internal function `_allocateLQTY()`. The issue has been fixed by passing `prevInitiativeState.averageStakingTimestampVoteLQTY` as second argument to function `_calculateAverageTimestamp()`:

```
if (status != InitiativeStatus.DISABLED) {
    state.countedVoteLQTYAverageTimestamp = _calculateAverageTimestamp(
        ...
        prevInitiativeState.averageStakingTimestampVoteLQTY,
        ...
    );
}
```

6.6 User's Average Timestamp Can Change Between Allocation and Deallocation of LQTY

Security High Version 1 Specification Changed

CS-V2Gov-032

The Governance only tracks one `averageStakingTimestamp` at any given time, for each user. This is used for all of the user's allocations and deallocations of LQTY to/from initiatives. However, this timestamp can change (move forward) when the users adds stake. If this happens before the user deallocates from an initiative, the function `allocateLQTY()` subtracts from the `initiativeState` a different line (same slope, different offset) from the one that was added to it when the user allocated LQTY to that initiative.
The subtracted line has a more recent timestamp (and thus a lower voting power), so this bug can be exploited. For example, a user who already intends to leave the system could allocate all his LQTY on his favourite initiative, then flash-loan a large amount of LQTY and add them to his stake (moving his timestamp very close to the present), then deallocate the LQTY from the initiative, unstake everything and repay the flash-loan. The deallocation will leave a constant offset "on top" of the previous `initiativeState`, roughly equal to his current voting power.

Specification changed:

The accounting of voting power for users have changed in Version 3. Instead of tracking `averageStakingTimestamp` for a user, the updated codebase uses the following struct:

```
struct UserState {
    uint256 unallocatedLQTY;
    uint256 unallocatedOffset;
    uint256 allocatedLQTY;
    uint256 allocatedOffset;
}
```

Furthermore, for each allocation of a user towards an initiative, the following data are stored:

```
struct Allocation {
    uint256 voteLQTY;
    uint256 voteOffset;
    uint256 vetoLQTY;
    uint256 vetoOffset;
    uint256 atEpoch;
}
```

This information allows the accounting to reset correctly when deallocating LQTY from an initiative.

6.7 Vote Tallying Is Incorrect

Correctness High Version 1 Code Corrected

CS-V2Gov-004

The global `votesSnapshot` is taken at the beginning of the epoch, and it stays the same for the whole epoch. It counts the number of "YES" votes, and will be used as the denominator in `claimForInitiative()`, to calculate the payout; it is therefore crucial that the contract's operations preserve the following invariant: the sum of the numerators used by all initiatives that can be claimed during an epoch must equal `votesSnapshot.votes`. This invariant is not maintained: it can happen that the sum of the numerators is smaller than the denominator (in which case not all the `boldAccrued` can be claimed), or worse, it can be larger (in which case the system is insolvent).
There are several individual snippets across the code that concur to this issue, but the root cause can be pinpointed as follows: the function `_snapshotVotesForInitiative()` recomputes whether the initiative is Claimable or not, but in a way such that the result is unrelated to the `counted` flag; it can happen that an initiative is `counted` but it is then not Claimable in the epoch, or worse, it can be not `counted` but still Claimable. The following mock scenario exemplifies this latter case:

Epoch x ends, x+1 begins. Global snapshot: YES vote count = 101

3 votes for initiative 1 -> not counted

98 votes for initiative 2 -> counted

Running global state is only counting initiative 2

Epoch x+1 ends, x+2 begins. Global snapshot: YES vote count = 98

No new votes, just re-snapshot both initiatives

Claim initiative 1. Re-snapshot, now above threshold -> claim 3/98 of the BOLD

Claim initiative 2: Re-snapshot -> claim 98/98 of the BOLD -> insolvency

Here is a list of places in the code that concur in breaking the invariant:

1. The function `_snapshotVotesForInitiative()` recomputes whether an initiative is Claimable, using a newer `votingThreshold`, different from the one used in the previous epoch by `allocateLQTY()` to decide whether the initiative should be `counted`.

2. The function `allocateLQTY()` computes the `votesForInitiative` evaluating the staking line at `block.timestamp`, rather than the beginning of the next epoch. This leads to a different result from the line evaluation performed in `_snapshotVotesForInitiative()` in the next epoch.
3. The function `allocateLQTY()` does not take into account whether the "YES" has a relative majority, to decide whether the initiative should be `counted`, instead only this check is performed: `votesForInitiative >= votingThreshold`.

Code corrected:

In Version 2, the code has been refactored and redesigned. What fixes this issue is a simplification of the logic: all initiatives are now counted, whether or not they will end up being CLAIMABLE. This prevents insolvency as described above; note, however, that it can still happen that the denominator is *larger* than the sum of the numerators (this will happen every time there is a non-claimable initiative), which leads to part of the rewards being not redeemable in the current epoch and getting carried over to the next one. Liquity is aware of this limitation.

6.8 Bribes Are Capped at Total Balance

Correctness Medium Version 2 Code Corrected

CS-V2Gov-033

The function `BribeInitiative.claimBribes()` caps the payout at the total balance of the contract (in `BOLD` and `bribeToken`, respectively), rather than the total bribe allocated for that epoch. This means that potential cases of insolvency are mitigated by giving out tokens that were allocated for other purposes: this can be exploited, in conjunction with the accounting bug described in Votes in BribeInitiative are evaluated at current timestamp.

Code corrected:

The abovementioned accounting bug has been fixed in Version 3, thus removing the possibility for an exploit.

6.9 Conflicting Functionalities for ForwardBribe

Design Medium Version 2 Specification Changed

CS-V2Gov-034

The contract `ForwardBribe` inherits from `BribeInitiative`, and therefore includes all its bribe-payout logic. This conflicts with its additional logic that forwards all funds (including bribes) to an immutable `receiver`.

Specification changed:

The contract `ForwardBribe` has been removed from the codebase in Version 3.

6.10 Governance Does Not Enforce Warm-Up Period for Initiatives

Design Medium Version 2 Specification Changed

CS-V2Gov-035

The function `Governance.getInitiativeState()` does not have a specific return value in case an initiative is at most `REGISTRATION_WARM_UP_PERIOD` epochs old; in fact, this variable is unused in the codebase Version 2. This enables a griefing attack against initiatives that do not receive enough votes in the firts week after they are registered. An attacker can allocate veto power to such initiatives and then unregister them.

Specification changed:

In Version 4, the parameter `REGISTRATION_WARM_UP_PERIOD` has been removed altogether: new initiatives can be vetoed and unregistered the next epoch. Furthermore (already starting form Version 3) the `lastEpochClaim` of a newly-registered initiative is set to the previous epoch, so that the `UNREGISTRATION_AFTER_EPOCHS` period also applies to new initiatives.

6.11 Initiative's Timestamp Can Be Manipulated

Security Medium Version 2 Specification Changed

CS-V2Gov-046

The choice of representing a line using the x-intercept inherently leads to rounding errors, as explained in _calculateAverageTimestamp() could be off by one second, since the x-intercept (the `averageStakingTimestamp`) of a staking line is potentially a fractional number, which the function `_calculateAverageTimestamp()` rounds up to an integer. A peculiar consequence of this is that a user allocating some LQTY on an initiative, and then de-allocating them in the same transaction, does not necessarily leave the initiative's line untouched overall: its timestamp could end up being strictly higher (more recent) than its previous value. This can be exploited by an attacker flash-loaning a large amount of LQTY and doing this repeatedly to all of his competitor initiatives, so as to bring their timestamp as close to the present as possible: this would reduce their voting power (evaluated at the epoch boundary), compared to the attacker's, granting him an unfair share of the total `boldAccrued`.
Here follow the details of the attacks. Suppose an initiative has a staking line $v_0(t) = m(t - t_0)$, and an attacker has a staking line $v_a(t) = m_a(t - t_a)$.
Allocating m_a LQTY on the initiative brings the initiative line to $v_1(t) = m(t - \lceil t_1 \rceil)$ where

$$t_1 = \frac{mt_0 + m_a t_a}{m + m_a}$$

and $\lceil \cdot \rceil$ is the ceiling function. | When the attacker deallocates the same LQTY amount, that brings the initiative line to $v_2(t) = m(t - \lceil t_2 \rceil)$ where

$$t_2 = \frac{(m + m_a)\lceil t_1 \rceil - m_a t_a}{(m + m_a) - m_a}$$

Let us compare t_0 (the initial timestamp of the initiative) with $\lceil t_2 \rceil$ (its final timestamp of the initiative's line). To simplify the analysis, let us actually focus on t_2: applying the ceiling function to it will only make the impact worse.
Noticing that $\lceil x \rceil \geq x \quad \forall x$, and applying this to the formula for t_2, quickly yields

$$t_2 \geq \frac{(m + m_a)t_1 - m_a t_a}{(m + m_a) - m_a} = \frac{mt_0 + m_a t_a - m_a t_a}{(m + m_a) - m_a} = \frac{mt_0}{m} = t_0$$

so the final timestamp will always be at least as high as the initial one. | Let us now use the other inequality of the ceiling function, $\lceil x \rceil < x + 1 \quad \forall x$; notice that this inequality can be arbitrarily tight: it

suffices that the mantissa of x be very small (e.g. 0.0001) for $\lceil x \rceil$ to be very close to $x + 1$. | Applying this to the formula for t_2 yields:

$$t_2 < \frac{(m + m_a)(t_1 + 1) - m_a t_a}{(m + m_a) - m_a} = \frac{mt_0 + m_a t_a + m + m_a - m_a t_a}{(m + m_a) - m_a} = \frac{mt_0 + m + m_a}{m} = t_0 + 1 + \frac{m_a}{m}$$

This means, for example, that an attacker having a stake equal to roughly 30 times that of a "victim" initiative can push forward its timestamp by 30 seconds. He just needs to choose m_a and t_a just right so that the mantissa of t_1 is small.
This procedure can be iterated, multiple times over the same initiative, and over several initiatives (possibly choosing different m_a and t_a every time), all in one transaction, with only one flash loan. This can be repeated at every epoch; the attack does not become more expensive over time, because the victim initiatives never get to age properly if the attacker "brings forward" their timestamp every time, so it will always be ~1 week old every time the attack is performed.

Specification changed:

The accounting has changed in Version 3. The timestamp is not used anymore for the voting line of an initiative, hence the issue is not applicable anymore.

6.12 Bribe Accounting Is Based on LQTY Amounts Instead of Votes

Correctness | Medium | Version 1 | Code Corrected

CS-V2Gov-005

The accounting on `BribeInitiative` is based on the LQTY amounts allocated for an initiative:

```
function onAfterAllocateLQTY(..., uint88 _voteLQTY, uint88 _vetoLQTY) ... {
    ...
    uint88 newVoteLQTY = (_vetoLQTY == 0) ? _voteLQTY : 0;
    ...
    _setLQTYAllocationByUserAtEpoch(_user, _currentEpoch, newVoteLQTY, true);
}
```

Therefore, a user `A` staking 1 LQTY for one year and allocating all its voting power to the initiative, would receive the same bribe as a user `B` staking 1 LQTY for one month and allocating all its voting power to the initiative. This contradicts the intended behavior is to distribute bribes to users according to their voting power. In the scenario described above, user `A` should receive a bribe that is roughly 12x of the bribe received by user `B`.

This design enables an attacker to manipulate the bribe accounting by taking short term loans. The attacker executes the following steps:

- At the last block of epoch `x`, take a loan of LQTY tokens from a 3rd party protocol.
- On the same transaction, stake all LQTY and allocate all voting power to one or more initiatives that pay bribes.
- At the first block of epoch `x+1`, unstake all LQTY and repay the loan and any interest for 12 seconds.
- Claim the bribe.
- Repeat the process for each epoch boundary.

This allows the attacker to receive large amount of bribes that are not proportional to its voting power.

Code corrected:

The function hook `BribeInitiative.onAfterAllocateLQTY()` has been revised in Version 2 to take additional input arguments that enable the calculation of vote contributions per user:

```
function onAfterAllocateLQTY(
    uint16 _currentEpoch,
    address _user,
    IGovernance.UserState calldata _userState,
    IGovernance.Allocation calldata _allocation,
    IGovernance.InitiativeState calldata _initiativeState
) external virtual onlyGovernance
```

Therefore, the bribes are now calculated based on user's votes and the initiative's total votes at a given epoch:

```
...
uint240 totalVotes = governance.lqtyToVotes(...);
if (totalVotes != 0) {
    ...
    uint240 votes = governance.lqtyToVotes(...);
    boldAmount = uint256(bribe.boldAmount) * uint256(votes) / uint256(totalVotes);
    bribeTokenAmount = uint256(bribe.bribeTokenAmount) * uint256(votes) / uint256(totalVotes);
}
```

However, this fix has introduced another vulnerability as described in Votes in BribeInitiative are evaluated at current timestamp

6.13 EOAs Are Not Supported as Initiatives

Correctness | Medium | Version 1 | Code Corrected

CS-V2Gov-006

The specifications of Governance state that an initiative can be any Ethereum address, including externally-owned accounts (EOAs):

```
As Initiatives are assigned to arbitrary addresses, they can be used for any purpose
including EOAs, Multisigs, or smart contracts designed for targetted purposes.
```

The implementation of the function `onRegisterInitiative()` triggers inside a `try/catch` block a hook in the initiative address:

```
try IInitiative(_initiative).onRegisterInitiative(currentEpoch) {} catch {}
```

Because the call into initiative is not expected to return any data, Solidity compiler adds checks to ensure that the called address has non-empty code. This check is executed before the `try/catch` block, which means that if the check fails, the revert is not caught by the `try/catch` logic. Hence, the registration of any EOA as an initiative would revert when triggering `onRegisterInitiative()`.

Code corrected:

In Version 2, hooks are called via the helper function `safeCallWithMinGas()`, which supports EOAs, since it performs a simple low-level call without checking its boolean return value.

6.14 Initiative Deregisterability Is Inconsistent

Design | Medium | Version 1 | Code Corrected

CS-V2Gov-007

The vetoes for an initiative are not snapshot at epoch boundaries. The function `unregisterInitiative()` instead decides whether an initiative is Deregisterable by recomputing the "NO" votes, evaluating the corresponding staking line at `block.timestamp`, thus "inflating" the "NO" votes compared to the "YES", which are taken from the snapshot. In fact, it is possible for an initiative to be at the same time Claimable (because the "YES" had a majority at the epoch start) and Deregisterable (because the "NO" were later recomputed and found to be higher than the snapshot "YES"): the outcome will then depend on the result of a race condition, which the use of epochs is meant to eliminate. Considering that an epoch is 7 days, the inflation in the voting power for 1 LQTY that has been staked for a year is roughly 2% during one epoch (7/365).
On the same note, based on the reasoning exposed in Vote tallying is incorrect, it should never happen that a `counted` initiative is Deregisterable, therefore the final update to the `GlobalState` should never take place.

Code corrected:

A new function `getInitiativeState()` has been added in Version 2 that returns the state of an initiative. This ensures that initiatives are treated consistently by the codebase. An initiative can be in one of the following states:

```
enum InitiativeStatus {
    NONEXISTENT,
    WARM_UP,
    SKIP,
    CLAIMABLE,
    CLAIMED,
    UNREGISTERABLE,
    DISABLED
}
```

6.15 Initiative Is Deregisterable Right After Claiming

Correctness | Medium | Version 1 | Code Corrected

CS-V2Gov-008

The function `claimforInitiative()` resets the number of "YES" votes to 0, to prevent double claiming in the same epoch. However, this has the unintended side effect of giving the "NO" a relative majority, thus possibly making the initiative deregisterable immediately afterwards (if other conditions, like the warmup period, are satisfied).
Additionally, this breaks the assumptions from external callers of the public view function `votesForInitiativeSnapshot()` on the semantics of its result.

Code corrected:

The number of "YES" votes is no longer reset to 0. Instead, an explicit field has been added that stores the last epoch at which the initiative has been claimed: this is now enough to prevent double claiming, and does not give the "NO" a relative majority.

6.16 Possible Frontrun in CurveV2GaugeRewards

Security Medium Version 1 Specification Changed

CS-V2Gov-009

The function `depositIntoGauge()` in `CurveV2GaugeRewards` should be called once in an epoch to move the claimable BOLD tokens from the Governance to a Curve gauge:

```
function depositIntoGauge() external returns (uint256) {
    uint256 amount = governance.claimForInitiative(address(this));
    ...
    gauge.deposit_reward_token(address(bold), amount, duration);
    ...
}
```

However, one can frontrun this call and trigger the execution of `claimForInitiative()` in the Governance contract. In this case the funds will be moved to `CurveV2GaugeRewards` but get stuck. Any subsequent call to `depositIntoGauge()` in the same epoch would deposit zero tokens into the gauge.

Specification changed:

The specifications for the contract `CurveV2GaugeRewards` have changed in Version 2. The external function `depositIntoGauge()` has been removed and the initiative hook `onClaimForInitiative()` automatically forwards the rewards to the Curve gauge.

6.17 Stuck Allocations

Correctness Medium Version 1 Code Corrected

CS-V2Gov-010

After an initiative is unregistered, its variables are erased (i.e. reset to 0). These variables (in particular the registration epoch) are checked in `allocateLQTY()` to be non-zero. Therefore, it is impossible to reclaim votes from an initiative once it has been deregistered, without registering it again, which costs 100 BOLD, and is only effective starting from the next epoch.
An attacker may exploit this to temporarily DOS competitors and effectively reduce the total voting power for this epoch.

Code corrected:

It is now possible to reclaim LQTY from a deregistered initiative.

6.18 Unsafe Casting on Math Functions

Security Medium Version 1 Code Corrected

CS-V2Gov-011

Functions `add()` and `sub()` in `Math.sol` perform unsafe casting when converting a value of a larger type to a smaller type. The function `add()` casts an input value of type `int192` into `uint88` without any range check, hence ignoring the high-order bits of the input value `b`:

```
function add(uint88 a, int192 b) ... {
    if (...) {
        return uint88(a - uint88(uint192(-b)));
    }
    return uint88(a + uint88(uint192(b)));
}
```

Similarly, the function `sub()` performs unsafe casting when converting a value of type `uint256` into a `uint128`.

The unsafe casting in function `add()` enables a severe vulnerability in the accounting of the votes, see Initiative votes can be manipulated.

Code corrected:

The functions have been revised to use a new function `abs()` that avoids the unsafe casting. All the operations are performed on 256-bit integers as of Version 3

6.19 Possible Underflow in withdrawLQTY

Design | Low | Version 3 | Code Corrected

CS-V2Gov-050

Function `withdrawLQTY()` updates user's state with the new offset and unallocated LQTY as follows:

```
// Update the offset tracker
if (_lqtyAmount < userState.unallocatedLQTY) {
    ...
} else {
    ...
    userState.unallocatedOffset = 0;
}

// Update the user's LQTY tracker
userState.unallocatedLQTY -= _lqtyAmount;
```

The function doesn't perform any check that amount of LQTY being withdrawn is less than the user's stake. Furthermore, `LQTYStaking.unstake()` caps the amount to the user's stake. Therefore, if one calls `withdrawLQTY()` with a larger LQTY amount than staked, the subtraction above reverts with an underflow error.

Code corrected:

A sanity check on `_lqtyAmount` has been added in Version 4, with a custom error message in case of too large a value.

6.20 Rounding Error on Offset Calculation

Correctness | Low | Version 3 | Code Corrected

CS-V2Gov-051

Users can allocate their voting power to different initiatives. For each allocation two values are stored: allocated LQTY and the user's offset. Function `allocateLQTY()` rounds down when computing the offset that should be added to an initiative:

```
for (...) {
    absoluteOffsetVotes[x] =
        _absoluteLQTYVotes[x] * int256(userState.unallocatedOffset) / int256(userState.unallocatedLQTY);
    absoluteOffsetVetos[x] =
        _absoluteLQTYVetos[x] * int256(userState.unallocatedOffset) / int256(userState.unallocatedLQTY);
}
```

This means that, in particular, when the user allocates all of his LQTY to many initiatives, he could be left with 0 `unallocatedLQTY` and non-zero `unallocatedOffset`.
The rounding error is accounted in favor of an initiative if user allocates YES votes (against initiative if user allocates NO votes).

Code corrected:

In Version 4, the in-memory copy of the user's vote trackers are updated (decreased) at each iteration of the loop. This removes the abovementioned edge case: in case of a complete allocation, the `unallocatedOffset` will be 0. We highlight that the rounding error is now path dependent, i.e., user can influence for which initiatives the rounding error is in favor.

6.21 Allocation of Zero LQTY Triggers State Updates and Emit Events

Design | Low | Version 2 | Code Corrected

CS-V2Gov-047

Function `allocateLQTY()` executes fully and updates the initiative and global state even if both `_absoluteLQTYVotes` and `_absoluteLQTYVetos` are zeros. Furthermore, the respective events are emitted. Therefore, the function can also be triggered by users that do not have any staking in the governance contract.

Code corrected:

In Version 3, checks are added to require that `_absoluteLQTYVotes` and `_absoluteLQTYVetos` be not all-zeros, as well as that the user have a stake in the system (positive `unallocatedLQTY`).

6.22 CurveV2GaugeRewards Does Not Check `remainder`

Correctness | Low | Version 2 | Code Corrected

CS-V2Gov-036

The function `CurveV2GaugeRewards._depositIntoGauge()` only checks the newly-accrued `amount`, rather than the `total = remainder + amount`, to decide whether to queue or deposit the rewards into the gauge.

Code corrected:

The internal function `_depositIntoGauge()` has been revised to consider both new Bold tokens and the remainder when deciding whether to queue or deposit the rewards into the gauge:

```
function _depositIntoGauge(uint256 amount) internal {
    uint256 total = amount + remainder;

    // For small donations queue them into the contract
    if (total < duration * 1000) {
        remainder += amount;
        return;
    }
    ...
}
```

6.23 Incomplete Events

Design | Low | Version 2 | Code Corrected

CS-V2Gov-040

The internal function `_snapshotVotes()` emits the snapshotted votes and the respective epoch, but it does not include the `boldAccrued`. Similarly, the function `_snapshotVotesForInitiative()` emits the votes recorded for an initiative, but it does not include vetos.

Furthermore, governance does not emit an event with the `success` flag returned from `safeCallWithMinGas()` when an initiative hook is triggered.

Code corrected:

The abovementioned fields have been added to the events in Version 3

6.24 Meaningless Checks in `BribeInitiative._claimBribe()`

Design | Low | Version 2 | Specification Changed

CS-V2Gov-039

The checks `require(lqtyAllocation.value != 0 ...)` and `require(totalLQTYAllocation.value != 0...)`, present in `BribeInitiative._claimBribe()`, are not meaningful, since the `value` now encodes the timestamp as well, and therefore will never be 0.

Specification changed:

The struct `Item` in the library DoubleLinkedList has been changed in Version 3 to store in separate fields `lqty` and `offset` instead of a single field (`value` in Version 2):

```
struct Item {
    uint256 lqty;
    uint256 offset;
    ...
}
```

6.25 Mismatch of Code With the Error Message

Correctness Low Version 2 Code Corrected

CS-V2Gov-048

The function `BribeInitiative.depositBribe()` performs the following check:

```
function depositBribe(..., uint16 _epoch) external {
    uint16 epoch = governance.epoch();
    require(_epoch >= epoch, "BribeInitiative: only-future-epochs");
    ...
}
```

The code accepts an input `_epoch` that is same as the ongoing epoch, while the error message highlights that `_epoch` should only be in the future.

Code corrected:

The error string has been changed in Version 3 to `"BribeInitiative: now-or-future-epochs"`

6.26 Remaining ToDos in the Codebase

Design Low Version 2 Code Corrected

CS-V2Gov-041

There are remaining `TODO` and `@audit` comments in the codebase. Addressing remaining notes help improve the quality and readability of the code.

Code corrected:

The previous comments have been removed from the contracts in scope.

6.27 Unsafe Casting in lqtyToVotes

Design Low Version 2 Code Corrected

CS-V2Gov-049

The function `lqtyToVotes()` has `public` visibility and takes as one of the inputs the variable `_currentTimestamp` which is of type `uint256`. The function is called with a timestamp variable of type

`uint32` whenever called internally, however if the function is called externally and a value greater than `type(uint32).max` is passed, the function performs an unsafe casting and returns an incorrect result.

Code corrected:

The unsafe casting has been removed, and the variable `_currentTimestamp` is now used at full width in the multiplication

6.28 Inconsistent Behavior for Retrieval of Tokens From Staking V1

Correctness Low Version 1 Code Corrected

CS-V2Gov-014

When unstaking or realizing the gains accrued in Staking V1, functions `withdrawLQTY()` and `claimFromStakingV1()` in Governance, exhibit different practices:

- Function `claimFromStakingV1()` allows the caller to specify any address as the recipient of funds, while `withdrawLQTY()` hard-codes the recipient to `msg.sender`.
- Function `withdrawLQTY()` emits an event which includes the unstaked LQTY amounts and the respective gains, while `claimFromStakingV1()` does not emit any event.

Code corrected:

Both functions have been revised in Version 3 to be aligned. Both `withdrawLQTY()` and `claimFromStakingV1()` allow users to specify a recipient, and emit the same event.

6.29 Incorrect Specifications

Correctness Low Version 1 Code Corrected

CS-V2Gov-015

We provide a non-exhaustive list of incorrect natspec comments in the codebase:

Version 1:

1. The documentation comment for the function `Governance._snapshotVotes()` says that it "accrues funds for the current epoch", whereas it "snapshots votes for the previous epoch". This could be misinterpreted: the number of votes are snapshot at the end of the past epoch, while BOLD are accrued until the first activity in the current epoch. The accrued BOLD are valid throughout all of the current epoch.
2. The natspec of function `DoubleLinkedList.getNext()` states that the head of the list is returned if `id` is zero, however `next` pointer stores the tail of the list.
3. Similarly, the function `getPrev()` state that the tail of the list is returned if `id` is zero, however the `prev` pointer stores the head of the list.

Version 3:

4. Function `claimBribes()` includes a note from previous versions of the codebase which is outdated.

5. Function `unregisterInitiative()` has an inline comment that is outdated: `weeks * 2^16 > u32`. Registration epoch for disabled initiatives is now set to `type(uint256).max`.
6. Function `getInitiativeState()` has an incorrect inline comment for `Unregister Condition` as an initiative is unregistable at 5th epoch that would result in SKIP.

Code corrected:

The abovementioned natspec comments have been fixed in Version 4

6.30 Initiative Snapshot Is Never Deleted

Correctness Low Version 1 Code Corrected

CS-V2Gov-016

The function `unregisterInitiative()` does not `delete initiativeSnapshot[_initiative]` which stores outdated values in case the initiative is registered again.

Code corrected:

It is no longer possible to re-register an initiative that was previously unregistered

6.31 Initiatives Can Be Registered With No Staking in First Weeks

Design Low Version 1 Code Corrected

CS-V2Gov-056

The deployer of the Governance contract can pre-register a set of initiatives which will be voted from epoch 2 and onwards. This means that the total votes for the first two weeks will be zero, hence anyone can register initiatives, bypassing the staking threshold check, however the registration fee must be paid.

Code corrected:

A new check has been added in Version 3 that ensures `registerInitiative()` can only be called after the second epoch:

```
require(currentEpoch > 2, "Governance: registration-not-yet-enabled");
```

6.32 Possible Simplification of onAfterAllocateLQTY

Design Low Version 1 Code Corrected

CS-V2Gov-017

The function `BribeInitiative.onAfterAllocateLQTY()` is triggered whenever a user changes their vote allocation in Governance, and it implements the logic to update user snapshots and global snapshots per an initiative. The implementation handles possibilities for scenario that both `_voteLQTY` and `_vetoLQTY` are non-zero.

However, the Governance ensures that a user either votes, or vetoes an initiative, it cannot do both in the same epoch. Taking this into consideration, the logic in `onAfterAllocateLQTY()` could be simplified. For instance, the following branches could be combined to achieve the same behavior:

```
if (_vetoLQTY == 0) {
    ...
    _setTotalLQTYAllocationByEpoch(_currentEpoch,
        totalLQTYAllocationByEpoch.items[_currentEpoch].value + _voteLQTY - prevVoteLQTY,
        false
    );
    _setLQTYAllocationByUserAtEpoch(_user, _currentEpoch, _voteLQTY, false);
} else {
    ...
    _setTotalLQTYAllocationByEpoch(_currentEpoch,
        totalLQTYAllocationByEpoch.items[_currentEpoch].value - prevVoteLQTY,
        false
    );
    _setLQTYAllocationByUserAtEpoch(_user, _currentEpoch, 0, false);
}
```

Code corrected:

The function `onAfterAllocateLQTY()` has been revised and its logic has been simplified.

6.33 Return Value of Ether Transfer

Design | Low | Version 1 | Code Corrected

CS-V2Gov-045

The following code does not check the return value of the Ether transfer via low-level call:

```
if (ethAmount > 0) {
    (bool success,) = payable(_lusdEthRecipient).call{value: ethAmount}("");
    success;
}
```

Code corrected:

A check on `success` has been added in Version 2.

6.34 Special Check in onAfterAllocateLQTY

Design | Low | Version 1 | Code Corrected

CS-V2Gov-054

The function `BribeInitiative.onAfterAllocateLQTY()` performs this check:

```
if (_currentEpoch == 0) return;
```

However, the condition can never be satisfied as the epoch number starts at 1.

Code corrected:

The check has been removed in Version 3.

6.35 Unnecessary Approval Given to LQTYStaking

Design | Low | Version 1 | Code Corrected

CS-V2Gov-020

The function `UserProxy.stake()` provides an approval to `LQTYStaking` before triggering the staking in Liquity V1:

```
lqty.approve(address(stakingV1), _amount);
stakingV1.stake(_amount);
```

The approval is unnecessary as the staking contract has a privileged role in the `LQTYToken` and does not require an approval to pull funds.

Code corrected:

The unnecessary approval has been removed.

6.36 Votes for UNREGISTERABLE Initiatives

Correctness | Low | Version 1 | Code Corrected

CS-V2Gov-055

The function `Governance._allocateLQTY()` forbids placing additional votes on an initiative that's in the `UNREGISTERABLE` state. However, a comment at the beginning of the for-loop in `_allocateLQTY()` says it should be allowed:

```
// Can vote positively in SKIP, CLAIMABLE, CLAIMED and UNREGISTERABLE states
```

However, a require statements does not allow voting on initiatives in the unregistable state:

```
require(
    status == InitiativeStatus.SKIP || status == InitiativeStatus.CLAIMABLE
        || status == InitiativeStatus.CLAIMED,
    "Governance: active-vote-fsm"
);
```

Currently, the only way for an initiative to leave unregistable state is if vetoes are removed from it.

Code corrected:

The inline comment has been revised to reflect the code behavior.

6.37 _calculateAverageTimestamp() Could Be off by One Second

Security | Low | Version 1 | Code Corrected

CS-V2Gov-021

The function `Governance._calculateAverageTimestamp()` performs an integer division between the number of votes and the LQTY amount, to obtain the average age. This could be rounded *down* by one second, resulting in the final average timestamp to be rounded *up* by one second. This rounding error can theoretically be used as a griefing attack to lower an initiative's average timestamp (allocated voting power), therefore reduce its claimable rewards.

Code corrected:

The code in Version 3 has been refactored to avoid the rounding error decribed above: the representation of a staking line has been changed to consist of the slope (LQTY amount) and a y-intercept (offset) which is always an integer, instead of the x-intercept (average timestamp) which could be fractional.

6.38 Confusing Name for Initiative Status

Informational | Version 2 | Code Corrected

CS-V2Gov-042

The enum variant `InitiativeStatus.WARM_UP` has a misleading name, since it represents initiatives that were registered in the current epoch; this is a different semantics from the one intended for the immutable variable `REGISTRATION_WARM_UP_PERIOD`.

Code corrected:

The immutable `REGISTRATION_WARM_UP_PERIOD` has been removed in Version 3

6.39 Encoding Functionalities Are Unused

Informational | Version 2 | Code Corrected

CS-V2Gov-043

The following newly functions are not used in the codebase:

- `BribeInitiative._encodeLQTYAllocation()`
- `EncodingDecodingLib.encodeLQTYAllocation()`

Code corrected:

The functions have been removed in Version 3

6.40 Governance Interface Is Incomplete

Informational Version 2 Code Corrected

CS-V2Gov-044

The interface `IGovernance` does not include all the public and external functions defined in the `Governance` contract. Here is a list of missing functions.

Version 2:

1. `getTotalVotesAndState()`
2. `getInitiativeSnapshotAndState()`
3. `calculateVotingThreshold()`

Version 3:

1. `resetAllocation()`

Code corrected:

All the Governance functions are in the interface, as of Version 4.

6.41 Inconsistent Use of safeTransfer

Informational Version 1 Code Corrected

CS-V2Gov-024

`UserProxy` uses the safe transfer functionality when sending LQTY and LUSD tokens to the user in function `unstake()`, but uses the token's transfer function when staking. Both LQTY and LUSD are already deployed and they return `true` on success, or revert on failure.

Code corrected:

Client has chosen to use safe transfer for all transfers in `UserProxy`.

6.42 Misleading Function Name

Informational Version 1 Code Corrected

CS-V2Gov-025

The internal function `Governance._deposit()` has a misleading name as does not actually perform any deposit. The function only deploys the `UserProxy` if not existent, and computes the new average staking timestamp for the user and updates the state.

Code corrected:

The function has been renamed to `_increaseUserVoteTrackers()` in Version 3. By the new line representation choice, it now updates the line's y-intercept instead of the x-intercept.

6.43 Signed Integers Are Needlessly Wide

Informational Version 1 Code Corrected

CS-V2Gov-028

In `allocateLQTY()`, signed LQTY quantities are represented as `int176`. This is unnecessarily wide, as the range of `int88` is approximately -150M to +150M, which is enough given that the hard cap on the LQTY supply is 100M.

Code corrected:

The codebase has been revised in Version 2 to use types `int88` for LQTY amounts. In Version 3, all integers are now 256-bit wide.

6.44 Unused Function in Math

Informational Version 1 Code Corrected

CS-V2Gov-029

The function `sub()` implemented in `Math.sol` is not used in the current version of the codebase.

Code corrected:

The function is used in codebase Version 4

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Possible Code Simplifications

Informational Version 3 Acknowledged

CS-V2Gov-052

Some parts of the code may be simplified, to make them more easily understandable and maintainable. Below is a non-exhaustive list of such snippets:

1. In `withdrawLQTY()` there is an `if-else` clause ensuring that the user's `unallocatedOffset` goes to 0 in case of a complete withdrawal. However, the fraction formula used in the `if` branch would already yield `offsetDecrease == unallocatedOffset` in this case, without the need for a special case.
2. In `allocateLQTY()`, the FSM check on `DISABLED` initiatives is redundant: the `require` will never fail. This is because the previous check on "active votes" already implies that, for all other states, both deltas should be non-positive. In fact, this holds not just for the `DISABLED` state, but also `UNREGISTERABLE`.
3. In `_allocateLQTY()`, updating the `GlobalState` is done by subtracting the `prevInitiativeState` and adding the new `initiativeState`. However, this could be simplified by simply adding the relevant user-supplied delta, and suppressing the `prevInitiativeState` altogether.
4. All payout snippets (in `claimForInitiative()`, `BribeInitiative.claimBribes()`, `CurveV2GaugeRewards._depositIntoGauge()`) include some capping logic, to account for potential errors leading to insolvency. However, the main source of rounding errors that led to broken accounting (i.e. using the average timestamp to represent lines) has been removed in Version 3. Moreover, should some other sources of error be left undiscovered, this mitigation still does not adequately protect against them: it only makes sure that the "last" payout does not revert, possibly at the expense of other destinations of those funds, without addressing the root cause that led to incorrect payout calculations.

Acknowledged:

Liquity has decided to keep the code unchanged (except `BribeInitiative.claimBribes()`)

7.2 Superfluous Calculations in Voting Threshold

Informational Version 3 Acknowledged

CS-V2Gov-053

As emphasized in note Expected immutable parameters, the parameter `MIN_CLAIM` will be set to `0` on deployment. Therefore, the calculations of `payoutPerVote` and `minVotes` in function `calculateVotingThreshold` are superfluous.

Acknowledged:

Liquity has decided to keep the code unchanged.

7.3 BOLD Could Be Accrued for Previous Epoch

Informational | Version 1 | Acknowledged

CS-V2Gov-022

The function `Governance._snapshotVotes()` gauges the contract's current BOLD balance and writes it into the snapshot `boldAccrued` variable, the first time any user action is taken in an epoch. Still, funds might arrive early in an epoch, before any user action is taken: they would then be counted towards the previous epoch's `boldAccrued`.
Users with a position both in LiquityV2 core and in the Governance might optimise for this, and time their operations in the core just right so the corresponding fee ends up being counted in an epoch where they have already a large voting power.

Acknowledged:

Liquity is aware of the inconsistency but has decided to keep the code unchanged.

7.4 Default Values in DoubleLinkedList

Informational | Version 1 | Acknowledged

CS-V2Gov-023

The view functions `getNext()`, `getPrev()`, `getValue()`, and `getItem()` in the library `DoubleLinkedList` do not check if the input `id` exists in the list, instead they return the default values (`0`s) if `id` does not exist. Considering that the node with `id` zero has a special meaning, stores the head and the tail of the list, the default value might be misinterpreted by callers.

7.5 Possible Gas Optimizations

Informational | Version 1 | Code Partially Corrected

CS-V2Gov-026

The code could be optimized to reduce the gas consumption. Below, is a non-exhaustive list of potential optimizations:

1. Functions `stakeViaPermit()` and `unstake()` in `UserProxy` can be marked as `external`.
2. Function `onAfterAllocateLQTY()` in `BribeInitiative` sets the `newVoteLQTY` to `0` if `_vetoLQTY` is non-zero, however, this is not necessary as the Governance guarantees that only one of the two is non-zero.
3. Function `_calculateAverageTimestamp()` returns early if `_newLQTYBalance` is zero, however, the check is repeated in both branches of the `if/else` block.
4. The function `claimForInitiative()` returns early if the total snapshotted "YES" votes are 0, or if the initiative's snapshot "YES" votes are 0. The first condition implies the second, therefore the first check is redundant.

Version 2:

5. Function `Math.add()` performs an unnecessary call to `abs()` when b is a positive value.
6. Function `UniqueArray._requireNoDuplicates()` copies unnecessarely the input array from `calldata` to `memory`.

7. Function `Governance.getInitiativeState()` performs redundant SLOADs when accessing `registeredInitiatives[_initiative]`.
8. Function `Governance.unregisterInitiative()` performs redundant checks on the initiative status, e.g. `require(status != InitiativeStatus.NONEXISTENT, ...)`
9. The function `Governance.allocateLQTY()` could skip all the updates to the `InitiativeState` and the `GlobalState`, in case the initiative is `DISABLED`, and short-circuit to the updates of the user's `Allocation` and `UserState`.

Version 3:

10. Function `UniqueArray._requireNoNegatives()` copies unnecessarily the input array from `calldata` to `memory`.
11. Function `withdrawLQTY()` unnecessarily overwrites a storage pointer: `userStates[msg.sender] = userState;`.
12. A redundant variable `upscaledSnapshotVotes` is declared in `registerInitiative()`.
13. The function `_requireNoNOP()` could check that only one value (either `_absoluteLQTYVotes[i]` or `_absoluteLQTYVetos`) is non-zero to revert early if user allocates both votes and vetos to an initiative.
14. Struct `DoubleLinkedList.Item` uses type `uint256` for each field, hence occupying 4 storage slots, although the values require less space.
15. Function `registerInitiative()` gauges the user's total staked LQTY via an external call to `stakingV1`, whereas it could add up his allocated and unallocated LQTY.

Version 4:

16. Function `Governance.lqtyToVotes()` can be marked as `external`. Other functions in Governance should use the internal version `_lqtyToVotes()` instead.
17. Function `Governance._requireNoSimultaneousVoteAndVeto()` copies unnecessarily the input array from `calldata` to `memory`.

Code partially corrected:

The optimizations in points 1-9 have been implemented in Version 3.
Points 11 and 13 have been implemented in Version 4

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Cannot Partially Deallocate From DISABLED and UNREGISTERABLE Initiatives

Note Version 3

Partial deallocations are implemented by first resetting all allocations and then re-allocating part of the LQTY. This, combined with the strict FSM checks implemented in `_allocateLQTY()`, implies that partial deallocations are not possible on DISABLED and UNREGISTERABLE initiatives.

8.2 CurveV2GaugeRewards Functionalities

Note Version 1

The contract `CurveV2GaugeRewards` should have the role `distributor` in the Curve Gauge in order to call the function `deposit_reward_token()`.

The `distributor` of the Curve Gauge can also call `set_reward_distributor()`, however no functionality is implemented in `CurveV2GaugeRewards` to trigger it. This functionality would be relevant if a new initiative is deployed to direct funds to a Curve gauge.

Liquity replied:

```
Following our usual procedures, we won't keep any kind of "admin" role, meaning
that we won't be able to call `set_reward_distribution` nor `add_reward`. We are
aware that we would need to deploy a new pool and a new gauge if more rewards
wanted to be added.
```

8.3 Depositing and Withdrawing LQTY in the Same Transaction Reduce Voting Power

Note Version 1

Assuming a user has staked to 2 weis of LQTY for `60` seconds, their voting power is `120`. Depositing 2 more weis of LQTY, trigger the recalculation of the `averageStakingTimestamp` for the user, which results in `30` seconds, hence the voting power before and after the deposit remains the same. If the user withdraws the 2 newly deposited weis, their voting power actually halves to `60` (from `120`).
As of Version 4, this only applies to a user's unallocated staking line, see Unallocated staking line can have unequal (virtual) average timestamp.

8.4 Expected Immutable Parameters

Note Version 1

All of the numeric parameters present in this overview are contract immutables, set at deployment time. The intended values to be used upon deployment are as follows:

```
uint256 public immutable EPOCH_DURATION; // 7 days (604800)
uint256 public immutable EPOCH_VOTING_CUTOFF; // 6 days (518400)
uint256 public immutable MIN_CLAIM; // 0 BOLD (0)
uint256 public immutable MIN_ACCRUAL; // 0 BOLD (0)
uint256 public immutable REGISTRATION_FEE; // 1000 BOLD (1000e18)
uint256 public immutable REGISTRATION_THRESHOLD_FACTOR; // 0.01% (0.0001e18)
uint256 public immutable UNREGISTRATION_THRESHOLD_FACTOR; // 1x (1e18)
uint256 public immutable UNREGISTRATION_AFTER_EPOCHS; // 4
uint256 public immutable VOTING_THRESHOLD_FACTOR; // 2% (0.02e18)
```

Any significant deviation from these values can break certain functionalities of the system. For example, setting `EPOCH_DURATION` to less than the block time, or `EPOCH_VOTING_CUTOFF` to zero, breaks the contract; setting a low amount as `REGISTRATION_FEE` opens opportunity for DoS; or setting `UNREGISTRATION_THRESHOLD_FACTOR` to a value less than 1 (1e18) breaks the unregistration rules.

8.5 Opportunity Cost of Vetoes

Note Version 1

Users have a finite voting power for an epoch that can be split as YES votes, and NO votes (vetoes) on different initiatives. Voting YES for an initiative increases the share of the revenues that is claimable by that initiative. On the other hand, voting NO for an initiative might prevent that initiative from claiming revenues, however it does not directly increase the share of revenues for a preferred initiative.

8.6 Staking With Permit Restrictions

Note Version 1

The function `UserProxy.stakeViaPermit()` requires that `_lqtyFrom` matches the `owner` in the `PermitParams`. Given that this function can be called only from the Governance, and `_lqtyFrom` is set to `msg.sender`, the functionality requires that a user issuing the permit needs to also submit the transaction on-chain.

8.7 Theoretical Overflow in Math Function

Note Version 3

Function `abs()` in file Math.sol takes as input a which is of type `int256`. If a is negative, then the following casting is performed: `uint256(-int256(a))`. Theoretically, an overflow can happen if a is `-2**255` as the intermediary positive value `2**255` cannot be stored in a type `int256`. This means that the function `abs()` only returns the correct value for integers in the range `[-2**255, 2**255-1]`.

8.8 Unallocated Staking Line Can Have Unequal (Virtual) Average Timestamp

Note **Version 4**

In **Version 4** of the codebase, staking and unstaking no longer requires a user to completely deallocate his LQTY, and instead only affects his unallocated staking line. When this requirement was still in force, it ensured that the ratio between the offset and the LQTY (i.e. the "virtual" average staking timestamp) of the unallocated line be equal to that of *all* the allocated lines. Now this no longer holds: while a withdraw preserves the timestamp, a deposit moves it closer to the present. The timestamp of the unallocated line can also be moved back by partially deallocating through `resetAllocations()`. These "throw out of sync" the unallocated line with respect to the allocated ones. This is however inconsequential, because the average staking timestamp is no longer used anywhere in the code, and no invariant relies on an exact proportionality of all staking lines (allocated *and* unallocated) of a given user.
Notice, however, that the allocated lines still have all the same x-intercept, since they were drawn right after a complete reset, which merged everything back into the unallocated line. This means that now, for the allocated lines to be all proportional, resetting before allocating is strictly necessary (unlike a previous version of the codebase).

8.9 Voting Thresholds and Rewards Depend on Allocated YES Votes

Note **Version 3**

We emphasize that the voting threholds and rewards do not depend on the sum of potential voting power (allocated and unallocated) of all users of the system. Instead, the global snapshot of votes tracks only the allocated YES votes. Therefore, the tresholds might be lower than expected in an epoch if there is significant amount of unallocated staked LQTY or NO votes (vetos).

8.10 `GlobalState` and `UserState` Are Stale if Queried in Hook

Note **Version 1**

The function `allocateLQTY()` calls the `onAfterAllocateLQTY()` hook on the initiative before updating the `GlobalState` and `UserState` variables, which therefore are stale if queried by the initiative, in the hook, via the view functions.

Furthermore, 3rd-party protocols querying the view functions of the Governance contract should consider that the returned data might be incorrect. Such view functions are susceptible to read-only reentrancy attacks.