PUBLIC

Code Assessment

of the Subsquid Smart Contracts

April 17, 2024

Produced for

SUBSQUID

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Subsquid Team,

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

Subsquid implements the on-chain part of the Subsquid protocol. The various parties of the system can stake their $SQD tokens in exchange for rewards for workers and stakers or computation units (CUs) for gateway operators.

The most critical subjects covered in our audit are the safety of the funds, the reward accumulation and distribution mechanism, the calculation of the computation units, and the vesting mechanism. The security of the funds is high as we were not able to uncover ways to steal user's funds. Reward distribution could be unfair in case a staker front-runs reward distribution (see Recent stakers get unfair yield). It could also be blocked if the number of workers grows a lot (see Reward distribution can run out of gas). The CU calculation could be improved as there are cases where CUs are double-counted (see Computation units are not split between an operator's gateways). The vesting could break in case the user claims their rewards through the vesting contract. All the issues have been addressed.

The general subjects covered include but are not limited to access control, rounding errors, the rollup (ArbitrumOne) where the contracts are to be deployed, documentation, and specification. The security regarding access control and rounding errors is high. Even though there exists a lot of documentation for the protocol itself, the interface of the on-chain part to the rest of the system is underspecified. Therefore, we had to make assumptions about how the system will be implemented e.g., what events are going to be observed. Hence, there could be more issues in this area that were not anticipated by the auditing team. Testing could also be improved as we uncovered a few issues that could be easily detected this way.

In summary, we find that the security of the codebase is satisfactory but there is room for improvement.

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	0
High-Severity Findings	0
Medium-Severity Findings	2
• Code Corrected	1
• Code Partially Corrected	1
Low-Severity Findings	12
• Code Corrected	11
• Acknowledged	1

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 Subsquid 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	20 March 2024	e0a94752031b0816a260901457475f26535de342	Initial Version
2	15 April 2024	2e48952fe8b7496224408f3e44ea0e4341eec7b0	Fixes
3	17 April 2024	294c8f739bf5b2b653d3b1c59131d54ec02aebad	2nd Round of Fixes

For the solidity smart contracts, the compiler version `0.8.20` was chosen. In particular the files in scope are the following files under the `packages/contracts/src/` directory:

- `Executable.sol`
- `Vesting.sol`
- `GatewayRegistry.sol`
- `RewardCalculation.sol`
- `Staking.sol`
- `DistributedRewardDistribution.sol`
- `WorkerRegistration.sol`
- `NetworkController.sol`
- `AllocationsViewer.sol`
- `SoftCap.sol`
- `TemporaryHolding.sol`
- `RewardTreasury.sol`
- `gateway-strategies/EqualStrategy.sol`
- `gateway-strategies/SubequalStrategy.sol`
- `Router.sol`
- `VestingFactory.sol`
- `TemporaryHoldingFactory.sol`
- `AccessControlledPausable.sol`

For the review of the $SQD token, please refer to the respective audit.

2.1.1 Excluded from scope

All the contracts not mentioned in the scope are considered out-of-scope. External libraries such as OpenZeppelin and prb are also excluded. The contracts will be deployed on Arbitrum One rollup. The system makes use of `libp2p` which is out-of-scope. The library is expected to assign non-predictable peer ids to the various nodes of the network.We assume that the semantics of the EVM operations as implemented for ArbitrumOne are similar to those of Ethereum mainnet. The transactions are assumed to be sent directly to the ArbitrumOne Sequencer, therefore no front-running should be possible, at least as far as the registration of various parties is concerned. The use of the on-chain information from the various off-chain components of the system was considered only on a best-effort basis as the documentation is lacking. The admins of the system are assumed to parametrize the system properly and assign the correct roles to the respective components.

2.2 System Overview

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

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

Subsquid implements an on-chain system for the reward distribution of workers and stakers in the Subsquid network. Subsquid network is built to provide a permission-less database with horizontal scalability. Workers provide the system with storage and computation resources and receive compensation of $SQD. Workers should be registered on-chain by bonding `100_000` $SQD. If a worker fails to adhere to the protocol, its bond will be slashed (not yet implemented). $SQD holders can also stake in a worker to guarantee its reliability and earn a portion of the rewards. The data are consumed by gateways controlled by operators. This system updates its internal state in time intervals of a duration called epochs.

In what follows, we delve into the on-chain system for registration of workers, staking, and distributing the rewards.

2.2.1 Worker Registration

Workers, the entities providing computation/storage in the network, can get registered by providing a bond. Later, they can be deregistered with the bond being redeemed.

`register()`: Any user, who possesses the required bond, can register a worker identified by a unique peerID (hex representation of the libp2p peer ID of the worker), through this function. If the worker was previously registered and then removed, the previous worker ID is reassigned to it, otherwise, it gets a fresh ID for this worker. The registration time for a worker is the start of the next epoch. Finally, the bond amount of $SQD is moved from the caller to the `WorkerRegistration` contract and the caller would be the creator of the worker.

`deregister()`: Any registered worker can deregister. A worker is considered as deregistered starting from the beginning of the next epoch. As long as its bond amount is still in the system, it is considered inactive but it remains in `activeWorkerIds`.

`withdraw()`: After getting deregistered, the creator can withdraw the bond. To call this function a certain lock period after deregistration must be passed. Then, its bond is returned to its creator.

A registered worker can be in two states:

1. **Fully Active**: `isWorkerActive()` returns true i.e., the registration epoch has passed but not the deregistration epoch (if such has been recorded).
2. **Bonded**: The worker is registered in the `activeWorkerIds` i..e, the bond of the worker hasn't yet been claimed.

Another useful devised functionality is the ability to withdraw the excessive bond amount. If after the registration of a worker, the bond amount goes down, the creator can withdraw the excessive amount. However, if the bond increases, this function reverts. We assume the bond amount only decreases over time.

2.2.2 Staking

This contract is the entry point for the stakers who want to stake (deposit) or unstake (withdraw) $SQD. Moreover, it's responsible for keeping the accounting of the rewards stakers have accumulated.

`deposit()`: Any user can stake an amount for any worker. Deposits fail if no rewards were distributed for the last 2 epochs, as it implies that the system is broken, or if the hard limit on the total amount staked into a specific worker has been reached.

`withdraw()`: Withdrawals are allowed after the end of the next epoch after last staking. Stakers can withdraw part of the $SQD they have staked.

Whether depositing or withdrawing stakes, the pending reward for the staker yielded by the worker since the last checkpoint gets calculated proportional to its staked value. Then, updates the last checkpoint of the caller to be the rewards cumulated per share of stake.

Any reward distributor (users holding `REWARDS_DISTRIBUTOR_ROLE`) can distribute (`distribute()`) rewards to the stakers of the particular workers. In this first experimental version of Subsquid, REWARD_DISTRIBUTOR_ROLE is not given out to users, it's only given to "bots" operated by the admins. There are multiple of them just to protect against accidental mistakes, they are assumed to be honest. This is done by increasing the cumulated reward per share.

2.2.3 Reward Distribution

This contract holds a set of distributors. During timeframes of 256 blocks, one distributor is selected. The current distributor, selected in a pseudo random way, can commit rewards for a range of blocks in the past for the workers and stakers. Committing automatically approves the commit. Other distributors can observe this commitment and approve this further by calling `approve()`. Once the required approval threshold is reached, the reward distribution takes place. During reward distribution it is checked that the distributions are sequential (i.e., if the previous distribution was for the range of `[A, B]` of blocks, the current distribution should be `[B + 1, C]`). This function calls into `Staking.distribute()`.

`claim()`: Reward treasury can claim for a given staker. This function calls into `Staking.claim()` which updates the checkpoint and sets the claimable amount for this staker to 0. Then it adds the claimable amount for each worker operated (owned) by this staker and sets each one to 0.

2.2.4 Reward Treasury

The reward treasury holds the funds to be distributed as rewards to the workers and the stakers. The reward distribution logic is implemented via some reward distributor contracts. A user can claim the rewards they are eligible for.

`claim()`: When the contract is not paused, a user specifies a whitelisted distributor contract e.g., DistributedRewardDistribution and optionally the receiver of the rewards. The reward distributor calculates the amount the user is eligible which is then sent to the receiver. Hence, before claiming, enough amount of reward should already be in the treasury.

The owner of the contract can whitelist distributors and reclaim all funds in case of emergency.

2.2.5 Gateway Registry

To query the network, data consumers have to operate a gateway. An operator can stake $SQD for a period of time to receive Computation Units (CUs). This contract keeps a list of whitelisted gateways. Computation units are distributed among different workers according to the strategy of each operator. If no strategy of preference is set, the default strategy is used.

Operators can register their gateway of interest. All the gateways operated by an operator form a cluster. The maximum size of a cluster is specified by the owner of the registry. A cluster is considered active as long as its operator has staked some amount of $SQD.

An operator of a given gateway can deregister and remove this gateway from the cluster as well as the active gateways.

Operators can stake $SQD for some time to activate their gateways. After staking, all the gateways present in the cluster of the operator get automatically activated, no matter if a non-zero amount of $SQD has been staked. Finally, the specified amount is transferred from the staker to the gateway registry. Once staked, and the lock start of the staking passed, the staker can increase the staked amount and set the start lock of staking to the next epoch.

When staking or increasing the stake, there is an option to stake for an infinite repetition (conceptually, like staking for a period and after the end of the lock instantly unstake and stake again). Stakers can choose this option by setting `withAutoExtension` when staking. Adding a stake might extend the staking period for another staking duration, but cannot change the `withAutoExtension`. To enable this option, the operator has to call `enableAutoExtension()`. And to reset it, `disableAutoExtension()` should be called, which not only disables the auto extension of the lock period but also sets the end lock to the beginning of the next interval of stake duration.

After the end of the lock period, operators can unstake the whole staked amount by calling `GatewayRegistry.unstake()` which deactivates all the gateways operated by the operator and transfers the staked amount to the operator.

Operators can change their preferred strategy by calling `GatewayRegistry.useStrategy()`. Note that although operators have this freedom to choose their strategy, they are allowed to choose it only from a set of whitelisted strategies.

All gateways operated by a single operator get the same computation units calculated as:

```
amount * durationBlocks * mana * boostFactor(durationBlocks * averageBlockTime)
```

`mana` describes how many CUs are accessible for a single $SQD during 1000 blocks. The longer the operator stakes its $SQD, the more its equivalent CU gets boosted according to the `boostFactor()` [in Reward Calculation]

2.2.6 Router

As described in the previous sections, this system consists of multiple units. Each unit might end up calling another one. Router lays in the middle and forwards the calls to the correct unit. It keeps track of the following contracts:

1. worker registration
2. staking
3. reward treasury
4. network controller
5. reward calculation

Admin of this contract can update the addresses of the core units after deployment.

2.2.7 Network Controller

Network parameters, like the length of epochs, get set through this contract. Later, other contracts query the Network Controller to fetch these parameters.

2.2.8 Reward Calculation

The network rewards are paid out to workers and stakers for each epoch and the reward calculation contract calculates the claimable rewards by each worker and staker. It provides the following interfaces:

1. `effectiveTVL()`: returns the sum of the bond amount plus capped staked (see below) for all the active workers. Please note that it considers the current bond amount. Hence, if during registration the bond amount was higher and the excessive amount was not reclaimed, it is not counted towards the TVL.
2. `baseApr()`: calculates the rewards based on the utilisation rate as follows: The APY is 70% if the utilization is over 90%. If it's positive then it follows the equation $25\% + u/2$. If the utilization is negative i.e, the actual utilization exceeds the target then the APY is $max(20\% + u/20, 0)$.
3. `apyCap()`: for a zero TVL, the APY can be up to 100%, otherwise it's $APR\_CAP(total\_staked) = 0.3 * INITIAL\_POOL\_SIZE/effectiveTVL$.

The actual APY would then be the value returned by `baseApr()` capped by `apyCap()`.

1. `boostFactor()`: to incentivise the gateways to stake their tokens for a longer period of time. In particular the boost factor is: 1 for staking less than 60 days, $1 + 2(d - 30days)/30days$, where d is the staking duration, if the duration is less than 180 days, 2 if the duration is less than 360 days, 2.5 is less than 720 days, 3 otherwise.

As mentioned above, during the calculation of `effectiveTVL()`, the staked amount of a worker gets capped. To cap this, the following function is used:

$$\left(\frac{2}{3}\right)^{(x-1)^4} - \left(\frac{2}{3}\right)$$

Where x is the staking share i.e., the portion of the staked amount for a worker over the sum of the staked amount and bonded amount.

2.2.9 Vesting and Temporary Holding

Funds can be distributed through two kinds of wallets, `SubsquidVesting` or `TemporaryHolding` that allow funds to be used within the protocol while limiting their usage outside of it. For `SubsquidVesting`, the release schedule contains an immediate releasable amount (cliff) plus a gradual release of the rest. In the holding wallets the funds are locked for a beneficiary to interact with the Subsquid network and after expiry they are claimable by the owner.

2.2.10 Trust Model and Assumptions

We assume the Router contract is correctly configured and forwards the queries to the correct contracts. Furthermore, the functionality of the system depends on the parameters set in the network controller. Therefore, the parameters should be set correctly.

Users having the role of `REWARDS_DISTRIBUTOR_ROLE` are trusted as they propose the commitments for the rewards distribution as well as approving them.

The vesting wallets and temporary wallets are created by users having the `VESTING_CREATOR_ROLE` and the `HOLDING_CREATOR_ROLE` respectively. They are assumed to appropriately set up the wallets.

Admin of the Reward Treasury is trusted to whitelist the distributors correctly. The admin of the system has the ability to pause specific contracts namely:

- TemporaryHoldingFactory
- DistributedRewardDistribution
- VestingFactory
- GatewayRegistry

- Staking
- WorkerRegistration
- RewardTreasury

As pausing between different contracts is not synced, we assume that the admin will appropriately pause the contracts they want to if the need arises.

In this assessment, we assumed the reward token is $SQD.

As this reward distribution system is supposed to be launched on Arbitrum, we assume bridging $SQD tokens from the mainnnet to Arbitrum is flawless.

2.2.11 Version 2

In the second version, the following changes were introduced:

- The selection of the distributor is not pseudo random but is determined in a round-robin fashion.
- The `Subequal` strategy. Gateway operators can specify a subset of workers they want to distribute their computation units to.

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 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:

- 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

- Recent Stakers Get Unfair Yield Code Partially Corrected

Low-Severity Findings	1

- Gateway Operator Can Add 0 SQD to His Stake Acknowledged

5.1 Recent Stakers Get Unfair Yield

Design Medium Version 1 Code Partially Corrected

CS-SQDC-001

In the `DistributedRewardDistribution` contract, the rewards being committed to and approved include the time range `[fromBlock, toBlock]` they were computed for. However, when the proposal is executed (when the last `approve()` arrives), the `distribute()` function calls `Staking.distribute()`, which gives out yield to the **current** stakers of the specified worker, regardless of whether they were already staking during the relevant timeframe.

This means that a staker joining after the period for which the rewards are computed, but before the last `approve()` arrives for that proposal, gets an unfair share of those rewards. Symmetrically, a staker leaving in the same window will lose their fair share of the rewards.

Code partially corrected:

Users are forced to stake for more epochs determined by `epochsLockedAfterStake`. This value is set by the admin.

5.2 Gateway Operator Can Add 0 SQD to His Stake

Design Low Version 1 Acknowledged

CS-SQDC-002

The function `GatewayRegistry.addStake()` does not revert if called with `amount = 0`; instead, it extends the lock period by one "segment", starting from the next epoch. This behaviour is harmless per-se (it is roughly equivalent to enabling auto-extension), but it is undocumented.

Acknowledged:

It is not an intended behaviour, but since there’s no harm in that, we will keep that and add a comment.

6 Resolved Findings

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

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

- Reward Distribution Can Run Out of Gas Code Corrected

Low-Severity Findings	11

- Distribution With Multiple Commitments Code Corrected
- Supporting the Same Worker in Subequal Strategies Code Corrected
- Claiming Can Run Out of Gas Code Corrected
- Delegation Limit Redundant With Soft Cap Code Corrected
- Distributor Index Code Corrected
- Gateway Operator Can Stake 0 SQD Code Corrected
- Gateway Staking for Less Than an Epoch, With Autoextension Code Corrected
- Missing Check When Removing Distributor Code Corrected
- Retiring a Small Worker Can DOS the Reward Distribution Code Corrected
- Stake Duration Is Not Sanity-Checked Code Corrected
- Transferring the Ownership of Vesting Code Corrected

Informational Findings	4

- Redundant Grant of Admin Role Code Corrected
- Redundant Role-Granting Function Code Corrected
- Event Rewarded Can Be Emitted With 0 Reward Code Corrected
- Two Different Implementations of effectiveTVL() Code Corrected

6.1 Reward Distribution Can Run Out of Gas

Design Medium Version 1 Code Corrected

CS-SQDC-016

The reward distribution system implicitly requires reward proposals to cover *all* active workers for their time window: this is because proposals have to cover consecutive timeframes (enforced through `lastBlockRewarded`), therefore one cannot, at a later time, "go back" and integrate an old proposal with worker rewards it did not include. If the system grows too big, the reward distribution would break because the one transaction to reward all workers would hit the block gas limit.

Code corrected:

The number of delegates is capped by `maxDelegations` which is settable by the admin.

-

6.2 Distribution With Multiple Commitments

Design Low Version 2 Code Corrected

CS-SQDC-017

In version 2, the system allows multiple distributors to call `DistributedRewardDistribution.commit()` for the same block range. Other distributors can approve this commitment for this range only once. However, if a second commitment takes place the approvals are reset. Moreover, the distributors already have approved this commitment they are not allowed to reapprove it.

Code corrected:

A second commitment doesn't reset the number of approvals.

6.3 Supporting the Same Worker in Subequal Strategies

Correctness Low Version 2 Code Corrected

CS-SQDC-005

Using the subequal strategy, gateway operators can choose to delegate their queries to a specific subset of workers by calling `SubequalStrategy.supportWorkers()`. Each time a worker is supported, the count of workers increases. However, there's no check that a worker has already been supported. Therefore, the worker count might be greater than the actually supported workers. The same issue exists for `SubequalStrategy.unsupportWorkers()`. Note that this issue could also lead to division by 0 when `SubequalStrategy.computationUnitsPerEpoch()` is called.

Code corrected:

A check for worker duplication was implemented.

6.4 Claiming Can Run Out of Gas

Design Low Version 1 Code Corrected

CS-SQDC-009

The function `Staking.claim()` iterates through all the workers this staker has staked into. There is no bound on the number of workers one can stake into: if it grows too large, the claiming transaction might hit the block gas limit, making it altogether impossible to claim yield without temporarily unstaking from some workers.

Code corrected:

The contract now enforces a hard limit of 100 delegations per staker. Note that the number of max delegations can be changed by the admin.

6.5 Delegation Limit Redundant With Soft Cap

Design | Low | Version 1 | Code Corrected

CS-SQDC-015

The `Staking.deposit()` function enforces a hard cap on the total amount of SQD staked in favour of any single worker. This is redundant with the soft cap induced by the law of diminishing returns implemented in `SoftCap`.

Code corrected:

The hard cap was removed.

6.6 Distributor Index

Correctness | Low | Version 1 | Code Corrected

CS-SQDC-014

The system decides who the current distributor is by calling `DistributedRewardDistribution.distributorIndex()` which implements the following logic:

```
uint256 slotStart = block.number / 256 * 256;
return uint256(blockhash(slotStart)) % distributors.length();
```

When the `block.number` is a multiple of 256, the slotStart equals to `block.number`. `blockhash` for the current block returns 0 instead. This means the distributor will always be the one with index 0 for the multiples of 256.

Code corrected:

The distributor index is not determined by `blockhash`. It changes in a round-robin fashion as follows:

```
return (block.number / roundRobinBlocks) % distributors.length();
```

However, the distributor index might not change the way it's expected. For example, consider the case where `roundRobinBlocks` is a multiple of `distributors.length()` then the index is always going to be 0.

6.7 Gateway Operator Can Stake 0 SQD

Design | Low | Version 1 | Code Corrected

CS-SQDC-008

The function `GatewayRegistry.stake()` does not check that `amount > 0`, so a gateway operator can call it with `amount` set to 0 and have all his gateways be marked as active.

Code corrected:

The function now includes a check that the staked amount is greater than `minStake` which is initially set to 1. `minStake` can be changed by the admin.

6.8 Gateway Staking for Less Than an Epoch, With Autoextension

Design | Low | Version 1 | Code Corrected

CS-SQDC-011

Gateway operators can set an "autoextension" option for their staking position, which is meant to prolong it indefinitely, in whole consecutive "segments" of the original duration, until the option is disabled. Yet, the function `GatewayRegistry.computationUnitsAvailable()` does not play well with this mechanism, if the stake duration is less than an epoch.

Say that the duration is one tenth of an epoch (and autoextension is enabled): then an epoch is "tiled" by 10 segments, so the available CUs in the epoch should be 10 times those afforded by a single segment (which is, instead, what the function returns). On the other hand, if it were actually implemented this way, one could spend all those CUs in less than an epoch, then "prematurely" disable the autoextension, and finally unstake, thus spending more CUs than what should be granted by the effective lock period.

Code corrected:

Gateway locking cannot be shorter than one epoch. However, should the epoch duration be increased, for stakers who have staked under the previous configuration, the staking duration could be less an epoch.

6.9 Missing Check When Removing Distributor

Design | Low | Version 1 | Code Corrected

CS-SQDC-006

In the `DistributedRewardDistribution` contract, the function `removeDistributor()` does not check that the resulting `distributors.length()` is greater than or equal to `requiredApproves`, as is instead done in the `setApprovesRequired()` function.

Code corrected:

The check was added to the function `removeDistributor()` as well.

6.10 Retiring a Small Worker Can DOS the Reward Distribution

Design | Low | Version 1 | Code Corrected

CS-SQDC-018

The function `Staking._distribute()` reverts if the worker in question has no SQD staked in his favour. Therefore, a malicious actor can register a worker doing only a tiny amount of work (just enough to earn some rewards), and also stake some SQD in his favour (he needs to be the *only* staker for that

worker). Then, when the reward proposal arrives to `DistributedRewardDistribution`, he can unstake everything from his worker. The proposal will then fail to execute because `DistributedRewardDistribution.distribute()` will revert, temporarily blocking the rewards for all other workers as well, until a new proposal is submitted to make up for it.

Code corrected:

The function `Staking._distribute()` now does nothing, instead of reverting, if the worker's total stake is 0.

6.11 Stake Duration Is Not Sanity-Checked

Design | Low | Version 1 | Code Corrected

CS-SQDC-013

In the function `GatewayRegistry.stake()`, the parameter `durationBlocks` is not checked to lie within some reasonable bounds. A user can therefore inadvertently plug in a disproportionately high value (e.g. thinking it is meant to be a duration in *seconds*) and lock their tokens for too long.

Code corrected:

The stake duration is now checked not to exceed 3 years.

6.12 Transferring the Ownership of Vesting

Design | Low | Version 1 | Code Corrected

CS-SQDC-003

`SubsquidVesting` aims to limit the usage of $SQD only within the protocol. However, the ownership of `SubsquidVesting` can be transferred to a contract which could issue transferrable shares of `SubsquidVesting` and therefore create a derivative of $SQD that can be traded.

Code corrected:

Ownership transferring was disallowed. We assume that the beneficiaries of the vesting accounts will not be contracts.

6.13 Event `Rewarded` Can Be Emitted With 0 Reward

Informational | Version 1 | Code Corrected

CS-SQDC-010

In the `Staking` contract, the `Rewarded` event is emitted by the functions `updateCheckpoint()` and `claim()`. However, while the former checks for the reward to be positive before emitting the event, the latter does not.

Code corrected:

The event is now only emitted for positive rewards.

6.14 Redundant Grant of Admin Role

Informational | Version 1 | Code Corrected

CS-SQDC-004

In the `DistributedRewardDistribution` contract, the constructor grants the `DEFAULT_ADMIN_ROLE` to the deployer. This is redundant with the constructor of `AccessControlledPausable`, which `DistributedRewardDistribution` inherits from.

Code corrected:

The redundant statement was removed

6.15 Redundant Role-Granting Function

Informational | Version 1 | Code Corrected

CS-SQDC-007

In contract `TemporaryHoldingFactory`, the function `allowTemporaryHoldingCreator()` is redundant with the public `grantRole()` function (in OpenZeppelin's `AccessControl`), inherited from `AccessControlledPausable`.

Code corrected:

The redundant function was removed

6.16 Two Different Implementations of `effectiveTVL()`

Informational | Version 1 | Code Corrected

CS-SQDC-012

The function `effectiveTVL()` is implemented both in `RewardCalculation` and in `WorkerRegistration`: in the former calculates the soft capped sum of the total bonded amount, the latter simply estimates the total bonded amount of all the workers.

Code corrected:

The function was removed from the `WorkerRegistration` contract.

7 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.

7.1 Cliff Value

Note Version 1

During vesting, the cliff depends on the `totalAllocation` i.e., the current balance of the token in the contract and the released amount. However, it ignores the `depositedIntoProtocol` amount. This means, the value of the cliff and he vested amount varies depending on the amount of assets deposited into the system.

7.2 Computation Units Are Not Split Between an Operator's Gateways

Note Version 1

The function `GatewayRegistry.computationUnitsAvailable()` calculates the CUs available to *a gateway* by applying some mathematical formulas to the stake of *its operator*, regardless of the presence of other gateways belonging to the same operator. Therefore, the CUs earned by an operator are "replicated" across all its gateways. According to Subsquid, the cluster is considered as a single instance of a gateway, with different endpoints. Therefore workers would have to track each cluster as a whole and monitor CU usage.

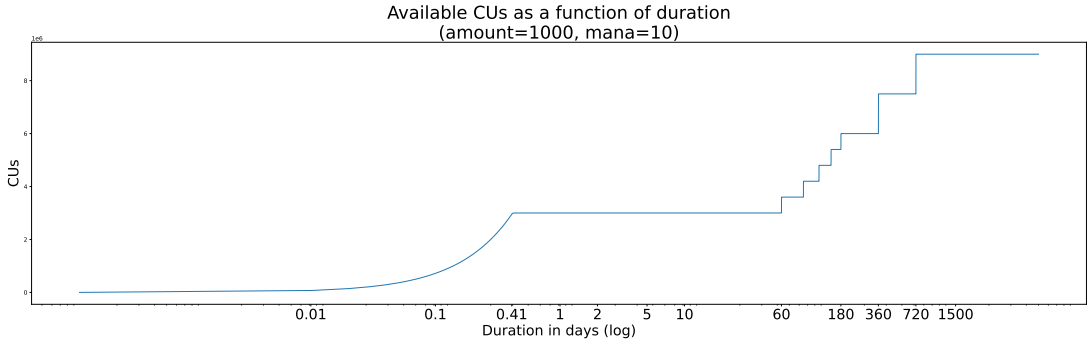
7.3 Incentives for Gateway Operators

Note Version 1

Gateway operators can lock their $SQD tokens for a period of time to get CUs in return. The number of CUs depends on the many factors:

- the staked amount
- the locked duration
- the mana factor i.e., CUs per $SQD per epoch
- the boost factor a step function for the most part with the exception of durations between 60 and 180 days where it's linear.

If the duration is set to a value greater than an epoch length then its effect is ignored. Therefore the CUs per epoch follow behave according to the graph below.



Note that the operators have no incentive to stake for longer than an epoch unless they want to stake for longer than 60 days. Then they don't have an incentive to stake for longer than 180 days unless they want to stake for 360 days. Moreover, an operator can use the autoextension option so their $SQD remains staked until they decide otherwise.

7.4 Incorrect Behavior of Variable `depositedIntoProtocol`

Note **Version 1**

The function `Vesting._vestingSchedule()` is the sole consumer of the storage variable `depositedIntoProtocol` inherited from `Executable`. It is meant to track the total $SQD that are currently not in the Vesting wallet itself, but have been deposited elsewhere in the protocol and can be withdrawn at a later time (e.g. locking $SQD as a gateway operator). It is used in `_vestingSchedule()` to compensate the fact that `totalAllocation` (equal to `SQD.balanceOf(address(this))` plus the already-released funds, see OZ `VestingWallet.vestedAmount()`) does not include such deposited funds.

However, the implemented behaviour for this variable, defined in `Executable`, does not match the description. Indeed, if after a call into the protocol (using `Vesting.execute()`) some $SQD are returned to the wallet (e.g. by calling `GatewayRegistry.unstake()`), the variable `depositedIntoProtocol` is simply reset to 0, instead of being decremented by the appropriate delta. This harms the user, in case they have multiple positions open in the protocol through the wallet, which will now be left unaccounted for in the vesting schedule

According to Subsquid, this is not considered to be an issue. However, users should be aware of this particular behavior of `SubsquidVesting` contract.

7.5 Malleability of `msg.data`

Note **Version 1**

In the `DistributedRewardDistribution` contract, the functions `commit()` and `approve()` calculate a commiment hash as `keccak256(msg.data[4:])`, whereas the function `canApprove()` calculates it by explicitly using `abi.encode()`, namely as `keccak256(abi.encode(fromBlock, toBlock, recipients, workerRewards, _stakerRewards))`. This is a slight discrepancy, since `msg.data` is malleable: a caller to `commit()` or `approve()` can construct `msg.data` in many different ways, all encoding the same logical parameters (see https://docs.soliditylang.org/en/v0.8.25/security-considerations.html#minor-details). On the other hand, `abi.encode()` always serialises the parameters in the same way, regardless of how they are encoded in `msg.data`.

Besides the two calculations potentially mismatching (which leads to a potentially wrong answer by `canApprove()`), the very exposure to the malleability of `msg.data` in the `commit()` function is an

issue. The current distributor can inadvertently call the function with a "non-default" encoding: if the other distributors then try to `approve()` using the "default" encoding, the call will revert. Moreover, the current distributor can reset the approval count to 1, by re-submitting the same commit with a different encoding.

7.6 Wrong Initialisation of Epoch Variables

Note Version 1

In contract `NetworkController`, the constructor initialises `firstEpochBlock` as `nextEpoch()`. However, the function `nextEpoch()` is not yet able to return the correct value at this early stage, since it itself relies on the value of `firstEpochBlock` being correct (and not 0).

This leads `firstEpochBlock` to take a value lower than it should (although still in the future), thus reducing the effective duration of epoch 0.

7.7 `addDistributor()` and `removeDistributor()` Change `distributorIndex`

Note Version 1

In the `DistributedRewardDistribution` contract, the functions `addDistributor()` and `removeDistributor()` modify `distributors.length()`, thus changing the return value of `distributorIndex()`.