TheGraph Protocol Audit

OpenZeppelin Security

Introduction

The The Graph team asked us to review and audit their Protocol smart contracts. We looked at the code and now publish our results.

Scope

We audited commit `54f6fa8f1425b4831fec5b4b1a6a9ff64bf9575c` of the The Graph protocol contracts repository. In scope are the smart contracts in the `contracts` directory. However, the following contracts and directories where deemed as out of scope:

- contracts/bancor/BancorFormula.sol
- contracts/discovery/erc1056/EthereumDIDRegistry.sol
- contracts/discovery/erc1056/IEthereumDIDRegistry.sol
- contracts/staking/libs/abdk-libraries-solidity/ABDKMathQuad.sol
- contracts/tests/

High-level overview

The Graph team developed a protocol that takes advantage of Web2 and Web3 technologies to create a decentralized network of servers called indexers that will be the ones in charge of querying information from the Ethereum Blockchain and making it available through an HTTP API, so that DApps can obtain the data they need to work in a decentralized manner. Since it is an important piece of infrastructure for the whole Ethereum ecosystem and taking into account the huge amount of monetary value that will be supporting, the protocol aims to build an incentive design that discourages malicious activities and promote the correct behavior of indexers.

Curation

Curators are the link between customers that need information from the blockchain and indexers, who present this information through a server to facilitate the access to it. For this, The Graph uses an elegant design by which curators signal specific subgraphs, which will take care of specifying important data on-chain, using a bonding curve. When customers pay for the information, a portion of those fees are deposited in the curation pool, so that curators can get an economic revenue for letting indexers know which subgraphs are in current demand.

Staking

Indexers provide easy access to demanded blockchain data by running indexer nodes. Indexers need to stake GRT tokens to be eligible for inflationary rewards and fees from the customers. Indexers can also receive delegation over GRT owners that want to earn rewards but are not tech-savvy enough to run their own nodes. Both indexers and delegators have to wait a specific period of time set by the Governance once they want to remove their funds from the protocol. During this time, only indexers can be slashed with a portion of their stake if they misbehave.

Disputes

For indexers to be slashed, incentivized actors called "fisherman" are allowed to start a dispute by providing proof that the indexer misbehaved. Some of these disputes require the fisherman to send an initial deposit, which will be returned if the dispute is accepted altogether with a reward for keeping the protocol safe.

Roles and their security assumptions

All external code and contract dependencies were assumed to work correctly. Additionally, we assumed that the administrators are available, honest, and not compromised during this audit.

Governor

The protocol defines a `Governor` role which we assume to be a decentralized governance system, which is in charge of doing changes on important protocol variables such as, but not limited to:

- Adding and removing minters of the Graph Token.
- Adding and removing slashers from the `Staking` contract.
- Modifying the thawing period where staker's funds are locked before withdrawal.
- Modifying the percentage of the fees the protocol will burn.
- Pausing and unpausing funds deposits/withdrawals from and to the protocol.
- Modifying the epoch length of the protocol.

If this role is given to an externally owned account or multisig wallet owned by the developers, users should only use the protocol if they trust their owners.

Slasher

The Slasher role is given to the `DisputeManager` contract. However, the `Governor` of the protocol can give slashing permissions to other addresses. They can slash a portion of the amount of staked tokens of an indexer, and give a portion of these tokens to any beneficiary address.

Arbitrator

The Arbitrator role is in charge of accepting, rejecting, or drawing disputes created in the `DisputeManager` contract by the fisherman. If disputes are accepted, misbehaving indexers will be slashed by a percentage of their staked tokens defined by the `slashingPercentage` variable, which is set by the governor.

Enforcer

The Enforcer role allows to blocklist subgraphs so that indexers do not get rewards for indexing these specific subgraphs.

Pause Guardian

The Pause Guardian role is in charge of pausing sections of the protocol in case of unexpected conditions. This state aims to disallow withdrawals and deposits into the protocol.

Proxy Admin

The administrator of the `GraphProxy` is allowed to change the implementation address of the contracts that are upgradeable, thus modifying the whole inner workings of the protocol. They also can transfer the adminship of the proxy to others. If this role is given to an externally owned account or multisig wallet owned by the developers users should only use the protocol if they trust their owners.

Update #1: *All issues listed below have been either fixed, partially fixed or acknowledged by the The Graph team. We address below the fixes introduced in individual pull requests. During the review, the The Graph team did a complete rewrite of the upgradeability system and of the Cobb-Douglas function. Our analysis of the mitigations disregards these changes to the codebase.*

Update #2: *After delivering the review of the fixes for this phase, the The Graph team asked us to review and audit their new upgradeability system. We audited all of the contracts in the `upgrades` directory in commit `a6fc96fe54991dd21141976a241dc93f4e88ba40`. All our findings are present in this phase, marked with an [Upgradeability] tag.*

Here we present our findings.

Critical Severity

[C01] Anyone can steal indexing and query rewards

The `settle` function of the `Staking` contract can be called by an indexer (or delegator after the `maxAllocationEpochs` has passed) to settle a given allocation. This function, among other things, is in charge of closing the allocation and distribute the indexing rewards for both indexers and delegators when a proof of indexing is presented. The task of distributing the rewards is done in the `_distributeRewards` function, which will collect the rewards earned for the number of allocated tokens and then add a portion of those rewards to the indexer's delegation pool, so that delegators can earn tokens when undelegating and later withdrawing their shares.

The problem lies in the fact that there are no restrictions for other external actors to not continue delegating their tokens (which will be included in the indexer's delegation pool) after the indexer has allocated all their stake and the already delegated tokens. Therefore, it is possible for a malicious actor to delegate a huge amount of tokens in order to take a big portion of the delegation pool's shares and redeem a big amount of the accrued tokens.

An example of this attack is explained below:

1. Indexer stakes tokens

2. Delegators delegate tokens to the indexer until the delegation capacity is reached (even though this condition is not necessary to reproduce the attack).

3. Indexer allocates their stake and the delegated tokens to a subgraph deployment.

4. Some time passes, rewards are accrued and the indexer calls the `settle` function to settle the allocation.

5. The `settle` call is front-run by a malicious actor, who delegates to the indexer a big amount of tokens to take a big portion of the delegation pool's shares.

6. The `settle` function is executed, and rewards are sent to the delegation pool.

7. The `settle` function is also back-run by the malicious actor, by calling the `undelegate` function, getting their delegation plus a portion of the rewards.

8. The malicious actor withdraws all their tokens, including their initial stake plus the rewards. Even though there is a lock period defined for delegators, these funds are not at risk since there is no mechanism to slash them.

Note that the bigger the amount of delegated tokens by the malicious actor is, the bigger will be their shares in the pool and a bigger portion of the rewards will be stolen. Additionally, note that this same pattern can be exploited before the `claim` function is called, to steal the query rewards.

A step-by-step proof-of-concept exploit for this scenario can be found in this private gist.

Consider disallowing that newer delegators can receive indexing and query rewards over allocations that were initially made without their tokens.

Update: *Partially fixed in PR#383. A delegation tax was added to discourage this attack, but it is still possible for an attacker to take a portion of a delegation pool and therefore take a portion of the indexing rewards, causing other honest delegators to get less rewards than the amount they would earn if this attack would had been mitigated by disallowing newer delegators to receive rewards over allocations that were initially made without their tokens, which the The Graph team is aware of. Lastly, depending on the delegation parameters set by the indexer, it is also possible for the attacker to make profit from the attack. In words of the The Graph Team: "Our analysis showed that under reasonable assumptions, a properly set delegation tax would make the attack unprofitable. I.e., they would lose more from the tax than they would gain by front-running."*

[C02] Malicious indexer can make profit without indexing subgraphs

Indexers can stake tokens for later allocating them in one or more subgraphs deployments. Given that indexers might be capital-constrained, the system allows delegators to give their tokens to indexers so they can allocate them on their behalf, increasing the amount of allocated tokens in a subgraph and therefore increasing the amount of earned rewards. An indexer can `settle` an allocation anytime (after at least one epoch passes) by presenting a proof of indexing, and distribute the indexing rewards between them and delegators respecting the delegation parameters set by the indexer. If the proof of indexing presented is not valid, the indexer can go through a slashing process, in which a portion of their staked tokens can be taken from them.

The problem lies in the fact that it is possible for a malicious indexer to earn more tokens than the slashed amount. This can be accomplished by setting the delegation parameters in such a way that all the indexing rewards go to the delegators, avoiding an increase of the indexer's staked amount, and therefore limiting the slashed amount to the initial stake. For this attack to be possible, the malicious indexer has to also act as a delegator, and reach the indexer's delegation capacity to avoid other delegators to earn rewards from this attack (as long as the issue: *Anyone being able to steal rewards* is fixed).

A malicious actor can then:

1. Deploy and curate a subgraph, staking a relatively low amount of GRT to avoid other indexers to be able to discover it

2. Stake an amount of tokens as indexer

3. Delegate tokens to themselves reaching the indexer's delegation capacity

4. Allocate the staked and delegated tokens in the subgraph

5. Settle the allocation after some time passes, presenting an invalid proof of indexing and sending the indexing rewards to the delegator

6. Even though the indexer will be slashed, the malicious actor withdraws the rewards as a delegator without consequences, as there is no mechanism defined to slash delegators.

Additionally, the malicious actor can back-run their own call to the `settle` function, calling the `createIndexingDispute` function of the `DisputeManager` contract, and recover a portion of the slashed amount.

A step-by-step proof-of-concept exploit for this scenario can be found in this private gist. The values used for the exploit are within the parameters discussed with the The Graph team throughout the course of this audit, and a relatively low amount of tokens were used to curate the subgraph (0.003% of the total amount of tokens in the `Curation` contract).

Consider distributing delegation rewards after slashing the indexer, and avoid distributing them if an invalid proof of indexing was presented.

Update: *Partially fixed in PR#383. A delegation tax was added to discourage this attack, but it is still possible for an attacker to reproduce it under certain boundaries. In words of the The Graph Team: "Our analysis showed that under reasonable assumptions, a properly set delegation tax would make the attack unprofitable. I.e., they would lose more from the tax than they would gain by front-running."*

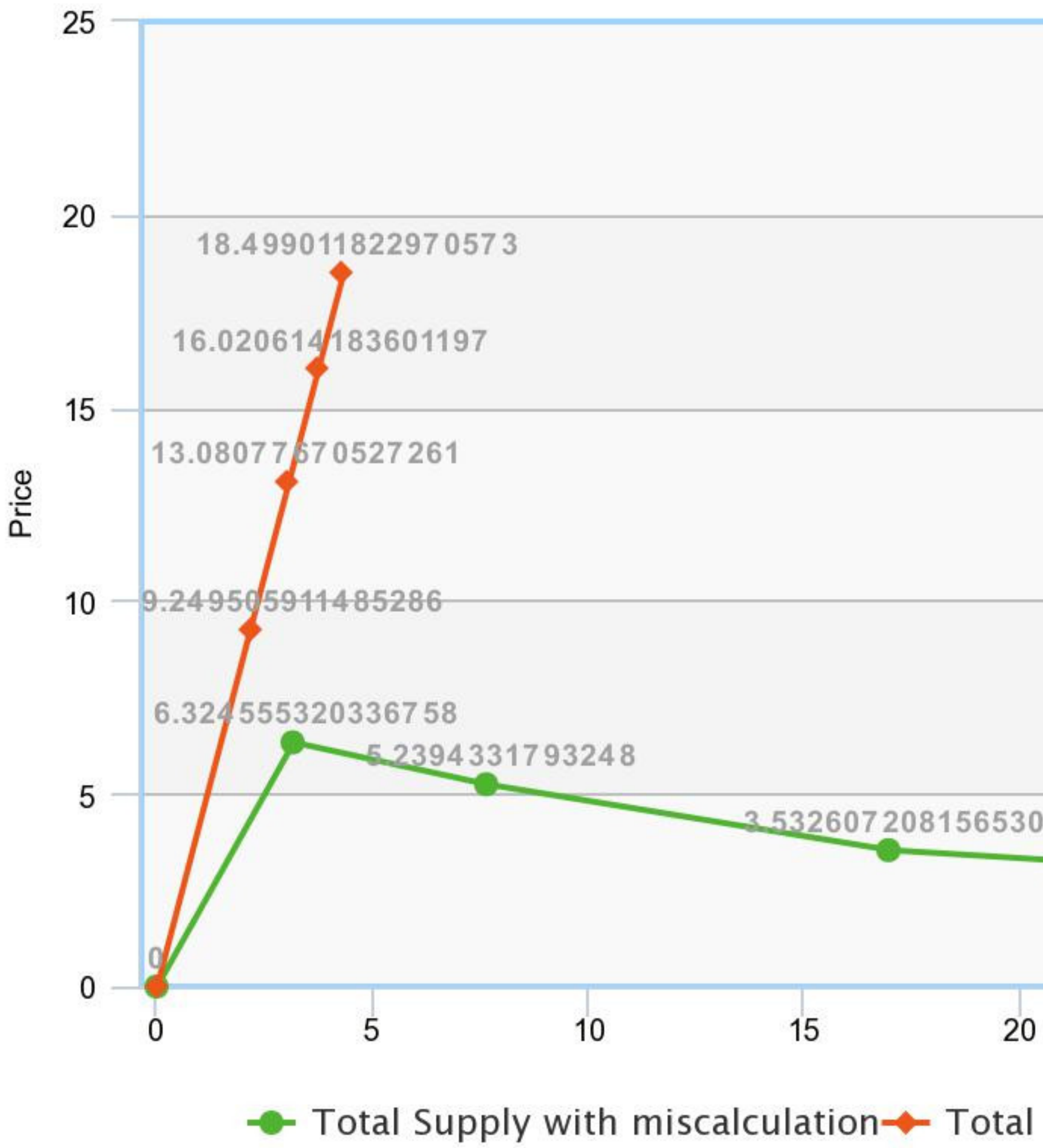
[C03] Miscalculation on Signal Token minting

The `Curation` contract allows curators to signal subgraphs that might be valuable to index by depositing an amount of GRT into a bonding curve, in exchange of GST. They do that by calling the `mint` function. Analogously, a curator can redeem their GRT anytime, by calling the `burn` function and receiving back an amount proportional to the total amount of GRT already staked in the pool.

These two functions make use of the `BancorFormula` contract, which provides the functionality needed to retrieve how many GST should be minted on a purchase and how many GRT should be retrieved on a sell operation.

The problem lies in the fact that the `tokensToSignal` function used in the `mint` function to convert GRT to GST is unnecessarily adding the current total supply of GST tokens to the amount of tokens to be purchased, which leads into a *bigger amount of GST minted*, and therefore into misbehavior in the bonding curve.

Given a `Reserve Ratio` of 50% where the bonding curve is expected to be a linear function, we can see the actual behavior in the graph below:



This shows that, in addition to an undesired bigger amount of GST minted, the price of GST *decreases* when its total supply increases (after a certain amount of tokens minted), instead of increasing. The step-by-step proof-of-concept example used for plotting the chart above can be found in this private gist.

Consider modifying the `tokensToSignal` function so it retrieves the value calculated by the `calculatePurchaseReturn` function of the `BancorFormula` contract instead of returning this value plus the total supply of GST for the given subgraph.

Update: *Fixed in PR#333. Now the `tokensToSignal` function is not adding anymore the total supply of GST token and it is exclusively adding `SIGNAL_PER_MINIMUM_DEPOSIT` whenever the pool needs to be initialized.*

[C04] Curator's GRT can get stuck in the `Curation` contract

The `Staking` contract collects query fees through the `collect` function. A portion of these fees are sent to the subgraph's curation pool in the `Curation` contract so that curators can be rewarded for curating subgraphs in the system. The amount of GRT that a curator receives when burning GST is directly proportional to the total amount of GRT deposited in the pool and is defined by the Bancor Formula as follows:

$$GRT\ Amount = Reserve\ token\ balance \times$$

In short, there are two ways of increasing the amount of GRT received when burning GST:

- By being an early curator in a pool, given a `Reserve Ratio < 1` and by unstaking GRT later in the curation process

- When the `Staking` contract collects fees from the asset holder in the `collect` function, since the `Reserve token balance` value in the formula above will increase.

The problem lies in the fact that, when the `Staking` contract sends these fees to the `Curation` contract by calling its `collect` function, only the pool's total reserve will be updated but not the contract's total reserve tracked in the `totalTokens` variable. Therefore, if a curator tries to redeem their GRT by calling the `burn` function, it will potentially revert as it will try to subtract a greater amount of GRT than the one available in the `totalTokens` variable, leading into the impossibility to redeem tokens.

Moreover, given that the `totalTokens` variable is used in the `RewardsManager` contract to calculate the rewards accrued per signal, the indexing rewards will be over-calculated. This means that the lower the amount of GRT tracked in the `Curation` contract is, the greater the accrued amount will be, and thus more tokens will be minted for rewards which will also potentially cause an undesired rise in the inflation rate defined by the `issuanceRate` variable.

Note that the more curation fees are collected, the greater the difference between the `totalTokens` amount and the sum of all the pools' reserves will be, ultimately worsening the impact of the issue.

A step-by-step proof-of-concept exploit for this scenario can be found in this private gist.

Consider either adding the collected tokens to the `totalTokens` variable in the `_collect` function of the `Curation` contract to prevent the unsynchronization between the amount stored in this variable and the sum of all the pools' reserves or removing the `totalTokens` variable and track the total amount of tokens by checking the balance of GRT of the `Curation` contract when appropriate.

Update: *Fixed in PR#336. The `totalTokens` variable has been removed and `balanceOf(curation)` is being used now both in the `Curation` and `RewardsManager` contracts.*

High Severity

[H01] Curation and protocol fees are not enforced to be collected

After an allocation is created, the protocol expects that the `collect` function is called by an allocation's `assetHolder` address, which will be the address collecting payments from the customers.

This function will collect fees only if customers send payments to the indexer over an indexed subgraph and will redistribute them to the curation pool, the protocol, and the indexer.

However, the `assetHolder` value is set by the indexer whenever the `allocate` function is called. This is an important role for the protocol incentive's health because if the fees are not sent to the curators, their only incentive would be to wait for other curators to signal the same subgraph curation pool and get returns on their investments, instead of relying also on the fees that the indexer is making by indexing that specific subgraph.

Since it is an important role, it should be trustlessly set by the system and not by the indexer's input. If the `assetHolder` address is set to an externally owned account or smart contract owned by the indexer, it is possible for them to keep the total amount of fees produced by the subgraph. If every indexer in the protocol do this in order to take more profit from their investment, then curators will not be correctly incentivized to signal subgraphs.

Consider enforcing that the `assetHolder` address is a smart contract deployed by the system that calls the `collect` function from the `Staking` contract with the correct `allocationID` for every payment received by the customers.

Update: *Fixed in PR#350. Asset holders addresses are not provided by indexers anymore when allocating tokens, but by the governor of the system.*

[H02] Slash can be bypassed

The `DisputeManager` contract takes care of slashing misbehaving indexers by calling the `slash` function of the `Staking` contract after a fisherman creates a query or indexing dispute. The `minimumIndexerStake` variable is used throughout the contract to validate that indexers *only* get slashed if they have at least this amount of tokens staked in the system.

However, the `Staking` contract does not establish a minimum amount of stake per indexer, allowing them to stake less than the `minimumIndexerStake` amount, which immediately makes them unslashable in the `_createQueryDisputeWithAttestation` and the `_createIndexingDisputeWithAllocation` functions of the `DisputeManager` contract.

By taking advantage of this design decision, a malicious indexer will be able to receive inflationary indexing rewards from the moment the allocation is created up to its settlement without the need of properly indexing a subgraph and without taking risks on their stake. The benefit of this attack can be magnified by attackers if they also delegate more tokens to their indexers from another account up to fill the indexer's capacity.

Since slashing is the primary defense mechanism over indexers returning malicious information to customers or presenting invalid proof of indexing values, consider modifying this behavior and avoid conditions where indexers are unslashable to maintain the security of the protocol.

Update: *Fixed in PR#347. The `minimumIndexerStake` is now enforced in the `Staking` contract and slashing can happen now for any non-null amount of stake. `minimumIndexerStake` is enforced when calling `stakeTo` or `unstake` functions. Notice that this is not enforced in the internal `_stake` function since it can be called also by "functions that increase the stake when collecting rewards".*

[H03] Functions that are missing pausing functionality

According to The Graph's Notion documentation file, the `Pause` and `RecoveryPause` mechanisms are expected to:

```
Paused pauses all protocol functions that involve the moving around of assets. Recovery_Paused sho
```

However, not every function "moving around assets" is using the `notPaused` modifier, and not every function that "enters assets into the protocol" is using the `notRecoveryPaused` modifier. Examples of these are:

- The `deprecateSubgraph`, `withdraw`, `mintNSignal`, `burnNSignal` and the `publishNewVersion` functions of the `GNS` contract, which move assets directly or in internal function calls.

- The `createQueryDispute` and `createIndexinispute`, the `acceptDispute` and the `drawDispute` functions from the `DisputeManager` contract, which moves funds by using internal function calls.

- The `slash` function of the `Staking` contract, which moves funds from the contract to the beneficiary of the rewards.

Consider carefully analyzing each of the reported cases and decide whether the lack of restriction of these functions is appropriate. Moreover, consider publicly documenting the tradeoffs of this decision so that users are aware of why and how these two states of the protocol are going to be used.

Update: *Fixed in PR#360. The team carefully reviewed this issue and decided to add pausing modifiers to some of those functions.*

[H04] Updating the proxy admin address emits no events

In the `GraphProxyStorage` contract, the `ADMIN_SLOT` memory pointer is where the proxy admin account is saved. This account is highly sensitive as it is the only one able to modify the Graph upgradable contract implementations, allowing changes in the protocol's behavior.

This variable can be updated by calling the `setAdmin` function of the `GraphProxy` contract, and since it updates the admin address without logging any events, users and indexers of the Graph protocol would need to inspect all transactions to notice that one address they trust is replaced with an untrusted one.

Consider emitting events when this address is updated. This will be more transparent, and it will make it easier for clients to subscribe to the events that want to keep track of the status of the system.

Update: *Fixed in PR#356.*

[H05] Sensitive functions inherited in implementations

The `GraphUpgradeable` contract inherits from the `GraphProxyStorage` contract which has sensitive internal functions to set the different addresses of the reserved storage slots (`admin`, `implementation` and `pendingImplementation`).

This means that all the contracts that inherit from `GraphUpgradeable` will also inherit all those functions.

Even if none of the contracts that inherit from `GraphUpgradeable` call directly the `GraphProxyStorage` functions, implementation contracts should not know anything about proxies, nor be able to modify their variables, unless following the EIP 1822. Moreover, the only reason why the `GraphUpgradeable` contract is inheriting from the `GraphProxyStorage` contract is to make use of the `onlyImpl` modifier.

To avoid this, consider the possibility of a new inheritance design, or consider using the OpenZeppelin upgrades library.

Update: *Fixed in PR#357.*

Medium Severity

[M01] Delegators can be deceived by indexers

Indexers are allowed to set delegation parameters through the `setDelegationParameters` function of the `Staking` contract, which updates the `indexingRewardCut` and the `queryFeeCut` of that particular indexer's delegation pool, ultimately determining the amount of fees that will be given to delegators.

By setting the `cooldownBlocks` variable, the protocol also enforces the amount of blocks in the future when these parameters can be changed again. However, once this cooldown period passes, there is no incentive in setting them to different values.

Nevertheless, this behavior allows for indexers to call the `setDelegationParameters` right before the `settle` or `claim` functions, which are the ones in charge of distributing the indexing and the query fees rewards respectively. As a result, it will be possible for indexers to withdraw the whole amount of fees without distributing them to delegators.

Consider modifying the `setDelegationParameters` so that delegators can opt-out of the modifications that the indexer does to the delegation parameters by implementing a timelock mechanism.

Update: *Acknowledged. In the words of the The Graph team: "Indexers that misbehave and cheat by repeatedly changing conditions will lose reputation and be socially penalized. Indexers might have the incentive to reset the cooldown or set high enough to create confidence".*

[M02] Rebate fees may get stuck on a certain epoch

The `settle` function of the `Staking` contract is called by an indexer or delegator to close a given allocation. Among other things, this function is in charge of sending both the collected fees by asset holders and the effective tokens allocation to the rebate pool of the epoch in which the allocation is settled. When the `claim` function is called, these values, together with any other collected fees after settling the allocation, are used to calculate the amount of query rewards to redeem from the rebate pool following the Cobb-Douglas production function which is used in the system as follows:

$$Y = \Theta \left(\frac{\omega}{\Omega}\right)^{\alpha} \left(\frac{\theta}{\Theta}\right)^{1-\alpha}$$

Where, for a given subgraph in a given epoch, `Y` is the rebate an indexer receives, `Î±` is the output elasticity and is defined as `0.5`, `Î¸` represents the collected fees of the indexer, `Î˜` represents the total collected fees, `Ï‰` represents the effectively allocated tokens by an indexer, and `Î©` represents the total effective allocated tokens.

Because of how this function works, chances are that even after all indexers redeemed their tokens from the rebate pool from a given epoch, some tokens will get stuck. This issue will be magnified by the following scenarios:

- When the second term of the formula above equals zero, that is, if no fees are collected by the indexer, as the effective allocation will anyway contribute to increasing the pool's size, but no query rewards will be redeemed.

- When an incorrect proof of index is presented, as the effective allocation will also contribute to increasing the pool's size, and therefore other honest indexer's shares in the pool will dilute, which will also contribute to leaving more collected fees stuck in the rebate pool.

Consider either burning or distributing the remainder rewards of each epoch, or consider redesigning the system in order to avoid this scenario.

Update: *Fixed in PR#335. The edge case where no fees are collected by the indexer is addressed in PR#368. Any stuck token amount is now burned whenever allocations are claimed. Notice that these changes have been introduced along the decision to change the contract implementation of the Cobb-Douglas formula. We assume these changes have not introduced any new issue or vulnerability.*

[M03] Query disputes can be front-run impeding slashing

The `DisputeManager` contract allows slashing malicious or mal-functioning indexers returning incorrect data to users by using the `createQueryDispute` and the `createQueryDisputeConflict` functions. These can be called by a fisherman who will submit a dispute that will be saved in the `disputes` mapping and will then be resolved by the `Arbitrator` by either accepting, rejecting, or drawing it.

Both aforementioned functions make use of the internal `_createQueryDisputeWithAttestation` function, which will check whether the `disputeId` already exists and only after this check the dispute will be created. As the `disputeId` variable is created by using the `keccak256` hashing function on the information in the `_attestation` parameter, malicious indexers can take advantage of this behavior by front-running any correct dispute against them with invalid dispute conflicts, created to be rejected by the arbitrator.

The way in which an invalid dispute conflict can be created is to make use of the `createQueryDisputeConflict` with the attestation information of the valid dispute we want to front-run and use an unrelated attestation information as the second parameter, such as an attestation with an invalid `responseCID`. This way, the arbitrator role will only have visibility of an incorrect dispute conflict which could be rejected because of its invalidity, but the indexer gets benefitted as the thawing period gets shorter for them to be slashed. This process can be repeated until the thawing period is over and the indexer removes their stake.

Moreover, as the `createQueryDisputeConflict` function does not require a deposit, they can do this without exposing themselves to more risks other than their staked capital.

Consider adding the `_fisherman` parameter information to the hash used to create the `disputeId` so that the same attestation information will create a different `disputeId` for every caller.

Update: *Fixed in PR#386.*

[M04] Delegators can trigger a slashing process when settling an allocation

It is possible for a delegator to call the `settle` function of the `Staking` contract presenting an invalid proof of indexing (on purpose or by mistake) after the `maxAllocationEpochs` has passed, and therefore trigger a slashing process to the indexer that created the allocation in the first place.

If it is mandatory for the system to accept allocation's settlements from delegators, consider implementing a `setProofOfIndexing` function that can be called by an indexer (only in case they will not be able to settle the allocation themselves) before the `maxAllocationEpochs` passes, and use that proof of indexing to execute the settlement triggered by the delegator.

UPDATE: *Fixed. In PR#385 delegators calling the `closeAllocation` function (former `settle` function in the audited commit) will not trigger the `_distributeRewards` function anymore. This should discourage delegators to call this function since no rewards will be distributed, but if they do call it,* ***both delegators and indexers will lose indexing rewards****. Since delegators can only settle an allocation after the `maxAllocationEpochs` period has passed, this would also discourage indexers to misbehave and leave an allocation opened indefinitely. Moreover, in PR#419, the team added a new field in the emitted event to inform whether the caller was a delegator or not. This should be useful for a `fisherman` when deciding to slash an indexer because of an invalid `_poi` presented by a malicious delegator.*

[M05] `subgraphsID`s are not unique

The `GNS` contract uses the `subgraphs` nested mapping to track the subgraph information and save a `_subgraphDeploymentID` variable which will be the id by which the subgraph will be referenced in the protocol. The value in this mapping is also saved to populate the `subgraphDeploymentID` of the `NamePool` struct of the `nameSignals` mapping in the `_enableNameSignal` function.

The `subgraphDeploymentID` variable is then used in both the `_mintNSignal` and the `_burnNSignal` private functions when calling the `mint` and `burn` functions of the `Curation` contract respectively.

As this parameter is supplied by the user in the `publishNewSubgraph` and `publishNewVersion` functions, its value may not be unique for different subgraphs on the `nameSignals` mapping of the `GNS` contract but point to the same subgraph in the `pools` mapping of the `CurationStorage` contract.

Consider establishing the subgraph ids as the result of the `keccak` operation applied on the concatenation of the `_graphAccount` and `_subgraphNumber`, to create unique values for each subgraph to assure that every `subgraphDeploymentID` variable of the `GNS` contract will point to a unique `pool` in the `CurationStorage` contract.

Update: *Acknowledged. In the words of the The Graph team: "The reason subgraphIDs are not unique is that two (or more) Subgraph Names can target the same subgraphDeploymentID.".*

[M06] Tokens ready for withdrawal can get locked again by future unstaking calls

The `unstake` function of the `Staking` contract allows an indexer to unstake a given amount of tokens that are not allocated in a subgraph, nor locked for withdrawal. This function will lock the given stake amount until the thawing period passes, moment from which the indexer will be able to withdraw their tokens by calling the `withdraw` function.

The problem lies in the fact that when an amount of tokens has been unstaked and is ready to be withdrawn (because the thawing period has passed), unstaking another amount of tokens before a withdrawal will potentially make the former tokens to get locked again, since a new locking period will be calculated for all non-allocated tokens.

Given an indexer that has staked 500 tokens, with no unstaked tokens, and a `thawingPeriod = 120`:

- Indexer calls the `unstake` function in block 2000 to unstake 200 tokens.
- 200 out of 500 tokens get locked for withdrawal until block 2120. After that block number, the indexer will be able to withdraw them.
- Indexer calls the `unstake` function in block 2150 to unstake the remaining 300 tokens. (note that before the call, the former 200 tokens were ready for withdrawal, as they were already unlocked in block 2120).
- A new `lockingPeriod` is calculated by calling the `getLockingPeriod` function, which will calculate a weighted average of periods based on the already unstaked tokens and the amount that is going to be unstaked.

Given the numbers mentioned above, the locking period, which follows the following formula will be calculated:

$$lockingPeriod = \frac{(periodA \times sta}{st}$$

Resulting in a new locking period of `2150 + 72 = 2222`, which is greater than the locking period of the former stake and will lock its withdrawal until that block, even though it was ready to be withdrawn.

In addition to this, note that there will be a miscalculation of the new locking period for the latter unstake, as the formula mentioned above will take into account the ready-to-withdraw amount of tokens in the denominator.

Consider tracking the amount of tokens ready for withdrawal in another attribute in the `Stakes.Indexer` struct, and subtract them from the `tokensLocked` variable.

Update: *Fixed in PR#342.*

[M07] Delegators stake can get soft-locked

There are no validations in the `delegate` function nor in the `_delegate` function to check whether the indexer to whom the tokens are going to be delegated has staked tokens. If a delegator sends a wrong `_indexer` value as a parameter, their tokens will be delegated to the delegation pool of an unexistent indexer, and if they notice this and want to undelegate and later withdraw or re-delegate their tokens by calling the `withdrawDelegated` function, they will have to wait until the delegation unbonding period passes before being able to do so.

Consider checking whether the indexer address sent by parameter to the `delegate` function has tokens staked, and revert otherwise.

Update: *Fixed in PR#341. Delegators trying to delegate tokens to an indexer that has no stake will fail in calling the `delegate` function.*

[M08] Initialization of contracts without setting parameters values

Upgradeable contracts throughout the code base do not initialize important variables in the `initialize` function, nor in the `constructor`, and rely on different transactions to set values to these variables.

Examples of this are:

- The `initialize` function of the `Curation` contract does not initialize the `withdrawalFeePercentage` variable.
- The `initialize` function of the `RewardsManager` contract does not initialize the `issuanceRate` nor the `enforcer` variables.
- The `initialize` function of the `Staking` contract does not initialize the `thawingPeriod`, `curationPercentage`, `protocolPercentage`, `channelDisputeEpochs`, `maxAllocationEpochs`, `delegationCapacity`, `delegationParametersCooldown` nor the `delegationUnbondingPeriod` variables.
- The `constructor` function of the `DisputeManager` contract does not initialize the `minimumIndexerStake` variable.

This behavior is error-prone and could lead to mistakes in the deployment process resulting in an unexpected functioning of the system.

Consider initializing every single state variable declared in the smart contracts in the `initialize` or the `constructor` functions.

Update: _Fixed in PR#364. _

[M09] Potential function clashing between proxy and implementation

The upgradeability system implemented in code base does not manage function clashing between the proxy contract and the implementation contract.

Clashing can happen among functions with different names. Every function that is part of a contractâ€™s public ABI is identified, at the bytecode level, by a 4-byte identifier. This identifier depends on the name and arity of the function, but since it is only 4 bytes, there is a possibility that two different functions with different names may end up having the same identifier. The Solidity compiler tracks when this happens within the same contract, but not when the collision happens across different ones, such as between a proxy and its logic contract.

Upgradeable contract instances (or proxies) work by delegating all calls to a logic contract. However, the proxies need some functions of their own, such as the `upgradeTo` function and the `acceptUpgrade` to upgrade to a new implementation. This means that there can be a function in the proxy contract with the same 4-byte identifier as one in the implementation contract, which would lead to the impossibility of accessing to the one defined in the implementation contract, as the call will not be handled by the fallback function but by the function with that signature in the proxy contract.

Considering implementing the transparent proxy pattern designed by OpenZeppelin to mitigate this issue. Additionally, consider thoroughly testing all functions implemented in each upgradeable contract to ensure that no collisions are possible.

Update: *The The Graph team did a complete refactor of their upgradeability system in PR#410. The review of this issue was limited to check whether the function clashing can still happen, which is not the case, so the issue was fixed. Any other modification in the upgradeability system was not thoroughly reviewed since it was out of the scope of this issue.*

[M10] Untested, undocumented assembly blocks

The following functions include untested and undocumented assembly blocks:

- The `sliceByte` function of the `Staking` contract
- The `_pow` function of the `RewardsManager` contract

While these do not pose a security risk on their own, they are at the same time complicated and important parts of the system. Moreover, since it's a low-level language that is harder to parse by readers, consider including extensive documentation regarding the rationale behind its use, clearly explaining what every single assembly instruction does, similar to how it is done in the `fallback` function in the `GraphProxy` contract. This will make it easier for users to trust the code, for reviewers to verify it, and for developers to build on top of it or update it.

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

Update: *Partially fixed. The `_sliceByte` function has been removed in PR#346 and the `_pow` function now has dedicated tests added in PR#354.*

[M11] Cancelling an ownership transfer can be front-run

The `Governed` contract use a two-steps transfer of ownership where the new potential owner must accept the transfer to finalize it.

Even though there is no direct function to cancel an initialized transfer, the `transferOwnership` function can be used to do so by setting the new pending owner to the zero address.

If a transfer needs to be canceled because the recipient owner got hacked or the wallet gets compromised somehow, the cancel operation can be easily front-run by the same person that took control of the recipient address. This cannot be avoided easily and it is a known limitation of the chosen pattern but it has to be carefully taken into considerations when initializing a transfer.

Update: Acknowledged by the The Graph team.

[M12] Lack of input validation

Throughout the code base, there are multiple situations where the input function's value is not checked before being used. In particular:

- The `setContractProxy`, the `updateController` and the `setPauseGuardian` functions of the `Controller` contract do not validate any of the inputs to be different from `address(0)`.
- The `transferOwnership` function of the `Governed` contract doesn't validate the `_newGovernor` parameter to be different than the current `governor` or `address(0)`.
- The `setController` function of the `Managed` contract does not check that the `_controller` parameter is different from the current `controller` value or `address(0)`.
- The `_setPauseGuardian` function of the `Pausable` contract does not check that the `newPauseGuardian` parameter is different from the current `pauseGuardian` value or `address(0)`.
- The `initialize` function of the `Curation` contract does not validate the `_controller`, `_defaultReserveRatio` and `_minimumCurationDeposit` parameters.
- The `setArbitrator` function of the `DisputeManager` contract does not validate that the `_arbitrator` parameter is different from the current `arbitrator` value or `address(0)`.

- The `setMinimumDeposit` and the `setMinimumIndexerStake` functions of the `DisputeManager` do not validate that their parameters are bigger than 0.

- The `setThawingPeriod`, `setCurationPercentage`, `setProtocolPercentage`, `setChannelDisputeEpochs`, `setDelegationCapacity`, `setDelegationParametersCooldown` and `setDelegationUnbondingPeriod` functions do not validate if their parameters are bigger than 0.

- The `setIssuanceRate` function of the `RewardsManager` contract does not validate that the `issuanceRate` variable is greater than 1.

Even though this issue does not pose a security risk if validations are being made before submitting the transactions, the lack of validation on user-controlled parameters may result in erroneous transactions. Consider adding the needed validation checks in the input parameters of the reported cases and in the entire code base.

Update: *Fixed in PR#402. Some of the mentioned functions have been intentionally left out to accept null values whenever the intention is to disable the feature.*

[M13] [Upgradeability] Incorrect event emission

The `setAdmin` function in the `GraphProxy` contract sets a new admin address for the proxy contract. This function calls the internal `_setAdmin` function from the `GraphProxyStorage` contract to do so. After setting the address of the new admin, `_setAdmin` emits the `AdminUpdated` event, defined as containing the old and new admin addresses. However, `_setAdmin` retreives the `oldAdmin` from the `ADMIN_SLOT` *after* updating the slot to store the `newAdmin`. This means that the event actually emits the new admin's address as both the `oldAdmin` and the `newAdmin`.

Consider updating the `_setAdmin` function to fetch the old admin's address before overriding the slot with the new admin.

Low Severity

[L01] Allocation is front-runnable by malicious indexers

The `allocate` function of the `Staking` contract makes use of the internal `_allocate` function to create a new allocation, which will be saved in the `allocations` mapping under the `allocationID` key.

The problem is that the `_channelPubKey` parameter that is used to derive the `allocationID` address is not related in any way to the `indexer`, and once the allocation is created with its associated `allocationID`, any subsequent call with the same `_channelPubKey` will revert. This can be used by malicious indexers to front-run others allocation creations, setting themselves as an indexer for that particular `allocationID`. Although this does not pose further problems in the protocol, we recommend to The Graph's team to validate whether this behavior is expected and whether this poses additional security risks in other parts of the protocol.

Update: *Fixed in PR#403. The malicious indexers now need to present proof of a message signed by the private key corresponding to the `allocationID` address. This way, there are no possibilities for malicious actors to provide a valid proof without having the corresponding private key. The `allocationID` is still enforced to be unique.*

[L02] Events not showing modified information

The `ParameterUpdated` event is declared in the `Managed` contract and used extensively in the `Staking`, `Curation`, `GNS`, `DisputeManager` and `RewardsManager` contracts.

Even though this event is trying to simplify and reuse the same event for each of the contracts' governable parameters, it does more harm than good, since clients will know that a certain parameter has been updated but will need to query the blockchain in order to fetch the old and new values. This adds complexity and inefficiency to programs depending on these parameter's values.

Consider creating a separate event for each of the governable parameters, showing both the previous and the updated value of the modified variable.

Update: *Acknowledged. In the words of the The Graph team: "We rely on the use of The Graph Subgraphs, when emitting one of those parameter change events, the contract can be queried for the proper information. We will provide the mappings for our Network Subgraph to the community for reference".*

[L03] Delegators can misbehave without consequences

A delegator can entrust an amount of tokens to an indexer by calling the `delegate` function so that the latter can allocate them in a subgraph deployment in order to earn indexing rewards when an allocation is settled, and query rewards when it is claimed.

In the case of settling an allocation, a proof of indexing must be presented in order to distribute indexing rewards. If the proof of indexing turns out to be invalid, slashers will be able to penalize the indexer before a thawing period passes and perform a slash for a given amount of tokens.

The issue lies in the fact that, even if the proof of indexing presented is incorrect, the `settle` function will still distribute rewards to both the indexer and delegators. In the case of the indexer, this could later be corrected by going through a slashing process if *Malicious indexer can make profit without indexing subgraphs* is fixed, but delegators will anyway receive inflation rewards that could later be withdrawn, since there is no mechanism to avoid the distribution them, nor a slashing mechanism to penalize misbehaviors, leading into scenarios such as the one reported in *Malicious indexer can make profit without indexing subgraphs*, and in *Anyone can steal indexing and query rewwards*.

Consider implementing a mechanism in which the system could redeem the delegator's rewards if their indexer misbehaves in any way, or consider not distributing rewards to delegators until the arbitrator validates that the proof of indexing presented is valid.

Update: *Although this issue was acknowledged by the The Graph team with the argument that "It is a design decision to not have the delegators be slashed", the issue is mitigated by the fact that now rewards are not distributed if the caller of the `closeAllocation` function (former `settle` function in the audited commit) is not the indexer as introduced in PR#385.*

[L04] Curator can receive an unexpected amount of tokens

Curation pools use a bonding curve to calculate how much GST or GRT will be returned by a `mint` or `burn` operation respectively. Given the fact that the bonding curve moves as a per-transaction basis, it presents frontrunning risks by design.

For instance, when a curator triggers a purchase of GST in exchange of GRT expecting a certain amount of the former given a particular moment in the bonding curve, if another `mint` transaction with a big amount of GRT is sent by another curator and executed before, the amount of GST that the initial curator receives will be lower than the expected and displayed by the system before doing the purchase. Note that this same case applies when doing the opposite transaction, that is, selling GST in exchange of GRT.

Consider allowing the user to send a minimum buy amount of either GST or GRT as a parameter and check whether the amount bought is lower than this minimum, or consider allowing the user to set a maximum slippage tolerance and check that the price variation is within the slippage, and otherwise revert to minimize exposure to this behavior.

Update: *Partially fixed in PR#369 and PR#395. Slippage protection was implemented by adding two new parameters in the `mint` and `burn` functions of the `Curation` contract, and also in the `burnNSignal` and `mintNSignal` functions of the `GNS` contract. No slippage protection has been given to the `_burnVSignal` and `_upgradeNameSignal` functions.*

[L05] Lack of event emission

In several parts of the code, there are sensitive functions that lack event emissions. This can make it difficult for users to track important changes that take place in the system.

Some examples of these are:

- The `_initialize` function of the `Governed` contract should emit a `NewOwnership` event.
- The `_initialize` function of the `Managed` contract should emit a `SetController` event.
- The `_initialize` functions of the `Staking` and `Curation` contracts should emit the `ParameterUpdated` event for each parameter set.

- The `constructor` of the `DisputeManager` contract should emit an event or make use of the different setter functions for each one of the storage variables being set.

Consider adding and emitting events to make it easier for offline services to track important contract storage changes.

Update: *Partially fixed in PR#364. Some of the proposed event emissions were implemented.*

[L06] Shortcomings in testing practices

While doing a general review of the project's test suite, a number of shortcomings were identified. In particular:

- External libraries and copy-pasted parts of the code are being used without testing. Most of the Solidity libraries are expected to be used under certain system properties and extrapolating them to another one can create misunderstanding issues on how these libraries should work. For this reason, it is extremely important that these libraries are heavily tested under The Graph's team development assumptions to check if these libraries are compatible with the protocol being developed.

- There are not integration tests demonstrating the whole workflow of actions through the protocol. Different parts of the system are tested in a self-constrained way which is not the best as there could be problems in the interaction between two parts of the code heavily related. An example of this is the lack of integration tests between the `GNS` and the `Curation` contract.

Having a healthy, well documented and comprehensive test suite is of utter importance for the project's overall quality, helping specify the expected behavior of the system and identify bugs early in the development process.

Consider enhancing the current testing practices to consistently test for the assumptions under the system will run under, making special efforts in external libraries and integration tests to assure that the system works as expected.

Update: *Acknowledged. In the words of the The Graph team "Tests for libraries like the rebate pool calculation and `_pow()` function used in `RewardsManager` where added in different PRs. Adding more scenario tests for the different flows is WIP".*

[L07] Contracts storage layout can be corrupted on upgradeable contracts

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

This upgradeability system consists of a proxy contract which users interact with directly and that is in charge of forwarding transactions to and from a second contract. This second contract contains the logic, commonly known as the implementation contract. When using this particular upgradeability pattern, it is important to take into account any potential changes in the storage layout of a contract, as there can be storage collisions between different versions of the same implementation. Some possible scenarios are:

- When changing the order of the variables in the contract
- When removing the non-latest variable defined in the contract
- When changing the type of a variable
- When introducing a new variable before any existing one
- In some cases, when adding a new field to a struct in the contract

This might be partially mitigated by separating the storage from the logic into two different contracts, but there is still no certainty that the storage layout will remain safe after an upgrade. Violating any of these restrictions will cause the upgraded version of the contract to have its storage values mixed up, and can lead to critical errors in the contracts.

Consider checking whether there were changes in the storage layout before upgrading a contract by saving the storage layout of the implementation contract's previous version and comparing it with the storage layout of the new one. Additionally, consider using the Openzeppelin Upgrades library, which already covers some of the scenarios mentioned above.

Update: *Acknowledged. In the words of the The Graph team "We are using separate storage and logic contracts to reduce the risk of changing the storage layout. It would be interesting to use the layout checker from `openzeppelin-upgrades` but it is not currently supporting structs".*

[L08] Externally Owned Accounts can be set as implementation contracts

The `upgradeTo` function in the `GraphProxy` contract, which calls the `_setPendingImplementation` function in the `GraphProxyStorage` contract does not check that the provided address is a contract or not.

This means that if the `upgradeTo` function is called with an EOA as a parameter, this address will be successfully saved in the `PENDING_IMPLEMENTATION_SLOT` as a valid implementation address. Then, if this EOA calls the `acceptUpgrade` function of the `GraphProxy` contract, it will successfully set its address in the `IMPLEMENTATION_SLOT`, as the validations in line 73 will pass all the checks.

Consider checking whether the address passed to the `acceptUpgrade` function is a contract or not using the `Address.isContract` function of the OpenZeppelin contracts.

Update: *Fixed in PR#358.*

[L09] Confusing, error-prone interest rates management

The `RewardsManager` contract defines an `issuanceRate` variable which represents the inflation rate that the `GraphToken` will accrue per block for minting the rewards that both indexers and delegators will receive for a subgraph that is successfully indexed. These rewards can be accumulated in each block following a simplified version of the compound interest formula:

$$a = p.(1 + r)^t$$

Where `t` is the number of blocks that passed since the last update, `a` is the total supply of GRT, and `p` is the inflated amount of the GRT's total supply for that period `t` when interest `r` is applied.

For simplicity, instead of defining the `issuanceRate` as the `r` variable of the formula above, it is defined as `1 + r`, but this is not part of the natspec, nor validated in the `setIssuanceRate` function, which could lead into undesired errors when setting or updating it.

Consider adding the appropriate documentation in the `param` tag of the `setIssuanceRate` function, and consider validating that the issuance rate sent as a parameter is within the expected values by the team.

Update: *Fixed in PR#352.*

[L10] [Upgradeability] Error-prone use of hex function selectors

The functions `getProxyImplementation`, `getProxyPendingImplementation`, and `getProxyAdmin` in the GraphProxyAdmin contract each perform a `staticcall` to a proxy contract to fetch the required data. A `staticcall` is required due to the fact that the `ifAdminOrPendingImpl` modifier is used on the functions being called, meaning that they cannot being marked as `view`.

Each `staticcall` is performed using a raw hex string specifying the GraphProxy function selector to be called - for example `staticcall(hex"5c60da1b")` is used to call the `implementation` function of the `GraphProxy` contract. This method of defining a function selector decreases code readability, and is error-prone. A small change to the function being called will cause the function selector to change and the code to break.

Consider using Solidity's built-in feature `.selector` to specify which function is being called, making the code more readable and less prone to error. For example, consider replacing `staticcall(hex"5c60da1b")` with `staticcall(abi.encode(_proxy.implementation.selector))`.

[L11] [Upgradeability] `initialize` functions in upgradeable contracts can be called more than once

All the upgradeable contracts in the system define an `initialize` function that replaces the constructor - a well-known practice when working with upgradeability in smart contracts. In Solidity, code that is inside a constructor is not part of a deployed contractâ€™s runtime bytecode. This code is executed only once, when the contract instance is deployed. As a consequence of this, the code within a contractâ€™s constructor will never be executed in the context of the proxyâ€™s state.

The problem lies in the fact that these `initialize` functions can be called more than once, which can lead to undesired outcomes if they are called accidentally a second time. In particular, if any of the variables set within the `initialize` function are altered by calling other functions in the system, and then `initialize` is called again, the values these variables held will be lost.

Consider adding an `isInitialized` flag to each of the upgradeable contracts with a default `false` value. Set it to `true` in the `initialize` functions, and check that the flag is set to `false` before executing the initialization code. Alternatively consider using the `Initializable` contract provided by the `openzeppelin/contracts` library.

[L12] Interfaces are missing functions present in their implementations

Some interfaces in the code base are not fully matching their implementation, defining only a part of the functionalities implemented in their respective contracts. Some examples of this +are:

- The `ICuration` interface is implemented by the `Curation` contract. The interface does not include all functions such as the `collect` function.
- The `IController` interface is implemented by the `Controller` contract. The interface does not include all functions such as the `setPaused` function.
- The `IGraphToken` interface is implemented by the `GraphToken` contract. The interface does not include all functions such as the `permit` function.

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

Consider adding these and all other missing functions to the corresponding interface, so the implementation directly matches the interface specification.

Update: *Fixed in PR#361. Some of the mentioned issues have been addressed along with additional interface changes listed in the PR.*

[L13] Lack of indexed parameters in events

Throughout the code base, there are events defined without any indexed parameters. Some examples of this are:

- The `SetController` event of the `IManaged` and the `Managed` contracts.
- The `SetContractProxy` event of the `IController` contract.
- The `NewPauseGuardian` event of the `Pausable` contract.
- The `PendingImplementationUpdated` and `ImplementationUpdated` events of the `GraphProxyStorage` contract.
- The `SetDefaultName`, `SubgraphMetadataUpdated`, `SubgraphPublished`, `SubgraphDeprecated`, `NameSignalEnabled`, `NSignalMinted`, `NSignalBurned`, `NameSignalUpgrade`, `NameSignalDisabled`, `GRTWithdrawn` of the `GNS` contract.
- The `RewardsDenylistUpdated` of the `RewardsManager` contract.

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

Update: *Fixed in PR#404.*

[L14] Constants not declared explicitly

There are some occurrences of literal values with unexplained meaning in the code base. For example, lines 752 to 753, line 798, and line 813 in `DisputeManager.sol`. Literal values in the code base without an explained meaning make the code harder to read, understand, and maintain, thus hindering the experience of developers, auditors, and external contributors alike.

Developers should define a constant variable for every magic value used (including booleans), giving it a clear and self-explanatory name. Additionally, for complex values, inline comments explaining how they were calculated or why they were chosen are highly recommended. Following Solidity's style guide, constants should be named in `UPPER_CASE_WITH_UNDERSCORES` format, and specific public getters should be defined to read each one of them.

Update: *Fixed in PR#387.*

[L15] `_pow` function can revert in extreme conditions

The `_pow` function of the `RewardsManager` contract is performing the exponentiation of a number `x` to the power of `n` using a `base` scale factor and is based on a `MakerDao`'s function.

The function is used in the `getNewRewardsPerSignal` function of the same contract in line 146. As this function is indirectly being used in different core functions of the protocol, it is extremely important to validate that will not revert and that works as expected.

If the difference between the current `block.number` and the block number saved in `accRewardsPerSignalLastBlockUpdated` is big enough, or if the `issuanceRate` is set too high, the `_pow` call can start reverting any subsequent call. An example would be an `issuanceRate` expressed as `x = 1,05e18`, a difference between the `block.number` and the `accRewardsPerSignalLastBlockUpdated` of `n = 993` and a base scale factor of `base = 1e18`. if `getNewRewardsPerSignal` is not called for more than `4 hours`, none of the `mint`, `burn`, or `set issuance rate` operations will be able to be performed anymore.

Even though the The Graph team specified that the `issuanceRate` will be a way smaller value than the one provided in the example above, the `_pow` function might be intended to be used with higher values of the base and the exponent in the future in other sections of the code.

Consider exhaustively testing the `_pow` function studying all the possible edge cases it can produce under different conditions to assure that will not revert and stale the protocol. Additionally, consider carefully verifying that other parts of the code copied from different projects behave as expected, to understand how it works and if it is a good fit for the code base.

Update: *Fixed in PR#354.*

[L16] Reimplementing ECDSA signature recovery

The `DisputeManager` contract includes an implementation of the ECDSA signature recovery function. Also, the `permit` function of the `GraphToken` contract uses the `ecrecover` function built in Solidity and only the returned value is checked to be valid (i.e., non zero). This function is already present in the OpenZeppelin Contracts project, where it has been audited and is constantly reviewed by the community. Consider importing and using the `recover` function from OpenZeppelin's ECDSA library not only to benefit from bug fixes to be applied in future releases, but also to reduce the code's attack surface.

Update: *Fixed in PR#353.*

[L17] Unnecessary return values

There are some places in the code base where functions are unnecessarily returning values, even though those values are not being assigned nor being read. Some examples are:

- The `_distributeRewards`, `_delegate`, and `_undelegate` functions in `Staking.sol`.
- The `_updateRewards` function in `Curation.sol`.

Consider refactoring those functions which are unnecessarily returning values, to promote code quality and readability. Additionally, consider always validating return values even of internal functions and react to unexpected results by reverting if they return an incorrect value, promoting gas savings to the caller, and easing transaction debugging issues.

Update: *Fixed in PR#388.*

[L18] [Upgradeability] Unnecessary access control defined in `ifAdminOrPendingImpl` modifier

The `ifAdminOrPendingImpl` modifier of the `GraphProxy` contract restricts calls so that only the admin or the address stored in the `pendingImplementation` storage slot can call certain functions. However this modifier is being used on `acceptUpgrade` and `acceptUpgradeAndCall` which can only be called by the `pendingImplementation` since they will revert on lines 142-145 if they are called by the admin.

Moreover, the `implementation`, `admin`, and `pendingImplementation` functions in the aforementioned contract also implement the `ifAdminOrPendingImpl` modifier, but they are never called from any of the implementation contracts and will therefore never be called by the `pendingImplementation`. These functions should therefore be restricted to be called only by the admin of the proxy.

Consider defining an `ifPendingImpl` modifier to restrict the `acceptUpgrade` and `acceptUpgradeAndCall` functions so that they can only be called by the contract address stored in the `pendingImplementation` storage slot. Additionally, consider using the `ifAdmin` modifier in the `implementation`, `admin` and `pendingImplementation` functions.

[L19] Undocumented operations to compute the Cobb-Douglas function

The `calcRebateReward` function of the `Rebates` library applies a number of arithmetic operations to compute the Cobb-Douglas production function in order to calculate the query rewards that should be distributed when either an indexer or a delegator claim tokens from the rebate pool. However, such sensitive operations were found to be undocumented, rendering them hard to follow and understand.

Explaining the rationale behind using the production function as a constant returns to scale instead of an increasing/decreasing returns to scale function, the reason of using equal output elasticities of `0.5`, or the reason of using the pool fees as the total-factor productivity are some examples of useful documentation that should be included in the code.

Consider thoroughly documenting all sensitive calculations made in the code base, making explicit the rationale behind them where appropriate. As a starting point, this can be done in inline comments and docstrings; although it is highly advisable to include references to external more-detailed end-user documentation. All of this will greatly improve the readability of the code, which should add to the platformâ€™s transparency and the usersâ€™ overall experience.

Update: *The implementation of the Cobb-Douglas function was re-implemented in PR#335, which was not reviewed since it is out of the scope of this issue.*

Notes & Additional Information

[N01] TODOs in code

There are â€œTODOâ€� comments in the code base that should be removed and instead tracked in the projectâ€™s issues backlog. See for example:

- Line 815 of the `Staking.sol` file.
- Lines 785 and L788 of the `Staking.sol` file.
- Line 468 of the `Curation.sol` file.
- Line 253 of the `RewardsManager.sol` file.
- Line 52 of the `Rebates.sol` file.
- Line 1015 of the `Staking.sol` file.
- Line 1019 of the `Staking.sol` file.

In particular, the first two items in the list refer to potential bugs and design decisions that may impact the security of the protocol and should be either fixed or added to the issue tracker to be fixed as soon as possible.

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

These `TODO` comments should at least have a brief description of the task pending to do, and a link to the corresponding issue in the project repository.

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

```
// TODO: Move to uint128 if gas savings are significant enough.
// https://github.com/graphprotocol/contracts/issues/1338
// --abarmat - 20200117
```

Update: *Partially fixed in PR#406. Some of the provided examples have been reviewed and fixed.*

[N02] [Upgradeability] Identical hex constants

The constant `IMPLEMENTATION_SLOT = 0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc` is defined both in

`GraphUpgradeable` and in `GraphProxyStorage`. While it is necessary for both contracts to have access to the constant, if they differ by just one hex character it will render the `onlyImpl` modifier useless, and therefore implementation contracts un-initializable.

Consider defining the `IMPLEMENTATION_SLOT` constant in a common contract, and inheriting this in `GraphUpgradeable` and `GraphProxyStorage`. This will ensure that any changes to `IMPLEMENTATION_SLOT` are applied to both contracts.

[N03] Erroneous or incomplete docstrings

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

- In line 399 of the `DisputeManager` contract, `_attestationData1` should be `_attestationData2`.
- In line 89 and 90 of the `GraphProxyStorage` contract, "implementation" should be changed to "pending implementation". The same applies in line 115 and 116.
- In line 11 of the `GNS` contract, the word "namings" should be "naming".
- In line 300 of the `GNS` contract, the word "publishing" should be "deprecating".
- In line 109 of the `Curation` contract, the word "exclusive" should be "inclusive" because actually the 100% is an accepted value.
- In line 499 of the `Curation` contract, the word "Extern" should be "External" but this function eventually lack of any useful docstring.
- In line 83 of the `Rebates` library, "covert" should be "convert".
- In line 118 of the `Stakes` library, the "@return True if staked" should be "@return the weighted locking period".
- In line 15 of the `DisputeManager` contract, "an" should be "and", and "a signed receipts" should be "signed receipts".
- In line 341 of the `Staking` contract, it should state "Set or unset an address as allowed slasher".
- In line 473 of the `Staking` contract, it should state "Authorize or unauthorize an address to be an operator".
- In line 13 of the `GNS` contract, it should state "The contract has no knowledge of human-readable names".
- In line 88 of the `EpochManager` contract, it should state "Return true if current epoch is the last epoch that has run".

Update: *Fixed in PR#412.*

[N04] `GraphProxy` does not match specification

In the internal documentation provided, when talking about the upgradeability of contracts, the The Graph team makes a reference to the possibility of setting the `admin` of the `GraphProxy` contract to the zero address.

It has to be noticed that the system is explicitly validating that this does not happen for security purposes. If such functionality is intended to be implemented for disallowing upgrades or remove the administrator's privileges, dedicated functions must be implemented.

Update: *Acknowledged. In the words of the The Graph team "A potential solution to remove ownership with the current implementation is to set the admin to the address of a contract that is not controlled by anyone".*

[N05] Inconsistency in struct declarations

Structs can either be defined in interfaces as in the `ICuration` file, or in contracts holding the storage definition such as the `RewardsManagerStorage.sol` file.

However, inconsistencies in where these structs are defined are not preferred as it may result in more convoluted and unclear code.

Consider choosing one of the two strategies and apply it throughout the complete set of contracts.

Update: *Fixed. Changes have been applied in PR#409.*

[N06] [Upgradeability] Incorrect comment

The comment above the declaration of `IMPLEMENTATION_SLOT` in `GraphUpgradeable` states that the value is "*the keccak-256 hash of "eip1967.proxy.implementation" subtracted by 1*", and that the value is "*validated in the constructor*. However, `GraphUpgradeable` does not have a constructor, and therefore the value is not validated.

Consider updating the comment to reflect this, or adding validation for the value.

[N07] Multiple getters for the same state variable

In the `Controller` contract, there are multiple getter functions that return the value of the `governor` state variable. Namely, `governor` (automatically generated by Solidity) and `getGovernor`.

To favor encapsulation and explicitness, ensure that there is at most one publicly exposed getter for each contract state variable.

Update: *Acknowledged. In the words of the The Graph team "The reason we had to include a `getGovernor` function call is to provide an interface for the `Managed` contract and avoid importing the complete implementation".*

[N08] Multiple Solidity versions are used

Throughout the code base there are two different versions of Solidity being used. For example, the `Curation` contract is using version `0.6.12` while the contract `Pausable` is using version `0.6.4`.

Given that different Solidity versions can lead to differences in the compiled code or raise incompatibilities, consider using the same version for all the contracts.

Update: *Fixed in PR#318 among other things which were not reviewed since they were out of the scope of this issue.*

[N09] [Upgradeability] GraphProxy does not inherit its interface

The interface `IGraphProxy` defines the function signatures for each of the functions in the `GraphProxy` contract, and is used to enable other contracts to interact with proxies. However, `GraphProxy` does not explicitly inherit from `IGraphProxy`. While this is not causing any issues at present, if either `GraphProxy` or `IGraphProxy` were to be altered, the compiler would not alert to the fact that the two no longer have identical interfaces.

Consider updating `GraphProxy` to inherit `IGraphProxy`, so that the compiler will inform you if ever the two have differing interfaces.

[N10] Repeated code

In the `DisputeManager` contract, the `acceptDispute`, `rejectDispute` and `drawDispute` functions are all sharing similar functionality, checking whether the dispute is created or not, storing the dispute in memory and then delete the mapping value associated to the `disputeID`.

Consider refactoring the repeated code by moving it to an internal function to be used by these functions to improve the quality and readability of the code base.

Update: *Fixed in PR#410.*

[N11] Not using the `SafeMath` library

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

Some examples of these are:

- In `line 239`, `line 376`, `line 383`, `line 458`, and `line 641` of the `GNS` contract.
- In `line 108` of the `GraphToken` contract.
- In `line 37`, and `line 58` of the `Rebates` library.

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

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

Update: *Fixed in PR#411.*

[N12] Service registry is untrusted input for other parts of the architecture

The `ServiceRegistry` contract allows anyone to register new unchecked service URLs with an associated geo-location.

Having in mind that The Graph protocol's smart contracts are just a part of the whole architecture and that developers tend to trust information coming from their own systems, it is important to highlight that this information could be maliciously crafted to exploit vulnerabilities in other systems that also take part of the protocol. An example of this would be an attacker crafting a malicious URL that could generate an Server-side request forgery attack, with the ability to reach services that are not meant to be reachable from the internet or crafting a malformed-input which may generate other problems in associated systems.

Consider always validating off-chain the information retrieved by the smart contracts before using it in other system's logic to avoid the exploitation of vulnerabilities.

Update: *Acknowledged. In the words of the The Graph team "Both the geohash and URL are not verified in the contract due to the cost it would imply. We acknowledge that this inputs need to be handled by the components that use this information and be taken as not sanitized".*

[N13] Typographical errors

There are multiple typos across the code base. Some examples are:

- In line 10 of the `Controller` contract, "convience" should be "convenience".
- In line 488 of the `Curation` contract, "Subgrapy" should say "Subgraph".

- In line 128 of the `GNS` contract, "their" should be "its".

Consider running codespell on pull requests to improve the readability of the code base.

Update: *Fixed. Mentioned and additional typos have been corrected in PR#408.*

[N14] Underflow is used to set token approval

In line 179 of the `GNS` contract, the hardcoded value of `-1` is used to approve the `Curation` contract as spender of all the `GRT` tokens of the contract.

Using underflows to mean the biggest possible number that can be stored in an `uint256` variable is unclear and very error-prone.

Consider declaring the same value using `2**256 - 1` instead, and using a state constant variable for it, improving clarity and readability of the code.

Update: *Fixed, among other changes, in PR#396.*

[N15] Unnecessary functions

The `setDefaultName` and the `updateSubgraphMetadata` functions of the `GNS` contract are exclusively emitting events.

While the `updateSubgraphMetadata` is at least called by the `publishNewSubgraph` function, the `setDefaultName` is not internally called in the code base.

Given the fact that a transaction must be executed to call them, and gas has to be spent, consider evaluating if those functions are really needed by the application and eventually remove them. An alternative to using events is to `return` the needed information at the end of the function.

Update: *Acknowledged. In the words of the The Graph team "The extra functions are used to emit events for the Network Subgraph logic".*

[N16] Unreachable condition in `tokensWithdrawable` function

The `tokensWithdrawable` function of the `Stakes` library checks whether the tokens staked are less than the tokens locked, but this condition will never be true even after a slash condition happens, as the `slash` function of the `Staking` contract will unlock the slashed amount of tokens for later releasing them.

Consider removing the `if` condition together with the comments above it.

Update: *Fixed in PR#351.*

[N17] [Upgradeability] Unused modifier in the `GraphProxyStorage` contract

The `onlyAdmin` modifier, defined in the `GraphProxyStorage` contract is not used in the repository. To favor simplicity and readability, consider removing the modifier entirely.

[N18] Wrong function visibility

There are some functions that are not being accessed locally but are being declared as `public` instead of `external`. *Some examples* are:

- In `RewardsManger.sol`: `onSubgraphSignalUpdate`, `getRewards`, and `setIssuanceRate`.
- In `EpochManager.sol`: `blockHash`, `currentEpochBlockSinceStart`, and `epochsSince`.
- In `Curation.sol`: `getCurationPoolToken`.

Moreover, *some examples* of functions that are only being accessed locally but are being declared as `internal` instead of `private` are:

- In `Curation.sol`: `_mintSignal`, `_burnSignal`, `_collect`, `_mint`, and `_updateRewards`.
- In `DisputeManager.sol`: `_createQueryDisputeWithAttestation`, `_createIndexingDisputeWithAllocation`, `_createQueryDisputeWithAttestation`, `_createIndexingDisputeWithAllocation`, `_isDisputeInConflict`, `_resolveDisputeInConflict`, `_pullSubmitterDeposit`, `_slashIndexer`, `_recoverAttestationSigner`, `_getChainID`, and `_parseAttestation`.
- In `RewardsManager.sol`: `_setIssuanceRate`, `_calcRewards`, and `_pow`.
- In `Staking.sol`: `_onlyAuth`, `_onlyAuthOrDelegator`, `_stake`, `_allocate`, `_collect`, `_delegate`, `_undelegate`, `_collectDelegationQueryRewards`, `_collectDelegationIndexingRewards`, `_collectProtocolFees`, `_getAllocationState`, `_sliceByte`, `_updateRewards`, and `_distributeRewards`.
- In `GraphToken.sol`: `_addMinter`, `_removeMinter`, and `_getChainID`.

Consider changing their visibility to improve the clarity and readability of the code base.

Update: *Fixed in PR#407.*

Conclusions

4 critical and 5 high severity issues were found. Even though The Graph team developed a stylish, well-documented, and clean code, some improvements can be done to improve the overall soundness and

security of the protocol. In particular, a more defensive approach when thinking about the different interactions between external actors and the system can potentially reduce the attack surface, especially considering that these workflows may not cover all the possible outcomes in the real world.