OpenZeppelin | security

The Graph Horizon Audit

The Graph

May 19, 2025

Table of Contents

Summary

Type	DeFi
Timeline	From 2025-03-03 To 2025-04-11
Languages	Solidity

Total Issues	42 (26 resolved, 2 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	2 (2 resolved)
Medium Severity Issues	5 (3 resolved)
Low Severity Issues	19 (11 resolved)
Notes & Additional Information	12 (6 resolved, 2 partially resolved)
Client Reported Issues	4 (4 resolved)

Scope

OpenZeppelin audited the ❄️ Graph Horizon and Subgraph Service ❄️ pull request at commit da95d8b. This pull request implements a new system version called Graph Horizon and reframes the subgraph indexing functionality into the Subgraph Service.

The following files have been audited in their entirety:

```
contracts/contracts
├── l2/staking/IL2StakingTypes.sol
└── rewards/IRewardsIssuer.sol
horizon/contracts
├── data-service
│   ├── DataService.sol
│   ├── DataServiceStorage.sol
│   ├── extensions
│   │   ├── DataServiceFees.sol
│   │   ├── DataServiceFeesStorage.sol
│   │   ├── DataServicePausable.sol
│   │   ├── DataServicePausableUpgradeable.sol
│   │   └── DataServiceRescuable.sol
│   ├── interfaces
│   │   ├── IDataService.sol
│   │   ├── IDataServiceFees.sol
│   │   ├── IDataServicePausable.sol
│   │   └── IDataServiceRescuable.sol
│   ├── libraries/ProvisionTracker.sol
│   └── utilities
│       ├── ProvisionManager.sol
│       └── ProvisionManagerStorage.sol
├── interfaces
│   ├── IAuthorizable.sol
│   ├── IGraphPayments.sol
│   ├── IGraphProxyAdmin.sol
│   ├── IGraphTallyCollector.sol
│   ├── IHorizonStaking.sol
│   ├── IPaymentsCollector.sol
│   ├── IPaymentsEscrow.sol
│   └── internal
│       ├── IHorizonStakingBase.sol
│       ├── IHorizonStakingExtension.sol
│       ├── IHorizonStakingMain.sol
│       └── IHorizonStakingTypes.sol
├── libraries
│   ├── Denominations.sol
│   ├── LibFixedMath.sol
│   ├── LinkedList.sol
│   ├── MathUtils.sol
```

```
│   ├── PPMMath.sol
│   └── UintRange.sol
├── payments
│   ├── GraphPayments.sol
│   ├── PaymentsEscrow.sol
│   └── collectors/GraphTallyCollector.sol
├── staking
│   ├── HorizonStaking.sol
│   ├── HorizonStakingBase.sol
│   ├── HorizonStakingExtension.sol
│   ├── HorizonStakingStorage.sol
│   ├── libraries/ExponentialRebates.sol
│   └── utilities/Managed.sol
├── utilities
│   ├── Authorizable.sol
│   └── GraphDirectory.sol
subgraph-service/contracts
├── DisputeManager.sol
├── DisputeManagerStorage.sol
├── SubgraphService.sol
├── SubgraphServiceStorage.sol
├── interfaces
│   ├── IDisputeManager.sol
│   └── ISubgraphService.sol
├── libraries
│   ├── Allocation.sol
│   ├── Attestation.sol
│   └── LegacyAllocation.sol
└── utilities
    ├── AllocationManager.sol
    ├── AllocationManagerStorage.sol
    ├── AttestationManager.sol
    ├── AttestationManagerStorage.sol
    └── Directory.sol
```

Out of the following files, only the differences (as they appear in the pull request) have been audited:

```
contracts/contracts
├── arbitrum/ITokenGateway.sol
├── curation
│   ├── Curation.sol
│   ├── CurationStorage.sol
│   └── ICuration.sol
├── disputes/DisputeManager.sol
├── epochs/IEpochManager.sol
├── gateway/ICallhookReceiver.sol
├── governance
│   ├── Controller.sol
│   ├── Governed.sol
│   ├── IController.sol
│   ├── IManaged.sol
│   └── Pausable.sol
├── l2
│   ├── curation
```

```
│   │   ├── IL2Curation.sol
│   │   └── Curation.sol
│   └── staking
│   │   ├── IL2Staking.sol
│   │   ├── IL2StakingBase.sol
│   │   └── L2Staking.sol
├── rewards
│   ├── IRewardsIssuer.sol
│   ├── IRewardsManager.sol
│   ├── RewardsManager.sol
│   └── RewardsManagerStorage.sol
├── staking
│   ├── IStakingBase.sol
│   ├── L1Staking.sol
│   ├── Staking.sol
│   └── libs/Stakes.sol
├── upgrades
│   ├── GraphProxy.sol
│   ├── GraphProxyAdmin.sol
│   ├── GraphProxyStorage.sol
│   ├── GraphUpgradeable.sol
│   └── IGraphProxy.sol
├── utils/TokenUtils.sol
└── token/IGraphToken.sol
```

System Overview

Graph Horizon is the next evolution of The Graph protocol, addressing the evolving needs of the data indexing ecosystem and incorporating the learnings of the development team. It introduces `HorizonStaking`, a staking primitive that provides economic security and enables the creation of modular and diverse data services. These services can cater to various data indexing needs, such as Amazon Firehose, SQL, Large Language Model queries, and legacy subgraphs for blockchain information.

Individuals can become **service providers** by staking funds in the `HorizonStaking` contract and locking part of their funds inside a **provision**. A provision corresponds to a specific **data service** and ensures economic security by allowing the slashing of funds if the service provider behaves maliciously. Funds within a provision are locked unless a service provider opts to withdraw them, requiring a thawing period during which the funds remain slashable. Service providers can serve multiple data services by creating separate provisions for each. Data services will have varying implementations and different requirements for the minimum amount to lock in a provision or the thawing period.

Graph Horizon also introduces changes to delegation, requiring delegators to stake funds in specific provisions. Delegators must conduct due diligence regarding both the service provider and the corresponding data service. The delegation tax has been removed, and now, delegation slashing can be enabled to enhance economic security. If enabled, delegators' shares can only be slashed after the service provider's share has been fully slashed. As delegated funds contribute to economic security, they also undergo a thawing period before withdrawal. It is important to note that while all tokens delegated to Horizon could be slashable, only part of the tokens will be considered usable for allocation or reward collection due to the delegation ratio mechanism.

The method by which data consumers pay for queries has now been abstracted. The Graph team provides a functional payments protocol framework based on escrows and **Receipt Aggregate Vouchers** (RAVs). A RAV is a signed data structure authorizing a collector to withdraw funds from a data consumer's escrow account. This modular payments protocol framework supports various types of collectors and authorization methods. If unable to seamlessly sign RAVs, a payer can authorize a signer to create RAVs on their behalf.

Graph Horizon also addresses the issue of service providers claiming query fees that are disproportionate to the amount of funds they have subjected to slashing. Each time query fees

are collected, a proportional amount of the corresponding provision is now "locked" for a set time, preventing further use for fee collection. If the entire provision becomes locked, query fees can no longer be collected and the service provider must wait for a portion of their tokens to get unlocked.

Upgradeability

Except for `GraphTallyCollector`, all other contracts within the new Graph Horizon will be initially upgradeable. This is intended as a security mechanism, to allow for intervention in case the system malfunctions or a bug is discovered. For some contracts, upgradeability is planned to be phased out over time once the system is deemed correct and stable.

Subgraph Service

The legacy subgraphs have now evolved into their own data service called the Subgraph Service. This service functions almost identically to its predecessor, allowing a service provider to receive query fees and indexing rewards for their work.

While the Subgraph Service is a new contract deployment, it maintains backward compatibility with existing allocations previously stored on the `Staking` contract. Upon upgrading to Graph Horizon, the system will enter a transition period during which participants will not be able to open new legacy allocations but can close them and gradually migrate to allocations within the `SubgraphService` contract. Delegations start as not slashable, and the governor can enable this feature if it is in the protocol's best interest.

The dispute and slashing mechanisms also remain mostly identical, with the addition that a fisherman can now cancel their dispute once the `disputePeriod` has passed. This lets fishermen recuperate their bond if the arbitrator has not yet resolved the dispute.

Design Limitations and Considerations

Several design limitations and considerations were identified that, while not necessarily a security risk, should be taken into consideration as they limit functionality or could hinder user experience:

1. Since the `delegationFeeCut` function can be instantly updated, a malicious service provider could sandwich a query fee/indexing reward collection with two calls to `setDelegationCall` and deny any rewards to the delegators. However, there is little

incentive to do so, as such activity could be observed and either punished through slashing or by the delegators withdrawing their stake.

2. An allocation can become over-allocated (containing more tokens than available in the legacy system or provisioned with Horizon), which would result in the indexing rewards for that allocation being slightly inflated. Note that security measures have been put in place to limit this inflation, such as:
    - the allocation automatically closing upon a collection.
    - anyone being able to close a legacy allocation after a minimum amount of epochs passed.
3. In the legacy system, slashing is not targeted towards a particular allocation but rather to the whole stake of an indexer. Hence, the accounting of `__DEPRECATED_tokensAllocated` and the amount of tokens in an allocation can be incorrect. This can lead to a slight inflation of indexing rewards, which is limited by the same security measures mentioned in point 2.
4. There is little on-chain data that can be used to differentiate between similar/identical query disputes, as there is no information about the address that requested the query or the time at which the request was created/served. Hence, two disputes that look identical on-chain could be different, and should both be slashable. Arbitrators should pay special attention when deciding on disputes, and leverage all available information as well as the Arbitration Charter.
5. It is technically possible that a dispute created for a past indexing mistake will slash newly provisioned stake. Imagine a scenario where a service provider provisioned 100 tokens and 3 disputes were created against him. If the first two disputes slash the whole amount of tokens and the provider deposits 50 more tokens, they are technically slashable by the third dispute, although the dispute was made against a past mistake and provision. Similarly to the above, arbitrators should take this into consideration.

Security Model

Graph Horizon's security model is built on the economic security provided by the staking protocol. The protocol requires that service providers stake GRT tokens and lock them inside provisions. Hence, service providers are allowed to serve queries only after subjecting themselves to being slashed by the data service for misbehavior.

Data services choose their own parameters and arbitration mechanisms to determine the conditions and amount of slashing. While the dispute mechanism of the Subgraph Services relies on a human or a quorum of humans to arbitrate a dispute, Graph Horizon allows custom

dispute resolution mechanisms to be implemented, including trustless ones such as zero-knowledge (ZK) proofs.

Privileged Roles and Trust Assumptions

In the long term, Graph Horizon aims to achieve censorship resistance by becoming fully permissionless and immutable. As such, the Graph team is trusted to correctly and completely execute the protocol setup steps, such as deploying and upgrading the contracts, setting appropriate values for each operational parameter, and correctly updating the whitelists of the `GraphTokenLockWallets` such that the vesting funds can not escape early. Moreover, The Graph team should wait until all legacy allocations are closed and ensure that they fully migrate all legacy IDs to Horizon.

Currently, the privileged roles of the Graph Horizon framework are:

- **Proxy Admin**: Can change the implementation of each upgradeable contract. The proxy admin will be set to The Graph Council.

  Note that while The Graph Council is currently an entity that has control over most of the Graph Horizon system, the project's intention is to phase out some of the council's powers and make the system more trustless and decentralized.

- **Governor**: Has administrative privileges inside many contracts. To enumerate a few, it can:

  - whitelist data services without escape hatches.
  - mark the end of the transition period.
  - set the GRT inflation through indexing rewards.

- **Service Providers**: They can be any entity that stakes GRT tokens and provisions them to a data service. A service provider can create a provision, reprovision tokens to another data service, withdraw unslashed tokens after the thawing period has passed, or collect query fees and indexing rewards within the Subgraph Service. Note that a service provider can also set operators, which are entities authorized to act on the provider's behalf with almost the same level of privileges.

- **Delegators**: They can delegate their stake to provisions, thereby contributing to the economic security of data services and getting a share of the rewards in return. Similarly, they can undelegate and withdraw any unslashed tokens once the thawing period has passed.

- **Data Services**: Custom implementations that manage the registration and operation of service providers, validate provisions, and enforce slashing rules. Note that custom data services are likely to require trust, so participants should do their due diligence around the services' implementations and the potential risks.

The privileged roles of the Subgraph Service are:

- **Pause Guardian**: Can pause and unpause important functionality such as registering an indexer, creating an allocation, and collecting rewards.
- **Owner**: Can set pause guardians and important operational parameters such as the minimum amount of tokens required in a provision or the ratio with which delegated funds can be leveraged for economic security. The owner is trusted to update all operational parameters in a sensible matter, as mistakes could create problems such as temporarily making some provisions invalid, making allocations appear overallocated and automatically closing after collection, or inability to collect fees.
- **Dispute Manager**: The only entity that can slash the `SubgraphService`, bubbling the slash up to the `HorizonStaking` contract. This role is held by the `DisputeManager` contract.
- **Fishermen**: Can put up bonds to create disputes within the `DisputeManager` contract. Each fisherman can cancel their own disputes after the dispute period has passed.
- **Arbitrator**: Can accept, draw, or reject a dispute within the `DisputeManager` contract.

The proxy admins, pause guardians, the governor, and the owner of the `SubgraphService` are trusted to act in the best interest of the protocol and update the system with appropriate code and parameters.

Lastly, it is important to note that verifiers can steal service providers' funds by maliciously slashing their provisions and receiving a percentage as a reward. Custom data services have this inherent risk and, thus, should be thoroughly researched and audited before they are engaged with.

High Severity

H-01 Legacy Slash Breaks Accounting

During the transition period, the amount of tokens staked by an indexer is composed of

- tokens allocated and not thawing, in the legacy system
- tokens allocated and thawing, in the legacy system
- tokens provisioned within Horizon
- the rest, which are considered idle

Note that while there is no overlap between the tokens allocated in the legacy system and the ones provisioned with Horizon, the legacy system can slash up to all the tokens staked, hence, potentially slashing tokens held in Horizon provisions.

The `legacySlash` function does not account for this and only updates the accounting of the legacy system. This can break the accounting of Horizon and cause issues, such as

- having an inflated service provider `tokensProvisioned` value, and, therefore, reporting a smaller idle stake.
- having an inflated provision `tokens` value which might make a provision look valid while there are no tokens backing it up. Such broken accounting could also prevent slashing by reverting when attempting to slash more tokens than are staked.

Note that a service provider can also carry over this inconsistent state to another provision by thawing and reprovisioning their funds.

Consider updating the `legacySlash` function to account for Horizon state and implementing a thorough testing suite which ensures that the system continues functioning as intended after a legacy slash. Alternatively, consider limiting the amount of tokens that are slashable by legacy functionality to the amount of tokens staked minus the amount of tokens provisioned.

Update: *Resolved in pull request #1143, which no longer allows legacy slashing on provisioned tokens. Arbitrators are trusted to act accordingly if any indexers escape slashing by provisioning their tokens. Note that this PR also introduces integration risks for the fishermen that are smart contracts. The Graph team is trusted to make a public announcement, letting the community know of this potential breaking change. The Graph team stated:*

> *We decided to limit the amount of slashable tokens for the legacy slash to whatever tokens are not provisioned. A downside of this approach is that indexers can escape slashing by just provisioning their tokens. To fix that, we added a new dispute type in the new `DisputeManager` contract which essentially allows arbitrators to slash an indexer that is suspected of this behavior. If tokens are provisioned during the transition period, it is guaranteed that they will be provisioned to `SubgraphService` so arbitration could settle any outstanding slash amount.*
>
> *There are two accepted risks with this approach:*
>
> - *If the legacy slash event is late, as in after the transition period, then the indexer could effectively escape the slashing by provisioning to a data service under their control. We think that, with proactive arbitration, we can minimize the chances of this happening.*
> - *The legacy `DisputeManager` contract assumes that the indexer is slashed for the full amount when the slash call to the staking contract happens and that the full rewards amount is awarded. The contract then emits an event with this information. This means that we would have to upgrade the contract to properly handle the actual slash and rewards amounts. We have decided not to do this and accept the fact that the event will produce incorrect data for these cases. We will patch this behavior in the relevant subgraph which is the main consumer for the event data.*
>
> *Finally, it is worth mentioning that we will add an integration test to cover the `legacySlash` + slash combination.*

H-02 Delegations Are Unlikely to Be Slashed If Delegation Slashing Is Enabled

The `DisputeManager` contract currently ties a provision's stake snapshot to the indexer's stake and a capped delegation amount (`delegatorsStakeMax`). As the indexer's stake decreases—especially if fully slashed—the delegation stake becomes unslashable, which weakens slashing incentives and misaligns delegator risk. The following is a step-by-step PoC:

1. The provision starts off with 1000 tokens from the service provider and 3000 from delegators. `maxSlashingCut` is set to 25% and `delegationRatio` to 3.
2. The service provider misbehaves several times.

3. A dispute is created and the provision is slashed for (1000 + min(3000, 1000 * 3)) * 25% = 1000 tokens. This leaves the provision with 0 tokens from the service provider and 3000 from delegators.
4. Another dispute is created and the provision is slashed for (0 + min(3000, 0 * 3)) * 25% = 0 tokens. Hence, the delegators cannot be slashed.

Consider updating the stake snapshot such that it reflects the total provision (indexer and delegators' stake) without considering `delegationRatio`, ensuring consistent slashing risk for both parties. In addition, consider reviewing any off-chain systems that could be involved in determining the `tokensSlash`, ensuring that the computations are aligned with the new limits.

Update: *Resolved in pull request #1160.*

Medium Severity

M-01 Service Providers Can Thaw Instantly to Escape Slashing

The amount of time required for funds to thaw is determined by the `provison.thawingPeriod` variable. This variable can be set as high as `type(uint64).max` unless it is explicitly limited to a lower value by the data service contract or the `_maxThawingPeriod` state variable of the `HorizonStaking` contract. If no reasonable limit is set for this variable, the thaw requests can be thawed immediately due to unsafe casting when thawing and undelegating. The unsafe casting results in a silent overflow when the addition of `block.timestamp` and `thawingPeriod` exceeds `type(uint64).max`. Below is a step-by-step description of a potential attack:

1. The service provider gives malicious responses to requests.
2. After getting disputed, the service provider front-runs the slashing by atomically:
    - calling the `setProvisionParameters` and `acceptProvisionPendingParameters` with `thawingPeriod` set to `type(uint64).max`
    - initiating a thaw request for their entire provision, which sets the `thawingUntil` parameter of the request to `block.timestamp - 1`
    - deprovisioning the thaw request

3. The service provider gets slashed, but since it has no provision, only the delegators get slashed if delegation slashing is enabled.

Aside from the overflow scenario, a high `thawingPeriod` value could lock delegations. A service provider can set `thawingPeriod` to a very high value such as 1000 years. This would not cause an overflow, but any thawing requests made in this state would practically become locked. Imagine the following scenario:

1. The service provider front-runs a delegator's thawing request by setting `thawingPeriod` to 1000 years.
2. The delegator's thawing request becomes practically locked.
3. The service provider sets back the `thawingPeriod` to a reasonable value to continue usual operations.

Consider initializing the `_maxThawingPeriod` variable in the `HorizonStaking` contract with a sufficiently low value to prevent both of the above-mentioned scenarios. In addition, consider introducing a limit to the `setMaxThawingPeriod` function and reverting if the new value exceeds it. Lastly, consider changing the thawing period upper bound of the `SubgraphService` contract to a sensible value.

Update: *Resolved in pull request #1128. The Graph team stated:*

> *We have implemented a few complementary fixes*:
>
> - *First, for the `SubgraphService`, we just limited the thawing period range to the `DisputeManager`'s dispute period (range lower bound = range upper bound = dispute period).*
> - *Then, we introduced a new property to `HorizonStaking` provisions: `lastParametersStagedAt`. As the name suggests this tracks the timestamp when parameters were last staged (not accepted). This allows data services to implement more robust mechanisms for provision parameter changes. For example, a data service might choose to disallow accepting a parameter change unless X amount of time has passed since they were staged, thus providing some form of protection for delegators.*
> - *Lastly, we will take special care to initialize `_maxThawingPeriod` variable to a reasonable value (56 days was actually the maximum value we were thinking of). We do not think it is worth encoding this with smart contract code so we just added a comment to highlight the importance of setting this value to a sufficiently low value to prevent the described attacks.*

M-02 Potential DoS During Slashing

The `slash` function of the `HorizonStaking` contract is susceptible to a DoS due to its validation check on `tokensVerifier`. The function accepts `tokensVerifier` as an argument, allowing the slasher to effectively dictate the amount of reward taken, while the remainder of the slashed tokens is burned. A check compares the computed maximum verifier tokens to the provided `tokensVerifier`. If `tokensVerifier` exceeds `maxVerifierTokens`, the transaction reverts. This condition can result in a DoS in multiple ways.

If `tokensVerifier` is computed off-chain, a service provider could potentially front-run a slashing transaction deprovisioning and decreasing `prov.tokens`. This decreases `maxVerifierTokens` and causes the check to fail, resulting in a revert and effectively DoSing the slashing process. Although the `SubgraphService` data service calculates `tokensVerifier` on-chain based on actual provision tokens, thereby mitigating this risk in that context, an alternative vector exists where the verifier's reward cut is computed using a mutable `fishermanRewardCut` in the `DisputeManager` contract. `prov.maxVerifierCut` cannot be set lower than `fishermanRewardCut`, but the `fishermanRewardCut` could be set higher than existing `prov.maxVerifierCut`. In this case, such provisions would always revert during slashing.

Consider modifying the `slash` function of the `HorizonStaking` contract to use the minimum of `maxVerifierCut` and `tokensVerifier` instead of enforcing a strict reversion when `tokensVerifier` exceeds `maxVerifierTokens`. If this change is implemented, the function must return the actual slashed amount. Consequently, any external contract invoking `slash` (e.g., `DisputeManager` in the `SubgraphService` contract) should be updated to check this return value and adjust its calculations accordingly, avoiding assumptions of a maximum slash.

Update: *Resolved in pull request #1159. The issue was fixed at the data service level instead. Hence, future data services ought to take this issue into consideration and implement a similar fix.*

M-03 Provision Validity Can Be Temporarily Griefed

One of the conditions for a provision to be considered valid by a data service is that the amount of tokens inside should be between a minimum and a maximum bound. Depending on the data service's implementation, the service provider might no longer be able to collect

rewards (e.g., `SubgraphService.collect`) or manage their operational parameters (e.g., `SubgraphService.resizeAllocation`) if the provision becomes invalid.

The `stakeToProvision` function does not have access control and can be used to increase the amount of tokens within the provisions of other users, making them invalid. This can be harmful especially for data services that enforce tight bounds (e.g., ETH staking, where the amount of tokens needed to provide service is fixed, so `minimum` == `maximum`). A malicious actor would only need to `stakeAndProvision` a small amount of funds to make another user's provision invalid. While such intentional griefing is temporary and can be fixed if the service provider starts thawing the maliciously added amount of tokens, it can be repeated indefinitely, disrupting service.

Consider addressing this issue by adding access control to the `stakeToProvision` function while ensuring that the `SubgraphService` can still distribute indexing rewards correctly.

Update: *Resolved in pull request #1132.*

M-04 Service Providers Could Be Left Without Payments if `SubgraphService` Is Paused

The `SubgraphService` contract inherits pausing functionality which causes the `collect` function to revert when the contract is paused. This prevents service providers from collecting any outstanding payments, making it important that such payments remain valid and not be denied in the meantime.

The `Authorizable` contract contains the `thawSigner` and `revokeAuthorizedSigner` functions that are used by an authorizer to thaw and revoke their signer, rendering all outstanding payments invalid. Note that the `GraphTallyCollector` contract (which inherits `Authorizable`) does not have pausing functionality. Thus, if the `SubgraphService` or any other data services that use the `GraphTallyCollector` are ever paused, there is an opportunity for payers to deny any outstanding payments by thawing and revoking the authorized signer.

Consider adding pausable functionality to `Authorizable`, which could be leveraged by the governor to pause signer thawing and revocation when payment collection is paused. However, this change would make `GraphTallyCollector` work with the single data service that controls its pausing. Therefore, such an update to `Authorizable` would require each data service to use a separate `GraphTallyCollector` contract.

Update: *Acknowledged, not resolved. The Graph team stated:*

> *We think that it is best to keep `GraphTallyCollector` as an isolated, un-governed contract and not one that can be controlled by a specific data service or entity. So, we will accept this as a known risk/trust assumption. This decision goes against the spirit of GraphTally (a trust minimized payments system). However, given that the contract is upgradeable, there is already trust placed on the governor as they could prevent any collection from happening by upgrading the contract at any time. We do not think that the pausing preventing collection changes the trust assumption/risk vector much.*

M-05 `DisputeManager` Excessively Slashes Indexer When Delegation Slashing Is Disabled

The `DisputeManager` contract permits slashing up to `maxSlashingCut` of the provision snapshot. The provision snapshot is computed based on both the indexer and delegation stake, without taking into consideration if delegation slashing is enabled or not. It is important to note that in Graph Horizon, delegation slashing starts as disabled, and will remain so until enabled by the governor. Hence, the indexer can be excessively slashed for a significant period of time.

Consider updating the `DisputeManager` contract such that, if delegation slashing is disabled, it only uses the indexer's stake when calculating `maxTokensSlash`.

Update: *Acknowledged, not resolved. The Graph team stated:*

> *We think that this is acceptable as the indexer is directly profiting from delegation regardless of the delegation slashing status. The indexer is getting more rewards/ collecting more payments thanks to the delegation, so it is fair that they can get slashed more. Any potential griefing vector whereby someone delegates just to inflate the snapshot can be remedied at the arbitration level.*

Low Severity

L-01 `getThawedTokens` Function Returns Incorrect Value

The `getThawedTokens` `view` function of the `HorizonStakingBase` contract returns the total thawed tokens of an owner by iterating through all thawing requests and calculating the

sum of completed thawing amounts. While the function correctly excludes requests still within their thawing period, it fails to account for requests with invalidated nonces that result from slashing all the tokens in a provision or delegation pool.

This oversight causes the calculation to include tokens from invalid requests, resulting in a reported total that exceeds the actual withdrawable amount. The disparity between the returned value and the truly withdrawable tokens can create integration problems with dependent systems and confuse users about their available balances.

Consider modifying the function to exclude thawing requests with invalidated nonces, ensuring the returned value accurately represents only withdrawable tokens.

Update: *Resolved in pull request #1129. The Graph team stated:*

> *Fixed. This PR also adds the nonce value to the `ThawRequestCreated` event which was missing.*

L-02 Temporary DoS on Thawing Requests or Stake Claims After Period Updates

The thawing request system processes requests in chronological order (head to tail). Each request's `thawingUntil` parameter is set when the request is created through the `thaw()` or `undelegate()` functions. Since it is calculated using the current `thawingPeriod` value and `block.timestamp`, an issue arises if `thawingPeriod` is reduced after some requests have already been created.

If a service provider reduces the `thawingPeriod` parameter, newer thawing requests could complete their `thawingUntil` earlier than older requests created with a longer thawing period. Since the contract processes thawing requests sequentially (from head to tail), the newer requests with earlier `thawingUntil` values remain blocked until older requests have fulfilled their thawing duration. This creates a temporary but potentially significant delay for users who made thawing requests after the parameter update.

Proof of Concept

1. At `block.timestamp` = 0 and `thawingPeriod` = 100, `thawRequest1` is created with `thawingUntil` = 100.
2. The service provider updates `thawingPeriod` to 50.
3. At `block.timestamp` = 10, `thawRequest2` is created with `thawingUntil` = 60.
4. `thawRequest2` cannot be withdrawn between timestamps 60 and 100.

A similar behavior can be observed for stake claims, which are ordered chronologically by `releaseableAt`, as long as the `disputePeriod` does not change. However, note that the owner of the `DisputeManager` contract can change the `disputePeriod` to a smaller value, potentially creating the same type of DoS.

The issues described above only create a temporary blockage, and it is likely not worth reworking these systems which are already complex. Hence, consider clearly documenting this behavior to users, service providers, and verifiers. Moreover, consider documenting this behavior in the NatSpec of the `getThawedTokens` function, which can report confusing information because of the above.

Update: *Resolved in pull request #1130. The Graph team stated:*

> *As suggested, we think that fixing this temporary blockage is not worth the added complexity. We have opted for documenting the behavior.*

L-03 Incomplete Slashings Can Lead to Inflation of Shares

Incomplete slashings can result in dramatic inflation of the share count relative to the underlying token balance, causing operational issues over time. For instance, starting with a pool of 1e18 tokens and 1e18 shares, a slashing event reducing the tokens to 1 while keeping shares at 1e18, followed by adding 1e18 tokens, results in approximately 1e36 shares but only 1e18+1 tokens. Repeating this cycle inflates the share count by roughly 1e18 times on each iteration. This inflation can eventually lead to a DoS on thawing provisions in the `HorizonStaking` contract, where operations involving `prov.sharesThawing * _tokens` may overflow and revert.

Consider modifying the slashing function to burn the remaining tokens in delegations and provisions, and nullify all thawing requests if the slashing leaves only insignificant or dust amounts of tokens, rendering the share inflation scenario impossible.

Update: *Acknowledged, not resolved. The Graph team stated:*

> *We think that this is a legitimate scenario that does not require additional mitigations. It can only happen due to repeated and/or heavy slashing which is solely under the service provider's control and ultimately only affecting them (delegators are affected by the slashing but not by the share inflation). If reckless behavior from a service provider leads*

> *to an unusable pool, there are two alternatives for them in case they want to continue to operate:*
>
> 1. *Unwind their operation and start over with a different identity.*
> 2. *Recover the delegation pool by adding tokens into it using `addToDelegationPool()`. This could be very costly depending on how bad the inflation was.*

L-04 Precision Loss in `tokensThawing` Calculations During Slashing

The slashing calculation in the `HorizonStaking` contract uses fixed-point arithmetic with intermediate rounding up:

```
uint256 provisionFractionSlashed = (providerTokensSlashed * FIXED_POINT_PRECISION +
prov.tokens - 1) /
    prov.tokens;
prov.tokensThawing =
    (prov.tokensThawing * (FIXED_POINT_PRECISION - provisionFractionSlashed)) /
    FIXED_POINT_PRECISION;
```

The current implementation rounds up the intermediate result to ensure that the final result is rounded down. Without this intermediate rounding up, the sum of thawing tokens would exceed the tokens in the provision, preventing withdrawal of the last thaw request. However, the intermediate rounding up introduces precision loss. In our calculation (see **Appendix A**), we found the error to be between `-tokensThawing / 1e18` and 0, meaning that there may be up to 1 wei loss per 1e18 thawing tokens. The same issue exists in the calculation for delegation thawing amounts.

Consider replacing both instances with a simplified direct proportion formula of the form `thawing = thawing * (tokens - slashed) / tokens`. The simplified formula would limit precision loss to 1 wei while still ensuring that the final result is rounded down, guaranteeing that the sum of thawing tokens does not exceed the tokens in the provision (or delegation).

Update: *Resolved in pull request #1136.*

L-05 Service Provider Might Be Forced to Update `thawingPeriod` Together With `maxVerifierCut`

A service provider can update a provision's `thawingPeriod` and/or `maxVerifierCut` through the `setProvisionParameters` function. The function checks the validity of both `newMaxVerifierCut` and `newThawingPeriod`, regardless of which one of the parameters actually changed.

However, there is a corner case where this can be problematic:

- The `HorizonStaking._maxThawingPeriod` variable is set to 100, and a service provider has a provision with `thawingPeriod` set to 80 and `maxVerifierCut` set to 50.
- Governance updates `_maxThawingPeriod` to 50. Note that this leaves existing provisions as valid and only applies to provisions created starting from the time of the update.
- The service provider wants to set their provision's `maxVerifierCut` to 40. To successfully pass the checks, they are forced to also update the provision's `thawingPeriod` to something smaller than the new `_maxThawingPeriod`, otherwise the call will revert.

To resolve this inconsistent behavior, consider only applying each check if the corresponding provision information has any change in it. Alternatively, if the above-described behavior is the intended one, consider clearly documenting the expected behavior of provision management if governance updates `_maxThawingPeriod` to a smaller value.

Update: *Resolved in pull request #1162. The Graph team stated:*

> *Updated. Now, the conditions are checked only if the specific parameter changes.*

L-06 Data Services Could Be Left in an Inconsistent State

The `ProvisionManager` contract is meant to help data services manage and interact with provisions. It inherits the `ProvisionManagerStorage` contract, which declares seven storage variables and sets six of them at the time of initialization. Note that `_delegationRatio` is not set, creating a chance for data service developers to forget setting it and, by doing so, make delegation irrelevant for a provision.

Consider leveraging the compiler to ensure that `_delegationRatio` is set, by passing it as a parameter and setting it together with the `ProvisionManager` initialization logic.

Update: *Resolved in pull request #1145. The Graph team stated:*

> *We fixed this by initializing the delegation ratio to the max `uint32` value. This is in line with what we are doing with the rest of the parameters where we have default values that are the most permissive but recommend data service developers to override them. There is a minor risk of overflow in `HorizonStaking.getTokensAvailable()` where the ratio is multiplied essentially by the service provider's tokens + 1. However, the number of tokens that would trigger this scenario is in the order of 10^57 times the current total supply so it is nothing to be worried about.*

L-07 Provision Slashing Does Not Update Tokens Locked When Collecting Query Fees

To prevent an indexer from collecting an enormous amount of query fees as compared to the slashable stake, the `SubgraphService` locks a part of an indexer's provision every time query fees are collected. This ensures that the amount of query fees to collect during a time interval is limited, which reduces the incentive of the indexer being dishonest. The locked tokens are released after some time, allowing the indexer to continue collecting query fees for their service. However, after a big slashing event, the accounting can become inconsistent and work to the indexer's disadvantage as follows:

- The indexer has provisioned 150 tokens and collected query fees which have locked 100 tokens.
- The indexer is slashed for 80 tokens, so the provision has 70 tokens left. The locked tokens are not updated and remain at 100.
- The indexer provisions 30 additional tokens, but is not able to collect any query fees (`70 + 30 - 100 = 0`). Since the slashing events did not update the locked tokens and corresponding stake claims, the service provider cannot claim query fees up to the expected full potential. This happens because some tokens are locked for economic security that has already been leveraged by slashing. So, in this sense, the service provider is "over-locked".

When slashing, consider updating the implementation such that the tokens locked in the `feesProvisionTracker` mapping are updated, as well as the `StakeClaims`. However, note that this change might not be easy to fit into the current implementation and could over-

complicate the system for little gain. Hence, alternatively, consider accepting this risk and clearly documenting it in the code and explicitly stating it for the end user.

Update: *Acknowledged, risk accepted. The Graph team stated:*

> *Given that it is only a temporary issue and it stems from the service provider getting slashed, we think that this is an accepted risk.*

L-08 Incorrect Domain Separator in Fork Scenarios

The `AttestationManager` contract sets the domain separator with the current `chain.id` during initialization and stores it in the `_domainSeparator` storage variable. However, this implementation is vulnerable in the case of blockchain forks, as both chains would retain the same domain separator despite having different chain IDs.

Consider using OpenZeppelin's EIP-712 Upgradeable library, which dynamically calculates the domain separator instead of caching it in storage. This would not only fix the issue but also make the `AttestationManager` contract consistent with the `AllocationManager`, which leverages `EIP712Upgradeable` instead of having a custom implementation.

Update: *Acknowledged, not resolved. The Graph team stated:*

> *The decision to keep the custom implementation for EIP-712 was made for backward-compatibility reasons with attestations. There is off-chain tooling that would need to be adapted which is not something we would like to do now as we are trying to minimize scope. It is definitely something we will revisit once the dust has settled post horizon upgrade. In the event that there is a hard fork, we would only support the protocol on one of the forks making this a less relevant issue.*

L-09 Inconsistencies or Missing Information in Arbitration Charter

The Arbitration Charter documents several limitations of the on-chain dispute resolution mechanism and provides rules for arbitrators to follow in order to solve these limitations. Multiple improvements that could be made to the document were identified:

1. Conflicting Query Attestation Dispute: The document describes three possible resolutions for a dispute, namely the first indexer winning, the second winning, or a draw.

There is one additional case that exists and is possible within the smart contracts, where both indexers have provided incorrect answers and are slashed. Consider including this case in the document as well. In addition, consider combining the first two cases into a more abstract one that uses wording such as "one of the indexers is slashed, the other is not" instead of having two separate cases where the indexers are differentiated by **first** and **second**.

2. Double Jeopardy: For indexing disputes, the smart contract requires uniqueness of the `disputeId`, which is composed of the `_allocationId` and `_poi`. This means that an indexer can be slashed at most once per indexing fault, and this rule is already enforced by the on-chain code. However, for query disputes, this is not the case: the `disputeId` includes the `fisherman` which created the dispute, meaning multiple disputes could be created for the same fault. Consider including in the charter all the relevant information an arbitrator should look at when assessing Double Jeopardy, such as:
    - the fisherman that created the dispute
    - the address that asked for the query
    - any other factor that would differentiate between queries and is not included in the `request` and `response` CIDs, such as the timestamps of the request being created and served

Update: *Acknowledged, will resolve. The Graph team stated:*

> *These are valid points. We have a big update planned for the Arbitration Charter so we will make sure to include these in the new version.*

L-10 Unsafe ABI Encoding

It is not an uncommon practice to use `abi.encodeWithSignature` or `abi.encodeWithSelector` to generate calldata for a low-level call. However, the first option is not typo-safe and the second option is not type-safe. The result is that both of these methods are error-prone and should be considered unsafe.

The `slash` function of the `HorizonStaking` contract uses `abi.encodeWithSelector`, which is unsafe.

Consider replacing this occurrence with `abi.encodeCall`, which checks whether the supplied values actually match the types expected by the called function and also avoids errors caused by typos.

Update: *Resolved in pull request #1146.*

L-11 The `takeRewards` and `getRewards` Functions Can Return Incorrect Values for Closed Allocations

The `getRewards` and `takeRewards` functions calculate the current rewards for an allocation based on multiple factors, one of which is the amount of `tokens` inside the allocation. This can result in returning/minting incorrect amounts of rewards, as neither the legacy system nor the subgraph service sets an allocation's tokens to 0 after closing it.

Note this should not pose an on-chain security threat as the `getRewards` function is not used on-chain and the `takeRewards` function is only called after ensuring the allocation in question is active. However, these functions might be used from off-chain, or rewards issuance bugs might appear if developers mistakenly remove the check that the allocation is active.

Consider encapsulating such checks within the `getRewards` and `takeRewards` functions.

Update: *Resolved in pull request #1149. The Graph team stated:*

> *Fixed in PR. It is worth noting that the fix for `Staking.sol` and `HorizonStakingExtension.sol` contains code that does not follow CEI pattern. The code is the same in both instances but:*
>
> - *The `Staking.sol` code will never be deployed and is deprecated. We just need it for unit tests at this stage.*
> - *The `HorizonStakingExtension.sol` code has two mitigating factors. First, the internal calls made prior to the state update that breaks CEI are to trusted contracts. Second, this code path will only be valid during the transition period, more specifically, it will only be reachable for indexers with legacy allocations open. We think this is an acceptable risk. However, we are also open to other alternatives like reverting `takeRewards()` to previous behavior but keeping the fix for `getRewards()`.*

L-12 Step-Wise Increase in Delegation Share Allows Fee Collection Frontrunning

Delegators can front-run the `addToDelegationPool` function to benefit disproportionately from large fee collections. While the thawing process prevents immediate withdrawal of these funds, this containment can be insufficient when query fee collection happens infrequently relative to the thawing duration. In scenarios where query fees are substantial and distributed

at lengthy intervals, delegators can strategically time their delegation immediately before collection events and then initiate the thawing process right after to maximize rewards while minimizing commitment duration.

To mitigate this front-running vulnerability, consider having service providers ensure that collection frequency is shorter than the thawing duration, forcing delegators to maintain continuous delegation to collect all available rewards. Additionally, consider implementing a mechanism to distribute reward collections in smaller amounts to prevent substantial one-time distributions.

Update: *Resolved through the considerations mentioned below and pull request #1150. The Graph team stated:*

> *This is an accepted risk mitigated by the following:*
>
> - *Based on current indexer behavior, collection frequency should be in the worst case equal to the thawing duration. It is worth noting that indexers with high query volume are also likely to be heavily optimizing allocations, meaning that they would be collecting much more frequently, sometimes on a daily basis.*
> - *Indexers with high query volume are also likely to have large delegation amounts, making the attack require more funds.*
> - *Delegated funds from the attack are still subject to the thawing period which means there is an associated loss from the cost of capital.*
>
> *Considering all these factors, we think that such attack is not very likely to be profitable except in very specific circumstances which are probably hard to predict.*

L-13 `SubgraphService` Cannot Collect Partial RAVs

The `GraphTallyCollector` contract implements two functions to collect `ReceiptAggregateVouchers`. The second function lets the caller specify how many tokens to collect out of a `RAV`. This functionality was implemented to allow a service provider to partially collect a `RAV` in case they cannot collect the whole amount due to not having enough available tokens to lock. However, while this functionality could plug into new data services, there is currently no way for the `SubgraphService` to invoke it, and, hence, indexers cannot perform partial `RAV` collections.

Consider accounting for this in `SubgraphService` and introducing the possibility of specifying how many tokens to collect out of the `RAV`.

Update: *Resolved in pull request #1150.*

L-14 Service Providers Could Gain Advantages By Delegating to Their Own Provision

If the amount of tokens in a provision's delegation pool is less than the possible amount that can be leveraged through the delegation ratio, a service provider will be incentivized to delegate tokens to their own provision, which can bring the following advantages:

- Part of the service provider's tokens are now less likely to be slashed as they are part of the delegation pool and only become slashable after the tokens inside the provision have been slashed. Moreover, a slashing event towards the delegation pool will only partly affect the service provider's tokens as it would be split between all participants.
- When collecting query fees and indexing rewards, the service provider will receive more tokens by also being part of the delegation pool instead of only provisioning tokens. This happens because the service provider receives the same percentage out of the reward, regardless of the size of the delegation pool or the provision. Hence, there is an incentive for the service provider to also participate in the delegation pool and receive part of the delegation cut.

Note that the above advantages are only relevant as long as other delegators are participating and the amount of tokens the service provider is delegating can be used towards creating allocations and collecting rewards. Moreover, there are other considerations that could reduce or nullify the profitability of the attack, such as

- the service provider wasting time to undelegate and provision when the strategy is no longer advantageous
- the delegators being less likely to delegate to provisions with smaller economic security, resulting in fewer tokens usable for allocation and reward collection

Since it is unclear the exact conditions in which such strategies could be profitable and further on-chain restrictions are not trivial to implement, consider documenting the above behaviors in your backlog and exploring them as part of the protocol's next iteration.

Update: *Resolved. The Graph team has noted this as an improvement for future iterations. The Graph team stated:*

> *Agreed with the observations noted here. There are certain cases where there could exist advantages, these appear to be very tricky and likely only work for periods of time, and more importantly dependent on third party actions which are impossible to predict*

> *(like other delegators joining in). We are not worried about this being exploited, but will note the comments and keep in mind for future iterations. It is also worth noting that horizon's design in this regard is a nice improvement over the current version of the protocol where delegation is not slashable.*

L-15 Escrow Tokens Remain Thawing After Collection

In the `PaymentsEscrow` contract, when a payer initiates thawing and then experiences a collection, the thawing state persists. If the collection reduces the account's `balance` below its `tokensThawing` amount, any new deposits might be immediately considered part of the thawing amount. This means that newly deposited tokens can inherit the thawing status instead of being available for service. This can cause confusion as payers depositing new funds would expect service to resume immediately but find their newly deposited tokens unavailable due to the ongoing thawing process.

Consider documenting this behavior clearly, advising payers who wish to deposit back into the escrow to first cancel their thawing request using the `cancelThaw` function if they intend the new deposits to be immediately available for service. Alternatively, consider modifying the `collect` function to reduce an account's thawing amount to the balance currently in the account, when needed.

Update: *Acknowledged, will resolve. The Graph team stated:*

> *Noted. We will make sure that off-chain components that interact with these contracts are aware of this.*

L-16 Repeated Event Emission After Authorization Revocation

The `thawSigner` and `cancelThawSigner` functions of the `Authorizable` contract can be called repeatedly even after a signer's authorization has been permanently revoked via `revokeAuthorizedSigner`. This behavior allows for the emission of `SignerThawing` and `SignerThawCanceled` events without any functional effect, since the signer cannot be re-authorized once revoked. This may confuse off-chain systems like user interfaces or indexers which could wrongly interpret repeated thawing events as a sign that re-authorization is possible, even after permanent revocation.

Consider adding a condition to both `thawSigner` and `cancelThawSigner` that prevents execution if the signer has already been revoked.

Update: *Finding is invalid. The Graph team stated:*

> *Both functions cannot be called unless the signer has a non-revoked authorization (see the `onlyAuthorized` modifier). So, it shouldn't be possible to call them repeatedly as described.*

L-17 Thawing End Timestamp Is Reset When Calling Thaw

In the `PaymentsEscrow` and `Authorizable` contracts, calling the thaw functions resets the thawing end timestamp from a previous thaw request, extending the thawing duration. This behavior is error-prone and can result in users having to wait longer than necessary for thawing.

In `Authorizable`, consider preventing thawing when the signer is already thawing. In `PaymentsEscrow`, documenting this behavior may be sufficient, as fully addressing it would require tracking individual thawing requests, introducing unnecessary complexity relative to the contract's intended functionality.

Update: *Acknowledged, will resolve. The Graph team stated:*

> *Noted. The contract interfaces already document this behavior but we will make sure to extend this to off-chain components.*

L-18 Verifier and Service Provider Collusion Enables Instant Withdrawal via Slashing

The `maxVerifierCut` value of a `provision` is designed to prevent verifiers from extracting the full value of a slashed provision, ensuring that malicious slashing remains economically irrational. However, a collusion attack vector exists: if a malicious verifier sets the accepted `maxVerifierCut` to 100% and colludes with a service provider, slashing will transfer the service provider's entire provision stake to the verifier. This effectively allows the service provider to bypass the thawing period and withdraw funds instantly, leaving only the delegated tokens at risk if delegation slashing is enabled. Moreover, the delegator stake could be maliciously burned without any economic consequences for the service provider.

Consider hard-coding a sensible maximum value for `maxVerifierCut` (e.g., 20%) within the `setProvisionParameters` function. This limit would ensure that instant withdrawal attempts burn at least 80% of the service provider's stake, rendering such withdrawal attempts economically unfeasible.

Update: *Acknowledged, risk accepted. The Graph team stated:*

> *This was considered during the design phase but discarded because we think there might be use cases where a high verifier cut, even 100%, might be a legitimate value. Due diligence from service providers and delegators should include checking what the data service parameters are and how those might impact them including understanding how verifier slashing works. Ultimately, when delegating or staking to a verifier participants accept the risk of the verifier being malicious and slashing them unfairly. This risk exists regardless of the `maxVerifierCut` value.*

L-19 Dispute Cancellation Time Can Change Retroactively

The `cancelDispute` function uses `dispute.createdAt + disputePeriod` to determine whether a dispute can be canceled. Since `disputePeriod` is a mutable protocol parameter, updating it retroactively affects the cancellation eligibility of existing disputes.

Consider storing a `cancellableAt` timestamp in the `Dispute` struct during dispute creation (e.g., `block.timestamp + disputePeriod`) and modifying the cancellation check to use `cancellableAt` instead of dynamically recalculating `createdAt + disputePeriod`.

Update: *Resolved in pull request #1151.*

Notes & Additional Information

N-01 Redundant Parameters in `_undelegate` and `withdrawDelegated` Functions

The `_undelegate` and `_withdrawDelegated` functions of the `HorizonStaking` contract receive `_requestType` as their first parameter. This is redundant because `_requestType` is always passed as `ThawRequestType.Delegation`. Similarly, the `_undelegate` function includes a `_beneficiary` parameter which is always passed as `msg.sender`.

To make the code easier to read and understand, consider removing these redundant parameters by replacing the instances of `_requestType` with `ThawRequestType.Delegation` and the instances of `_beneficiary` with `msg.sender`.

Update: *Resolved in pull request #1133.*

N-02 Unnecessary Restriction on `acceptProvisionPendingParameters`

The `acceptProvisionPendingParameters` function of the `SubgraphService` contract is used to change a provision's `thawingPeriod` and `maxVerifierCut` to the pending values if they are within the enforced bounds. Note that the `acceptProvisionPendingParameters` function also enforces the provision to contain an appropriate amount of tokens which can unnecessarily prevent the indexer from updating the provision's parameters after a slashing event. It should also be noted that having a more lax requirement should not bring a security risk, as a provision that does not meet the token requirements will still be considered invalid by the `SubgraphService` as well as slashable by `HorizonStaking`.

Consider updating the `acceptProvisionPendingParameters` function to only require the parameters that change to be within bounds.

Update: *Resolved in pull request #1152.*

N-03 Unused Code

Throughout the codebase, multiple instances of unused or redundant code were identified:

- The `onlyController` modifier of the `Managed` contract
- The `_graphTokenGateway` and `_graphProxyAdmin` functions of the `GraphDirectory` contract
- The `_getLegacyAllocation` and `_getAllocation` functions of the `AllocationManager` contract
- The `_updateRewards`, `_resizeAllocation`, and `_allocate` functions' return values that are not used by the callers
- The `using UintRange for uint256;` directive in the `UintRange.sol` library

To make the code easier to read and save some gas costs at deployment time, consider removing the above-mentioned instances of unused code.

Update: *Resolved in pull request #1153. The Graph team stated:*

> *Removed instances of unused code except for the `_graphTokenGateway` and `_graphProxyAdmin` functions of the `GraphDirectory` contract --> these can be used by future contracts extending `GraphDirectory`.*

N-04 Unnecessary Loop in `RewardsManager` Increases Code Complexity

The `RewardsManager` contract contains a loop to iterate through rewards issuers when calculating the subgraph-allocated tokens. The code creates an array of two addresses and loops through them to sum their allocated tokens, adding complexity to a simple operation that only needs to handle two fixed addresses.

Consider making the code more readable and efficient by directly calling the two reward issuers and removing the array creation and loop overhead.

Update: *Acknowledged, not resolved.*

N-05 Misleading PPM Calculation in the `GraphPayments` Contract

In the `collect()` function of the `GraphPayments` contract, each fee cut is calculated sequentially from the remaining token amount after previous deductions. When the delegation cut is defined as PPM (parts per million), it is not applied to the total initial token amount but rather to what remains after the protocol and data service cuts. This creates a situation where the actual percentage a party receives depends on upstream cuts, making the effective PPM value different from what might be expected.

Consider clearly documenting this cascading calculation behavior, so users understand that the specified PPM values do not represent absolute percentages of the original payment amount.

Update: *Acknowledged, will resolve. The Graph team stated:*

> *This is already the case with the current version of the protocol so somewhat expected already. We will make sure though that user-facing documentation reflects this.*

N-06 Breaking Changes in `external` Functions Could Disrupt Dependent Contracts

Several `external` functions in the codebase have been modified from their original implementations by changing parameter signatures or removing functions entirely. For example, the `getRewards` function in `RewardsManager` was changed from having a single argument to requiring multiple parameters.

Additionally, `HorizonStakingExtension` only includes some legacy functions but not all of them. These breaking changes can cause dependent contracts to malfunction if they rely on the original function signatures. Even seemingly minor modifications to `view` functions can lead to complete contract failure for dependent systems.

Consider performing an on-chain analysis to identify existing usage of these functions and provide adequate notice to the community about breaking changes, allowing sufficient time for updates to dependent systems.

Update: *Acknowledged, will resolve. The Graph team stated:*

> *Noted. To the best of our knowledge, there is only one other party that depends on the staking contract and we will notify them once the audit is finished.*

N-07 Inconsistent `require` Statement Error String

Both the `getRewards` and `takeRewards` functions require the caller to be either the `staking()` or `subgraphService` contracts. However, they revert with different error strings.

Consider unifying the behavior by using the same error string and using a custom error.

Update: *Invalid finding. The Graph team stated:*

> *`getRewards()` does not require the caller to be a rewards issuer but rather one of the function arguments must be one. That is why the messages are different.*

N-08 Missing Docstrings

Throughout the codebase, multiple instances of missing docstrings were identified:

- In `DataServiceFeesStorage.sol`, the `feesProvisionTracker` state variable does not have any docstrings.
- In `DisputeManager.sol`, the `MAX_FISHERMAN_REWARD_CUT` and `MIN_DISPUTE_DEPOSIT` state variables have comments that could be helpful to users. Consider changing the documentation to be externally visible, by using the `@notice` NatSpec tag.

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

Update: *Resolved in pull request #1154.*

N-09 Shadowed State Variables

Throughout the codebase, multiple instances of shadowed state variables were identified:

- The `maxPOIStaleness` state variable is shadowed here.
- The `rewardsDestination` state variable is shadowed here and here.
- The `subgraphService` state variable is shadowed here.
- The `arbitrator` state variable is shadowed here.

- The `disputePeriod` state variable is shadowed here.
- The `disputeDeposit` state variable is shadowed here.
- The `stakeToFeesRatio` state variable is shadowed here.

To avoid undesirable behavior and improve code clarity, consider renaming variables that shadow any state variable.

Update: *Resolved in pull request #1155.*

N-10 Misleading, Incorrect, or Incomplete Documentation

Throughout the codebase, multiple instances of misleading or incorrect documentation were found:

- This comment is incorrect, as the `HorizonStakeWithdrawn` event can also be emitted throughout the transition period.
- This comment is incorrect, as the `tokens` parameter is used as the total amount of tokens locked, not the amount unstaked just now.
- This diagram is incorrect, as the functions used to deposit into the `PaymentsEscrow` are `deposit` and `depositTo`.
- This comment is incorrect, as the `ProvisionPendingParametersAccepted` event is not emitted in the `_acceptProvisionParameters` internal function, but rather in the `acceptProvisionPendingParameters` function.
- This comment is incorrect, as it is possible for a service provider to have multiple allocations for the same subgraph. Hence, closing an allocation does not necessarily mean he stopped indexing the subgraph.
- This comment is outdated, as the `disputeId` is created from different information.
- There are several typos, such as wether, addres, and these parameters of the `outboundTransfer` function.
- This comment could mention that a timestamp of 0 means the authorization is not thawing. Consider adding this information.
- This comment is incorrect, as the `exp` function reverts for `x > 0`.
- This comment is incorrect, as the result can also be equal to the smallest of the two values.
- Similarly, this NatSpec could mention that a `_tokensToCollect` of 0 means all tokens from the `RAV` should be collected.

To make the codebase documentation more consistent and easier to understand, consider applying the suggestions mentioned above.

Update: *Partially resolved in pull request #1156.*

N-11 Misleading Naming

Throughout the codebase, multiple instances of misleading naming were identified:

- The pointer to the next element in a linked list is called `nextClaim` for the `StakeClaim` struct and `next` for the `ThawRequest` struct. Consider being consistent and naming both `next` or changing the name of the `ThawRequest` field to `nextRequest`.
- The amount of tokens to rescue in the `_rescueTokens` function is called `_tokens`, which hints of a list of tokens rather than an amount. Consider renaming it to `_amount`, and applying similar changes to the `rescueGRT` and `rescueETH` functions.
- Consider renaming the `EIP712_ALLOCATION_PROOF_TYPEHASH` constant to `EIP712_ALLOCATION_ID_PROOF_TYPEHASH`, matching the name of the struct it represents.

Update: *Partially resolved in pull request #1157.*

N-12 Gas Optimizations

Throughout the codebase, multiple opportunities for gas optimization were identified:

- Swaping the storage read order in HorizonStaking so that the provision is read from storage only after verifying that `_tokens` is non-zero, thereby saving gas on reversion and ensuring logical consistency.
- Optimizing the fallback assembly function in HorizonStaking by eliminating the free memory pointer (using a constant such as `0`) and removing memoization of `returndatasize()`, thus reducing gas consumption.
- Changing the function in ExponentialRebates from external to internal to leverage compiler optimizations and avoid unnecessary external call overhead.
- Condensing the two allocation assignment lines in LegacyAllocation into a single statement, simplifying the code.
- Removing the redundant `whenNotPaused` and `whenPaused` modifiers in the public `pause` and `unpause` functions within DataServicePausable and

`DataServicePausableUpgradeable`, since these checks are already enforced in the internal `_pause` and `_unpause` functions.

When performing these changes, aim to reach an optimal trade-off between gas optimization and readability. Having a codebase that is easy to understand reduces the chance of errors in the future and improves transparency for the community.

Update: *Resolved in pull request #1158.*

Client Reported

CR-01 Missing field from GraphTally's `EIP712_RAV_TYPEHASH`

The `collectionId` field of the `ReceiptAggregateVoucher` struct is not included in the `EIP712_RAV_TYPEHASH` or the `hashStruct` of the voucher. Hence, the signer of the voucher will be recovered incorrectly, making the data service provider unable to receive query fees payment.

Consider including the `collectionId` field in both the `EIP712_RAV_TYPEHASH` and the voucher's `hashStruct` calculation.

Update: *Resolved in pull request #1126.*

CR-02 Indexers Can Be Locked Out of the `SubgraphService` By Creating a Provision With Incorrect Parameters

Horizon provision parameters can be directly set when creating the provision, but for a provision that's already created, the only way of modifying the provision parameters is with a two-step process:

1. The service provider stages a parameter change
2. The changes are not effective until the verifier (data service) validates and accepts the new parameters

The `SubgraphService` allows a service provider to initiate step 2 by calling the `acceptProvisionPendingParameters` function which is protected by the `onlyRegisteredIndexer()` modifier. Hence, it will revert if it is not called by a service provider already registered in the `SubgraphService`.

However, a service provider that inadvertently creates a provision with incorrect parameters will not be able to register themselves since the `register` function will only allow a provider to register if the provision's parameters are valid. This creates a deadlock, where the service provider is effectively locked out of the `SubgraphService`.

Consider removing the `onlyRegisteredIndexer` modifier from the `acceptProvisionPendingParameters` function to allow a service provider to alter incorrect provision parameters without having to be registered first.

Update: *Resolved in pull request #1127.*

CR-03 Misleading Custom Error Name in `slash` Function

The `slash()` function of the `HorizonStaking` contract contains a `require` check that reverts with the `HorizonStakingInsufficientTokens` custom error if there are no tokens in the provision. Note that this error name can be misleading, as it makes sense in other contexts, such as not having enough tokens to thaw or enough tokens left after undelegating.

Consider creating and throwing a different error which is more appropriate in the context of slashing a provision with no tokens inside.

Update: *Resolved in pull request #1141.*

CR-04 Several Events Are Missing Important Parameters

Throughout the codebase, multiple instances of events that do not emit any key information were identified:

- The `TokensUndelegated` event does not emit the amount of shares that have been undelegated.
- The `QueryFeesCollected` event does not emit the `allocationId` and `subgraphDeploymentId`.

To give more context and relevant information to off-chain entities that observe the events, consider adding the aforementioned parameters to the events.

Update: *Resolved in pull request #1142. The PR also introduces a* `forceClosed` *argument to the* `AllocationClosed` *event and updates encompassing code accordingly.*

Conclusion

Graph Horizon is the next evolution of The Graph protocol, addressing the evolving needs of the data indexing ecosystem and incorporating the learnings of the development team. It introduces `HorizonStaking`, a staking primitive that provides economic security, enabling the creation of modular and diverse data services.

Throughout the audit, The Graph team was extremely responsive and collaborative, promptly addressing any questions posed and providing valuable insights. The technical documentation paired with the codebase was found to be comprehensive, while the overall architecture of the Horizon upgrade appeared well-thought-out and a major step forward for The Graph protocol.

After thoroughly reviewing the codebase and the accompanying documentation, multiple high- and medium-severity issues were identified, along with various low-severity issues and codebase improvement opportunities. No critical issues or significant gaps were identified in the protocol, and the design limitations have been documented in the introduction.

Appendix

Thawing Token Calculation Analysis

Exact Calculation Using Real Number Arithmetic

To determine the exact thawing token amount, we begin with the intended formula using real number arithmetic:

$$H = \frac{H_0 \cdot (10^{18} - F)}{10^{18}}$$

where H represents the remaining thawing tokens, H_0 is the initial thawing token amount, and F is the fraction of slashed tokens scaled by 10^{18}, defined as:

$$F = \frac{10^{18} \cdot T_{\text{slash}}}{T}$$

where T_{slash} is the slashed token amount and T is the total token amount. Substituting the expression for F into our original equation:

$$H = \frac{H_0 \cdot \left(10^{18} - \frac{10^{18} \cdot T_{\text{slash}}}{T}\right)}{10^{18}} \tag{1}$$

$$= H_0 \cdot \frac{T - T_{\text{slash}}}{T} \tag{2}$$

This final formula represents the mathematically exact value of H. Intuitively, this formula applies the slashing ratio directly to the thawing tokens.

Integer Arithmetic Implementation Analysis

In Solidity's integer arithmetic environment, divisions always round down (floor division). We'll use the notation $\lfloor x \rfloor$ to denote integer division that rounds toward zero.

The current implementation calculates F with a ceiling division by adding $(T - 1)$ to the numerator before division:

$$F_{\text{ceiling}} = \left\lfloor \frac{10^{18} \cdot T_{\text{slash}} + T - 1}{T} \right\rfloor$$

This ceiling division can be written as:

$$F_{\text{ceiling}} = \left\lceil \frac{10^{18} \cdot T_{\text{slash}}}{T} \right\rceil$$

where $\lceil x \rceil$ represents ceiling division.

The error introduced by this ceiling operation can be expressed as:

$$F_{\text{ceiling}} = F + \epsilon$$

where $0 \leq \epsilon < 1$.

Error Analysis

When using F_{ceiling} ($F + \epsilon$) in our thawing calculation:

$$H_{\text{int}} = \left\lfloor \frac{H_0 \cdot (10^{18} - (F + \epsilon))}{10^{18}} \right\rfloor \tag{3}$$

$$= \left\lfloor \frac{H_0 \cdot (10^{18} - F)}{10^{18}} - \frac{H_0 \cdot \epsilon}{10^{18}} \right\rfloor \tag{4}$$

The first term is equivalent to the exact H value we derived earlier. The second term represents our error:

$$H_{\text{int}} = \left\lfloor H - \frac{H_0 \cdot \epsilon}{10^{18}} \right\rfloor$$

Since $0 \leq \epsilon < 1$, the error term falls within the range:

$$0 \leq \frac{H_0 \cdot \epsilon}{10^{18}} < \frac{H_0}{10^{18}}$$

This means the computed value using integer arithmetic will be:

$$H - \frac{H_0}{10^{18}} < H_{\text{int}} \leq H$$

Summary

The mathematical analysis confirms that the current implementation introduces a small negative error. The integer arithmetic result can be up to $\frac{H_0}{10^{18}}$ tokens less than the mathematically exact value. This means thawers may lose up to 1 wei per 10^{18} thawing tokens.