OpenZeppelin | security

The Graph Horizon Audit

The Graph

September 5, 2024

Table of Contents

Summary

Type	Infrastructure
Timeline	From 2024-06-10 To 2024-07-26
Languages	Solidity

Total Issues	49 (32 resolved, 1 partially resolved)
Critical Severity Issues	5 (5 resolved)
High Severity Issues	11 (11 resolved)
Medium Severity Issues	6 (6 resolved)
Low Severity Issues	9 (8 resolved, 1 partially resolved)
Notes & Additional Information	16 (0 resolved)
Client Reported Issues	2 (2 resolved)

Scope

We audited the ❄️ Graph Horizon and Subgraph Service ❄️ pull request at commit f47cf91. This pull request implements a new system version called Graph Horizon.

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
│   ├── IGraphPayments.sol
│   ├── IGraphProxyAdmin.sol
│   ├── IHorizonStaking.sol
│   ├── IPaymentsCollector.sol
│   ├── IPaymentsEscrow.sol
│   ├── ITAPCollector.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/TAPCollector.sol
├── staking
│   ├── HorizonStaking.sol
│   ├── HorizonStakingBase.sol
│   ├── HorizonStakingExtension.sol
│   ├── HorizonStakingStorage.sol
│   ├── libraries/ExponentialRebates.sol
│   └── utilities/Managed.sol
└── utilities/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/ICuration.sol
├── epochs/IEpochManager.sol
├── gateway/ICallhookReceiver.sol
├── governance
│   ├── Controller.sol
│   ├── Governed.sol
│   ├── IController.sol
│   ├── IManaged.sol
│   └── Pausable.sol
├── l2
│   └── staking
│       ├── IL2Staking.sol
│       ├── IL2StakingBase.sol
│       └── L2Staking.sol
├── rewards
│   ├── IRewardsManager.sol
│   ├── RewardsManager.sol
│   └── RewardsManagerStorage.sol
├── staking
│   ├── IStakingBase.sol
│   ├── L1Staking.sol
│   └── Staking.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, enabling the creation of modular and diverse data services. These services can cater to various data indexing needs, including 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 requirements for provision-locked amounts and thawing periods.

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 is removed and now the entire delegation is slashable to enhance economic security. 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.

The method by which data consumers pay for queries is now abstracted. The Graph team provides a functional payments protocol framework based on escrows and **Receipt Aggregate Vouchers** (RAVs). An 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.

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," preventing its 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 before collecting additional fees.

Upgradeability

Except for `TAPCollector` and `PaymentsEscrow`, all other contracts within the Graph Horizon system 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. In addition, delegations will not be slashable throughout the transition period.

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

Deployment and Backwards Compatibility

Graph Horizon is a complete redesign of the legacy protocol architecture, modifying most of the protocol contracts. After careful consideration, The Graph team has decided to follow a Brownfield approach when upgrading the protocol, meaning that Graph Horizon contracts are implemented to fit the legacy storage structure and state. It is important to note that while we have validated backwards compatibility of storage and functionality, the deployment strategy was not part of this audit. We encourage The Graph team to create a comprehensive plan around deployment and ensure its correctness through thorough fork and integration testing.

Security Model and Trust Assumptions

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 provide service but 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 `SubgraphService` 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

Graph Horizon aims to be as decentralized as possible and to minimize privileged roles to enhance censorship resistance. In the long term, the goal of the protocol is to achieve a fully permissionless and immutable protocol.

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.
- **Governor**: Has administrative privileges inside many contracts. To enumerate a few, it can:
    - set the cross-chain counterparty address that is allowed to transfer the indexer or delegator stake,
    - set data services without escape hatches for provisioning with vesting tokens,
    - trigger the end of the transition period, or
    - set the GRT inflation through issuance for indexing rewards.

  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

- **Service Providers**: These 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 and do the same actions.

- **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 after the thawing period 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 potential risks.

In addition, 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 upgrade the Subgraph Service's implementation, set the pause guardian, and set important operational parameters like the minimum amount of tokens required in a provision or the ratio with which delegated funds can be used for economic security.
- **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, governors, pause guardians, 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.

Critical Severity

C-01 `DelegationPool` Accounting Can Be Manipulated Allowing Malicious Delegators to Drain Funds

The `addToDelegationPool` function lacks proper access control and does not ensure that the necessary tokens are transferred from the caller. This allows any user to arbitrarily increase `pool.tokens`, inflating the value of a share and potentially draining the entire delegation pool.

Consider updating the `addToDelegationPool` function to include the token transfer from the caller. In addition, to prevent users from unintentionally calling this function directly instead of calling the `delegate` function, consider restricting access to only the `SubgraphService` and `GraphPayments` contracts.

Update: *Resolved in pull request #988. The Graph team stated:*

> *We'd like to keep `addToDelegationPool` open for everyone so as to not limit potential use cases in the future. Documentation already suggests Delegators should not use this to delegate and there won't be client side apps using this function.*

C-02 `GraphPayments` `collect` Function Locks Delegators' Fees

When assets are funneled through the `GraphPayments` contract, the `collect` function splits the funds for the protocol tax, delegators, data service, and receiver. However, when distributing the cut to the delegation pool, the `addToDelegationPool` function is called, which only updates the accounting without actually transferring the tokens. This results in the tokens intended for the delegators becoming stuck in the `GraphPayments` contract, disrupting the delegation pool's accounting.

Consider updating the `GraphPayments` `collect` function to transfer the delegators cut to the `HorizonStaking` contract. Alternatively, consider modifying the `addToDelegationPool` function and granting the `HorizongStaking` contract approval to pull the delegator tokens.

Update: *Resolved in pull request #988.*

C-03 `Curation` Contract's `collect` Function Will Always Revert When Called by the `SubgraphService`

When collecting query fees for a curated subgraph, a portion of the fees is distributed to curators and the `Curation.collect` function is called to update the curation accounting. The `collect` function is restricted to be callable only by the `staking()` contract, resulting in a revert when `SubgraphService` attempts to call it.

Consider updating the `collect` function to accept calls from both the `HorizonStaking` and `SubgraphService` contracts.

Update: *Resolved in pull request #989 and commit b71eca0 of pull request #1023.*

C-04 Lack of Proper Resource Access Control in the `SubgraphService`

In the `SubgraphService` contract, there is a lack of proper access control for functions that require the caller to have authorization for specific resources, such as allocations.

The current modifiers, including `onlyProvisionAuthorized`, `onlyValidProvision`, and `onlyRegisteredIndexer`, validate authorization using the indexer as the sole parameter. These modifiers can ensure that the caller is authorized to interact with provisions from the indexer, that the indexer has a valid provision, and that the indexer is registered. However, they fail to verify that the caller of the function has access to the other resources passed as parameters.

For instance, there is no validation in the `stopService` function that the `allocationId` corresponds to the indexer validated by the modifiers, allowing a malicious actor to close any allocation, stopping indexers from accumulating rewards and boosting the share of the malicious actor.

Similarly, when collecting `SubgraphService` rewards, the `collect` function allows a service provider to collect rewards for an invalid provision by specifying any indexer with a valid provision. The `onlyProvisionAuthorized` modifier is missing, and there is no validation that the information stored inside the `data` parameter corresponds to the `indexer`.

Additionally, in the `SubgraphService` `collect` function, the indexer's provision is required to be valid and the indexer must be registered with the `SubgraphService`. However, the `indexer` parameter can be arbitrarily specified for `QueryFee` payment types and is not checked against the `SignedRAV`. This means that payment can be collected for a `RAV` whose indexer is not necessarily registered or has a valid provision.

Consider fixing the above and double-checking every function to make sure that the correct resource access control is implemented.

Update: *Resolved in pull request #990 and pull request #992.*

C-05 Indexing Rewards Can Be Stolen by Resizing Closed Allocations

After provisioning tokens with the `SubgraphService`, a service provider can use free tokens to create allocations. Tokens are locked either when creating an allocation or when resizing one to a higher token amount. Similarly, tokens are released when closing an allocation or when resizing one to a smaller token amount.

The `_resizeAllocation` function does not ensure that only open allocations can be resized, allowing a malicious actor to release tokens twice by first closing an allocation and then resizing it to zero. This loophole enables the creation of an indefinite number of allocations without increasing the number of provisioned tokens, resulting in a disproportionate share of the indexing rewards for the malicious actor, with honest service providers receiving reduced or no indexing rewards.

Consider adding a check which ensures that only an open allocation can be resized.

Update: *Resolved in pull request #992.*

High Severity

H-01 Malicious Service Provider Can Inflate Their Share of Indexing Rewards

The `SubgraphService` contract allows an indexer to resize their allocation to a smaller or bigger amount. To prevent losing the accumulated rewards, the contract tracks them in the `accRewardsPending` variable of an allocation. This value represents the amount of tokens to reward per allocated token, not the total rewards accumulated with the allocated tokens that are subject to change.

A malicious actor can exploit this by allocating a small amount of tokens and resizing their allocation to a larger amount after a sufficient time. Upon claiming indexing rewards, the `accRewardsPending` value is incorrectly applied to each of the newly allocated tokens. This is an error as these rewards were accumulated while the indexer had a small amount allocated. This would allow a malicious indexer to grab a disproportionate share of the total indexing rewards, potentially denying other indexers their fair share. Conversely, an unsuspecting user resizing their allocation to a smaller amount could receive a reduced amount of the indexing rewards than he is owed.

Consider using the `accRewardsPending` variable to track the total amount of rewards pending after a resize instead of using the rewards accumulated per allocated token.

Update: *Resolved in pull request #993 and commit 157bfc6 of pull request #1023.*

H-02 Service Provider Can Force Slashing to Revert

When slashing a service provider's provision, the maximum number of tokens a provider allows to be slashed is calculated using the provision's tokens and `maxVerifierCut`. A service provider can choose any `maxVerifierCut` value they want, as long as it is between the range allowed by the data service. In the `SubgraphService` contract, this range is between the `DisputeManager`'s `verifierCut` and `uint32.max`.

A malicious service provider can exploit this by setting a provision's `maxVerifierCut` to a number exceeding a valid parts-per-million representation, causing the calculation of `maxVerifierTokens` to revert within the `PPMMath.mulPPM` function.

Consider setting the upper boundary of a provision's `maxVerifierCut` to a number smaller than the maximum parts-per-million value.

Update: *Resolved in pull request #998 and commits 99af0a6 and ddbc4d2 of pull request #1023.*

H-03 Delegator Receives Fewer Shares When Transferring Delegation from L1 to L2

When sending a delegation from L1 to L2, the shares received are calculated without accounting for the delegation pool tokens that are thawing, unlike the `_delegate` function. This oversight results in the delegator transferring funds but receiving fewer shares in return than intended, thereby unintentionally distributing part of their tokens to other participants in the delegation pool.

Consider including thawing tokens in the calculation to ensure that the correct number of shares corresponds to the transferred delegation.

Update: *Resolved in pull request #1001.*

H-04 Query Fees for `SubgraphService` Cannot Be Withdrawn

In the `SubgraphService`, the `_collectQueryFees` function calculates and distributes query fees, allocating a portion to the `SubgraphService` contract itself. However, the contract lacks a mechanism to withdraw tokens, resulting in them being permanently stuck.

Consider adding a mechanism to withdraw query fees from the `SubgraphService` contract.

Update: *Resolved in pull request #1004 and commit e2ac05d of pull request #1023. The Graph team stated:*

> *We decided to remove the subgraph service cut. We were going to set it to 0% initially and send 100% to curators anyways.*

H-05 Slashing Can Be Denied by Overwriting a Provision

In the `HorizonStaking` contract, the `_createProvision` function does not revert if the provision already exists, allowing for overwriting the existing provision with different parameters.

This enables service providers to bypass the parameters that were accepted by the data service when the service was started. A malicious actor can provision the minimum amount of tokens possible, start the service, and then exploit any of the following attack vectors:

- By setting the `maxVerifierCut` to 0, all slashing events can be forced to revert.
- By setting the `thawingPeriod` to 0, the service provider and delegators can thaw and immediately withdraw funds.
- By setting the `tokensThawing` and `sharesThawing` to 0, provision accounting might be broken leading to unexpected behavior.

After overwriting a provision, the service provider would lose the initially provisioned tokens. However, this exploit can still be made profitable by blocking slashing and having no deterrent to claiming rewards without indexing data or serving correct responses to queries.

Consider adding a check to the `_createProvision` function to ensure that the provision does not already exist.

Update: *Resolved in pull request #1005.*

H-06 Participation in a `DelegationPool` Can Be Blocked

In the `HorizonStaking` contract, because of a division by zero, the `_delegate` function will revert if all tokens in the delegation pool are thawing. This would block any other participants from delegating.

This state can be achieved accidentally by participants who want to thaw their funds and withdraw. In addition, an attacker can force this state by choosing an empty pool, delegating, and immediately thawing any amount of tokens. The attacker only bears the gas cost and blocks delegation for anyone until he withdraws his funds.

Consider adjusting the `shares` calculation to account for cases where all tokens in the pool are thawing and reward the shares accordingly.

Update: *Resolved in pull request #1006.*

H-07 `TAPCollector` Contract Incorrectly Recovers `RAV` Signers

The `collect` function of the `TAPCollector` contract expects a `SignedRAV` that can be used to recover the RAV's payer by utilizing EIP-712.

However, the `EIP712_RAV_TYPEHASH` is incorrectly calculated due to an additional space between `dataService` and `address  serviceProvider`. This error would cause any signature generated with an `EIP-712` tool to be incompatible with the on-chain signer recovery, leading to either reverts or unexpected behavior.

Consider adhering to the `EIP-712` standard by removing the additional space inside `EIP712_RAV_TYPEHASH` to ensure compatibility and correct signer recovery.

Update: *Resolved in pull request #1009.*

H-08 Unsigned Metadata in `RAV` Allows Bypassing Curator Fee Distribution

The signature of the `RAV` (Receipt Aggregate Voucher) does not include the metadata field. This is particularly problematic because when claiming query fees, there is no mechanism to verify that the query fees correspond to the actual allocation for which they were generated. This allows a malicious indexer to specify an arbitrary metadata field that encodes an `allocationId` of a subgraph without curation, causing the `getQueryFeePaymentCuts` function to return a curation cut of zero. This would allow the indexer to collect the full amount of the query fees while bypassing the intended fee distribution.

Consider including the metadata field in the signed data to ensure that it is verified and correctly linked to the relevant allocation by the payer.

Update: *Resolved in pull request #1014 and commit d8bce8c of pull request #1023.*

H-09 POIs Can Be Submitted by Anyone in the `_collectIndexingRewards` Function

The `_collectIndexingRewards` function lacks validation to ensure that only the allocation indexer or one of its authorized addresses can submit a Proof of Indexing (POI). This lack of validation allows a malicious actor to submit a POI to any allocation, making the indexer receive indexing rewards but also exposing them to the risk of slashing due to the submission of an incorrect POI.

Consider implementing validation to restrict the submission of POIs to the allocation indexer or its authorized addresses only.

Update: *Resolved in pull request #990 and refactored in pull request #1015. The Graph team stated:*

> *This was fixed initially by pull request #990. Pull request #1015 refactors the code a bit and introduces a new function to allow force closing allocations that are stale.*

H-10 Undercollateralized Provisions Can Be Created Using Tokens Backing Legacy Allocations

In the `HorizonStaking` contract, the `_createProvision` function calls the `_getIdleStake` function to ensure that the service provider has sufficient idle tokens to collateralize the new provision.

However, the calculation of the idle stake is a subtraction that does not take into consideration the `__DEPRECATED_tokensAllocated` state variable: `tokensStaked - tokensProvisioned - __DEPRECATED_tokensLocked`. During the transition period, this would allow creating provisions that are collateralized by tokens already being used to back a legacy allocation. If the legacy allocation or the provision were to be slashed, the other structure using the tokens could remain undercollateralized, leading to unexpected behavior or lack of deterrent to misbehave.

Consider excluding `__DEPRECATED_tokensAllocated` from the pool of available provision funds. Alternatively, consider ensuring that a service provider no longer has any legacy allocations when they want to create a provision.

Update: *Resolved in pull request #1019. The Graph team stated:*

> *Fixed in pull request #1019. Initially we were going to allow using legacy allocated tokens for new provisions, i.e allow stake double dip. We later pivoted from this idea.*

H-11 `MaxVerifierCut` Is Calculated Based on the Total Provision Instead of the Slash Amount

The `maxVerifierCut` variable of a `Provision` ensures that when a provision is slashed, only a previously agreed percentage of the slashed funds will be rewarded to the verifier, while the rest are burned. This is intended to deter malicious verifiers from attempting to seize provision tokens.

However, the `slash` function of `HorizonStaking`, `maxVerifierTokens`, incorrectly calculates the maximum amount of tokens that the service provider has agreed to send to the verifier. This is because `prov.maxVerifierCut` is supposed to be a percentage of the slashed tokens, not the whole provision. If used maliciously, this miscalculation allows a verifier to reward themselves with all of the funds in the provision.

For example, consider that there are 1000 tokens inside the provision and the verifier calls the `slash` function with `tokens` set to 500 and `tokensVerifier` set to 500. The `maxVerifierTokens` calculation will evaluate to 500, allowing the verifier to receive all of the slashed tokens. By repeating this process, a verifier can potentially empty a provision without burning any tokens. This can also be exploited by a previously vetted verifier, that is now locked, to withdraw funds that are still vesting.

Consider correcting the `maxVerifierTokens` calculation to reward a percentage of the slashed tokens instead of a percentage of the whole provision.

Update: *Resolved in pull request #1018.*

Medium Severity

M-01 Resizing an Allocation to Less Tokens Will Revert

In the `_resizeAllocation` function, the `_tokens - oldTokens` subtraction will revert if `oldTokens` is bigger than `_tokens`. This effectively prevents resizing an allocation to fewer tokens. While it is possible to close the allocation and reopen it with fewer tokens, this is not the desired behavior.

Consider revising this calculation to properly handle underflows.

Update: *Resolved in pull request #992.*

M-02 Repeated Slashing May Become Ineffective in the `DisputeManager`

The `DisputeManager` contract slashes a percentage of one's provisioned stake. If a service provider has been punished multiple times, the resulting slashed amount can become so small that it no longer serves as an effective deterrent.

Consider implementing a minimum slashing amount, ensuring that the system always slashes the greater of the calculated percentage or the minimum slashing amount to maintain effective deterrence.

Update: *Resolved in pull request #995. The new functionality takes a snapshot of the slashable stake at the time of dispute creation, and allows slashing up to a percentage of it.*

M-03 Early Return in `HorizonStaking` Contract's `_withdraw` Function Unexpectedly Locks Tokens

In the `HorizonStaking` contract, the `_unstake` function calls the `_withdraw` function with the `_revertIfThawing` parameter set to `false` to ensure that any unlocked tokens are withdrawn before locking more tokens.

However, the `_withdraw` function currently returns early if `_revertIfThawing` is set to `false`, resulting in the unlocked tokens not being returned to the user and instead being locked again along with the new tokens.

Consider revising the `_withdraw` function to ensure that it does not return early if `_revertIfThawing` is `false`, allowing the `_unstake` function to work as intended.

Update: *Resolved in pull request #994.*

M-04 Payer Can Potentially Rug Pull a Collector if Not Actively Monitoring Thawings

In the `PaymentsEscrow` contract, if a payer calls [`approveCollector`] to approve a collector and then immediately calls [`thawCollector`], the `REVOKE_COLLECTOR_THAWING_PERIOD` will start.

However, the collector will remain in the `authorizedCollectors` mapping until the payer calls [`revokeCollector`]. Unless the collector actively monitors all `thawCollector` events, including those from payers the collector is not currently interacting with, the payer can wait until the `REVOKE_COLLECTOR_THAWING_PERIOD` has passed and then start working with the collector. The collector might see themselves in the `authorizedCollectors` mapping of the payer, without knowing that once the payer accrues a significant debt, the payer can immediately call `revokeCollector` before the debt can be settled.

Consider adding a helper function that returns whether a payer is authorized for a collector and, if the payer is currently thawing the collector, the remaining time until the payer can revoke the collector. Moreover, consider adding this information to The Graph off-chain systems and UIs that the payers and service providers are using.

Update: *Resolved. The Graph team stated:*

> *We decided not to include helper functions. This information is already available with `authorizedCollectors` being public. There will also be subgraphs tracking this information. When a service provider detects that a payer started thawing them as collector they should stop doing business with them.*

M-05 Slashing Reverts When Exceeding Provision Balance

In the `HorizonStaking` contract, the [`slash` function] reverts if the verifier [attempts to slash more] than the available balance of the provision. Moreover, the `slash` function also reverts if the amount of tokens to slash the delegators for [is larger than the balance of the delegation pool]. This can render a provision "immune" to slashing if the slashing amount exceeds the available balance, potentially allowing malicious behavior.

Consider revising the logic so that the function does not revert in these cases but rather slashes the existing balance without causing a revert.

Update: *Resolved in [pull request #997].*

M-06 Delegators Can Lose Funds When Creating Thaw Requests

If multiple slashing events occur, the protocol can enter an inconsistent state, leading to unexpected behavior, such as a delegator losing part of their stake when creating a thaw request.

For instance, consider a scenario where three delegators each stake 1000 tokens and initiate thaw requests for the same amount, resulting in the delegation pool accounting for 3000 `tokensThawing` and 3000 `sharesThawing`. If the service provider misbehaves, causing the entire provision to be slashed, the delegation pool updates to reflect 0 `tokensThawing` but still shows 3000 `sharesThawing`.

Then, if the service provider locks more funds and other delegators stake additional tokens, the first delegator to attempt withdrawal will lose part of their stake because the `thawingShares` calculation will incorrectly evaluate to 1000 `thawingShares`. Consequently, the delegation pool will incorrectly account for 4000 `sharesThawing`, leading to each of the initial three delegators only being able to 250 tokens, resulting in a 750 token loss for the fourth delegator.

Upon completely slashing a provision, consider deleting any remaining state such as active thaw requests and delegation pool state. Alternatively, consider revising the accounting logic to correctly handle cases like the one aforementioned.

Update: *Resolved in pull request #999. The Graph team stated:*

> *If a service provider gets slashed entirely such that there's no tokens left in the delegation pool we decided to mark the pool as invalid so it won't be possible to delegate/undelegate/withdraw. Service providers shouldn't get slashed entirely unless they did something very wrong. They could bring it back to life by calling `addToDelegationPool`, it would also serve as a possibility for service providers to compensate their delegators if they wished to undo their wrong doings.*

Additionally, the OpenZeppelin team recommends to clearly document and convey to delegators what is the expected behaviour if an entire provision is slashed.

Low Severity

L-01 Lack of Incentive for a Fisherman to Deposit More than the Minimum

In the `DisputeManager` contract, a fisherman must make a deposit to open a dispute, which needs to be larger than the `minimumDeposit`.

However, there is no incentive or reason for a fisherman to deposit more than the minimum amount. In fact, depositing more only increases the risk of losing a larger amount if the dispute is rejected, as the bond is forfeited.

Consider simplifying the design to require a fixed bond amount for deposits.

***Update:** Resolved in pull request #1002.*

L-02 Allowing 100% Reward of Slashed Indexer Tokens to Fisherman Creates Bad Incentives

In the `DisputeManager` contract, it is possible to configure the percentage reward that a fisherman receives when slashing occurs.

Rewarding fishermen with 100% of the slashed indexer tokens creates an incentive for indexers to front-run a valid claim and create it themselves to avoid losing tokens. This allows indexers to preemptively penalize themselves in a controlled manner, mitigating their losses and undermining the intended deterrent effect of the slashing mechanism.

Consider revising the upper limit of the reward that a fisherman can receive from slashing.

***Update:** Resolved in pull request #1003.*

L-03 Lack of Upgradeability and Pausing Functionality in the `PaymentsEscrow` Contract

The `TAPCollector` and `PaymentsEscrow` are the only two contracts that lack upgradeability. The `PaymentsEscrow` contract maintains a mapping that identifies which collectors are authorized to withdraw funds on behalf of payers. Payers can only withdraw their escrowed funds from the contract after a specified thawing period. If a security vulnerability is discovered in the `TAPCollector`, users could be at risk of fund theft due to the inability to immediately retrieve their funds from the `PaymentsEscrow`.

Consider adding a pausing functionality to `PaymentsEscrow`, temporarily preventing withdrawals by collectors in case an issue is found. This pausing functionality could be removed later on once more trust is established in the correct working of the collectors.

Update: *Resolved in pull request #1007. The Graph team stated:*

> *We made `PaymentsEscrow` upgradable and pausable with the commitment to throw away the keys after a period. `TAPCollector` will remain non-upgradable.*

L-04 `transfer` and `send` Calls Are No Longer Considered Best Practice

When `transfer` or `send` calls are used to transfer ETH to an address, they forward a limited amount of gas. Given that gas prices for EVM operations are sometimes repriced, code execution on the receiving end of these calls cannot be guaranteed in perpetuity.

In line 69 of the `DataServiceRescuable` contract, ETH is transferred via `transfer`.

Instead of using `transfer` or `send`, consider using `address.call{value: amount}("")` or the `sendValue` function of the OpenZeppelin `Address` library to transfer ETH.

Update: *Resolved in pull request #1008.*

L-05 Payer Can Approve More Funds Than Intended

A payer can use the `approveCollector` function to increase a collector's allowance. The allowance decreases either when the payer successfully thaws the funds and revokes the collector or when payment is collected.

However, an issue arises if a collector withdraws right before the payer increases the allowance. For instance, if the current allowance is 500 and the payer intends to increase it by 100 (to 600), they may accidentally increase it by 150 if the collector withdraws 50 just before the transaction executes. This can occur due to an unfortunate ordering of transactions or intentional collection through the `SubgraphService`.

Consider modifying the `approveCollector` function to take as a parameter the amount by which to increase the allowance. This change would also simplify the code by removing the need to check for allowance increases.

Update: *Resolved in pull request #1010.*

L-06 Inconsistent Coding Style

Throughout the codebase, multiple instances of inconsistent coding style were identified:

- This occurrence of `DelegationPoolInternal` explicitly specifies the `IHorizonStakingTypes` library, which is unnecessary and inconsistent with other occurrences.
- Functions only move tokens after checking that the amount is not zero. This is not the case for the first and second `delegate` functions. Consider pulling the `require` check out of the `_delegate` function and into the external functions.
- Most events that change a storage variable to a new value only emit the updated value. This is not the case for the `SubgraphServiceSet`, `PendingImplementationUpdated`, `ImplementationUpdated`, `AdminUpdated`, `NewPendingOwnership`, `NewOwnership`, and `NewPauseGuardian` events. Consider only emitting the new value of a state variable to be consistent and save gas.
- Tokens are transferred using the `TokenUtils.pushTokens` function everywhere except these occurrences: 1, 2 and 3. Consider updating them to make the code consistent.
- When updating the `accRewardsPerAllocatedToken` variable of an allocation, the codebase leverages the `snapshotRewards` function call. Consider updating this occurrence to do the same.
- The `ATTESTATION_SIZE_BYTES` constant is calculated using other constants that are declared after it. While this will compile, it is hard to read and is inconsistent with the rest of the codebase. Consider reordering the constants under question.
- The `IDisputeManager`, `IGraphPayments`, and `ISubgraphService` interfaces specify an `initialize` function. This is inconsistent with the rest of the codebase and is considered a bad practice. Consider removing these occurrences and only mentioning the `initialize` functions inside the implementations.

Consider reviewing the above suggestions and implementing them to make the codebase more consistent and easier to read.

Update: *Resolved in pull request #1011 and commit 58ba769 of pull request #1023. The Graph team stated, as a response to following recommendations in the issue:*

> *Most events that change a storage variable to a new value only emit the updated value. This is not the case for the SubgraphServiceSet, PendingImplementationUpdated, ImplementationUpdated, AdminUpdated, NewPendingOwnership, NewOwnership, and NewPauseGuardian events. Consider only emitting the new value of a state variable to be consistent and save gas.*

Some of these events are already deployed and being used so we'll leave them unchanged.

> *When updating the accRewardsPerAllocatedToken variable of an allocation, the codebase leverages the snapshotRewards function call. Consider updating this occurrence to do the same.*

We do this differently to calculate `accRewardsPending`. We feel code is clearer this way.

L-07 Lack of Validations

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

- The `_delegationSlashingEnabled` flag should only be switched on once, when the transition period is over. However, the implementation allows subsequent calls that could switch it back off, leading to unexpected behavior. Consider adding a check to ensure that the `_delegationSlashingEnabled` flag can only be switched on.
- When registering with the `SubgraphService`, the `url` is required to not be empty. Consider adding such a check for the `geohash` as well.
- When receiving cross-chain delegation, consider adding a check ensuring that the provision it is added to exists.
- The `delegationFeeCut`, `protocolPaymentCut`, and `paymentCuts` variables should be within specific bounds. Otherwise, they can cause unexpected behavior and reverts.
- The `addTail` function does not verify if the provided ID is `bytes32(0)`, which contradicts the library documentation.

Consider implementing the above validations to reduce user error and minimize the attack surface of the codebase.

Update: *Resolved in pull request #1012.*

L-08 Missing or Incorrect Documentation

Throughout the codebase, multiple instances of missing or incorrect documentation were identified:

- This documentation does not apply as it references a check that occurs only when starting the service in the data service, not upon the creation of a provision.
- This documentation incorrectly states that it checks if an allocation is closed. It actually checks if the allocation is altruistic, meaning that it is set up by a backstop indexer for subgraphs that are not indexed, without collecting rewards, just to provide query responses for those subgraphs.
- This documentation incorrectly references the percentage assigned to the delegation pool. It should be the percentage to be given to the indexer, with the rest going to the delegation pool.
- This documentation is incorrect as it is the `undelegate` function that creates the `ThawRequest`.
- The `mulPPMRoundUp` function documentation states that one of the two values should be in `PPM`, but it only requires the `b` parameter to be so. Consider updating the documentation or the function implementation accordingly.
- While correct, it is difficult to understand why the `_toUint8` function adds only `0x1` for the `bytes` array length instead of `0x20`. Consider adding thorough documentation explaining where the `0x1` comes from.
- The intention and necessity of the `provisionLocked` and `setOperatorLocked` functions are not documented. Consider detailing why these functions are needed and how to set up the vesting wallets to create locked provisions.

In addition, throughout the codebase, there is inconsistency around where docstrings are written. Some contracts have them in the implementation while some have them in the interfaces.

To improve the consistency and clarity of the codebase, consider addressing the aforementioned instances of missing or incorrect documentation and settling on one approach regarding docstring placement.

Update: *Partially resolved in pull request #1013. The Graph team stated, as a response to following recommendations in the issue:*

> *The intention and necessity of the provisionLocked and setOperatorLocked functions are not documented. Consider detailing why these functions are needed and how to set up the vesting wallets to create locked provisions.*

Creating vesting contracts is outside of the scope of Horizon contracts.

> *settling on one approach regarding docstring placement.*

We'll update this after the audit.

L-09 Gas Inefficiencies

Throughout the codebase, multiple instances of gas inefficiencies were identified:

- The `NewPauseGuardian` event is emitted using the `pauseGuardian` state variable. Consider using the `newPauseGuardian` memory variable instead, and similarly updating the `DisputePeriodSet`, `MinimumDepositSet`, `FishermanRewardCutSet`, `MaxSlashingCutSet`, and `ArbitratorSet` events.
- In the `LinkedList.traverse` function, instead of having a `traverseAll` variable and an `if` statement, consider setting the `iterations` variable to `self.count` instead. Additionally, consider not caching the `processInitAcc` variable and using it directly.
- In the `_getDelegationPool` function, consider returning the pool directly instead of caching it in a secondary variable. This would save gas and simplify the function.
- The `Thaw` event emits both the `tokens` and `account.tokensThawing` variables. Consider only emitting one of the two since they are guaranteed to have the same value.
- The `SubgraphService` contract inherits both the `GraphDirectory` and `Directory` contracts. Consider only inheriting the `Directory` contract and adding any necessary, missing address to it.
- Consider removing the `>= 0` check from the `isValidPPM` function as `uint256` is guaranteed to be bigger or equal to zero.
- Consider only pushing tokens to a fisherman if the dispute were not a conflicting attestation. Otherwise, the fisherman would have posted no bond when creating the dispute and hence should not be sent tokens back. Similarly, consider doing the same when canceling a dispute.
- The `collector.authorized` flag can be replaced by considering a collector as authorized if they have an allowance larger than zero. Consider removing it to minimize storage space.

- The `onlyValidProvision` modifier of the `SubgraphService` contract is redundant and could be replaced with the one inside the `ProvisionManager` contract. This would save gas at deployment time.
- The `_fulfillThawRequest` function can be refactored such that instead of doing multiple calculations every iteration, it would only accumulate the thawed shares and only calculate the `tokensThawing` and `tokensThawed` variables at the end. The `getThawedTokens` function can also be similarly updated.
- In the `weightedAverageRoundingUp` function of the `MathUtils` library, consider caching the `weightA + weightB` calculation to save gas.

When implementing the aforementioned 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 #1016. The Graph team stated, as a response to following recommendations in the issue:*

> *The SubgraphService contract inherits both the GraphDirectory and Directory contracts. Consider only inheriting the Directory contract and adding any necessary, missing address to it.*

`GraphDirectory` and `Directory` serve different purpose so we'll leave both. We see that `CURATION` is part of both contracts, we'll remove it after the audit.

> *The `_fulfillThawRequest` function can be refactored such that instead of doing multiple calculations every iteration, it would only accumulate the thawed shares and only calculate the tokensThawing and tokensThawed variables at the end. The getThawedTokens function can also be similarly updated.*

This function is already quite complex and we feel being explicit here helps following the logic.

> *In the weightedAverageRoundingUp function of the MathUtils library, consider caching the weightA + weightB calculation to save gas.*

We'll leave this as it is for readability.

Notes & Additional Information

N-01 Cannot Filter by the `paymentType` Variable of the `PaymentCutsSet` Event

The `PaymentCutsSet` event is emitted when service and curation cuts are set or changed for a `paymentType`.

Considering indexing the `paymentType` variable of the event in order for off-chain systems to easily filter by the different types of payment.

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

> *Thank you for raising this issue. We will not be addressing it now, as we are going to do thorough integration tests after the audit and subsequent reviews that will lead to further improvements in codebase quality.*

N-02 Unused Errors

Unused errors can negatively affect code intent and readability.

The `AllocationManagerZeroTokensAllocation`, `DataServiceFeatureNotImplemented`, and `LegacyAllocationAlreadyMigrated` errors are unused.

Consider deleting the `AllocationManagerZeroTokensAllocation` error since creating allocations with no tokens is allowed for altruistic allocations. In addition, consider deleting the `DataServiceFeatureNotImplemented` error and using `LegacyAllocationAlreadyMigrated` to prevent migrating a legacy allocation twice.

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

> *Thank you for raising this issue. We will not be addressing it now, as we are going to do thorough integration tests after the audit and subsequent reviews that will lead to further improvements in codebase quality.*

N-03 Lack of Security Contact

Providing a specific security contact (such as an email or ENS name) within a smart contract significantly simplifies the process for individuals to communicate if they identify a vulnerability in the code. This practice is quite beneficial as it permits the code owners to dictate the communication channel for vulnerability disclosure, eliminating the risk of miscommunication or failure to report due to a lack of knowledge on how to do so. In addition, if the contract incorporates third-party libraries and a bug surfaces in those, it becomes easier for their maintainers to contact the appropriate person about the problem and provide mitigation instructions.

Throughout the codebase, multiple instances of contracts without a security contact were identified:

- The `DisputeManager` contract
- The `GraphPayments` contract
- The `HorizonStaking` contract
- The `HorizonStakingExtension` contract
- The `PaymentsEscrow` contract
- The `SubgraphService` contract
- The `TAPCollector` contract

Consider adding a NatSpec comment containing a security contact above each contract definition. Using the `@custom:security-contact` convention is recommended as it has been adopted by the OpenZeppelin Wizard and the ethereum-lists.

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

> *Thank you for raising this issue. We will not be addressing it now, as we are going to do thorough integration tests after the audit and subsequent reviews that will lead to further improvements in codebase quality.*

N-04 TODO Comments in the Code

During development, having well-described TODO comments will make the process of tracking and solving them easier. Without this information, these comments might age and important information for the security of the system might be forgotten by the time it is released to production. These comments should be tracked in the project's issue backlog and resolved before the system is deployed.

Throughout the codebase, multiple instances of TODO comments were found:

- line 378 of `HorizonStaking.sol`
- line 11 of `HorizonStakingStorage.sol`
- line 5 of `IHorizonStakingTypes.sol`
- line 7 of `Managed.sol`

Note that there are also TODO comments around the steps required after the transition period.

For each existing TODO comment, consider tracking them in the issues backlog and linking the inline comment to the corresponding entry.

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

> *Thank you for raising this issue. We will not be addressing it now, as we are going to do thorough integration tests after the audit and subsequent reviews that will lead to further improvements in codebase quality.*

N-05 Inconsistent Parameter and Mapping Order

In the `HorizonStaking` contract, the `_isAuthorized` function has the following parameter order: `_operator`, `_serviceProvider`, `_verifier`. On the other hand, the `_operatorAuth` mapping uses this order: `_serviceProvider`, `_verifier`, `_operator`. In addition, the `_legacyOperatorAuth` variable is defined as a mapping from an operator to a service provider to a boolean status. However, it is being used as a mapping from a service provider to an operator to a boolean status.

These inconsistencies can lead to confusion and potential errors during integration or modification, as developers may misinterpret the correct parameter order.

Consider aligning the parameter order in both the `_isAuthorized` function and the `_operatorAuth` mapping for improved consistency. Moreover, consider updating the keys and value names in the `_legacyOperatorAuth` mapping definition to match its usage.

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

> *Thank you for raising this issue. We will not be addressing it now, as we are going to do thorough integration tests after the audit and subsequent reviews that will lead to further improvements in codebase quality.*

N-06 Variable Could Be `constant`

If a variable is only ever assigned a value when it is declared, then it could be declared as `constant`.

The `EIP712_ALLOCATION_PROOF_TYPEHASH` state variable could be `constant`.

To better convey the intended use of variables, consider adding the `constant` keyword to variables that are only set when they are declared.

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

> *Thank you for raising this issue. We will not be addressing it now, as we are going to do thorough integration tests after the audit and subsequent reviews that will lead to further improvements in codebase quality.*

N-07 Typographical Errors

Throughout the codebase, multiple instances of typographical errors were identified:

- `Wether` should be "Whether" (also in the `IRewardsIssuer` contract).
- `poriver` should be "provider".
- `bookeping` should be "bookkeeping".
- `unkock` should be "unlock".
- `Trown` should be "Thrown".
- `Retrun` should be "Return" (similarly here).
- `thawn` should be "thawed".
- `verifieres` should be "verifiers".
- `fulfulling` should be "fulfilling".
- `delegting` should be "delegating".
- `undelegting` should be "undelegating".
- `stil` should be "still".
- `Bookeeping` should be "Bookkeeping".
- `ownsership` should be "ownership".
- `Emmited` should be "Emited".
- `allocationl` should be "allocation".
- `disput` should be "dispute".
- Throughout the `IHorizonStakingTypes` interface, "inludes" should be "includes".
- Throughout the `Curation` and `L2Curation` contracts, `poool` should be "pool".

Consider fixing the aforementioned typographical errors in order to improve the readability of the codebase.

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

> *Thank you for raising this issue. We will not be addressing it now, as we are going to do thorough integration tests after the audit and subsequent reviews that will lead to further improvements in codebase quality.*

N-08 Inconsistent Licensing

Several files of the codebase have the `GPL-2.0-or-later` SPDX license, whereas others have the `GPL-3.0-or-later` license.

To ensure clarity and uniformity, consider settling on a single license for all files in the codebase.

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

> *Thank you for raising this issue. We will not be addressing it now, as we are going to do thorough integration tests after the audit and subsequent reviews that will lead to further improvements in codebase quality.*

N-09 Missing `override` Keyword in Functions

Throughout the codebase, multiple instances of functions lacking the `override` keyword were identified:

- The `pause` and `unpause` functions of the `DataServicePausable` contract
- The `pause` and `unpause` functions of the `DataServicePausableUpgradeable` contract
- The `setRewardsDestination` function of the `SubgraphService` contract

Functions that override the base contract functions ought to have the `override` keyword. Otherwise, it can be confusing for readers and developers, leading to potential misunderstandings about a function's origin and behavior. As such, consider using the `override` keyword for all functions that override base contract functions to enhance code readability and clarity.

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

> *Thank you for raising this issue. We will not be addressing it now, as we are going to do thorough integration tests after the audit and subsequent reviews that will lead to further improvements in codebase quality.*

N-10 Redundant Return Values

Throughout the codebase, multiple instances of functions with unused return values were identified:

- The `_closeAllocation` function of the `AllocationManager` contract
- The `presentPOI`, `clearPendingRewards`, `close`, and `snapshotRewards` functions of the `Allocation` library

Consider removing these unused return values to simplify the code and improve readability, ensuring that functions only return values that are actively utilized by their callers.

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

> *Thank you for raising this issue. We will not be addressing it now, as we are going to do thorough integration tests after the audit and subsequent reviews that will lead to further improvements in codebase quality.*

N-11 Use Constants for Default Values

In the `ProvisionManager` contract, the `__ProvisionManager_init_unchained` function initializes multiple default values such as `type(uint256).min`, `type(uint256).max`, `type(uint32).min`, `type(uint32).max`, `type(uint64).min`, and `type(uint64).max` directly within the function body.

This approach introduces magic numbers, reducing code readability and maintainability. Moreover, it could potentially introduce errors since the same literal values are used in another part of the codebase as well. In addition, the `verifierCutRange` is set between `0` and `uint32.max` but the maximum should be set to `MAX_PPM`.

Consider revising all uses of literal values, defining constants for them where applicable, and setting a more sensible maximum value for `verifierCutRange`.

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

> *Thank you for raising this issue. We will not be addressing it now, as we are going to do thorough integration tests after the audit and subsequent reviews that will lead to further improvements in codebase quality.*

N-12 Incorrect Parameter Order in `mulPPM` Function Call

In the `mulPPMRoundUp` function, the parameters of the `mulPPM` function call are passed in the order `(MAX_PPM - b, a)` instead of `(a, MAX_PPM - b)`.

While the order of parameters does not affect the result of the multiplication, using the correct order would enhance readability and maintain consistency with the `mulPPM` function definition.

Consider updating the function call to match the function definition.

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

> *Thank you for raising this issue. We will not be addressing it now, as we are going to do thorough integration tests after the audit and subsequent reviews that will lead to further improvements in codebase quality.*

N-13 Naming Suggestions

Throughout the codebase, multiple instances where code would benefit from renaming were identified:

- `function releaseStake(uint256 n)` could be `function releaseStake(uint256 numClaimsToRelease)`.
- `function _setPartialPaused(bool _toPause)` could be `function _setPartialPaused(bool _toPartialPause)`.
- `lastPausePartialTime` could be `lastPartialPauseTime`.
- `acceptProvision` could be `acceptProvisionPendingParameters`. Additionally, `staged` could be `pending`.
- Multiple variables follow the convention of `__DEPRECATED_{variable_name}`, whereas they will only be deprecated after the transition period. Consider renaming them to `__DEPRECATING_{variable_name}` and switching to the current naming after they are truly deprecated.

- `IHorizonStakingMain` could be `IHorizonStaking` to ease reading and match the implementing contract.
- `maxVerifierCut` and `thawingPeriod` could be `newMaxVerifierCut` and `newThawingPeriod`.
- `sharesThawing` and `tokensThawing` could be `sharesStillThawing` and `tokensStillThawing`.
- `onlyProvisionAuthorized` could be `onlyAuthorizedForProvision`.
- `ProvisionAccepted` could be `ProvisionPendingParametersAccepted`.
- `releaseAt` could be `releasableAt`.
- `_tokens` could be `amount`.
- `Resolve the conflicting dispute` could be `Draw the conflicting dispute`.
- In order to avoid confusion, consider limiting the use of `thawing` and `unthawing` to only instances referring to provisions. We suggest renaming `MAX_THAWING_PERIOD` to `MAX_WAIT_PERIOD` and taking a likewise approach when dealing with all the similar occurrences in the the `PaymentsEscrow` contract.
- In the `DisputeManager` contract, the terms "challenger" and "submitter" are used interchangeably with "fisherman", leading to potential confusion. Consider consistently using the term "fisherman".

Consider reviewing the aforementioned suggestions and applying them to improve code clarity and readability.

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

> *Thank you for raising this issue. We will not be addressing it now, as we are going to do thorough integration tests after the audit and subsequent reviews that will lead to further improvements in codebase quality.*

N-14 Event Emission Suggestions

Throughout the codebase, multiple instances of events that could emit additional information were identified:

- The `AuthorizedCollector` event could emit the amount for which the collector was authorized.
- The `TokensRescued` event could emit the token that was rescued.

Consider updating these events to enhance off-chain tracking and ensure comprehensive event logging.

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

> *Thank you for raising this issue. We will not be addressing it now, as we are going to do thorough integration tests after the audit and subsequent reviews that will lead to further improvements in codebase quality.*

N-15 Redundant Getter Functions

When state variables use `public` visibility in a contract, a getter function for the variable is automatically included.

Throughout the codebase, multiple instances of redundant getter functions were identified:

- Within the `ProvisionManager` contract, the `getDelegationRatio` and the `_getDelegationRatio` functions are redundant. Consider removing both or combining them into a `public` getter.
- Within the `DisputeManager` contract, the `getDisputePeriod` function is redundant.
- Within the `SubgraphService` contract, the `getAllocation` function is redundant.
- Within the `SubgraphService` contract, the `getLegacyAllocation` function is redundant.
- Within the `Allocation` library, the `_get` function is redundant and the `get` function could implement its logic.

To improve the overall clarity and readability of the codebase, and save some gas at deployment, consider removing the redundant getter functions.

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

> *Thank you for raising this issue. We will not be addressing it now, as we are going to do thorough integration tests after the audit and subsequent reviews that will lead to further improvements in codebase quality.*

N-16 Function Visibility Overly Permissive

Throughout the codebase, multiple instances of functions with unnecessarily permissive visibility were identified:

- The `_verifyAllocationProof` function in `AllocationManager.sol` with `internal` visibility could be limited to `private`.

- The `_getDelegatedTokensAvailable` function in `HorizonStakingBase.sol` with `internal` visibility could be limited to `private`.
- The `_receiveIndexerStake` and `_receiveDelegation` functions in `HorizonStakingExtension` with `internal` visibility could be made `private`.
- The `_deposit` function in `PaymentsEscrow.sol` with `internal` visibility could be limited to `private`.
- The `acceptDispute` and `drawDispute` functions in `DisputeManager.sol` with `public` visibility could be limited to `external`.

To better convey the intended use of functions, consider changing a function's visibility to be only as permissive as required.

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

> *Thank you for raising this issue. We will not be addressing it now, as we are going to do thorough integration tests after the audit and subsequent reviews that will lead to further improvements in codebase quality.*

Client Reported

CR-01 Pausing The `SubgraphService` Does Not Allow Slashing

The `SubgraphService` contract can be paused in case of unexpected issues. However, the `slash` function of the `SubgraphService` has the `whenNotPaused` modifier, which prevents any service provider from being slashed while the contract is paused. This allows service providers to act maliciously as they can thaw and de-provision their tokens from the `HorizonStaking` contract, without the worry of being slashed while the `SubgraphService` is paused.

Consider allowing slashing of service providers even when the `SubgraphService` is paused and adding this feature as a design guideline for other data services.

Update: *Resolved in pull request #1017.*

CR-02 Indexers Can Become Overallocated

A service provider for `SubgraphService` can lock their provisioned tokens inside allocations and start collecting the indexing rewards. It is crucial that the amount of tokens provisioned is always larger than or equal to the amount of tokens locked after reward collection. Similarly, the amount of tokens provisioned should be larger than or equal to the amount of tokens inside all of the service provider's allocations with the `SubgraphService`.

The current implementation does not enforce this and the `SubgraphService` contract implementation remains unaware if an indexer becomes overallocated, meaning that it has more tokens inside the allocations than the tokens left in its provision. This can happen in several cases and can have unexpected effects:

- After provisioning and allocating funds, a malicious service provider can deprovision most of their funds and continue to collect indexing rewards. The `SubgraphService` does not account for this and will continue accumulating rewards for the indexer proportionally to the initially allocated amount. The service provider can take their deprovisioned funds, reprovision them using a different address, and repeat, stealing indexing rewards from honest indexers.

- After a slashing event, a service provider can be left with fewer funds than those locked inside allocations. The `SubgraphService` does not account for this, accumulating indexing rewards for the indexer proportionally to the initially allocated amount.

If a service provider becomes overallocated, consider closing their allocation when they collect indexing rewards. It is possible that even after this fix, a malicious service provider will become overallocated and intentionally not collect indexing rewards, only to make other service providers' indexing rewards shares smaller. For this, consider allowing any protocol participant to close the allocation if the service provider does not collect the indexing rewards and the proof of indexing becomes stale.

Update: *Resolved in pull request #1020.*

Recommendations

Dispute Manager

We believe that the incentive structure for the `DisputeManager` contract could be improved by reducing its reliance on expecting the different actors to behave correctly. While this is less important for the Subgraph Service as it operates under the Graph Council, we believe the following recommendation would benefit other implementations that use the `DisputeManager` as a template and might not have the same level of trust.

Consider rewarding verifiers or arbitrators for solving disputes, and making the rewarded amount independent of the outcome of a dispute. This would encourage arbitrators to solve disputes and would give them less incentives to not be impartial.

Provision Management

Currently, if a service provider decides to thaw their tokens but then changes their mind, they must wait until the thaw request is over and then reprovision the tokens to the same provision.

Consider adding functionality to allow a service provider to cancel a thaw request and instantly have the funds rejoin the idle funds of the provision.

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.

The Graph team initiated this audit engagement while still improving the Graph Horizon codebase and actively developing the test suite, indicating that this was not yet a production version. The lack of unit and integration tests led to a significant amount of issues being discovered, with severities ranging from low to critical, underscoring the fact that the Horizon upgrade is not yet ready for deployment. Moreover, the Graph Horizon upgrade is following the Brownfield approach which requires extensive backwards compatibility tests to ensure that the new deployment does not put at risk the funds already in the protocol.

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

In summary, we assess that the Graph Horizon requires further development and extensive unit, integration, and backwards-compatibility testing. Once the protocol is considered production-ready, a final audit should be conducted before deployment to ensure a smooth and secure transition.