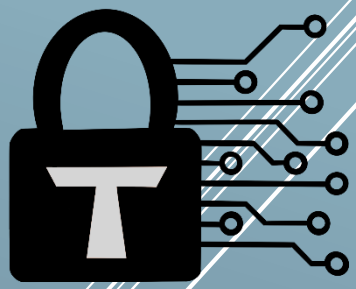
Trust Security

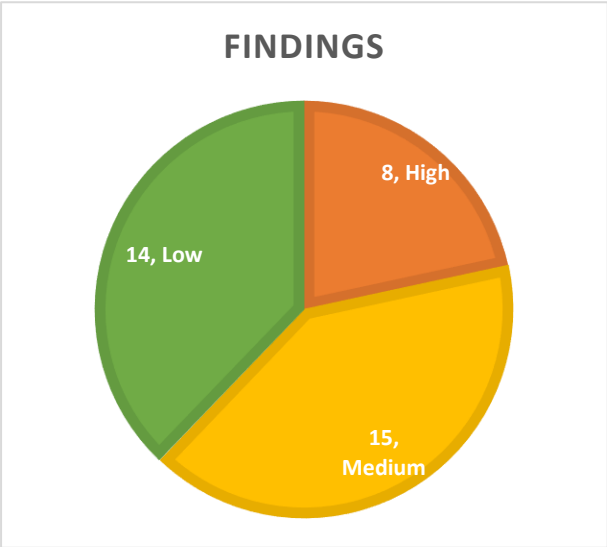


Smart Contract Audit

The Graph - Horizon Upgrade

17/12/24

Executive summary

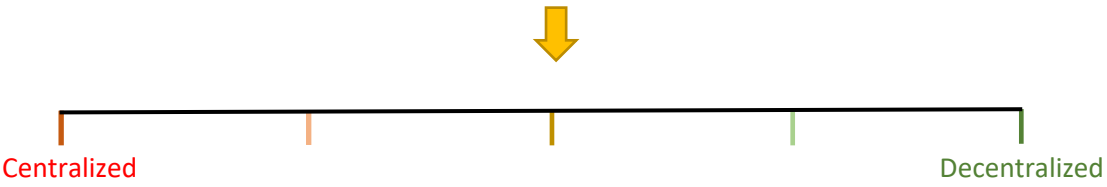


Category	Services Infrastructure
Audited file count	40
Lines of Code	3803
Auditor	Trust
Time period	11/10/24-26/11/24

Findings

Severity	Total	Fixed	Acknowledged
High	8	8	-
Medium	15	8	7
Low	14	7	7

Centralization score



Signature

Digitally signed by Trust
Date: 2024.12.17 11:57:50 +02'00'

Document properties

Versioning

Version	Date	Description
0.1	26/11/24	Client report
0.2	12/12/24	Mitigation review
0.3	17/12/24	Mitigation review #2

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

- contracts/rewards/RewardsManager.sol
- contracts/l2/curation/L2Curation.sol
- contracts/staking/HorizonStaking.sol
- contracts/staking/HorizonStakingExtension.sol
- contracts/staking/HorizonStakingBase.sol
- contracts/libraries/LibFixedMath.sol
- contracts/payments/collectors/TAPCollector.sol
- contracts/payments/PaymentsEscrow.sol
- contracts/data-service/utilities/ProvisionManager.sol
- contracts/utilities/GraphDirectory.sol
- contracts/data-service/extensions/DataServiceFees.sol
- contracts/libraries/LinkedList.sol
- contracts/payments/GraphPayments.sol
- contracts/staking/HorizonStakingStorage.sol
- contracts/data-service/libraries/ProvisionTracker.sol
- contracts/data-service/extensions/DataServiceRescuable.sol
- contracts/staking/libraries/ExponentialRebates.sol
- contracts/data-service/extensions/DataServicePausableUpgradeable.sol
- contracts/data-service/DataService.sol
- contracts/staking/utilities/Managed.sol
- contracts/data-service/extensions/DataServicePausable.sol
- contracts/libraries/MathUtils.sol
- contracts/libraries/PPMMath.sol
- contracts/data-service/utilities/ProvisionManagerStorage.sol
- contracts/data-service/extensions/DataServiceFeesStorage.sol
- contracts/libraries/Denominations.sol
- contracts/libraries/UintRange.sol
- contracts/data-service/DataServiceStorage.sol
- subgraph-service/contracts/DisputeManager.sol
- subgraph-service/contracts/SubgraphService.sol
- subgraph-service/contracts/utilities/AllocationManager.sol
- subgraph-service/contracts/libraries/Allocation.sol
- subgraph-service/contracts/libraries/Attestation.sol
- subgraph-service/contracts/utilities/AttestationManager.sol

- subgraph-service/contracts/utilities/Directory.sol
- subgraph-service/contracts/libraries/LegacyAllocation.sol
- subgraph-service/contracts/DisputeManagerStorage.sol
- subgraph-service/contracts/utilities/AllocationManagerStorage.sol
- subgraph-service/contracts/SubgraphServiceStorage.sol
- subgraph-service/contracts/utilities/AttestationManagerStorage.sol

Repository details

- **Repository URL:** https://github.com/graphprotocol/contracts
- **Commit hash:** 0c0d09090f6f8cff63a7cb4d499d94f579d159ba

Mitigation review fix commits:

- **PR 1072:** a06580420f485169c42b17df6b2109a340f3a23f
- **PR 1073:** 6e00d1764ded92317f9830d5ec7cc0403ba035d0
- **PR 1074:** 6ec971069ea91451bfb3827fb18b315e02c5e702
- **PR 1075:** 85de5bdf4879b323ab9e05b2fb512f6674d7185b
- **PR 1076:** 4d62209eafed4bc5aafe2d64413f38b11a7d8e64

Mitigation review #2 fix commits:

- **PR 1072:** dc79bd401974a605789f38a5d812b731a9006143
- **PR 1073:** 399b7a91f89248d95a224dc5369286b3f39ec590
- **PR 1075:** 4ba215c8925e38d0ec60a29ecfb9ac223b2faef4

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Since its inception it has safeguarded over 30 clients through private services and over 30 additional projects through bug bounty submissions.

About the Auditors

Trust has established a dominating presence in the smart contract security ecosystem since 2022. He is a resident on the Immunefi, Sherlock and C4 leaderboards and is now focused in auditing and managing audit teams under Trust Security. When taking time off auditing & bug hunting, he enjoys assessing bounty contests in C4 as a Supreme Court judge.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Good**	The code is well modularized to reduce complexity and attack risks.
Documentation	**Excellent**	Project is mostly very well documented.
Best practices	**Excellent**	Project consistently adheres to industry standards.
Centralization risks	**Moderate**	The Governance has significant privileges and is trusted with user funds.

Findings

High severity findings

TRST-H-1 A payer could bypass the escrow mechanism and avoid payments

- **Category:** Logical flaws
- **Source:** TAPCollector.sol
- **Status:** Fixed

Description

The TAPCollector's *collect()* is used by operators to collect payments, by calling a data service's function (like SubgraphService's *collect()*) which would trigger a collection. It receives a **signedRAV**, signed by the signer, and the RAV will be charged to the appropriate payer. Note that *collect()* must be called by the **dataService** which is listed in the **signedRAV**. When collection occurs, the escrow system will deduct the payment from the payer's balance. It is crucial that a payer's balance remains secure and exclusive to a particular provider, so that they can be confident they would be able to cash out a RAV at some point in the future.

An issue occurs due to the lack of validation that the **dataService** (i.e. msg.sender) is trusted by the provider. This opens the door to an exploit where a malicious payer could "recycle" the escrowed balance into their account while reducing it to zero. At that point, the provider will not be able to claim payments.

To execute the attack, a malicious signer will sign the following RAV:

- **dataService** – an EOA performing the *collect()* call.
- **serviceProvider** – the victim address provider
- **timestampN** – any value
- **valueAggregate** – the remaining balance in the escrow
- **metadata** – any value

They will then call TAPCollector's *collect()* passing the malicious RAV and **dataServiceCut** of 100%, minus protocol and delegator cuts. This will be accepted by the collector's msg.sender check and proceed to decrease the balance of the escrow mapping, while transferring almost the entire balance to the signer's account.

Recommended mitigation

The **dataService** must be validated to be trusted by the provider.

Team response

Fixed.

Mitigation review

The code now validates the service provider has provisioned tokens to the data service, fixing the issue.

TRST-H-2 Core thawing logic of Horizon staking can be broken due to collision in data structures

- **Category:** Mapping collision attacks
- **Source:** HorizonStaking.sol
- **Status:** Fixed

Description

The HorizonStaking contract allows providers and delegators to withdraw their deposited tokens after a thawing period. That is done in *_thaw()* and *_undelegate()* respectively, by calling *_createThawRequest(),* which inserts an item in the **_thawRequestLists** mapping. Note that the same mapping is used for both requests, where the keys for the 3-layered mapping are: **serviceProvider, dataService, beneficiary**. An item in the **thawRequestLists** includes the thawing share count to be burnt (from the **provision.sharesThawing** amount for provision thaws and **pool.sharesThawing** amount for delegator thaws).

The critical issue is that the mappings could collide: a delegator for a given provider's pool can pass **beneficiary = provider**, which will overlap with the provider's thaws. This would directly break the share accounting, since the amount of shares in the delegator pool thawing shares might be worth a significantly different amount compared to the provision thawing shares. It would lead to various different impacts, including bypassing of thawing period and theft of funds from other delegators.

Recommended mitigation

There should be a separate mapping for delegator pool thawing requests and provision thawing requests.

Team response

Fixed.

Mitigation review

The requests are now split between three categories via mapping keys, therefore no collisions are possible.

TRST-H-3 An attacker could prevent a delegator from ever withdrawing their tokens

- **Category:** DoS attacks
- **Source:** HorizonStaking.sol
- **Status:** Fixed

Description

A delegator could undelegate their stake and pass a beneficiary through the *undelegate()* function. This would create a thawing request in the beneficiary's list. However, consider that the list is capped to 100 items. Therefore, anyone can easily deny a delegator from starting a withdrawal by passing the victim's address as the beneficiary of a negligible withdrawal (e.g. 1 wei). This could easily be repeated indefinitely with a bot once the withdrawal has matured.

Recommended mitigation

Users should not be able to pass an arbitrary beneficiary – consider requiring their approval before adding a thaw request. Alternatively, add another layer in the request list mapping, which would be the initiator of the request.

Team response

Fixed.

Mitigation review

The code change makes undelegation requests to other beneficiaries separate from self-undelegations and deprovision requests, therefore the direct attack specified cannot work. However, a user can still prevent anyone from using *undelegateWithBeneficiary()* by filling the beneficiary's queue. If this functionality is counted on by ecosystem participants, this would constitute a valid griefing attack. Consider requiring a minimum undelegation token amount to disincentive DoS attacks, or further separate withdrawals by the undelegator.

Team response

Fixed.

Mitigation review #2

The issue is addressed by requiring a minimum undelegation amount of 10e18 GRT, as well as enlarging the withdrawal list to maximum length of 1000. This would likely provide sufficient deterance from filling the beneficiary queue, although in theory, an attacker could still block another user's undelegation with beneficiary requests.

TRST-H-4 The last withdrawal from a provision or delegation could permanently revert due to accounting flaw in slashing

- **Category:** Rounding issues
- **Source:** HorizonStaking.sol
- **Status:** Fixed

Description

In the Staking contract, it is a critical invariant that any pool's **tokensThawing** amount is smaller or equal to the pool's **tokens**. Otherwise, the last withdrawal request would underflow the token count and revert. This can in fact be broken due to a rounding error in the *slash()* code:

```
uint256 delegationFractionSlashed = (tokensToSlash *
FIXED_POINT_PRECISION) / pool.tokens;
pool.tokens = pool.tokens - tokensToSlash;
pool.tokensThawing =
    (pool.tokensThawing * (FIXED_POINT_PRECISION -
delegationFractionSlashed)) /
    FIXED_POINT_PRECISION;
```

Slashing attempts to reduce the **tokensThawing** by a proportional amount to the tokens slashing. However, **delegationFractionSlashed** is rounded down, while it is used in the latter calculation by reducing from **FIXED_POINT_PRECISION**. In other words, **tokensThawing** is rounding up with respect to **delegationFractionSlashed**.

The following possible scenario demonstrates the issue:

- Initial state is: **tokens = tokensThawing = 1e18 + 1**
- Now 1 wei is slashed. **delegationFractionSlashed = 1 * 1e18 / (1e18+1) = 0**
- **tokens = (1e18+1) – 1 = 1e18**
- **tokensThawing = (1e18+1) * (1e18 - 0) / 1e18 = 1e18 + 1**

At this point, **tokensThawing <= tokens** is broken and the final withdrawal will revert.

The same issue affects the service provider's **tokens** and **tokensThawing** structure.

Recommended mitigation

If the calculation of **delegationFractionSlashed** would round up, **tokensThawing** would round down which is the desired behavior.

Team response

Fixed.

Mitigation review

The code now rounds the fractions correctly, fixing the issue.

TRST-H-5 Allocations can be closed and historic rewards lost due to allocation power structure

- **Category:** Logical flaws
- **Source:** SubgraphService.sol
- **Status:** Fixed

Description

Whenever an allocation is closed through *forceCloseAllocation()* or voluntarily with *closeService()*, all rewards which have already been reserved for the allocation in the RewardsManager will be both forfeited by the indexer, and forever unclaimable. This is because these functions do not claim rewards, and the points allocated to the closed allocation are not redistributed across other allocations.

In our opinion this is already an issue in the current version of the protocol, because stuck rewards mean loss of value that could be sent to honest indexers of that period of time. However, it is definitely amplified by the fact that anyone can force-close an allocation when the index is overallocated. Indexers gain allocation power by getting delegated to, within a defined delegation factor. Indexers could be expected to allocate the additional power, otherwise that mechanic is unused. However, whenever they use delegated power, they are at risk of delegators removing their stake which immediately affects the total allocation power. At that point, anyone could close their allocation and make them (and honest

delegators) lose historic rewards. It should not be assumed that providers can trust their delegators to hold for a presumed duration.

Recommended mitigation

Any flow that closes allocations should claim historic rewards. This would make it far less risky for indexers to make use of delegated power.

Team response

“For the cases of “closing the allocation via stopService()” and the case of “force closing via forceCloseAllocation() when the allocation is stale” we believe the described finding is an acceptable outcome that can be avoided if the indexer does a good job at running their indexer:

-Delegators are subject to losing rewards in these two cases, this is akin to delegators getting slashed. On the long run delegation should gravitate towards indexers with good reputation and metrics (like historical APY) whereas those with a poor track record should have a harder time getting it.

-As for stuck rewards that could have gone to honest indexers/delegators, that is also an accepted risk. Worth noting that it’s not an unbounded amount that could get stuck, for one you need capital to get rewards stuck (which means cost of capital associated) and then once the allocation becomes stale anyone can close it to stop the dilution. This is something that the indexing community will for sure be on the look as it’s something they already do (or it’s equivalent) in the current version of the protocol.

Then there is the case of “force closing via forceCloseAllocation() when the indexer is over allocated”. This could be a serious problem, creating the wrong incentives in the protocol (like not using all your delegation, or having to equally split your stake on all your allos). We could not find a good solution to keep the feature without making it overly complicated (like a two step force close with a grace period for the indexer to “top up”), so we decided to remove it. This means accepting that overallocated indexers will gain unjust rewards. This is bounded however by the POI cycle and delegation thawing, when collecting rewards if the indexer is overallocated the allocation will automatically get closed. So at most the indexer will get one POI cycle of undeserved rewards (which is exactly what happens on the current version of the protocol, except the cycle will be shorter in Horizon). This fix is addressed by https://github.com/graphprotocol/contracts/pull/1074.

It’s worth noting that in order to distribute rewards a valid POI needs to be presented which is not possible in the cases where allocations are force closed (it’s assumed it’s not the service provider who is force closing).”

Mitigation review

The severe part of the issue has been fixed through disabling of the *forceCloseAllocation()* function, so providers cannot immediately lose rewards unless the allocation is stale, in which case the behavior is intended.

TRST-H-6 During the transition period, legacy allocations cannot be slashed

- **Category:** Missing functionality issues
- **Source:** HorizonStakingExtension.sol
- **Status:** Fixed

Description

During the transition period, new allocations cannot be created in the staking contract, and existing ones can be closed. As allocations are active, it's important that they can be slashed at this time. However, during the upgrading from the old staking contracts, the legacy *slash()* functionality has been removed, disabling any form of slashing during the transition period.

Recommended mitigation

Include the *slash()* implementation of the legacy StakingExtension.

Team response

Fixed.

Mitigation review

The functionality has been implemented as suggested. Note the refactor introduced issues with the new slashing function which are detailed separately.

TRST-H-7 Delegators that began undelegation before the transition would not be able to withdraw them

- **Category:** Missing functionality issues
- **Source:** HorizonStaking.sol
- **Status:** Fixed

Description

After the upgrade, delegators withdraw their tokens by calling *undelegate()* and then *withdrawDelegated()* after the thawing period. These operate on the new thaw request lists structure. However, the legacy undelegation model uses the delegator's **tokensLocked** and **tokensLockedUntil** fields. In Horizon, these are never used, meaning that any undelegation request that started and didn't complete before the Horizon upgrade, cannot be completed.

Recommended mitigation

Include the legacy *_withdrawDelegated()* logic in the new *_withdrawDelegated()* function.

Team response

Fixed.

Mitigation review

The issue has been fixed as suggested.

TRST-H-8 Legacy slashing functionality could revert, making it impossible to slash

- **Category:** Underflow issues
- **Source:** HorizonStakingExtension.sol
- **Status:** Fixed

Description

During the fix round, *legacySlash()* was introduced to support legacy slashing during the transition period. It calculates the tokens available to be directly confiscated using the calculation below:

```
uint256 tokensAvailable = indexerStake.tokensStaked -
    indexerStake.__DEPRECATED_tokensAllocated -
    indexerStake.__DEPRECATED_tokensLocked;
```

The issue is that the line can easily overflow if **tokensAllocated + tokensLocked > tokensStaked**, which is possible because allocation can use delegation boost. The current deployed codebase is not affected because *tokensAvailableWithDelegation()* is careful to return zero before causing any overflow.

Recommended mitigation

Check for underflow as done in *tokensAvailableWithDelegation()*.

Team response

Fixed.

Mitigation review

Fixed as recommended.

Medium severity findings

TRST-M-1 Curators are receiving a smaller share of fees than they are expecting due to taxes

- **Category:** Logical flaws
- **Source:** HorizonStakingExtension.sol
- **Status:** Fixed

Description

The documentation of curation fees describes that the curation fee is taken from *all* query fees. However, the behavior seen in HorizonStakingExtension first collects the protocol tax, and only then is the curation fee cut taken.

```
// -- Collect protocol tax --
protocolTax = _collectTax(queryFees,
__DEPRECATED_protocolPercentage);
queryFees = queryFees - protocolTax;
// -- Collect curation fees --
// Only if the subgraph deployment is curated
curationFees = _collectCurationFees(subgraphDeploymentID,
queryFees, __DEPRECATED_curationPercentage);
queryFees = queryFees - curationFees;
```

When the protocol tax is charged earlier, a lower amount remains to be collected, so in effect curators are receiving less than the amount they should be expecting.

Note that the Extension behavior is the same as the behavior currently used by the protocol.

Recommended mitigation

The behavior should be the same as in SubgraphService, where the cut is taken directly from the RAV without applying any tax.

Team response

No action taken.

We're going to keep cuts at the same percentage as in the current protocol. We also updated GraphPayments to calculate curation fees after charging protocol tax.

Mitigation review

The change GraphPayments aligns the protocol tax with the existing method, the team is encouraged to document this behavior in user-facing docs.

TRST-M-2 Pool shares could become inflated, preventing withdrawal of tokens

- **Category:** Amplification attacks
- **Source:** HorizonStaking.sol
- **Status:** Acknowledged

Description

Each pool and provision has its own share/token ratio for both current and thawing tokens. This ratio starts at 1:1 when inserting the first tokens but can fluctuate based on rewards and slashing activity.

An issue can occur if this ratio becomes very large, because legitimate operations would overflow the uint256 data type used for storing the share counts.

Consider the following sequence:

- A pool is being thawed for 1e18 shares, 1e18 tokens.
- While thawed, the pool is slashed down to 1 token, 1e18 shares remain.
- A new thaw for 1e18 tokens is initiated, there are now 1e36+1e18 shares, 1e18 + 1 tokens.
- Again pool is slashed down to 1 token, while ~1e36 shares remain.
- After a few more times, there could be 1e77 shares and just 1 token.

At this point, no one can thaw more than 10 tokens, since 1e78 exceeds uint256. Attacker can perpetually block the pool by never withdrawing their thawed token (last withdraw will reset shares to zero).

The above example is extreme, but there are various ways of getting to a highly inflated share/token ratio which would have severe consequences.

Recommended mitigation

Consider introducing a function to divide both shares and tokens when shares count is dangerous high while tokens is low. This accounting change should consider live thawing requests to ensure there is no corruption.

Team response

No action taken.

Service providers will have to option to use addToDelegationPool if they care to bring their delegation pool to a more balanced state after being slashed. From what we looked there was no possibility of griefing so if this is a result of a service provider being slashed multiple times and they don't want to add tokens to bring it back to an acceptable level then it's their risk.

TRST-M-3 An attacker could make the minimum delegation amount arbitrarily large and prevent competing delegations

- **Category:** Share manipulation attacks
- **Source:** HorizonStaking.sol
- **Status:** Acknowledged

Description

When the first delegation into a pool is made, the share amount minted equals the amount of tokens deposited. Meanwhile, through *addToDelegationPool()*, anyone can donate an arbitrary amount into a delegation pool's token balance, to appreciate the value of shares. The issue is that these functions can be combined to make the minimum delegation amount extremely large. An attacker could do the following:

- Monitor service providers getting created.
- Target a service provider and be the first to mint shares by depositing one wei.
- Use *addToDelegationPool()* to increase the tokens, e.g. by $10,000.
- Their share is now worth ~$10,000. In order for anyone to also delegate to the pool, they must deposit at least $10,000, or they would receive zero shares.

Recommended mitigation

The most robust fix would be to implement a dead share minting model similar to Uniswap. As long as shares amount is always reasonably large, it would be difficult to inflate the pool and sizably change the share value.

Team response

Fixed.

Mitigation review

The code enforces a reasonable minimum delegation in the *delegate()* and *undelegate()* flow, so it would be difficult for an attacker to severely inflate share value. However, note that through slashing, it is still possible for a delegation to have a very low value. This could theoretically be abused through frontrunning a *slash()* call. Consider either accepting the risk, or refactoring the slashing function to address the possibility.

Team Response

We understand and accept the risk. It is challenging to set a minimum delegation amount that also accounts for all thawing tokens. This scenario would arise only in the event of a slashing event where a service provider is slashed by their entire stake and the majority of their delegated stake. In practice, if a service provider is slashed to this extent, it would effectively signal the end of their participation.

TRST-M-4 Insufficient incentives to close overallocated positions may cause unjust gaining of rewards

- **Category:** Game-theory issues
- **Source:** SubgraphService.sol
- **Status:** Acknowledged

Description

When an indexer is overallocated, they are gaining more rewards than they should across their entire allocation list. Since the rewards are coming at the expense of other indexers of the same subgraph ID, these are the only parties that can be expected to “call-out” the overallocation and cancel it. However, any on-chain call costs resources, and the allocations can be engineered to tire out the competition, causing unfair rewarding to accrue.

- Allocations can be continuously overallocated (deposit and thaw some tokens).

- Allocations can be split to many micro-allocations, which multiply the effort of closing overallocations, to the point where it is losing too much value for the caller compared to their lost reward gains.

Recommended mitigation

Introduce some price to be paid by indexers for being overallocated. This needs to be thought through together with H-5 and a new mechanic for predictable allocations may need to be introduced.

Team response

With the changes introduced by the fix to H-05 we accept the risk of unjust rewards being gained as part of the rules of the game. We think the limitations that the POI cycle introduce are good enough to contain the "exploit" to an acceptable level. No further changes will be made to address this issue .

Regarding resource requirements for indexers to force close others' allocations, given this is on Arbitrum One the gas cost should typically be less than the potential rewards difference from closing any decently sized allocation. In the current version of the protocol we already see a similar setup with indexers keeping an eye out for this scenario and force closing allocations without any extra incentive.

TRST-M-5 Lack of chunking functionality of new RAVs may cause them to not be processable

- **Category:** Logical flaws
- **Source:** SubgraphService.sol, TAPCollector.sol
- **Status:** Fixed

Description

A RAV commits on-chain to paying some amount to a provider. RAVs are incremental, so RAVs for 100, 150 and 200 tokens could be consumed to collect 100, 50, 50 respectively. Alternatively just the last RAV could be collected for the entire 200 tokens.

When RAVs are collected, the economic security model forces sufficient tokens to be available for the indexer until the fees are cleared. The larger the RAV, the higher the token requirement.

The issue is that a payer could provide a valid, but very large RAV for some service, if they believe the provider will not be able to cash it out. They will then thaw their tokens from the PaymentsEscrow, putting a timer on how long the provider has to increase their allocation sufficiently to claim the RAV. Also note that a payer could time the RAV after the provider's tokens are already mostly occupied for staking previous RAVs.

Recommended mitigation

Introduce an option to partially collect RAVs, in this way provider knows how much of a specific RAV they would be able to collect on reception.

Team response

Fixed.

Mitigation review

The code now supports incremental processing of RAVs.

TRST-M-6 Allocations could be created in both legacy and Horizon Staking, breaking assumptions

- **Category:** Time-sensitivity issues
- **Source:** SubgraphService.sol
- **Status:** Fixed

Description

According to The Graph, off-chain logic assumes there is only one allocation with a particular **allocationID**. This is also assumed by the RewardsManager, where at least *getRewards()* assumes there is a single allocation with that ID.

The Graph plans to use *migrateLegacyAllocation()* on the SubgraphService to make it impossible to create the same **allocationID** on it. However, that is not sufficient, because an attacker could frontrun the migration call to create a parallel allocation. Even if it were to be executed atomically, attacker could make a large amount of legacy allocations, which can't all be migrated in the same TX because of the gas limit. Therefore, a better way of assuring an allocation is not created is necessary.

Recommended mitigation

When creating a new allocation in SubgraphService, check that the ID is not used in the legacy contract.

Team response

Fixed.

Mitigation review

The issue has been fixed as suggested.

TRST-M-7 The Arbitrator could be spammed with a large amount of disputes without any way to be made whole through fees

- **Category:** DoS attacks
- **Source:** DisputeManager.sol
- **Status:** Fixed

Description

All dispute types except conflicting disputes require some up-front security deposit by the fisherman. Arbitrator's time is valuable and expensive, so any dispute must result in either loss of the deposit or slashing of reasonable amount of stake. However, that is not enforced. A correct dispute could be created for a malicious attestation where the slashable amount is

as little as 1 wei. As a result, an arbitrator's time would be wasted at the cost of locking the deposit amount for the dispute period. Also, through the conflicting dispute route, an attacker could cause DoS of the arbitrator for virtually no cost except the blockchain TX .

Recommended mitigation

When submitting a dispute, the Manager should verify there are enough slashable tokens. Keep in mind that the currently thawing tokens should not be used to check the amount is sufficient, because they cannot be trusted to be available at resolution.

Team response

Fixed.

Mitigation review

The conflicting dispute vector has been mitigated by enforcing a deposit. However, the vector of creating a valid dispute on a near-zero stake has not been addressed.

Team Response

We've decided not to enforce minimum slashable tokens. We believe the attack vector is sufficiently mitigated by:

1 .The significant capital cost required for the attack - each dispute requires locking up the disputeDeposit amount

2 .The ability to adjust the disputeDeposit if needed

3 .The Arbitration Charter can be amended to allow arbitrators to rule against fishermen if a DoS attack is suspected, even for technically valid disputes.

While creating multiple service providers with minimal provisions is theoretically possible, the capital requirements of the disputeDeposit make this attack economically impractical.

TRST-M-8 The DisputeManager could enter a state where the arbitrator cannot abide by the Charter

- **Category:** Logical flaws
- **Source:** DisputeManager.sol
- **Status:** Fixed

Description

Two disputes could be submitted in parallel through *createQueryDisputeConflict()*. They don't require a deposit, because one of the attestations has to be wrong in which case it will be slashed. The issue is that it has not been considered that *both* attestations could be wrong, meaning two valid disputes are raised. There is no reason two wrong and conflicting attestations cannot be submitted. When resolving the disputes, the arbitrator has to accept them by the laws of the Arbitration Charter. However, in *acceptDispute()*, the following lines would reject the other dispute:

```
if (_isDisputeInConflict(dispute)) {
    rejectDispute(dispute.relatedDisputeId);
}
```

Therefore, a logical impasse has been created in the protocol. Also, note that disputing the malicious attestations again would not help, as the rules state only the first dispute for the same attestation could be resolved to a non-draw result.

Recommended mitigation

Allow *acceptDispute()* to accept both disputes.

Team response

Fixed.

Note that the Arbitration Charter will also be amended to ensure that edge cases for conflicting query disputes are correctly addressed.

Mitigation review

In *acceptDispute()* it is now possible to accept both disputes. An issue was introduced such that the same **tokensSlash** value is sent to both the original and conflicting dispute in case they are both accepted. However the disputes could be on different indexers with different profiles, so it should be possible to slash a different amount. For example, an exploit could be to combine two valid disputes, one with a near-zero stake, to limit the slashing on the other dispute.

Team response

A new function for slashing conflict disputes has been added. The new function takes tokensSlashRelated as an additional parameter, allowing arbitrators to configure how much to slash for each dispute.

Mitigation review

Fixed by introducing a function that handles conflicting disputes correctly.

TRST-M-9 Tokens which should be slashed can be removed from the token before the arbitrator's dispute period expires

- **Category:** Logical flaws
- **Source:** SubgraphService.sol
- **Status:** Acknowledged

Description

The dispute period is the amount of time in which the arbitrator has to respond to a dispute. The period is also used as a lower bound for thawing period in SubgraphService provisions. This raises an issue because it doesn't recognize the existence of a window of time between the malicious action to be disputed and when the dispute is actually created. Assume the following scenario:

- A malicious attestation is presented.
- Indexer thaws their provision.
- Ten minutes pass, dispute is recorded on-chain.
- After dispute period passes, if dispute is not resolved, indexer removes their provision. However, arbitrator should still have ten minutes to resolve the dispute.

Recommended mitigation

Introduce a buffer (for example, 48 hours) between the dispute period and the thawing period.

Team response

We acknowledge the finding is valid and could be solved by setting the periods with an offset. However we'll probably set both thawing period and dispute period to 14 days, at least initially. This puts a bit more strain on the human base arbitration process as the time to resolve a dispute is definitely shortened, but we think this is an acceptable tradeoff to have the shortest thawing period possible. Based on our experience with arbitration in the current version of the protocol we don't anticipate this to be an issue, however it's something that we will keep an eye out since we are also going from current 28 days to 14 days.

Also worth noting that it's the fisherman's responsibility to raise the dispute as soon as possible to give the arbitrators enough time. If the fisherman delays too much and the indexer removes their stake, then the fisherman loses out on the reward.

TRST-M-10 A RAV could be collected more than once, leading to double payment

- **Category:** Logical flaws
- **Source:** TAPCollector.sol
- **Status:** Fixed

Description

An interesting property of the RAV model is that it is not explicitly disallowed to pass the same RAV for collection more than once. The **valueAggregate** component of the RAV is a counter for the total amount to be paid for the service. If a RAV is passed again, the TAP collector would supposedly identify that amount has already been collected, and collect a total of zero additional tokens. However, the model falls short in case a payer is switched for some RAV signer. A signer can always authorize a new payer through *authorizeSigner()*. In *collect()*, it looks up the current payer belonging to the signer in the **authorizedSigners** mapping. A new payer will result in a new mapping entry for **tokensCollected**:

```
uint256 tokensAlreadyCollected =
tokensCollected[dataService][receiver][_payer];
```

This will make the collection be completed again from zero, for the new payer:

```
uint256 tokensToCollect = tokensRAV - tokensAlreadyCollected;
```

The impact where a new payer is liable for paying for RAVs already fulfilled is very dangerous in case signer needs to appoint a new payer (for example, if an old payer misbehaves for any

reason). Ordinarily it would be a high-severity finding, but the team stated it would be an unlikely occurrence for users of the Graph platform.

Recommended mitigation

Consider having **tokensCollected** identify the signer, instead of payer, as a mapping entry. Logically the RAV is coupled to a particular signer, so it should be credited to them, regardless of which account actually paid for it.

Team response

Fixed.

Mitigation review

The collector has been simplified and it is no longer possible to authorized a second payer from the same signer.

TRST-M-11 After the transition period, locked amount would still not be available for use

- **Category:** Accounting issues
- **Source:** HorizonStakingBase.sol
- **Status:** Acknowledged

Description

In HorizonStaking, *_getIdleStake()* is used by several functions to calculate the stake available for provisioning, taking into account the old staking state management.

```
function _getIdleStake(address _serviceProvider) internal view
returns (uint256) {
    return
        _serviceProviders[_serviceProvider].tokensStaked -
        _serviceProviders[_serviceProvider].tokensProvisioned -

_serviceProviders[_serviceProvider].__DEPRECATED_tokensAllocated
-


_serviceProviders[_serviceProvider].__DEPRECATED_tokensLocked;
}
```

During the upgrade timeline, after the transition period completes, it is documented that any stake previously locked would be released. However, in *_unstake()* the amount to be unstaked has to be below *_getIdleStake()*, which effectively keeps the tokens locked even after the transition. The intention was that the case below would allow withdrawal of all legacy tokens:

```
uint256 lockingPeriod = __DEPRECATED_thawingPeriod;
if (lockingPeriod == 0) {
    sp.tokensStaked = stakedTokens - _tokens;
    _graphToken().pushTokens(serviceProvider, _tokens);
    emit StakeWithdrawn(serviceProvider, _tokens);
```

As a result, a user would have to wait until the thawing cycle is over and call *withdraw()*, which would clear out the **tokensLocked** and make *unstake()* behave as expected. Also, a user would not be able to use the idle state for provisioning in data services until *withdraw()* would be called.

Recommended mitigation

The getIdleStake() logical should operate on the **__DEPRECATED_thawingPeriod** similarly to the *unstake()* logic.

Team response

Fixed by updating the documentation in the unstake() function. After the transition period any locked tokens will have to be removed by calling withdraw() We acknowledge that this introduces a timing issue where if an indexer unstakes the day before the transition period they'll be subject to thawing and if they would have waited one extra day they could have removed their stake immediately. We'll make sure to communicate this beforehand so indexer can plan accordingly.

TRST-M-12 The operator check in closeAllocations() will not work, indexers must close by themselves

- **Category:** Typo errors
- **Source:** HorizonStakingExtension.sol
- **Status:** Fixed

Description

When closing an allocation, there is a msg.sender check in case it is closing prematurely:

```
bool isIndexerOrOperator = msg.sender == alloc.indexer ||
    isOperator(alloc.indexer, SUBGRAPH_DATA_SERVICE_ADDRESS);
if (epochs <= __DEPRECATED_maxAllocationEpochs || alloc.tokens ==
0) {
    require(isIndexerOrOperator, "!auth");
}
```

```
function isOperator(address operator, address serviceProvider)
public view override returns (bool) {
    return _legacyOperatorAuth[serviceProvider][operator];
}
```

As seen above, the check would look up **_legacyOperatorAuth[indexer, SUBGRAPH_DATA_SERVICE_ADDRESS]**. But that doesn't make sense. The check does not use msg.sender in any way.

Recommended mitigation

Replace with:

isOperator(msg.sender, alloc.indexer)

Team response

Fixed.

Mitigation review

The issue has been fixed as suggested.

TRST-M-13 Legacy vouchers may not be cashed out in time due to no fault of the indexer

- **Category:** Time-sensitivity issues
- **Source:** HorizonStakingExtension.sol
- **Status:** Acknowledged

Description

During the transition period, old allocations can still call *collect()*, and the AllocationExchange may keep minting vouchers for them. However, as documented, after the transition period, the allocations would be closed and the functionality removed. Therefore, there exists a time-sensitive period near the end of the transition period, when the exchange creates vouchers, which may not be cashed out in time.

Recommended mitigation

Introduce a buffer period, only after that time, may allocations be closed and collection officially ends.

Team response

No action taken.

This is an accepted risk. What we've done in the past in a similar case is to make sure the indexing community is well aware of the upgrade, the steps they need to take and the consequences if they don't. The idea of the transition period is to give active indexers enough time to react and close allocations so that they don't lose any rewards or fees. It's possible that some less involved indexers miss the announcements and get left behind, in which case they would have a bigger problem as they would stop getting indexing rewards (because they would need to interact with the new SubgraphService contract).

TRST-M-14 Tokens may not be withdrawable despite passing the thawing period

- **Category:** Logical flaws
- **Source:** HorizonStaking.sol
- **Status:** Acknowledged

Description

Token thawing is implemented by a linked list of thaw requests. At any point the list could be iterated, and any item whose **thawingUntil** has passed, can be released, but when coming across an item that hasn't matured, it stops iterating the list. The approach is safe as long as the list is sorted with respect to **thawingUntil**. The list is sorted because there is a separate list for each provision, *except* when the **thawingPeriod** of that provision changes. When it does, a situation could occur where an item that has already matured is behind an item that

hasn't, effectively making the current **thawingUntil** a lower-bound for all future thaws. Consider the following scenario:

- A data service allows thawing period of at least 14 days and maximum of 90 days.
- A provider starts with 90-day period and thaws a tiny test amount.
- It then changes the period to 14 days and thaws a large amount.
- The large amount will only be withdrawable after 90 days.

Recommended mitigation

The minimal necessary refactor may be introducing a "force traverse" flag which could be passed to the traverse function. If it is on, the traversal would not break out when coming across a non-matured request. There would still need to be careful list linking logic to delink the appropriate items.

Team response

No action taken.

While the idea to have a force traverse could work it would imply that we would have to maintain the linked list which could introduce new issues and increase the complexity of the thaw requests which is already quite high. We'll document this so participants are aware that even if the thawing period changes the thaw requests will still need to be fulfilled in order so they'll have to wait for pending thaw requests to complete even if the new requests have shorter thawing periods.

TRST-M-15 The payer field of the RAV structure is not validated during signature check

- **Category:** Signature issues
- **Source:** HorizonStaking.sol
- **Status:** Fixed

Description

The **EIP712_RAV_TYPEHASH** value has changed during the fix changes in accordance with the new **payer** field of the structure. However, *_encodeRAV()* which should prepare a hash of all the fields, was not updated to consume that value. As a result, a RAV could be submitted with an arbitrary payer value.

Recommended mitigation

Add the **payer** in hash construction.

Team response

Fixed.

Mitigation review

Fixed as suggested.

Low severity findings

TRST-L-1 DataService cannot be initialized using __DataService_init_unchained() unlike stated in docs

- **Category:** Initialization issues
- **Source:** DataService.sol
- **Status:** Fixed

Description

The DataService abstract contract can be used by upgradeable contracts. The documentation states:

```
* - If the data service implementation is upgradeable, it must
initialize the contract via an external
 * initializer function with the `initializer` modifier that
calls {__DataService_init} or
 * {__DataService_init_unchained}. It's recommended the
implementation constructor to also call
```

A developer reading the documentation and initializing the contract with the *init_unchained()* variant will be surprised, as it actually does not perform any initialization.

```
// solhint-disable-next-line func-name-mixedcase
function __DataService_init_unchained() internal onlyInitializing
{}
```

Recommended mitigation

Amend the documentation so that integrators would not suffer from initialization issues, or mirror the logic into *unchained()*.

Team response

Fixed.

Mitigation review

Fixed as suggested.

TRST-L-2 Signer proofs for TAP Collector cannot be revoked

- **Category:** Cryptographic flaws
- **Source:** TAPCollector.sol
- **Status:** Acknowledged

Description

In the TAP Collector, a payer passes a proof provided by the signer that authorizes the payer. The proof includes a deadline, but importantly there is no way to cancel a proof that was already published. That leads to several issues:

- A payer could be revoked but then reuses the proof until the deadline.
- A signer is unable to cancel an already dispatched proof.

Recommended mitigation

The proof should include a nonce value, which the signer can always increment to invalidate proofs.

Team response

No action taken.

- A payer could be revoked but then reuses the proof until the deadline.” - This is fixed by M-10 fix.
- A signer is unable to cancel an already dispatched proof.” - We don’t think that’s a concern. The moment a signer hands over a signed proof they accept the payer will manage their authorization. Also, it's expected that payer and signer are always controlled by the same entity.

TRST-L-3 The getThawedTokens() function could return a wrong amount leading to integration risks

- **Category:** Logical flaws
- **Source:** HorizonStakingBase.sol
- **Status:** Fixed

Description

The *getThawedTokens()* function documents it gets the thawed amount for a provision. It loops over the thaw requests and cashes them out similarly to the *deprovision()* logic.

```
while (thawRequestId != bytes32(0)) {
    ThawRequest storage thawRequest =
_thawRequests[thawRequestId];
    if (thawRequest.thawingUntil <= block.timestamp) {
        tokens += (thawRequest.shares * prov.tokensThawing) /
prov.sharesThawing;
    } else {
        break;
    }
    thawRequestId = thawRequest.next;
}
```

The issue is that the calculation above doesn’t update **tokensThawing** / **sharesThawing** like the real calculation in *_fulfillThawRequest():*

```
if (validThawRequest) {
    tokens = (thawRequest.shares * tokensThawing) /
sharesThawing;
    tokensThawing = tokensThawing - tokens;
    sharesThawing = sharesThawing - thawRequest.shares;
    tokensThawed = tokensThawed + tokens;
}
```

When there are multiple withdrawals, the result may turn out differently as the numbers operated on would be different. Thus, integrations using this function are at risk.

Recommended mitigation

Line up the calculation to what is done in *_fulfillThawRequest()*.

Team response

Fixed.

Mitigation review

Fixed as suggested.

TRST-L-4 The getBalance() function could return a misleading result

- **Category:** Logical flaws
- **Source:** PaymentsEscrow.sol
- **Status:** Fixed

Description

In PaymentEscrow, *getBalance()* can be used to read the remaining available (non-thawing) tokens for some **(payer, collector, receiver)** tuple. However, it does not account for collector approval or thawing period, which makes any integration with the function very risky. Anyone only using *getBalance()* to determine a safe escrow amount, could be wrong when there is insufficient allowance for the collector, there is a collector thawing request, or the collector's thawing period is shorter than token thawing period, in which case the security guarantees are compromised.

Recommended mitigation

Consider refactoring the escrow and collector allowance management so that the API would be safer to use.

Team response

Fixed.

Mitigation review

The issue no longer exists due to refactoring of the PaymentsEscrow.

TRST-L-5 Slashes in SubgraphService may be executed too late if dispute period becomes longer

- **Category:** Time-sensitivity issues
- **Source:** SubgraphService.sol,HorizonStaking.sol
- **Status:** Acknowledged

Description

The Graph governance can change the dispute period maintained in DisputeManager. This is effectively the SLA which it can be expected that the arbitrator will respond by. The value is used to enforce a satisfactory minimal thawing period for SubgraphService provisions. However, if the dispute period is made longer after tokens have already been thawed, the arbitrator might only respond to a dispute after the tokens have already been withdrawn.

Recommended mitigation

Because of the abstract design structure of the staking contract and withdrawal requests not maintaining any state on the request origin, it is difficult to fix the issue without some refactoring. An option is to introduce an optional callback on withdrawal requests, where in this instance when processing the request, it compares in the Subgraph the registered dispute time and the current time, and would delay by the difference. Such a fix would need be done with M-14 in mind in order to not clog the request queue. Another option is to entirely disable increasing the dispute time. Finally, it is also possible to explicitly mandate that dispute time changes are only forward-facing, so it can be expected the arbitrator will rule on the previous disputes in time.

Team response

Acknowledged. This is also the case in the current version of the protocol, we think it's not worth addressing due to the unlikely chance of occurring.

TRST-L-6 Slashing logic is susceptible to front-running

- **Category:** Frontrunning attacks
- **Source:** DisputeManager.sol,HorizonStaking.sol
- **Status:** Acknowledged

Description

It is described in the documentation that slashing may be implemented as a percentage of the snapshotted stake amount in DisputeManager. Note that the snapshot includes thawing tokens, which may be withdrawn by the time *slash()* is called. Also, *slash()* will contain an amount which must be slashed, or the call would revert. These conditions allow for the *slash()* call to be griefed by frontrunning attacks. Suppose the stake to be slashed is 100, out of which 90 is thawing, and the % to slash is 20. Then before *slash(20)*, attacker could withdraw 81 tokens, leaving only 19. This would cause *slash()* to revert and the slasher only losing funds to execute a reverting call. Note that if a subsequent *slash(19)* is called, it too can be frontrun.

Recommended mitigation

The issue seems to point at a more fundamental flaw:

- If indexer's snapshot includes thawing tokens, one cannot be sure these (or % of them) would be available for slashing .
- If indexer's snapshot does not include thawing tokens, indexers could evade slashing by immediately thaw all tokens after their misbehavior and before a dispute is created.

While it will increase complexity, it seems that a long-term solution may need to include higher-grained visibility into the tokens being thawed.

Team response

No action taken. At the moment we only plan on slashing via DisputeManager by specifying the amount of tokens to be slashed. The finding references documentation that is likely outdated. It does raise two important considerations:

- Thawing tokens should be taken into consideration when taking the stake snapshot, otherwise it's trivial to frontrun and avoid the slashing
- It's possible that pre-existing thawing tokens would be withdrawn from the protocol before the slashing happens. Thus we should consider the stake snapshot as a stake cap rather than the actual stake at play.

TRST-L-7 Disputes involving resubmission of a disputed attestation may not be properly resolved

- **Category:** Logical flaws
- **Source:** DisputeManager.sol
- **Status:** Acknowledged

Description

By Arbitration Charter rules, if an attestation is disputed more than once, latter attestations should be drawn. Suppose a bad attestation A is made and disputed. Later, a fisherman provides two conflicting disputes, for A and A', a good attestation. At this point, the arbitrator cannot follow both rules:

- Same attestations should be drawn.
- Disputes of good attestations should be rejected.

This is because by drawing A, A' would also be drawn, and rejecting A' would revert because it verifies any conflicting dispute is handled first.

Recommended mitigation

Consider allowing to draw a single dispute, or reject a dispute despite being in conflict.

Team response

Fixes for M-07 and M-08 would provide an escape hatch for arbitrators by allowing conflicting query disputes to be both drawn as a special case. It's also worth noting that the Arbitration Charter can be amended in cases where edge cases expose conflicting rules such as this one. Arbitrators would typically recommend a change that will be reviewed and approved by governance.

TRST-L-8 Dispute snapshots are inherently unfair

- **Category:** Time-sensitivity issues
- **Source:** DisputeManager.sol
- **Status:** Acknowledged

Description

Whenever a dispute is created, the current provision amount is recorded and may be slashed. However, an indexer could have provisioned funds after the slashable event occurred. This would unfairly cause slashing of an amount unrelated to the problematic event.

Recommended mitigation

The attestation data could include a commitment to a particular staking amount, which would be the amount at risk.

Team response

Acknowledged. If we could easily implement the snapshot at the moment the slashable offense was done we would use that but we don't really have a way to know that and we don't want to have an arbitrary amount added in the attestation. The snapshot serves as helper for arbitrators when setting the slashing value but the human nature of arbitration could mitigate this issue by manually checking the service prov ider's stake at the time of the offense if necessary and setting an appropriate value.

TRST-L-9 Data services could consume the entire slashed amount

- **Category:** Logical flaws
- **Source:** HorizonStaking.sol
- **Status:** Acknowledged

Description

During slashing, a percentage up to **maxVerifierCut** of the slashed amount is sent to the data service, while the rest is burnt. The issue is that it does not cap **maxVerifierCut** accepted by the data service, which could be up to 100%. It allows data services to make use of the Horizon services without paying anything for the slashing service.

Recommended mitigation

Cap the **maxVerifierCut** at the HorizonStaking level.

Team response

For now, we're okay with not setting a maximum amount and allowing data services to define their own cuts, even if the cut is 100%. We might reconsider charging a tax for slashing in the future.

TTRST-L-10 The getBalance() function could revert when balance is lower than tokens thawing

- **Category:** Underflow issues
- **Source:** PaymentsEscrow.sol
- **Status:** Fixed

Description

The *getBalance()* function of PaymentsEscrow returns the available tokens after subtracting the thawing tokens.

```
function getBalance(address payer, address collector, address
receiver) external view override returns (uint256) {
    EscrowAccount storage account =
escrowAccounts[payer][collector][receiver];
    return account.balance - account.tokensThawing;
}
```

However, it is possible that the function would revert when there are tokens being thawed, but the balance has been consumed beyond **tokensThawing**. In this instance, the correct behavior should be to return a negative number (changing the API) or carefully documenting for consumers of the function, that it would revert under certain circumstances.

Recommended mitigation

Consider returning an int256 from the function.

Team response

Fixed.

Mitigation review

Fixed by returning zero instead of reverting.

TRST-L-11 Changes to the SubgraphService configuration could lead to instant loss of rewards for indexers

- **Category:** Logical flaws
- **Source:** SubgraphService.sol
- **Status:** Acknowledged

Description

The SubgraphService exposes several functions for governance, including *setDelegationRatio()* and *setMaxPOIStaleness()*. These immediately impact conditions that are checked in the *forceCloseAllocation()* code path:

- A lower maxPOIStaleness will make *isStale()* be true earlier.
- A lower delegationRatio will make *isOverAllocated()* be true at a lower threshold.

Therefore, parameter changes may well catch indexers of guard and cause closure of their allocation. While that may be important for the economic security of the Graph, it would certainly be unfair to make indexers lose rewards accrued up to this point.

Recommended mitigation

Consider changing these parameters behind a timelock in the SubgraphService contract. Additionally, make closing of allocations not lose historic rewards.

Team response

Fix for H-05 addresses the over allocated case.

For the POI staleness case a configuration change such as this one would be highly discussed in public forums and notified with enough time for indexers to plan accordingly. We have made similar upgrades or configuration changes where indexers were subject to similar "problem" and have had good experiences so far. We acknowledge that this still could result in an unfair loss of rewards for unaware indexers but we think it's not worth the added complexity.

TRST-L-12 The collection cuts could exceed 100% causing collect() to revert

- **Category:** Logical flaws
- **Source: S**ubgraphService.sol
- **Status:** Fixed

Description

When payments are collected through GraphPayment, there are three cuts split off:

- Protocol tax
- Delegator cut
- Curation cut

At any point, these have to be less or equal to the total token amount. The delegator cut is at the control of the provider, curation cut is in control of governance, and protocol tax is immutable. If curation cut is changed to a higher amount, it is possible that *collect()* would suddenly revert. The delegator cut would have to be dialed down and then *collect()* may be called again.

Recommended mitigation

It is recommended to make curation cut fixed or behind a time-lock to not cause any surprises for collection.

Team response

Fixed.

Mitigation review

The cuts are now calculated in a series, so it is not possible for them to exceed 100%.

TRST-L-13 Allocations could be created with a provision below the minimum valid amount

- **Category:** DoS attacks
- **Source:** SubgraphService.sol
- **Status:** Fixed

Description

Several functions in SubgraphService require that the provision is considered valid, which means it has to be above some minimum token amount. This is important because dust provisions are high-risk for DoS attacks, the removal of associated allocations and slashing

rewards would not be sufficient to offset the intrinsic blockchain costs. However, the validity check does not take into account the tokens being thawed from the provision.

```
function _checkProvisionTokens(IHorizonStaking.Provision memory
_provision) internal view virtual {
    _checkValueInRange(_provision.tokens, minimumProvisionTokens,
maximumProvisionTokens, "tokens");
}
```

As a result, a provision could pass the check despite most of its tokens being immediately withdrawable.

Recommended mitigation

Deduct the tokens being thawed for validity checks.

Team response

Fixed.

Mitigation review

Fixed as suggested.

TRST-L-14 Unexpected underflow in _getIdleStake()

- **Category:** Underflow issues
- **Source:** HorizonStakingBase.sol
- **Status:** Fixed

Description

In the legacy allocation tracking, it is possible for **tokensStaked** to be lower than **tokensAllocated**, because allocations can leverage delegation power. However, *_getIdleStake()* does not check for underflow before subtracting them. Therefore, any users of *_getIdleStake()* can unexpectedly revert. Fortunately it does not seem like it is blocking critical functionality, because the callers in HorizonStaking would not be able to provision or unstake regardless of the underflow. Therefore it is mostly a convenience and view-related issue.

Recommended mitigation

Return zero instead of underflowing in *_getIdleStake()*.

Team response

Fixed.

Mitigation review

Fixed as suggested.

Client-reported findings

TRST-CL-1 A payer could bypass the escrow mechanism and avoid payments through the vulnerable collector allowance mapping

- **Category:** Logical flaws
- **Source:** PaymentsEscrow.sol
- **Status:** Fixed

Description

The Escrow manages two trust structures:

- An escrow balance, mapped by a **(payer,collector,receiver)** tuple.
- A collector allowance, mapped by a **(payer,collector)** tuple.

When collection happens, both variables are reduced by the paid amount. At the same time, a receiver should trust that when their private escrow balance covers a payment and there is sufficient collector allowance by the payer, they will be able to collect the funds.

However, there is a problematic coupling of the two structures – a collector's allowance is shared across all potential receivers, including untrusted ones. Therefore, a malicious payer could immediately nullify their allowance by signing a RAV which would be paying themselves.

Recommended mitigation

The collector allowance should have another mapping layer to mirror the escrow balance mapping. In this way, each allowance is isolated.

Team response

Fixed.

Mitigation review

The Escrow has been refactored, there is no longer a collector mapping and therefore the attack is mitigated.

Additional recommendations

TRST-R-1 Improve event structure

The DelegationSlashingEnabled event accepts a bool, however it is always called with **true**. Consider removing the superfluous parameter.

TRST-R-2 Improve interoperability with smart wallets

In TAPCollector, it uses ECDSA.recover() to extract the signer from the signature. However, this only works for EOA accounts. Consider using *isValidSignatureNow()* from Open Zeppelin so that smart accounts can also sign RAVs.

TRST-R-3 Verify state transitions

In several locations in the codebase, a bool configuration parameter is changed (for example in DataServicePausable, the *_setPauseGuardian()* changes the guardian status). Whenever such changes are done, it is recommended that the function call acts as a switch, to avoid any errors. Consider checking the state is opposite of the requested state.

The recommendation also applies to functions like *thawSigner()* which have a hidden assumption the signer is not thawed yet.

TRST-R-4 Lack of storage slots in upgradeable contracts

DataServicePausableUpgradeable is an upgradeable contract, however it does not contain any gap slots to allow for future extension.

TRST-R-5 Equivalent contracts behave differently

One would expect DataServicePausable and DataServicePausableUpgradeable to be equivalent except the upgradeable structures. However, the upgradeable version disables setting of a pause guardian when the protocol is paused. Consider removing that restriction or introducing it in the non-upgradeable version.

TRST-R-6 Protect from developer errors by improving docs of inheriting contracts

The docs in DataService make it clear that *DataService_init()* or another variant must be called on initialization or construction. However, there are several extensions that inherit from DataService, like DataServicePausable and DataServiceRescuable. These don't initialize the base class by default, so it is important to also document here that it needs to be done.

TRST-R-7 Thawing shares should be rounded up to protect from early unlocking of tokens

In *_thaw()* and *_undelegate()*, it calculates thawing shares and increases the shares and token thawing counts. In fact, since it is rounding down the created shares, but adds the original amount of tokens, it is increasing the value of previously minted shares. Essentially it is rounding in favor of the user in respect to locking time, since more value would be withdrawn earlier.

On the other hand, in the delegation pool, rounding up the share count risks taking more tokens from other pool thawers, so it seems unfavorable impact would occur in either case. It would need to be carefully thought through.

TRST-R-8 Standardize the checks made in the list iteration functions

The codebase uses two functions for getting the next linked list item:

- *getNextStakeClaim()*
- *getNextThawRequest()*

The inconsistent behavior is that the first one checks the current item is valid (non-zero) while the second one doesn't. The list traversal logic ensures the function would only be called for non-zero items, so it seems redundant.

TRST-R-9 Documentation errors

- *getAllocationData()* in SubgraphService doesn't document the pending rewards return value.
- In the DisputeManager header, the "Indexing Disputes" section is outdated.
- The HorizonStakingExtension documents that the contract can be removed, however that is not accurate since there are some functions that are used by the RewardManager. Only some of the functions in the Extension can be removed.
- In HorizonStaking, *_createProvision()* docs refer to *createProvision()*, but that function doesn't exist. The intention is to refer to the *provision()* function.

TRST-R-10 Verify addresses are non-zero before use

There are some locations in the codebase where an address is called without checking it is non-zero, for example in the DisputeManager it uses the **subgraphService** without checks. The value will only be non-zero if initialized before use of the functions, but that is not guaranteed since the initialization doesn't set it.

TRST-R-11 Dispute resolution behavior is arbitrary after disputePeriod elapses

After the dispute period, the resolution functions can still be called, however the fisherman can call *cancelDispute()* as well. This means any resolution except cancellation should not be counted on. It is in fact a cleaner practice to make only *cancellation* possible after the dispute period. This way, behavior is more predictable and well-defined.

TRST-R-12 Apply a better sender check in L2Curation

The L2Curation contract has been modified to support multiple callers. In its *collect()* logic, caller can be either **subgraphService** or the legacy staking contract. However, the check could be more precise. During the transition period, either contracts can call it, but after the transition, only **subgraphService** is a valid caller. Also, before the transition period only the staking contract should be able to make a call.

TRST-R-13 Service providers can avoid sharing already accrued fees with delegators

Same as before the upgrade, there is a known attack where providers can avoid sharing fees with delegators by setting the delegator cut to 0% before calling *collect()*, and restoring it after the call. Despite being a known issue, it will be hard to detect, and also affects any data service in the new model. It merits a mention and consideration whether it is best to only acknowledge the issue.

TRST-R-14 Verify requested thawing shares is not zero

When calling *_createThawRequest()*, it should never create a request with zero shares. Consider adding a check to reduce likelihood of users burning tokens without creating shares.

TRST-R-15 Include requestType in thawing events to avoid ambiguity

Since the code update, there are now three kinds of request types for any given (provider, verifier, owner) mapping. However, ThawRequestsFulfilled event lacks context for which type of request was fulfilled.

Centralization risks

TRST-CR-1 The Graph Governance is trusted

Because of the upgradeable nature of the contracts, it is clear the Graph Governance must be trusted or else any behavior including the compromise of funds can occur. Any inspection of the particular privileges of Governance in the current codebase is therefore redundant.

Team Response

Acknowledged and agreed. The role of governance is a salient point in Horizon and is discussed in GIP-0066. Given the nature and complexity of the changes we are introducing we chose to go with mostly upgradeable contracts. The plan is to give up upgradeability over time as contracts are battle tested as we've done in the past for example with L2GraphToken contract in Arbitrum One. The form that will take will depend on how the protocol evolves, but likely dropping upgradeability entirely and/or using timelocks or DAO voting as mechanisms are things that we will explore.

TRST-CR-2 The Arbitrator is trusted

The arbitrator resolves disputes using off-chain logic, and therefore must be trusted by users of the SubgraphService. Up to the entire provision and delegation amounts can be slashed.

Team Response

Unfortunately at this point trusted human arbitrators is a requirement for the Subgraph Service dispute mechanism. We have ongoing initiatives that attempt to remove or minimize the role of a centralized authority for arbitration but those are on early stages of research.

Systemic risks

TRST-SR-1 Some off-chain trust is required in the escrow mechanism

The Graph payment escrow facilitates a trustless way of cashing out service fees in exchange for a data service potentially including attestations. However, it needs to be clear that it does not fully mitigation trust assumptions. Some off-chain mechanism must ensure that the exchange between a valid payment voucher and a slashable attestation is safe. If one of the sides receives the other's input before sending their own, the on-chain mechanism would not protect from malicious use of resources or funds by one of the parties.

Team Response

We usually refer to Graph Payments and specially TAP as trust minimized systems as opposed to trustless. We agree that is a very important distinction to be made and will ensure going forward this is not mischaracterized.

TRST-SR-2 Reward dilution in case not all legacy allocations are closed

After the transition period, all old allocations should be closed, however it is not clear this could be done because there is a theoretically uncapped amount of allocations that could be created before that which would each need to be closed individually, which would cost very high expenses in gas. In case any old allocation is not closed, it will dilute rewards for the registered subgraph for legitimate allocations.

Team Response

Noted. We think the likelihood of a dilution attack like this to happen is negligible considering the required cost of capital of the stake that is left stuck. In order to make a significant dilution the amount of stake involved in the attack would have to be of comparable size to the rest of the productive stake.

TRST-SR-3 Overallocated indexers cannot be closed, incentivizing dishonest behavior

Following the code refactor to address other issues, it is now impossible to force close an allocation when it is overallocated. This allows gaining up to an entire-epoch's worth of allocation rewards while they are not allocated (e.g. in the unfreezing period). The risk is acknowledged by The Graph and is no worse than the currently deployed and operational version of the protocol.

Team Response

We acknowledge the issue of overallocated indexers gaining undeserved rewards. However, this is bounded by the POI cycle and delegation thawing. When collecting rewards, if the indexer is overallocated, the allocation will automatically be closed. At most, the indexer can

gain one POI cycle of undeserved rewards, which is consistent with the current version of the protocol, though the cycle will be shorter in Horizon.