OpenZeppelin | security

Exponential Query Fee Rebates Audit

The Graph

June 05, 2023

This security assessment was prepared by OpenZeppelin.

Table of Contents

Summary

Type	DeFi
Timeline	From 2023-05-08 To 2023-05-19
Languages	Solidity

Total Issues	13 (10 resolved, 2 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	1 (1 resolved)
Medium Severity Issues	3 (1 resolved, 1 partially resolved)
Low Severity Issues	2 (1 resolved, 1 partially resolved)
Notes & Additional Information	7 (7 resolved)

Scope

We audited pull request #824 at commit 49cfd5.

In scope were the following contracts:

```
contracts
├── discovery
|   └── GNSStorage.sol
└── staking
    ├── IStakingBase.sol
    ├── IStakingData.sol
    ├── IStakingExtension.sol
    ├── IStaking.sol
    ├── StakingExtension.sol
    ├── Staking.sol
    ├── StakingStorage.sol
    └── libs
        ├── Cobbs.sol
        ├── Exponential.sol
        └── Rebates.sol
```

Overview of Changes

We audited a new mechanism for The Graph to distribute query fee rebates to indexers that serve queries.

Query fees are a form of compensation that is paid by consumers to the protocol for processing their queries. A query fee rebate is a mechanism that returns a portion of these query fees back to the indexers, who are required to stake GRT into subgraphs in order to collect these rebates.

The amount of the rebate rewards that an indexer receives is currently determined by a mechanism that uses the Cobb-Douglas Production Function. The rebates that are collected by an indexer are influenced by the collective GRT allocations for all indexers whose allocations closed at the same epoch. This makes it difficult for indexers to predict their potential earnings, which in turn can lead to query fees being burned at a higher rate.

The proposed change from the Cobb-Douglas Production Function to an exponential rebate function aims to simplify this mechanism. The percentage of rebates that an indexer can collect would depend solely on the ratio between an indexer's own stake and the query fees that they have generated. This would make it easier for indexers to understand and predict their potential earnings, and it would also reduce the overall amount of query fees that are burned by the protocol.

Security and Deployment Considerations

Changing from the Cobb-Douglas function to exponential rebates is not a security concern as it is only modifying how the rebates are calculated. However, changing the flow of how indexers collect their rebates and modifying the state transition of allocations is a significant change that requires special attention.

Because of the arithmetic change and removal of the rebate pools, this upgrade will not be backward compatible. Once the upgrade is completed, indexers will not be able to claim rebates from old rebate pools.

The upgrade based on the changeset in the audited pull request has the following considerations:

- Open pools with pending rebates will not be claimable by indexers. After the upgrade, indexers will forfeit any fees that have not been claimed from these pools.
- During the upgrade, there will be a specific time window when indexers will not be able to claim collected fees. The rebate fees can only be claimed from a rebate pool once the corresponding allocation has reached the Finalized state, which currently takes approximately 7 days after the allocation is closed. However, before the upgrade, The Graph's team plans to reduce this timeframe to just 1 day.
- Re-staking behavior will change. After the upgrade, indexers will need to be aware that rebated fees for indexers will be automatically re-staked unless there is a `rewardsDestination` set.

High Severity

H-01 Allocations that had a `Claimed` state will now appear with an `Active` state

As part of the rebate pool logic, there were two additional states that an allocation could be in: `Claimed` and `Finalized`. `Claimed` was the final state for an allocation that had been closed and had collected its query fees via the `_claim` function. This was accomplished by setting the `createdAtEpoch` and `closedAtEpoch` values for an allocation to zero in the `_claim` function, which would in turn cause the `_getAllocationState` function to return the `Claimed` allocation state. Now that the `Claimed` and `Finalized` states have been removed as part of the exponential rebates changeset, allocations whose state was `Claimed` will now return the `Active` state when calling the `_getAllocationState` function.

This will allow an indexer or operator to call the `_closeAllocation` function on any allocation whose previous state was `Claimed`. No additional rewards will be distributed due to the `alloc.tokens` being set to 0, but sending an additional `poi` for an allocation that had already been closed could cause an accidental dispute and subsequent slash.

Consider adding the `Claimed` and `Finalized` states back into the `AllocationState` enum and clearly indicating that these states are deprecated for new allocations. This will ensure that the original states for allocations that were claimed before the upgrade do not change after the exponential rebates upgrade.

Update: *Resolved in pull request #835. Note that this will still change the state of `Claimed` allocations to `Closed`. However, this does not appear to cause undefined behavior within the contract. The Graph team stated:*

> *We discussed the proposed solution internally, but ultimately decided against it and came up with an alternative approach that does not require keeping old/unused code in the contract (allocation states and `getAllocationState` logic) and does not require making an expensive call to epoch manager to check for the `Finalized` state. We think the fix proposed in the linked PR also addresses the issue while keeping the previous bullet points true. Additionally, we are adding upgrade tests with pull request #838 that help validate state transitions.*

Medium Severity

M-01 Problematic edge cases on `collect` function with multiple gateways

In the `Staking` contract, the `collect` function allows indexers to claim their rebates from an allocation. This function is called by the `AllocationExchange` contract.

If The Graph adds support for multiple gateways, an indexer could generate query fees for a given subgraph on such multiple gateways. When redeeming a voucher from a gateway, the query fee value that is being collected may not represent the total amount of fees that have been generated. This would mean that an indexer could redeem multiple vouchers for the same allocation. In that case, we have identified two edge cases in the current implementation:

- In the scenario where multiple vouchers are issued for a single indexer allocation and all the vouchers are emitted synchronously: the total amount of rebates collected may not align with what the indexer should receive because there may be more drift in the actual rebated amount than expected. This would be particularly true if `alpha` and `lambda` are modified between multiple collect calls.
- Because the protocol and curator fees paid by the indexer are not tracked in the data structure for an allocation, it may be problematic to generate vouchers sequentially (only after the first one has been collected). The problem would arise if voucher emitters need to know the total amount of query rebates that an indexer has already collected. Currently, the indexer-collected rebates are stored in `alloc.distributedRebates`, which does not take into account any protocol or curator fees that have been paid. In the case that a voucher is generated after a separate one has been collected, the second voucher could be emitted for the remainder of the query fees that the indexer is entitled to, but it would also add the fees that have already been paid. If this is the case, indexers would be overcompensated as they would only effectively pay their fees on the last voucher they claim for a given allocation.

Consider revising the validations in the `collect` function so that the logic tracks the total amount of query rebates that an indexer is entitled to in an allocation, and ensures that an indexer gets that total amount, independently of the number of vouchers it receives or the order in which they are redeemed.

Update: *Resolved. The Graph team stated:*

> *Initially we did not provide enough context about how vouchers work for the audit team to make the correct assumptions about vouchers. This was rectified via Slack and after discussing with the OZ team we agreed that these edge cases are not possible with the current protocol design. We do take note however for future consideration when thinking/designing the decentralized gateway solution. For reference, here is some additional context that we shared:*
>
> *Indexers get receipts for each query they serve, they then exchange N receipts for a voucher that allows the indexer to get paid an amount. The gateway signs the voucher and marks the receipts as redeemed. So receipts (and vouchers, by extension) are completely independent from each other, and the gateway does not care whether the voucher was actually collected on-chain. As far as the gateway is concerned once they hand out the voucher it is already collected and they will not issue another one for those same receipts.*

M-02 Lack of test coverage

The latest changes have reduced the code coverage of the test suite. There is also a lack of tests that cover possible state transitions during the upgrade period. This is a major concern as the rebates upgrade involves multiple contracts and state transitions that can not be easily simulated in a local environment. It is crucial that the upgrade is thoughtfully tested and the largest amount of edge cases and state transitions are covered to ensure the upgrade will not behave unexpectedly.

Consider adding extensive tests to increase code coverage and simulating it on a mainnet fork before deploying to the mainnet. In particular:

- Use tooling and tests to ensure that there are no storage collisions resulting from the upgrade.
- Test all the possible intermediate states for allocations during and after the upgrade window.
- Test that the upgrade can be executed on a mainnet fork without issues while simulating normal allocation activity.

Update: *Partially resolved. The Graph team stated:*

> *Regarding the code coverage's percentage reduction, I am not 100% sure what happened, but the patch coverage percentage is 100% (so any changes introduced in the pull request are covered). My best guess is that the overall lines of code count went down, so previously uncovered code paths now account for a higher percentage and*

> *that is causing the code coverage reduction. Additionally, the linked pull request introduces a few scripts and utilities that allow us to easily fork from the mainnet, run a test suite on the protocol before upgrading, perform the upgrade and then run another set of tests on the upgraded protocol. This already allows testing a few cases like allocation state transitions, testing the upgrade on a mainnet fork and more. We will probably keep adding small tests as we come up with them.*

M-03 Decreasing the `__channelDisputeEpochs` increases the likelihood of accidentally burning 100% of the query fees for an allocation pre-upgrade

As part of the upgrade process for exponential rebates, The Graph intends to decrease the `__channelDisputeEpochs` from ~7 days to ~1 day. This increases the chance that an indexer who is in the process of closing their allocation and collecting their query fees accidentally burns 100% of the query fees. This is because in the rebate pool version of the `collect` function, if the state of the given allocation is `Finalized` or `Claimed`, then `MAX_PPM`, or 100% of the query fees will be burned as part of the protocol tax. Note that the `Finalized` allocation state is returned by the `_getAllocationState` function when the difference between the `closedAtEpoch` and the current epoch is greater than or equal to `__channelDisputeEpochs`.

Therefore, when The Graph decreases the value of `__channelDisputeEpochs` before the upgrade, they also reduce the time it takes before an allocation enters the `Finalized` state by approximately 6 days. If an indexer is not paying attention to the timing of their `closeAllocation` and `collect` calls, they could accidentally call `collect` and have all their query fees burned.

The Graph should communicate to indexers that they should only close allocations during the upgrade window that would otherwise exceed the `__maxAllocationEpochs` duration, to avoid force-closing. Additionally, The Graph should consider not issuing rebate vouchers in advance of the upgrade and through the upgrade window to further minimize the risk of accidental loss of query fees (due to either burning or the upgrade itself).

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

> *This is certainly a concern we had and discussed with indexers but it's highly unlikely that it happens because of how the indexer software works. To the best of our knowledge, there are no indexers interacting directly with the smart contracts, they use*

> *the "indexer agent" to do it. The indexer agent will automatically call* `AllocationExchange.redeem()` *(which calls* `Staking.collect()`*) whenever an indexer closes an allocation. So it is safe to assume that most if not all of the query fees are collected into rebate pools right after allocations are closed, leaving no chance of the fees being burnt. The only cases where this would not happen is if it doesn't make economic sense to collect the query fees (if they are worth less than what the tx costs). Despite this we have been communicating with the indexer community through various channels, raising awareness of the upgrade and what the consequences for them are. We plan on announcing the upgrade with enough time for the community to react.*

Low Severity

L-01 Exponential rebates arithmetic does not support business logic when `alpha` is not 1

In the exponential rebates calculation, it is assumed that when either `stake`, `fees`, or `lambda` are equal to 0, the full calculation can be short-circuited to return 0 query fees for the given allocation. When `fees` is 0, this calculation can be correctly short-circuited as taking a percentage of 0 will always return 0. However, if either `stake` or `lambda` are equal to 0, the exponential rebates arithmetic does not always support the claim that the query fees returned will also be 0.

When an integer's power is 0, the resulting value will always be 1, so when taking the power of `e`, we get `e^0 = 1`. When `alpha` is set to 1, we know that the resulting percentage of query fees to distribute will be `1 - 1 * e^0 = 0`. However, when `alpha` is not equal to 1 the arithmetic will result in `1 - alpha * e^0`, which means the percentage of query fees to distribute will be the result of `1 - alpha`. Therefore, while the business logic for returning 0 query fees when either `stake` or `lambda` are 0 makes sense, this logic should not be baked into the arithmetic as it does not always singlehandedly support this conclusion.

Consider moving the business logic that short-circuits the exponential rebates arithmetic to return 0 query fees when either `stake` or `lambda` are 0 out of the `exponentialRebates` function and into the `collect` function.

Update: *Resolved in pull request #840. The Graph updated* `LibFixedMath` *to short-circuit in the* `_mul` *function when either input passed-in is zero. This change is unrelated to the finding. The team stated:*

> *The pull request solves this issue. Note that we cannot add tests for the condition `lambda=0` on `Staking.collect()`. We short-circuit the calculation just in case but the current implementation will not allow lambda to be set to zero. The pull request also fixes a bug found in `LibFixedMath`.*

L-02 `getRewards` will always return a non-zero value for rewards even when an allocation is closed

As part of removing the `claim` function and its associated logic, `alloc.tokens` will never be reset to zero (unless the allocation was for zero tokens). As a result, the `getRewards` function will always return a non-zero value, even when the state of the corresponding allocation is `Closed`. This behavior differs from the previous logic, which would set `alloc.tokens` to zero once an allocation had claimed their query rewards. Therefore, this change may confuse indexers who are expecting the previous behavior of the `getRewards` function.

Consider adding a check for the allocation state in the `getRewards` function, and if the state is `Closed` then return zero.

Update: *Partially resolved in pull request #841. The Graph added a check to ensure the `getRewards` function only returns a non-zero value if the allocation state of the passed-in `_allocationID` is `Active`.*

Notes & Additional Information

N-01 Adding a new member to the `Allocation` struct could cause a memory collision

In the `IStakingData.sol` `Allocation` struct, a new member called `distributedRebates` has been added.

Due to their unpredictable size, mappings in Solidity cannot be stored between the state variables preceding and following them. A single 32 bytes slot is used for the mapping, and the

elements that it contains are stored starting at a different storage slot that is computed using a Keccak-256 hash of the key.

Because of this, adding a new member to the `Allocation` struct has a non-zero chance of causing a collision.

While it is almost impossible for this to happen, a storage collision could be catastrophic for the state of the contract. So consider testing in a testnet or a fork environment for any storage collision occurrences.

Update: *Resolved. The Graph team stated:*

> *Noted, though we believe the probability of a storage collision happening because of the inclusion of the new allocation struct member is negligible. Since the size of the stuct is only 9 slots the probability of a collision is almost the same as the probability of finding a keccak256 collision. Adding one field means increasing the struct from 8 to 9 slots, which we think changes the size of the hash space from 253 bits to 252 bits, which we still consider cryptographically secure.*

N-02 `require` statement with multiple conditions

Within `Staking.sol` there is a `require` statement on line 656 that requires multiple conditions to be satisfied.

To simplify the codebase and raise the most helpful error messages for failing `require` statements, consider having a single require statement per condition.

Update: *Resolved in pull request #842 at commit 06a8c15.*

N-03 Missing docstrings

Line 7 in `Exponential.sol` is missing docstrings.

Consider thoroughly documenting all functions (and their parameters), contracts, and libraries. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Resolved in pull request #844 at commit #aaa7f98.*

N-04 Lack of indexed event parameter

Within `IStakingBase.sol`, the `ExtensionImplementationSet` event does not have its parameters indexed.

Consider indexing event parameters to improve the ability of off-chain services to search for and filter for specific events.

Update: *Resolved in pull request #842 at commit 7471ab3.*

N-05 Non-explicit import

The use of non-explicit imports in the codebase can decrease the clarity of the code and may create naming conflicts between locally defined and imported variables. This is particularly relevant when multiple contracts exist within the same Solidity files or when inheritance chains are long.

In `Exponential.sol`, line 5 has a global import.

Following the principle that clearer code is better code, consider using named import syntax (`import {A, B, C} from "X"`) to explicitly declare which contracts are being imported.

Update: *Resolved in pull request #842 at commit b334198.*

N-06 Unused imports

There are imports in `IStaking.sol` that are unused and could be removed. For instance:

- Import `Stakes`
- Import `IStakingData`

Consider removing unused imports to improve the overall clarity and readability of the codebase.

Update: *Resolved in pull request #842 at commit 0f7f04b.*

N-07 Incorrect documentation

The following instances of incorrect documentation have been identified:

- The comment on line 18 in the `Exponential` library explaining the exponential rebates equation is incorrect. The exponent value should be `-lambda * stake / fees` and not `-lambda * fees / stake`.
- The comment on line 211 in `IStakingBase` is missing the parameters `_lambdaNumerator` and `_lambdaDenominator`.

Consider resolving these instances of incorrect documentation to improve the clarity and readability of the codebase.

Update: *Resolved in pull request #842 at commit #0271bc2.*

Conclusions

One high-severity and two medium-severity issues were identified during this audit. These issues stem from the complexity of upgrading the protocol while accounting for the state transitions of the existing rebate mechanism.

Appendix

Monitoring Recommendations

While audits help in identifying code-level issues in the current implementation and potentially the code deployed to production, The Graph's team is encouraged to consider incorporating monitoring activities in the production environment.

Ongoing monitoring of deployed contracts helps in identifying potential threats and issues that may affect the protocol. It can also help monitor the status of the upgrade. For this reason, we think that monitoring the following may be beneficial:

- Rebates collected by indexers: this can be done by monitoring `RebateCollected` which includes useful information regarding the amounts of tokens collected, protocol tax and curation fees. By monitoring this event, The Graph's team can ensure that the upgrade is working as expected and that the new rebate mechanism is working as intended.
- The balance of the staking contract should be the same after a `collect`: all the GRT tokens pulled to the `Staking` contract during a `collect` call should be either distributed as rewards or burnt, so the balance of the Staking contract should be approximately the same (with only slight variations due to rounding error) as it was before a `collect` call.
- Invalid state transitions of previous allocations: this can be done by checking for duplicate `allocationIDs` in the `AllocationClosed` event. Having a duplicated allocation ID could potentially indicate a regression of H-01.