OpenZeppelin | security

The Graph Staking and Vesting L2 Migration

The Graph

May 05, 2023

This security assessment was prepared by OpenZeppelin.

Table of Contents

Summary

Type	DeFi, L1/L2 Bridge
Timeline	From 2023-03-06 To 2023-03-31
Languages	Solidity

Total Issues	37 (21 resolved, 3 partially resolved)
Critical Severity Issues	1 (1 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	9 (6 resolved, 2 partially resolved)
Low Severity Issues	9 (2 resolved)
Notes & Additional Information	18 (12 resolved, 1 partially resolved)

Scope

We audited the following repositories:

- The graphprotocol/contracts repository at the `83fffa32a908ceba4aceaea29bba1f83ea25a462` commit of pull request #786.
- The graphprotocol/token-distribution repository at the `1dc1eb230189343a5c6509d844e78bf8da851a23` commit of pull request #64.

In scope were the following contracts:

```
https://github.com/graphprotocol/contracts:
└── contracts
    ├── staking
    │   ├── Staking.sol
    │   ├── StakingExtension.sol
    │   ├── IStakingBase.sol
    │   ├── L1Staking.sol
    │   ├── IStakingExtension.sol
    │   ├── IL1StakingBase.sol
    │   ├── IStaking.sol
    │   ├── StakingStorage.sol
    │   ├── IL1GraphTokenLockMigrator.sol
    │   ├── L1StakingStorage.sol
    │   ├── IL1Staking.sol
    │   └── IStakingData.sol
    ├── l2
    │   └── staking
    │       ├── L2Staking.sol
    │       ├── IL2Staking.sol
    │       └── IL2StakingBase.sol

https://github.com/graphprotocol/token-distribution:
└── contracts
    ├── L1GraphTokenLockMigrator.sol
    ├── L2GraphTokenLockManager.sol
    ├── L2GraphTokenLockMigrator.sol
    ├── arbitrum
    │   └── ITokenGateway.sol
    ├── L2GraphTokenLockWallet.sol
    ├── ICallhookReceiver.sol
    ├── MinimalProxyFactory.sol
    ├── GraphTokenLockWallet.sol
    └── GraphTokenLockManager.sol
```

Overview of Changes

We conducted an audit of the code in pull request #786, which expands upon the previous GNS and Curation migration audit by introducing migration helpers for GRT staking and delegation to L2. Throughout this report, pull request #786 and its associated codebase will be referred to as "Staking." Additionally, the code in pull request #64 facilitates the migration of vesting GRT Lock Wallets, which manage a significant portion of GRT used in staking and delegation. Pull request #64 and its codebase will be referenced as "Vesting" for the remainder of this report. The "Staking" and "Vesting" modifications are coupled to enable the migration of locked funds in a manner that restricts GRT transfers from escaping their vesting contract, while ensuring the funds can both participate in staking and delegation as well as migrate to L2 with as little friction as possible.

The staking contracts were refactored to keep their size within the mainnet's limit while adding the new functionality needed for migration. Some functions have been moved to a `StakingExtension` contract, while others have been removed. The `StakingExtension` contract is invoked via a chained proxy pattern using a `fallback` in the Staking contract.

The staking migration from L1 to L2 is done using `migrateStakeToL2`, which allows indexers to migrate their stake to L2 by specifying an L2 beneficiary address. On L2, the upgraded `L2Staking` contract receives and processes the stake migration in its `onTokenTransfer` token gateway callhook. Partial migration of indexing stake is allowed as long as the minimum amount of stake is sent in the first migration transaction, and as long as the remaining indexing stake left on L1 for an indexer satisfies the minimum stake amount.

Delegation migration from L1 to L2 is done using `migrateDelegationToL2`, which allows delegators to migrate their delegated GRT to their counterpart indexer address on L2. The `L2Staking` contract receives and processes the delegation migration in its `onTokenTransfer` token gateway callhook. Partial delegation is allowed, and an additional `unlockDelegationToMigratedIndexer` function is provided in `L1Staking` which allows delegators that do not wish to migrate their delegation to L2 to avoid the unbonding delay and withdraw as soon as an indexer fully migrates to L2.

Locked stake and delegation that belong to a vesting contract are migrated using the `migrateLockedStakeToL2` and `migrateLockedDelegationToL2` functions, respectively. These functions are designed to be invoked by the vesting lock wallets, which do not have the ability to pass the ETH that is required for cross-chain transactions. These functions query the `L1GraphTokenLockMigrator` in Vesting to obtain the L2 vesting

contract address for the caller and perform the same migration process described earlier, but with the L2 vesting contract as the beneficiary.

The reliance on `L1GraphTokenLockMigrator` for the locked stake and delegation migration requires a prior migration of the vesting lock, which is done using the `L1GraphTokenLockMigrator` contract by calling the `depositToL2Locked` function. Additionally, vesting contract beneficiaries that wish to migrate to L2 must deposit ETH into the `L1GraphTokenLockMigrator` for the gas payment required for all the mentioned cross-chain transactions. Optionally, they can withdraw any remaining ETH from the `L1GraphTokenLockMigrator`. To allow moving locked funds between the two vesting lock counterparts, `L2GraphTokenLockMigrator` allows sending GRT back to the L1 counterpart via `withdrawToL1Locked`.

To allow cross-chain message passing, `L1Staking` and `L1GraphTokenLockMigrator` will be added to the callhook allowlist in the `L1GraphTokenGateway`.

Privileged Roles

- The Governor:
  - Can `setExtensionImpl` on the `L1Staking` contract (inherited from the `Staking` contract).
  - Can `setCounterpartStakingAddress` on the `L1Staking` contract (inherited from the `Staking` contract).
  - Can `setL1GraphTokenLockMigrator` on the `L1Staking` contract.
- The GraphTokenLockManager owner:
  - Can `setL2LockManager` on the `L1GraphTokenLockMigrator` contract.
  - Can `setL2WalletOwner` on the `L1GraphTokenLockMigrator` contract.
- Indexers:
  - Can `migrateStakeToL2` from the `L1Staking` contract.
- Delegators:
  - Can `migrateDelegationToL2` from the `L1Staking` contract.
  - Can `unlockDelegationToMigratedIndexer` from the `L1Staking` contract.

- Vesting lock beneficiaries (through the wallet lock contract `fallback`):
  - Can `withdrawETH` from the `L1GraphTokenLockMigrator` contract.
  - Can `depositToL2Locked` from the `L1GraphTokenLockMigrator` contract.
  - Can `withdrawToL1Locked` from the `L2GraphTokenLockMigrator` contract.
  - Can `migrateLockedStakeToL2` from the `L1Staking` contract.
  - Can `migrateLockedDelegationToL2` from the `L1Staking` contract.

Additional Security Assumptions and Considerations

Vesting Lock Methods' Signatures and Targets Allowlist

The vesting wallet's signatures-to-targets allowlist should be handled with care. Only methods specifically designed for the vesting migration should be added to the allowlist. Adding the non-locked methods should not be done despite their inability to forward ETH, since this is a fragile assumption, and can change in the future if Arbitrum and the token bridge allow zero-gas L1-to-L2 transactions.

ETH Handling Methods

The ETH handling methods in `L1GraphTokenLockMigrator` increase the surface area for attacks and bugs due to the additional considerations related to making arbitrary destination calls, and handling ETH balances. While using WETH may add steps for some users, it is possible that the increased simplicity and security may be worthwhile.

Arbitrum Sequencer Liveness and Transaction Ordering Assumptions

Arbitrum's transaction sequencer is a singular, centralized entity. Therefore, it is possible for the sequencer to stop working, either intentionally or unintentionally, which halts the processing of

new transactions from being sequenced. Should this occur, Arbitrum has a failsafe mechanism that allows users to forcibly include a transaction to be sequenced. However, this mechanism can only be activated once a transaction has been submitted to the delayed inbox and 24 hours have gone by since the submission occurred.

In The Graph's case, indexers can close an allocation so long as one epoch (~24 hours) has elapsed. An allocation is active for `__maxAllocationEpochs` (currently 28 epochs), at which point any user can forcibly close an allocation and forfeit the rewards that would have been granted to an indexer and its delegates. If an indexer or allow-listed operator attempts to close an allocation during its last epoch, and they are forced to use the `failsafe` mechanism to include their transaction, it would take over 24 hours for their transaction to be forcibly sequenced. This would allow the `__maxAllocationEpochs` window to elapse, which means any user could submit a transaction to close the allocation. Additionally, if the Arbitrum sequencer is not working, there is no guarantee on the transaction's ordering, meaning the `closeAllocation` call by a user could be placed before the `closeAllocation` call by the indexer. Therefore, it is possible for a situation to occur where an indexer who intended to correctly close their allocation and receive rewards was unable to do so due to Arbitrum's sequencer not working as intended, and ends up unfairly forfeiting their indexing rewards.

Contracts' Deployment State Assumptions

The following assumptions were made regarding the current state of deployed contracts on L1 and L2 that were in scope for this audit:

- All Staking contracts are deployed on both L1 and L2, and are to be upgraded on both layers to the new version.
- All Vesting migration contracts have not yet been deployed on either L1 or L2.
- Wallet and manager contracts have only been deployed on L1.

Client-Reported Issues

Minimum Stake Invariant Is Broken by Partial Migration

Partial stake migration allows migrating less than the minimum stake amount if the migration is not the first. If the indexer unstakes their initial migration amount on L2, they are able to migrate arbitrarily low stakes from L1 (e.g., 1 wei of GRT), which will result in a violation of the minimum staked amount on L2. This will break the economic incentives of slashing, which ensure the integrity of the data provided by the indexers.

Although reported by The Graph's team as a known issue ahead of time, the ability to break this invariant seems unnecessary for the purpose of allowing partial migrations. The partial migration of amounts of at least the minimum stake amount can still be allowed, which will ensure that the stake on L2 is either zero or the minimum amount. Additionally, the flexibility only makes a difference for indexers with a stake size larger than twice the minimum stake amount, since if an indexer initially has, for instance, 1.5 times this amount, they must migrate all of it anyways to maintain the invariant on L1.

Consider ensuring that at least a minimum stake requirement is migrated for all amounts sent (beyond just the first one).

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

> *We acknowledge that the invariant is broken as it is now possible to have less than the minimum stake by doing two partial migrations and unstaking on the L2 side. However, we are also introducing a check for the minimum stake when opening allocations, so indexers trying to exploit this would be unable to do anything with this less-than-minimum stake, other than unstaking and waiting to withdraw. Many indexers have much more stake than the minimum, so having some flexibility in the migrated amounts is useful.*

Different Implementations of Lock Contracts On-Chain

During the audit, The Graph's team identified old implementations of `GraphTokenLockManager` and `GraphTokenLockWallet` that are different from the ones

in the current version of the codebase. Specifically, the old wallets lack the `changeBeneficiary`, `acceptLock`, and `cancelLock` mutative functions, and the `isAccepted` and `releasableAmount` `view` functions.

`GraphTokenLockWallets` that do not have the `isAccepted` function would revert when calling `depositToL2Locked` and would thus be unable to migrate using the current implementation of `L1GraphTokenLockMigrator`. Therefore, the `isAccepted` check in `depositToL2Locked` would need to be removed from the migration code to accommodate the older locks. Furthermore, the lack of a `changeBeneficiary` also changes some findings in the report in relation to these older wallets.

The old versions of the wallets or managers and their compatibility with the migration in this scope have not been audited.

Update: *Resolved in pull request #66. The check using the `isAccepted` function was removed to account for token lock wallet contracts that do not have that function.*

Migration of Fully-Vested Lock Contracts Still Requires Forwarding ETH

During the audit, The Graph's team identified an issue with their proposed migration plan for fully-vested `GraphTokenLockWallet` contracts. Originally, the team intended to allow fully-vested `GraphTokenLockWallet` contracts to use the unrestricted migration helper functions, which includes `migrateStakeToL2` and `migrateDelegationToL2`. This allows the contracts to specify any L2 address as a beneficiary, thus freeing stake and delegation from being transferred to another restricted `GraphTokenLockWallet` contract on L2. However, all `GraphTokenLockWallet` contracts are unable to forward ETH, and therefore would be unable to use migration helpers that expect ETH directly from the `msg.sender`.

Update: *Resolved in pull request #67 and pull request #74.*

Critical Severity

C-01 Funds will be lost during migration if the L1 beneficiary is a contract [Vesting]

During the initial vesting lock migration, the L2 beneficiary is set from the L1 beneficiary. However, if the L1 beneficiary is a contract on L1, its address on L2 may not be controlled by the same owner due to address aliasing. This will cause the migrated funds to be unrecoverable.

Furthermore, since the migration destination address can only be set once during the first migration transaction on L2, any subsequent migration transfers will still be sent to the same address and lost. Thus, even if a user recognizes this mistake during an initial small "test" migration and subsequently changes the L1 beneficiary to an EOA to fix the issue, their funds will still be lost in subsequent migrations as this change will not have an effect.

Although some L1 locks have the ability to change their beneficiaries ahead of migration to prevent this issue, it should not be considered a user error if they do not do so. This is because the L2 beneficiary is not an explicit input to the migration transaction, but rather an implicit input that the user is not in direct control of. Additionally, some older vesting lock implementations do not have the ability to change their beneficiaries, while other locks may not be willing or able to change the beneficiary to an EOA due to other reasons (e.g., due to being controlled by a DAO or a collective).

Consider checking that the L1 beneficiary is set to an EOA during the L1 part of the migration. For locks that are controlled by contracts and are not able to change their beneficiary, consider allowing the caller to pass an L2 beneficiary address in the call parameters. Additionally, consider reverting during any subsequent migrations if the beneficiary is different from the initial beneficiary, to prevent users from assuming a different beneficiary will be used.

Update: *Resolved in pull request #66 at commit 38539d4.*

Medium Severity

M-01 Refunded ETH will be stuck on L2 [Staking, Vesting]

In all transactions that use the `L1GraphTokenGateway`, any excess `msg.value` and any L2 gas refund will be sent to the aliased address of the sender in the context of the gateway. During the migration transactions, the migration contracts will be the senders from the point of view of the gateway. Since the aliased addresses of these L1 contracts on L2 will most likely not be controlled by The Graph or users, these funds will become unrecoverable.

For example, during `_sendTokensAndMessageToL2Staking` in `migrateStakeToL2`, the `L1Staking` contract will be the `from` address parameter on the token gateway. The `L1Staking` contract address will also be the intended refund and value recipient on L2 passed to the inbox, which will result in the aliased address of `L1Staking` on L2 receiving the funds. Similarly, for `migrateLockedStakeToL2`, the excess gas fee payment will be sent to the same address.

Furthermore, this will be the case for the `L1GraphTokenLockMigrator`'s `depositToL2Locked` method, and for `L1GNS`'s `sendSubgraphToL2` and `sendCuratorBalanceToBeneficiaryOnL2` methods as well (out of scope for this audit).

As a consequence of these migration paths, a substantial amount of ETH may get trapped in unrecoverable addresses. This is both because gas payments on L1 are likely to overpay for L2 gas to ensure that transactions do not fail, and because the payable methods do not restrict `msg.value` to the minimum gas payment needed.

Consider checking that `msg.value` is exactly the `_maxSubmissionCost + _gasPriceBid * _maxGas` in all migration paths to prevent any overpayment. Additionally, consider updating the `L1GraphTokenGateway` so that setting an explicit `submissionRefundAddress` (i.e., `excessFeeRefundAddress`) to a beneficiary is possible during the call to `outboundTransfer`.

Update: *Resolved in pull request #810 at commit ff8ce4d, and pull request #68 at commit cfe3a1d.*

M-02 Locked migrated stake and delegation may be lost due to beneficiary address mismatch [Staking, Vesting]

The address of the L2 lock on L1 is computed and stored during the initial lock migration. However, the actual L2 address is created and stored when the initial migration is completed. There are multiple scenarios that may cause an address mismatch between the L2 lock address stored on L1 and the one on L2.

First, if L2 transactions are executed out of order with regards to L1 transactions, it is possible that the first successful L2 lock migration transaction will correspond to a second (or subsequent) L1 transaction, instead of the first one. This can happen, for instance, due to the first transaction's retryable ticket initially failing due to gas reasons, or due to the sequencer executing them out of order (there is no order guarantee for L1 transactions before they are "force-included"). In this case, if the payload of the second transaction is different from the first, due to an updated wallet beneficiary on L1, the address on L1 and L2 will not match. This is because the message payload affects the address of the new locked contract, due to being used in the `create2` salt on L1 and on L2.

A second set of mismatches can occur if the `l2Implementation` or `l2Manager` addresses are different (on L2) from what was initially used to calculate the expected L2 token lock contract address on L1, when the migration transaction happened. This can happen due to a misconfiguration, or during an upgrade to the migration mechanism, since an upgrade cannot be performed atomically on L1 and L2 at the same time, and some transactions may be in-transit.

In the context of the lock migration itself, these scenarios do not create a problem, since the L2 manager will correctly match the addresses using its internal mappings. However, in the context of the staking and delegation migrations, the address stored on L1 is used as the L2 beneficiary. An address mismatch will prevent the migrated balances from being claimed on L2.

Consider not specifying a beneficiary on L1 during message encoding, and instead relying on the L2 lock manager to resolve the beneficiary on L2 using its `l1WalletToL2Wallet` mapping. To ensure that the beneficiary exists on L2, consider requiring that the L1 lock migrator has already migrated that account. Additionally, in case of any upgrades to the L2 contracts referenced on L1, consider pausing any L1 migration initiation functionality well ahead of the L2 upgrade by removing the migration method from the allowlist to prevent any mismatches due to in-transit migrations.

Update: *Resolved in pull request #66 at commit c15d7b3.*

M-03 Inconsistent initialization of delegation parameters [Staking]

During the first staking deposit, `_setDelegationParameters` is called to set the indexer's delegation parameters. However, during the finalization of migration on L2, this parameter initialization is not done. As a result, the default parameters after the first staking deposit and following a migration are different.

Specifically, due to not setting delegation parameters on stake migration, the indexer's migrating delegators will not receive any rewards on L2 unless the indexer updates the parameters after the migration. Until that happens, due to the undelegation delay, the delegators' GRT will be locked and temporarily unavailable.

Additionally, the default parameters set during initial staking are set to values that will also disable distributing indexing and query rewards to delegators due to the boolean expressions in `_collectDelegationQueryRewards` and `_collectDelegationIndexingRewards`, respectively.

Consider using the same initialization logic in both code paths to ensure the behavior is consistent. Additionally, consider setting the default values in a way that will be functional and sensible for both indexers and delegators. The default values should favor delegators in order to protect them in case the indexer is inactive, and also to incentivize the indexer to set the correct values.

Update: *Resolved in pull request #811 at commit 1df3808. The Graph's team stated:*

> *Note the default value must be the one we use in L1, as any other value would mean the indexer shares rewards without consent (unless we migrated the values from L1 but that could pose additional issues). Indexers are incentivized to set these parameters to attract delegators, and delegators could choose not to migrate if the indexer does not set attractive values.*

M-04 Indexers can delay slashing by migrating [Staking]

When an indexer is acting maliciously, such as providing bad indexing results or not indexing in a timely manner, they can be slashed accordingly. Before the migration changes, indexers were

unable to avoid slashing as there was a mandatory period of time between unstaking GRT and withdrawing GRT. This allowed fishermen and arbitrators enough time to identify, dispute, and slash an indexer before they could unstake and withdraw their GRT. However, the new migration path does not have a wait time that must elapse before an indexer is allowed to transfer their stake to Arbitrum L2. Therefore, if an indexer is aware they are going to be slashed, they can simply migrate their stake to L2 and thus avoid slashing on L1. However, there is no on-chain verification required for slashing, so it would be possible to slash the indexer accordingly on L2.

Consider monitoring for `slash` function calls that revert, which may be an indication that an indexer has temporarily avoided slashing on L1 by migrating to L2. Additionally, The Graph's Arbitration Charter may need to be updated to include a process for disputes and slashing that occur while indexers are migrating to L2.

Update: *Acknowledged, the fix will be to modify the Arbitration Charter as suggested. This is mentioned in pull request #13, which updates GIP-0046.*

M-05 Migrated token lock wallet can be canceled on L2 [Vesting]

Token lock wallets that are created on L2 as part of the migration path in the `depositToL2Locked` function in `L1GraphTokenLockMigrator` are not initialized with the `isAccepted` flag set to `true`. This means it is possible for the token lock wallet owner on L2 to call `cancelLock` and transfer whatever amount of GRT is in the token lock to itself. While the beneficiary on L2 can call `acceptLock` which will subsequently set `isAccepted` to `true`, the check to ensure only accepted tokens locks are migrated in `depositToL2Locked` may lead the beneficiary to incorrectly assume the token lock created on L2 is instantiated with the lock accepted.

Consider setting `isAccepted = true` in the `initializeFromL1` function in `L2GraphTokenLockWallet`.

Update: *Resolved in pull request #69 at commit 26c8808.*

M-06 Vesting migration contracts are not upgradeable [Vesting]

The following vesting migration contracts do not use `GraphUpgradeable`:

- `L1GraphTokenLockMigrator`
- `L2GraphTokenLockMigrator`
- `L2GraphTokenLockManager`

This means any logic changes to the above contracts would require them to be redeployed. Not having an upgradeable abstraction on top of immutable smart contracts typically makes them less complex. However, various issues identified during this audit would have required the vesting migration contracts to be updated and redeployed. This would have led to the following consequences:

- Deploying a new contract does not transfer the state of the previous contract. Therefore, the migration state in `L1GraphTokenLockMigrator` and `L2GraphTokenLockManager` would need to be repopulated into the new contracts if these contracts were deployed again. Since a mapping is used to hold the migration's state, it would be difficult to retrieve all the relevant information from the previous migration contracts, and missing out on any state for the migration contracts could cause mismatches between the L1 and L2 contracts. This could impact a vesting contract's ability to migrate, or continue migrating.
- Deploying a new `L2GraphTokenLockManager` changes the address involved in the `CREATE2` deployment of the minimal proxy `GraphTokenLockWallet` contracts. This changes the calculation of the counterpart L2 token lock wallet address in the `L1GraphTokenMigrator`, which could cause the L1 and L2 migration states to become mismatched.

Therefore, consider making the above contracts upgradeable to avoid causing mismatches between the L1 and L2 migration states by having to deploy new versions of the contracts instead of upgrading the existing ones.

Update: *Partially resolved in pull request #70, and pull request #73. Only `L1GraphTokenLockMigrator` and `L2GraphTokenLockMigrator` were modified to be upgradeable. The Graph's team stated:*

> *We would rather keep `L2GraphTokenLockManager` immutable to match the level of trust from L1, but we added transparent proxies for the migrators.*

M-07 Migration functions do not use a pause modifier [Staking, Vesting]

The following migration functions do not use the `notPartialPaused` or `notPaused` modifiers, which revert if either the `_partialPaused` or `_paused` internal `boolean` values are set to `true`:

- `migrateStakeToL2` in `L1Staking`
- `migrateLockedStakeToL2` in `L1Staking`
- `migrateDelegationToL2` in `L1Staking`
- `migrateLockedDelegationToL2` in `L1Staking`
- `unlockDelegationToMigratedIndexer` in `L1Staking`
- `depositToL2Locked` in `L1GraphTokenLockMigrator`

In the event of a full pause, the `outboundTransfer` function in the `L1GraphTokenGateway` (which makes use of the `notPaused` modifier) will revert, and subsequently the five migration methods will revert. This helps mitigate the issue in case of a full protocol pause. In the event of a partial pause, however, these methods will continue functioning, which may have dangerous consequences due to the delicate state consistency that the contracts have to maintain, and due to the outflow of GRT from the protocol as a result.

`unlockDelegationToMigratedIndexer` will allow circumventing a partial pause and withdrawing GRT on L1, since `undelegate` is guarded by `notPartialPaused`, but `withdrawDelegated` is only guarded by `notPaused`.

Consider adding the `notPartialPaused` modifier to the above functions to be consistent with other migration functions that make use of the `notPartialPaused` modifier.

Update: *Resolved in pull request #812 at commit c076852. The Graph's team stated:*

> *The linked pull request fixes all the listed instances of this issue on the Staking contract. For the vesting, `depositToL2Locked in L1GraphTokenLockMigrator`, we do not think it is necessary since the vesting contracts are not affected by a protocol partial pause and this particular function does not interact with protocol state other than the GRT token contract, which is also not affected by a partial pause.*

M-08 Minimum stake invariant can be broken on L1 by restaking rewards after migration [Staking]

Addresses that want to become indexers are required to stake a minimum amount of GRT as determined by the `__minimumIndexerStake` state variable. This ensures the incentives and deterrents within The Graph's protocol work as expected. However, the minimum stake invariant can be broken on L1 as part of the migration path due to the timing of query rewards.

When closing an allocation, there is a period of time that must elapse before an allocation is finalized and query rewards can be claimed. These rewards can optionally be restaked to the indexer. If an indexer closes their remaining allocations and then immediately migrates all of their staked GRT, the indexer will have migrated before receiving the query rewards from their recently closed allocations. When the allocations are finalized, it is possible for the indexer or an allow-listed operator to claim the query rewards and restake back to the indexer. During the internal call for restaking rewards, the resulting amount will not be checked to ensure the minimum indexer stake has been met. This is because the private `_stake` function lacks this check.

Therefore, it is possible that restaking the query rewards to an indexer who has fully migrated their stake to L2 results in an indexer with a stake amount that is below the `__minimumIndexerStake` on L1. However, the indexer is unable to allocate using this stake due to a check that ensures the indexer holds the minimum amount of stake. Additionally, an indexer having a non-zero amount of stake on L1 blocks the instant withdrawal path for delegators who now have to wait the entire duration of the unbonding period if they choose not to migrate to L2.

Consider moving the minimum stake check from `stakeTo` to `_stake` to ensure the minimum stake invariant holds when restaking query rewards.

Update: *Resolved in pull request #813 at commit 520ff1d, and in pull request #823 at commit bc506ba. The minimum stake check has been moved to the `_stake` function. As a consequence, the `_receiveIndexerStake` in `L2Staking` has been updated to duplicate the logic in `_stake` without the minimum stake check. This is to ensure that migrating stake to an indexer on L2 that does not meet the minimum stake value does not cause the cross-chain transaction to revert. This can happen if multiple stake migrations occur, as an L1 indexer only has to send the minimum stake amount on the first migration to an L2 beneficiary, but not on subsequent migrations.*

M-09 Vulnerable rounding in delegation pool calculations [Staking]

There is a known vulnerability in ERC-4626 vaults regarding share calculation rounding. While the delegation pools are not explicitly ERC-4626 vaults, the calculations used to deposit GRT for delegation pool shares and redeem delegation pool shares for GRT make the delegation pools behave in a "vault-like" manner. Thus, the delegation pools are susceptible to the following rounding vulnerabilities:

In the first scenario, the `shares calculation` done as part of the `delegate` logic is susceptible to unfavorable rounding. While there is a check that ensures zero shares cannot be minted, this does not prevent the calculation from being manipulated to round down, thus potentially devaluing a delegator's contribution to an indexer. The following steps outline the attack scenario:

1. A malicious address stakes on L1 and becomes an indexer.
2. The malicious indexer delegates 1 wei to themselves.
3. The indexer then allocates to a subgraph to receive rewards.
4. Query fees and indexing rewards are distributed after at least one epoch, and since the indexer had a non-zero amount of delegation, it will disperse rewards to its delegation pool. This skews the ratio between the delegation pool shares and tokens by, at most, the total reward tokens that were distributed to the delegation pool for the allocation. Therefore, the ratio of shares to tokens in the delegation pool for the malicious indexer is now `1 share / (1 wei + X rewarded GRT)`. For the sake of this example, we assume 100 GRT is rewarded to the delegation pool.
5. A victim delegator delegates 200 GRT to the malicious indexer. The shares' calculation returns `(200 GRT * 1 share) / (1 wei + 100 GRT) = 1 share`, which rounds down and gives the victim delegator 1 share instead of 2 shares. The malicious indexer (who is also a delegator) has gained 50 GRT from the victim delegator due to rounding.

In the second scenario, the `_receiveDelegation` function used as part of the delegation migration path lacks the zero shares minted check that the normal `delegate` logic contains. The check was most likely removed in order to minimize failure conditions when migrating delegation that could in turn cause cross-chain tickets to be un-redeemable on L2, and subsequently lock GRT on the gateway. However, by removing this check, the shares' calculation when migrating delegation to the new indexer's delegation pool is susceptible to a

modified version of the ERC-4626 inflation vulnerability. The following steps outline the modified attack scenario:

1. A malicious address stakes on L2 and becomes an indexer.
2. The malicious indexer delegates 1 wei to themselves.
3. The indexer then allocates to a subgraph to receive rewards.
4. Query fees and indexing rewards are distributed after at least one epoch, and since the indexer had a non-zero amount of delegation, it will disperse rewards to its delegation pool. This skews the ratio between the delegation pool shares and tokens by, at most, the total reward tokens that were distributed to the delegation pool for the allocation. Therefore, the ratio of shares to tokens in the delegation pool for the malicious indexer is now `1 share / (1 wei + X rewarded GRT)`.
5. The malicious indexer initiates the migration of their L1 stake to the L2 address with the skewed delegation pool.
6. If a victim delegator migrates their delegation to L2, depending on how much GRT they have delegated to the malicious indexer, they may be susceptible to floating point truncation during the division of their new shares' calculation on L2. Truncation will occur if the delegator migrates Y GRT such that `Y migrated GRT <= X rewarded GRT`. If truncation does occur for a delegator, the missing shares check will cause the delegator to lose their GRT as 0 shares will be minted to them.

However, in both circumstances there are several mitigating factors that limit the severity and impact of the vulnerable rounding conditions:

- These rounding attacks cannot be accomplished in one transaction due to the requirement for at least one epoch (~24 hours) to elapse before an allocation can be closed. This allows time for other delegators to delegate to an indexer, which would balance the ratio of shares to tokens in the delegation pool.
- The rewards distributed to a delegation pool cannot be manipulated in a controlled manner, so it is not possible to set up a rounding attack for an arbitrary number of GRT that a victim delegator may delegate to the pool.
- Due to the prior knowledge of a delegation pool that is required to set up an attack, the only user that would be able to perform either rounding attack would be the indexer of the delegation pool. The indexer of a delegation pool is vulnerable to slashing, where the slashed amount most likely exceeds the amount that could be gained from manipulating their delegation pool, which scales with the rewards accumulated to it during the attack.

There are various mitigation strategies that can be used to prevent rounding vulnerabilities, such as the ones mentioned here. However, each strategy has its own tradeoff. For this

particular set of vulnerabilities, consider burning a small amount of shares (for example 1000 wei) during the initial pool delegation.

Update: *Partially resolved in pull request #814 at commit 79fbd32 (only that specific commit). In `L2Staking`, if the delegation pool share calculation returns 0, the GRT will be transferred directly to the L2 delegate beneficiary instead of being added to the pool. Note that this does not address the rounding concern in `StakingExtension`. Regarding `StakingExtension`, the Graph's team stated:*

> *Good find, though we would not want to modify the core delegation shares calculation in this upgrade, so instead we will add the zero-shares check as in L1, acknowledging it only fixes part of the issue.*

Low Severity

L-01 Updates were made to immutable L1 `GraphTokenLockWallet` contracts [Vesting]

The following functional changes were made to the `GraphTokenLockWallet` contract to prevent ETH from being received:

- The `fallback` function now reverts when receiving ETH.
- A reverting `receive` function was added.

The token lock contracts on L1 are not upgradeable, so previously-deployed instances of the contract will not be updated with this functionality. If the source code is allowed to diverge from the deployed code in functionally important aspects, future maintainers may work under incorrect assumptions, which is error-prone.

Consider only adding the `receive` function to the `L2GraphTokenLockWallet`, and refactoring the `fallback` function so that only the `L2GraphTokenLockWallet` will revert when receiving ETH. This can be achieved by using an internal function:

```
contract oldA {
    fallback() external payable virtual {
        // old code
    }
}

contract newA {
```

```
    fallback() external payable virtual {
        _fallback();
    }

    function _fallback() internal {
        // old code
    }
}

contract B is newA {
    // new methods

    fallback() external payable override {
        // some new fallback code

        _fallback();
    }
}
```

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

> *Will not fix. The change in this pull request does not introduce a new risk. The source code of `GraphTokenLockWallet` was already modified in the past after instances of it were deployed.*

L-02 Contracts can be initialized multiple times [Staking]

The `initialize` function in both the `Staking` and `StakingExtension` contracts lacks the `initializer` modifier used in other contracts. This allows the two contracts to be re-initialized by the governor, potentially leading to an inconsistent or unexpected state.

Consider using the `Initializable` mixin and ensuring initialization can only be called once by using the `initializer` modifier.

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

> *Will not fix. We would rather not add more code to the staking contracts as they are already very close to the maximum contract size. The current staking contract can also be initialized multiple times so we are not adding risk here.*

L-03 `fallback` check allows call from implementation [Staking]

The contract address check in the `fallback` function in `Staking` will always pass when called from the implementation contract since the contract's `address(this)` can never be the implementation address. In the proxy context, the proxy's address will be different from the `_implementation()` value, and in the implementation context, `_implementation()` should return `address(0)`, which can never correspond to `address(this)`. This causes the check to always pass even when invoked on the implementation.

Since the implementation contract cannot be properly initialized, `extensionImpl` cannot be set. When calling from the implementation, the `extensionImpl` address will be `delegatecalled` where `extensionImpl` is the zero address. A `delegatecall` to the zero address will result in a no-op transaction.

If the intention is to check that the `fallback` function is not called on the implementation contract, the condition can be altered to check that `_implementation()` returns a non-zero address. Alternatively, the implementation contract's own address can be stored as an immutable constant during construction and checked to be different from `address(this)`.

Update: *Resolved in pull request #798 at commit 00f5c94.*

L-04 `withdrawETH` can call any contract's `fallback` function [Vesting]

`withdrawETH` will call any address' `fallback` function without the need to deposit ETH, and will not send ETH with the call if the `_amount` passed in is 0. This can potentially be used for phishing (e.g., originate a fake airdrop), or for reputation attacks. However, even if a zero-amount check is added, calling an arbitrary address' payable fallback would be possible by depositing 1 wei prior to the call using `depositETH`. This however does make the scenario less likely to be used for any low-value attacks.

In general, performing a call to an arbitrary destination increases the attack surface, which is a broad problem and may not be justified if it can be avoided. In order to avoid arbitrary destination calls completely, WETH can be transferred instead of ETH. This will limit the call to the known WETH contract, which will limit the added surface area. Furthermore, WETH can be used to remove the need for both `depositETH` and `withdrawETH` methods in the contract by allowing users to transfer WETH to the lock contracts and approving the migrator to pull it as needed.

Consider adding a zero-amount check to prevent the easiest type of misuse. Additionally, consider transferring WETH instead of ETH to prevent arbitrary destination calls. Alternatively, consider using WETH for both deposit and withdrawal flows to remove the need for the contract to accept and handle ETH.

Update: *Resolved in pull request #71 at commit db13017. The Graph's team stated:*

> *The linked pull request adds a zero-ETH check to rule out the simple misuse case. We considered using WETH as suggested, but feel that the added UX complexity does not justify the change.*

L-05 Test coverage cannot be measured [Vesting]

The repository tests are set up in a way that is not compatible with coverage measurement. This hinders the task of assessing the extent of the test coverage, as well as identifying specific coverage misses that are often indicative of correctness or security issues.

Consider updating the test setup in a way that allows coverage measurement, as well as adding coverage metrics and checks into the repository's workflow to ensure that test coverage is maintained and improved on an ongoing basis.

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

> *We will not fix this for the moment but will add to our backlog.*

L-06 Missing address validation [Staking]

The `StakingExtension` address, `extensionImpl`, may be set to zero on initialization or during `setExtensionImpl`. `extensionImpl` is only called using the low-level `delegatecall`, which means the `delegatecall` in the `initialize` function and the `fallback` function will not revert if the address is the zero-address or is set to an EOA. The behavior of the low-level `delegatecall` can mask the mistake of incorrectly setting the `extensionImpl` address, as the contracts will appear to be functioning when called (except no logic will be executed for some methods).

Consider validating that a contract is deployed at the address intended for `StakingExtension` when it is being set during `initialization` or when `setExtensionImpl` is called.

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

> *Will not fix. We would rather not add more code to the staking contracts as they are already very close to the maximum contract size. Furthermore, this is a governance-only function, so the risk can be mitigated.*

L-07 Delegators can miss out on indexing and query rewards when migrating or undelegating [Staking]

Delegators can miss out on both indexing and query rewards depending on when they decide to either migrate their delegation to L2 or undelegate and withdraw their GRT on L1. This occurs for two reasons:

- Indexers are allowed to partially migrate their stake, which means allocations on L1 may still be open while delegators are migrating to L2.
- There is a period of time between when an allocation closes and when an allocation is finalized to allow for disputes. Query fees can only be claimed and distributed to delegators once an allocation is finalized.

Delegators that migrate to L2 before all allocations are closed and rewards are distributed will lose out on their portion of the rewards. These rewards will either be distributed to the delegators that still remain on L1, or sent to the indexer. The loss of rewards during migration is similar to the opportunity cost associated with the unbonding period when undelegating from an indexer.

Consider communicating the potential costs of migrating early to delegators. This may include adding UI elements to the dApp that warn delegators who may attempt to migrate before rewards are distributed on L1 for their indexer.

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

> *We will ensure the migration UI clearly communicates the risks associated with an early migration.*

L-08 Token lock migration tests do not sufficiently simulate on-chain conditions [Vesting]

The token lock migration tests in `l1TokenLockMigrator.test.ts`, `l2TokenLockMigrator.test.ts`, and `l2TokenLockManager.test.ts` use mocks. Mocks do not sufficiently simulate on-chain conditions, which could mask issues that may otherwise occur when testing on a testnet or deploying to the mainnet. Therefore, tests that use mocks can provide a false sense of security and test coverage.

Consider implementing tests that simulate on-chain conditions by instantiating local instances of the L1 and L2 blockchains.

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

> *We will address this with a test plan on testnet before deploying. Setting up an automated `devenv` to simulate cross-chain interactions is something we have been working on but it is still in progress.*

L-09 Signed messages used for allocations and redemption vouchers are not unique cross-chain [Staking]

There are two instances where signed messages are being used and the message content does not include information that is unique across different blockchains:

- The proof required for subgraph allocation, where the signed message content is defined as `keccak256(indexerAddress,allocationID)`
- The redemption vouchers for an allocation, where the voucher itself is a signed message and the content is defined as `keccak256(abi.encodePacked(_voucher.allocationID, _voucher.amount))`

In both cases, there are additional protections in place to ensure only authorized addresses can interact with functions that use the signed messages. For the allocation signature, the function `_isAuth` is used to ensure that only the indexer or an authorized operator address can allocate stake to a subgraph. For redemption vouchers, the signing authority is set by the governor where the authority is an EOA but is checked off-chain by governance to ensure the same address is not set on multiple different blockchains.

While there are additional protections in place to ensure the aforementioned signatures are protected against unauthenticated replay cross-chain, it is best practice to ensure signed messages contain information that is unique in cross-chain applications. Consider adding the `chainID` to the contents of each signed message to ensure uniqueness.

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

> *We are aware of the risk in `AllocationExchange` and will ensure signers are different on each chain, and we are keeping this in mind for future iterations of the fee vouchers. On the `allocationIDs`, there should be no issues as far as we can see if the `allocationID` is reused cross-chain, but indexers will generally auto-generate random accounts for this.*

Notes & Additional Information

N-01 The `StakingExtension` implementation's address can be updated in different ways [Staking]

The function `setExtensionImpl` in `Staking` is used to update the implementation address of the `StakingExtension` contract. However, in contrast to the way this address is set during initialization, no call to its initialization is made. This may cause issues in the event the contract implementation is updated, as additional initialization steps may need to be executed.

Additionally, because the chained proxy pattern used between `Staking` and `StakingExtension` is different from the normal transparent proxy pattern used by the rest of the protocol, it may be safer to avoid additional complications and risks of the upgrade flow that `setExtensionImpl` possibly adds (e.g., storage compatibility).

Consider removing `setExtensionImpl` if there is no concrete need to use it operationally, and instead use the normal upgrade process to update implementation contracts. Alternatively, consider documenting the planned usage, and allowing the execution of an initialization step for consistency.

Update: *Resolved. The Graph's team stated:*

> *The `setExtensionImpl` will be needed to set the `StakingExtension` implementation after upgrading the existing Staking contracts to the new version that uses `StakingExtension`, as we generally do not call the `initialize()` function again when running an update (as the contracts are already initialized and `initialize()` would set other things that shouldn't be set again). And we should still keep the setting in `initialize()` so that new deployments (e.g., to testnets, or future networks) are properly initialized. So both ways to update the extension implementation are needed.*

N-02 Unused imports [Vesting]

Throughout the codebase, imports on the following lines are unused and could be removed:

- Import `IGraphTokenLock` of `L2GraphTokenLockManager.sol`
- Import `IGraphTokenLock` of `L2GraphTokenLockMigrator.sol`

Consider removing unused imports to avoid confusion that could reduce the overall clarity and readability of the codebase.

Update: *Resolved in pull request #72 at commit d3fb094.*

N-03 Unsafe ABI encoding [Staking, Vesting]

`abi.encodeWithSignature` is used to generate calldata for a low-level call. However, this option is not typo-safe, which is error-prone and should be considered unsafe. Some examples are seen:

- On line 156 of `L2GraphTokenLockManager.sol`
- On line 126 of `Staking.sol`

Consider using the safer `abi.encodeWithSelector` which is available in the used version of Solidity. Alternatively, when a newer version of Solidity is used, consider using `abi.encodeCall`, which checks whether the supplied values actually match the types expected by the called function and also avoids errors caused by typos.

Update: *Resolved in pull request #815 at commit c4917d2, and pull request #72 at commit 552897a.*

N-04 Partial lock migration allows partial sale of unvested tokens [Vesting]

Allowing partial migration creates the opportunity to split the unvested tokens into two beneficiaries, thus selling them over the counter in a manner that removes custody or counterparty risk. As the migration destination cannot be changed, this forgoes the ability for the portion remaining on L1 to migrate to a separate address and reduces the usefulness of these tokens. However, it may still be advantageous to be able to sell a portion of the unvested tokens (this could impact the GRT market).

Although the design choice of allowing a partial migration is safer and provides better UX, the vesting split implications should also be considered by the community.

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

> *We will ensure lock manager owners (Edge & Node, Graph Foundation) are aware of this.*

N-05 Missing test cases [Staking]

Throughout the codebase, several code paths that are lacking test cases have been identified:

`Staking`:

- The fallback `require` statement logic is not tested.
- The `setExtensionImpl` function has no tests.
- The `_operator` address check is not tested.
- The `isAllocation` function has no tests.
- The `claimMany` function has no tests.
- The `collect` asset holders check logic is not tested.
- The `rebatePool.unclaimedAllocationsCount == 0` `if` branch's logic is not tested.
- The `_updateRewards` and `_distributeRewards` empty address branches are not tested.

`StakingExtension`:

- Slashing more than the stake is not tested.
- The `subgraphAllocations` function has no tests.
- The `isDelegator` function has no tests.

`L1Staking`:

- The checks that ETH was successfully pulled in `migrateLockedStakeToL2` and `migrateLockedDelegationToL2` are not tested.

Consider ensuring the above positive and negative cases are covered by the test suite.

Update: *Partially resolved in pull request #815 at commit e8a24d5. The following test cases remain unimplemented:*

- *The `collect` asset holders check logic is not tested.*
- *The `rebatePool.unclaimedAllocationsCount == 0` `if` branch's logic is not tested.*
- *The `_updateRewards` and `_distributeRewards` empty address branches are not tested.*
- *Slashing more than the stake is not tested.*
- *The `subgraphAllocations` function has no tests.*

Regarding the unimplemented test cases, the Graph's team stated:

> *Added more tests to improve coverage, but left out some cases that were already missing from before this pull request.*

N-06 Return result is unused [Vesting]

The result returned from `withdrawToL1Locked` appears to be unused and not needed in the calling contract. Consider simplifying the interface by removing this return value.

Update: *Resolved in pull request #75 at commit `c256ca6`.*

N-07 `Ownable` can be imported as `OwnableInitializable` for clarity [Vesting]

There are two `Ownable` versions used in this codebase. The main difference between the version used in `L2GraphTokenLockWallet` and the other one is that it is upgradeable and initializable. Consider importing it as `OwnableUpgradable` or `OwnableInitializable` to highlight this difference and implicitly clarify the need for two different versions.

Update: *Resolved in pull request #72 at commit a678ed7.*

N-08 Inconsistent error message [Staking]

The check for the delegation token lock is done on `tokensLockedUntil`, but the error message mentions `tokensLocked`. This is inconsistent, and it appears that checking `tokensLocked` instead of `tokensLockedUntil` would be a more robust check. This is because it would not rely on the semantic overloading of the lock time variable, but would instead check the amount of tokens locked.

Consider updating the check to use the `tokensLocked` value instead.

Update: *Resolved in pull request #815 at commit 63f64ed.*

N-09 Unused struct [Staking]

Consider removing the `CloseAllocationRequest` struct since it is no longer used following the refactor, due to the removal of the `closeAllocationMany` method.

Update: *Resolved in pull request #815 at commit 25f5f7e.*

N-10 `require` statements with multiple conditions [Staking]

Throughout the codebase there are `require` statements that require multiple conditions to be satisfied. For instance:

- The `require` statement on line 207 of `L1Staking.sol`
- The `require` statement on line 652 of `Staking.sol`

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

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

> *Acknowledged, but these conditions are tightly related so we would rather leave them as a single check.*

N-11 Gas optimizations [Staking]

The following instances of storage reads can be removed or adjusted in order to optimize gas consumption:

- In `_migrateDelegationToL2`, storage variables are loaded multiple times: `indexerMigratedToL2[_indexer]`, `pool.tokens`, `pool.shares`, and `delegation.shares`.
- In `migrateStakeToL2`, similarly, `indexerStake.tokensStaked` and `indexerStake.tokensAllocated` (during `indexerStake.tokensUsed()`, which is redundant since there are no locked tokens) are read multiple times.

Consider caching in temporary variables to reduce storage-related gas costs.

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

> *Acknowledged, but this is a conscious trade-off between gas and contract size.*

N-12 Typographical error [Staking]

Consider addressing the following typographical error:

- `adddress` should be "address."

Update: *Resolved in pull request #815 at commit 495ea0c.*

N-13 Contracts are used instead of interfaces [Vesting]

In the `withdrawToL1Locked` function in `L2GraphTokenLockMigrator`, contract implementations are used for casting instead of interfaces. Consider using interfaces for clarity.

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

> *Acknowledged, but it feels like overkill to create interfaces just for use in this function, so we will leave it like this for now.*

N-14 Dangerous storage packing assumption [Staking]

The `fallback` function in `Staking` assumes the `extensionImpl` state variable is the first value stored in its respective storage slot. This assumption is typically used by transparent proxies, whose implementation addresses are stored such that they are in unique storage slots. However, this particular use of a `fallback` function retrieves a state variable from storage that may be packed with other state variables. While the logic does use a bitmask to clear any values after the `extensionImpl` state variable, this will not work if `extensionImpl` is not the first value in a storage slot. Coincidentally, the current packing of `extensionImpl` in storage ensures that it will be the first value in a new storage slot.

Consider adding a comment to line 56 in `Staking` to indicate that the packing of state variables allows the code to use `.slot` without having to calculate an offset. This ensures the code snippet is not simply copy-pasted into a future codebase where the storage packing assumption may not hold.

Update: *Resolved in pull request #815 at commit 19c55b7.*

N-15 Incorrect documentation [Staking, Vesting]

The following instances of incorrect documentation have been identified:

- The docstring for the `initialize` function in `IStakingBase` states the `_thawingPeriod` is the "Number of epochs that tokens get locked after unstaking," but `_thawingPeriod` is in blocks, not epochs.
- The comment on line 68 in the `withdrawToL1Locked` function in `L2GraphTokenLockMigrator` is incorrect. The comment should be placed above the call to `outboundTransfer` instead, and say "Send the tokens through the L2GraphTokenGateway to the L1 wallet counterpart".

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

Update: *Resolved in pull request #815 at commit 5fcd2dc, and pull request #72 at commit 0e6f783.*

N-16 Insufficient input checks in `_setAuthFunctionCall` [Vesting]

The `GraphTokenLockWallet` contracts are designed to handle approvals using the `approveProtocol` and `revokeProtocol` functions; however, a misconfiguration of the general target allowlist is possible. If the `approve` or `transfer` methods are added for GRT, all tokens can be unlocked.

While this is unlikely to happen, consider explicitly disallowing those signatures for the GRT contract target when adding signatures in `_setAuthFunctionCall` for the new L2 contracts.

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

> *Acknowledged, but will not fix. Lock Manager owners could legitimately want to allow beneficiaries to do an early exit for whatever reason and this would be a way to do it; it would be very unlikely for this to be done accidentally as any `allowlist` inclusions will be carefully evaluated.*

N-17 Migrated token lock wallets may miscalculate `surplusAmount` [Vesting]

L1 token lock wallets that have been migrated using the `depositToL2Locked` function do not pass all of their internal state data to their counterpart L2 token lock wallet. More specifically, `releasedAmount` is not encoded as part of the migration data. `releasedAmount` is used in the `totalOutstandingAmount` function, which is then used by the `surplusAmount` function to calculate if there is any GRT in excess of the amount that is a part of the remaining tokens locked that can be withdrawn immediately. If `releasedAmount` is non-zero on L1 but set to the default value of zero on L2, this will skew the calculation of `surplusAmount`. Additionally, as tokens are released from the vesting lock wallet on L1, `releasedAmount` will not be reflected on L2 as tokens are not released on L2 until the wallet is fully vested.

Consider adding documentation to `L2GraphTokenLockWallet` that explains that the `surplusAmount` state variable in L2 token lock wallets will be skewed, and that beneficiaries should bridge their L2 locked tokens back to the token lock wallet on L1 if they want to release tokens and withdraw surplus tokens from the token lock wallet.

Update: *Resolved in pull request #72 at commit f6e4612.*

N-18 Use of deprecated `safeApprove` function [Vesting]

The `safeApprove` function used by the `approveProtocol` and `revokeProtocol` functions in `GraphTokenLockWallet` is deprecated. Additionally, `safeApprove` disallows token approvals where the original approval is non-zero and a new approval is non-zero due to the ability to double-spend approvals. This, combined with the logic in `approveProtocol` which requires all protocol addresses within the enumerable set to be re-approved for a non-zero amount, means that vesting contract beneficiaries will first have to revoke and then re-approve all token approvals if a new contract address is added to the allowlist. This is cumbersome and potentially expensive for the beneficiaries of a vesting contract.

While the `GraphTokenLockWallet` contract is immutable on L1, it is yet to be deployed on L2. Therefore, consider replacing `safeApprove` with `safeIncreaseAllowance` and `safeDecreaseAllowance` respectively.

Update: *Resolved in pull request #72 at commit ccfc254. `approve` was used instead of `safeIncreaseAllowance` and `safeDecreaseAllowance` as the token lock wallet is only able to set allowances that are either `type(uint256).max` or 0. Thus allowance front-running is not a concern in this use case.*

Conclusions

One critical-severity issue and multiple medium-severity issues were identified during this audit. The majority of the more severe issues stem from either cross-chain assumptions about addresses and states, or from the complex set of economic assumptions and user flows for the different roles in the protocol. We also recommend implementing monitoring and/or alerting functionality.

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 incorporate monitoring activities in the production environment. Ongoing monitoring of deployed contracts helps in identifying potential threats and issues that may affect the protocol. To achieve a comprehensive security assessment, it is necessary to identify trust assumptions and out-of-scope components that could benefit from on-chain monitoring:

- Retryable tickets failing on L2: if a retryable ticket for an L2 staking, delegation, or vesting migration has not been successfully executed, this could be an indication of a problem on the `L2GraphTokenGateway`. Consider monitoring L2 migration retryable tickets to ensure that the `sequenceNum` emitted by the `RedeemScheduled` event does not indicate that the transaction has been retried multiple times and failed.
- `slash` calls failing on L1: if `slash` function calls abruptly start failing for indexers on L1 due to insufficient funds, this could be an indication that an indexer migrated to L2 before they were slashed on L1. This can be verified by subsequently checking an indexer address that was the target of a `slash` call. If it was, the `IndexerMigratedToL2` event would be emitted with that address and an amount of GRT that was migrated. In that case, the GRT value which was subtracted from their newly-existing balance on L1 becomes smaller than the proposed `slash` amount, causing the call to revert.
- Delegation migration calculating zero shares on L2: if the `StakeDelegated` event is emitted on L2 with zero shares, this is an indication that the delegation pool has a skewed ratio of shares to underlying GRT, and is giving zero shares to delegators that migrate to L2.
- L2 token wallet address discrepancies: there may be a scenario where an L1 `LockedFundsSentToL2` event is emitted with an `l2Wallet` address that is different from the `contractAddress` value emitted from the corresponding `TokenLockCreatedFromL1` event on L2 for the same L1 token lock. This is an indication of a discrepancy between the expected L2 token lock wallet address calculated on L1 and the actual L2 token lock wallet address deployed on L2.
- Unrelated addresses on L1 migrating to the same beneficiary address on L2: if there have been multiple emitted `IndexerMigratedToL2` events with the same address as

the `l2Indexer` parameter for different L1 users, this could be an indication of a phishing attack against indexers who are migrating to L2.