OpenZeppelin | security

The Graph Protocol Drip Keeper Rewards Mechanism

The Graph

September 1, 2022

This security assessment was prepared by **OpenZeppelin**, protecting the open economy.

Table of Contents

Summary

The Graph team asked us to review and audit the smart contract changes from pull request #582 in the graphprotocol/contracts repository. This pull request introduces a new deployment of The Graph to Arbitrum Nitro which will co-exist with the deployment on the Ethereum Mainnet. The introduced changes modify the way that rewards are issued and distributed by adding a keeper reward.

Type	L1/L2 Bridge	Total Issues	14 (11 resolved)
Timeline	From 2022-08-01 To 2022-08-12	Critical Severity Issues	0 (0 resolved)
Languages	Solidity	High Severity Issues	3 (3 resolved)
		Medium Severity Issues	3 (2 resolved)
		Low Severity Issues	5 (4 resolved)
		Notes & Additional Information	3 (2 resolved)

Scope

We audited the graphprotocol/contracts repository at commit `120753f2`.

In scope were the following contracts with respect to the changes in PR #582:

```
contracts
├── l2
│   └── reservoir
│       ├── IL2Reservoir.sol
│       ├── L2Reservoir.sol
│       └── L2ReservoirStorage.sol
└── reservoir
    ├── L1Reservoir.sol
    └── L1ReservoirStorage.sol
```

Out of scope

Contracts not explicitly listed above were out of scope for this phase of the audit.

Overview of the change

Pull request #582 compliments pull request #571, which added deployment on Arbitrum One layer 2 and refactored rewards distribution to periodically drip rewards on L1 using Arbitrum's retryable tickets. Pull request #582 compliments this by adding a keeper reward to whoever called drip on L1, to incentivize and offset the gas costs of doing so. In addition, part of the keeper reward is distributed to whoever redeemed the retryable ticket on L2, if the ticket was not auto-redeemed.

Keepers can be indexers, or allowed addresses as specified by governance. The reward per block for calling drip is also set by governance, in addition to a minimum interval, which prevents drip from being called too often.

Security considerations around Arbitrum

Many of the state variables for the reward dripping mechanism are closely related (e.g. the drip intervals, min drip intervals, and rewards per block), thus users rely on governance to set the variables to the correct values to ensure the correct operation of the protocol. In addition, the drip rewards mechanism is dependent on external market factors such as the current gas cost, the market value of GRT and the current price of ETH since these factors affect the economic incentives of calling the `_drip` function. The goal of the protocol is to have the `_drip` function called in regular intervals that range from 4 to 7 days. This requires monitoring from governance to ensure that the different variables of the mechanism are updated to keep the keeper incentive aligned with the desired drip time window. Our findings are a reflection of the complexity of this critical mechanism.

Regarding L2 more specifically, sending transactions from L1 to Arbitrum's L2 requires complex gas calculations to occur off-chain, with the risk being that a L2 transaction fails due to insufficient gas parameters. The codebase accounts for this by generating Arbitrum retryable transaction tickets. However, the guarantee that retryable tickets always succeed in being created was only recently introduced in Nitro so it has not been time-tested to ensure its reliability. Deploying a system that has communication between a L1 and a L2 is inherently dangerous. However, that is especially relevant in Arbitrum's case as the network itself is still in beta. This means that the behavior of the L2 could be unpredictable or even change and create integration issues.

Everything previously mentioned makes the drip rewards mechanism highly dependent on the correct functioning of Arbitrum Nitro network as well as The Graph team governance participation. For this reason, we strongly recommend that The Graph team deploy monitoring solutions to fully ensure that the state sync between L1 and L2 is working as expected.

Findings

Here we present our findings.

Several high severity issues were identified in the `_drip` rewards mechanism which are rooted in unvalidated assumptions regarding the integration with Arbitrum Nitro, insufficient coverage by test cases and a high complexity of the mechanism in general.

High Severity

H-01 Missing L2 address alias conversion during auto-redeem

The `getCurrentRedeemer` function call in the `receiveDrip` function in `L2Reservoir.sol` is used to determine the address that is redeeming the current retryable transaction ticket. This informs `receiveDrip` of whether the entire calculated `keeperReward` goes to the `_l1Keeper` address or whether a portion of the reward will go to the current redeemer address in addition to the `_l1Keeper`. If the redeemer address returned by `getCurrentRedeemer` is the same address as the `l1ReservoirAddress`, the entire reward is sent to the `_l1Keeper` address parameter as this implies that the current retryable ticket is being auto-redeemed. However, the if statement does not take into account that if a retryable ticket is being auto-redeemed, `getCurrentRedeemer` will return the fee refund address of the retryable ticket. In this instance, the refund address will be the `L1Reservoir` contract address. However, when L1 contract addresses are designated as refund addresses in a retryable ticket, the Arbitrum docs specify the following behavior:

> ...if the provided beneficiary or credit-back-address is an L1 contract address, the address will be converted to its address alias

Therefore, the `if` statement to check whether `redeemer != l1ReservoirAddress` will always be true because it is incorrectly assumed that `getCurrentRedeemer` would return the `l1ReservoirAddress` without translating the address to its L2 alias.

Consequently, in the situation where an auto-redeem is occurring, the `keeperReward` will be erroneously split between the `_l1Keeper` and the L2 alias of the `l1ReservoirAddress`, resulting in the keeper not receiving the full reward for an auto-redeem. The GRT rewards erroneously transferred to the L2 alias of the `l1ReservoirAddress` on Arbitrum would be difficult to recover as the L2 aliasing done by Arbitrum is specifically meant to prevent a direct mapping of L1 contracts to L2 contracts. The amount of rewards lost depends on the size of the `l2KeeperRewardFraction`. This bug disincentivizes auto-redemption because in order for keepers to receive the full `keeperReward` they would need to intentionally prevent an auto-redemption from occurring and instead manually redeem the retryable ticket on L2.

Consider using Arbitrum's helper function `applyL1ToL2Alias` to convert the `l1ReservoirAddress` to its L2 alias.

Update: *Resolved in PR #677 at commit d36aab9.*

H-02 Potential lack of transaction atomicity on L2 token transfer

In the `L1Reservoir` contract, the `_sendNewTokensAndStateToL2` function uses the `L1GraphTokenGateway` to send tokens to the `L2Reservoir`, and encodes a callhook that updates its state. However, the call to the `L1GraphTokenGateay` lacks error handling and a mechanism to verify that the L2 transaction was successful. This is problematic as the Arbitrum Nitro official docs about the base submission fee state:

> If an L1 transaction underpays for a retryable ticket's base submission fee *(sic)*, the retryable ticket creation on L2 simply fails.

This would break the atomicity between transactions on L1 and L2 and would result in a loss of synchronization between the state of L1 and L2.

Currently, it is not possible to know if the L2 transaction was successful until the Arbitrum team implements this feature in a future release. Consider waiting for this feature and implementing it once released, or creating a robust monitoring solution for this issue.

Update: Resolved.

The Graph was already aware of the current limitations of Arbitrum Nitro, as stated in GIP-0034, and has reiterating their intention to wait for the necessary Nitro upgrade before deploying the audited code.

Statement from The Graph:

> _This is actually addressed in GIP-0034 - We've explicitly marked the drip rewards change as dependent on the Nitro upgrade that will make retryable ticket transactions revert in L1 if the max submission cost is too low.

H-03 `keeperReward` is included twice when `l2RewardsFraction` is non-zero

The `keeperReward` is included twice when calculating `tokensToSendToL2` when `l2RewardsFraction` is non-zero and has not been updated. As part of the calculation of `tokensToSendToL2` for this specific case, `tokensToMint` is multiplied by the unchanged and non-zero `l2RewardsFraction` which is then divided by the

`FIXED_POINT_SCALING_FACTOR` in order to ensure the proper magnitude after the multiplication has occurred. Then `keeperReward` is added to the resulting fraction of `tokensToMint`; however, `tokensToMint` already contains the `keeperReward` as part of its own calculation. The `tokensToMint` is calculated earlier in the `_drip` function by subtracting `mintedRewardsTotal` from `newRewardsPlusMintedActual`, which is the summation of `newRewardsToDistribute`, `mintedRewardsActual`, and `keeperReward`.

Including the `keeperReward` in `tokensToMint` is necessary as the GRT reward for calling `drip` needs to be minted in addition to the calculated indexer rewards. However, by using `tokensToMint` in the calculation of `tokensToSendToL2` it incorrectly increases the total GRT rewards that will be sent to the `L2Reservoir` contract. The amount of extra GRT rewards sent to the `L2Reservoir` contract depends on the `l2RewardsFraction` where the total amount of GRT rewards sent is given by `keeperReward + (keeperReward * l2RewardsFraction)`. By including `keeperReward` twice in the calculation of `tokensToSendToL2`, the amount of GRT rewards sent to the `L2Reservoir` is larger than what was intended.

Consider reducing the `tokensToMint` by the `keeperReward` before multiplying by the `l2RewardsFraction`, in this case, to avoid sending too many tokens to the L2 reservoir.

Update: *Resolved in PR #685 at commit 2eea58c.*

H-04 Drip function will always revert in edge cases

Update: This issue was reclassified as Low in severity. Please see L-02 for details.

Medium Severity

M-01 Lack of validation

The following instances have been identified in the codebase where validation was lacking or could be improved:

- `setMinDripInterval` does not validate that `minDripInterval` is less than `dripInterval`. If `minDripInterval` is larger than `dripInterval`, the reservoir may dry out before the next `drip`.
- `setL1ReservoirAddress` does not validate that the address is not the zero address.
- `_drip` does not validate that the `_keeperRewardBeneficiary` is not the zero address. In addition, consider further restricting the address by requiring it be an indexer's or allowed dripper's address.
- `msg.value`, `_l2MaxGas`, `_l2GasPriceBid`, and `_l2MaxSubmissionCost` are not validated to be greater than zero when the `l2RewardsFraction` is greater than zero. This can cause otherwise auto-redeemed transactions to fail.

Consider adding validation to address the instances above and enforce more strict conditions around function parameters.

Update: Resolved.

The Graph team has let us know that `msg.value` and L2 gas parameters are validated by the bridge which was outside the scope of this audit. The remaining validations are fixed in PR #689 at commit f8fb06.

M-02 No fund recovery mechanism

The `L1Reservoir` is handling Ether and GRT, while going through drip cycles to remain in sync with L2. Depending on several factors, there is a potential for an imbalance in Ether or GRT to accrue. If funds should ever become stuck there is no fallback mechanism to retrieve them. Consider adding a generic, governor-only mechanism to transfer Ether or ERC20s out of the L1 reservoir.

Update: Resolved.

The Graph has acknowledged that for GRT there can be a surplus of tokens however, this situation would actually be beneficial for the protocol. It would provide some leeway before the `drip` *function must be called. In the case of ETH any surplus would inevitably be sent to the L1GraphTokenGateway which is the desired behavior.*

M-03 Insufficient test suite

The test suite does not perform proper integration tests with Arbitrum L1/L2 messaging and instead it relies on inaccurate mocks that do not represent the real behavior. This leads to a false sense of test coverage and is not reliable for ensuring code quality when refactoring the code base.

Consider adding a robust test suite that integrates with Arbitrum for proper insight on test cases.

Update: *The Graph has acknowledged this issue and is working on a detailed test plan as well as integrating a test suite that uses an actual Arbitrum environment.*

Low Severity

L-01 Unnecessary complexity in rewards distribution

In the `L1Reservoir` contract, the `_drip` function subtracts the `minDripInterval` from the `keeperReward`. This adds uneccesary complexity to the calculations that a `keeper` needs to make before calling `drip`. The reason for this is that, in order for a `drip` call to be profitable, the `keeperRewards` must be large enough to cover the cost of the transaction gas. To make this calculation, the `keeper` already has to take into account the current price of gas, Ether, and GRT.

By subtracting the `minDripInterval`, the `keeper` has to consider this variable as well in its calculation. This results in a delay when calling this function and introduces uneccesary complexity. This complexity makes it harder to predict when calling `drip` would be profitable and makes it more difficult to set the correct parameters such that the `drip` function is called on a timely basis.

Consider removing the subtraction of `minDripInterval` from the `keeperRewards` calculation.

Update: *Resolved in PR #675 at commit 00d454.*

L-02 Drip function will always revert in edge cases

In the `L1Reservoir` contract the `drip` function has a require statement that ensures the `L1Reservoir` does not send more tokens to `L2` in the case that the `l2RewardsFraction` has decreased. This check prevents the transaction from executing until enough time has passed to compensate the `l2OffsetAmount`.

However in the case that the `l2RewardsFraction` is set to zero from a large value while the `issuanceRate` is lowered at the same time, a `drip` call will then revert. This is because the `mintedRewardsTotal` would be larger than `mintedRewardsActual`, while `tokensToSendToL2` would be zero (due to the new rewards fraction), and the transaction will always revert in this check.

Consider refactoring the drip function so that it handles this edge case.

Update: *The Graph has acknowledged these issues, as they were raised in a previous audit. The Graph has taken steps to mitigate the issue and document proper function usage in comments.*

The Graph replied:

> *1) This is an edge case that governance would rarely introduce.*
>
> *2) The consequence is that drip will have to be called exactly at the time when the Reservoirs are running out of funds, so there should be no DoS (or in any case a very brief period where closing allocations will revert).*
>
> *3) Doing anything else would require making this quite more expensive, for little gain.*

L-03 Unstrict require inequality in drip function

In the `L1Reservoir` contract the `drip` function checks that `newRewardsPlusMintedActual` is larger or equal to `mintedRewardsTotal` to prevent trying to mint a negative amount of tokens.

However, because the inequality is not strict, the value of `tokensToMint` can be zero if the `mintedRewardsTotal` equals the `newRewardsPlusMintedActual`. This would not have a material impact as this conditional would prevent a call with zero tokens to mint. However, the transaction will not revert as likely intended.

Consider making the comparison strict or checking that the value `tokensToMint` is larger than zero.

Update: *Resolved in PR #685 at commit 2eea58c.*

L-04 Missing docstrings

Throughout the codebase there are several parts that do not have docstrings. For instance:

- Line 13 in `L2ReservoirStorage.sol`
- Line 25 in `L1ReservoirStorage.sol`

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

Update: *Resolved in PR #690 at commit 365e30.*

L-05 Require statement with multiple conditions

Within `L1Reservoir.sol` there is a `require` statement on line 65 that requires multiple conditions to be satisfied.

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

Update: *Resolved in PR #691 at commit 91a021.*

Notes & Additional Information

N-01 Lack of indexed parameters in events

None of the parameters in the events defined in the contracts are indexed. Consider indexing event parameters to avoid hindering the task of off-chain services searching and filtering for specific events.

Update: *Resolved in PR #692 at commit 80cc2f.*

N-02 Non-explicit imports are used

Throughout the codebase, global imports are being used. For instance:

- line 5 of `IL2Reservoir.sol`
- line 6 of `L2Reservoir.sol`
- lines 9-12 of `L2Reservoir.sol`
- lines 6-12 of `L1Reservoir.sol`

Non-explicit imports reduce code readability and could lead to conflicts between names defined locally and those imported. This is especially important if many contracts are defined within the same Solidity files or the inheritance chains are long.

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 PR #693 at commit b99d31.*

N-03 Files specifying outdated Solidity versions

Throughout the codebase there are `pragma` statements that use version `^0.7.6` of Solidity. Consider taking advantage of the latest Solidity version to improve the overall readability and security of the codebase. Regardless of which version of Solidity is used, consider pinning the version consistently throughout the codebase to prevent the opening of bugs due to incompatible future releases.

Update: *The Graph has acknowledged this is intentional.*

> *Version 0.7.6 is the Solidity version used consistently across the codebase. Updating this would require either changing contracts that are already deployed and that we can't or shouldn't unnecessarily update, or to adapt our tooling to support multiple Solidity versions which is out of scope for this change.*

Conclusions

We found three high severity issues in the drip rewards mechanism that are rooted in the integration with Arbitrum Nitro, insufficient coverage by test cases, and a high complexity of the mechanism in general.

We strongly recommend The Graph team consider refactoring the `_drip` function. During the audit we found this function to be quite complex and difficult to read. This impacts both the ability for external auditors to review this codebase as well as long term maintenance. For instance, issue H-03 in the `_drip` function was overlooked in our preliminary pass and only found later after multiple read-throughs.

Refactoring this function into smaller discrete functions, while avoiding nested if-else statements and adding more comments, will improve code readability and the quality of the codebase.

As a final recommendation, we think that The Graph team would benefit greatly from performing more integration tests on an Arbitrum Nitro environment.

Appendix

Monitoring Recommendation

Consider using a robust monitoring solution to track critical values, such as the contract reserves, the keeper rewards, and the last time that the `_drip` function was called. This can enable the team to react quickly if there are incidents and be alerted in cases where governance is required to step in.