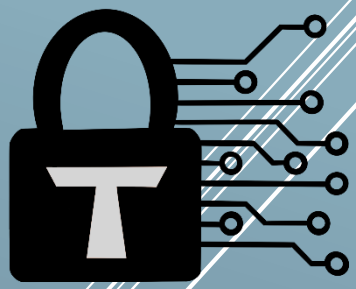
Trust Security

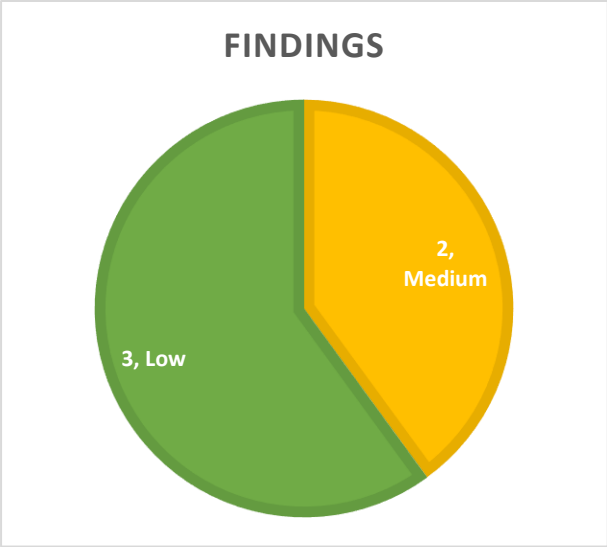


Smart Contract Audit

The Graph – IssuanceAllocator

16/12/25

Executive summary

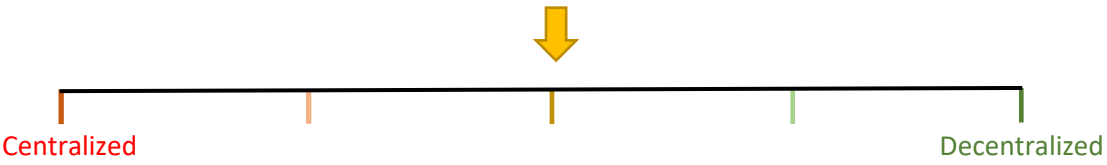


Category	Indexing
Audited file count	4
Auditor	Trust

Findings

Severity	Total	Open	Fixed	Acknowledged
High	0	0	-	-
Medium	2	0	2	-
Low	3	2	1	-

Centralization score



Signature

Digitally signed by Trust
Date: 2025.12.16 10:45:02 +02'00'

Document properties

Versioning

Version	Date	Description
0.1	16/12/25	Client report

Contact

Trust

trust@trust-security.xyz

Introduction

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

Scope

- DirectAllocation.sol
- IssuanceAllocator.sol
- RewardsManager.sol
- RewardsManagerStorage.sol

For files that have existed prior to the commit hashes below, only a differential audit has been undertaken.

Repository details

- **Repository URL:** https://github.com/graphprotocol/contracts
- **PR #1257 commit hash**: dbf5d2b7d98d829a8ff481344700b6733206859b
- **PR #1267 commit hash**: cbe2bd1eb018d1490bed669385170ad91ceb1f89
- **Additional commit hash:** 88ce412963add2e65259b5cc8e749e0c15c2fc78

About Trust Security

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

About the Auditors

Trust has established a dominating presence in the smart contract security ecosystem since 2022. He is a resident on the Immunefi, Sherlock and C4 leaderboards and is now focused in auditing and managing audit teams under Trust Security. When taking time off auditing & bug hunting, he enjoys sharing knowledge and experience with aspiring auditors through X or the Trust Security blog.

Disclaimer

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

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

Methodology

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

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Excellent**	Project kept code as simple as possible, reducing attack risks
Documentation	**Excellent**	Project is very well documented.
Best practices	**Excellent**	Project consistently adheres to industry standards.
Centralization risks	**Good**	Project employs RBAC mechanism to distribute permissions for sensitive operations.

Findings

Medium severity findings

TRST-M-1 Accounting configuration changes during pause period are likely to lead to unexpected distribution of tokens

- **Category:** Accounting issues
- **Source:** IssuanceAllocator.sol
- **Status:** Fixed

Description

The Allocator is designed so that when paused, funds are accumulated up to the point where an allocation configuration change occurs. Upon resumption, funds are distributed according to the new allocation set. The accumulation logic is provided below:

```
if (0 < blocksToAccumulate) {
    uint256 totalIssuance = $.issuancePerBlock * blocksToAccumulate;
    // There can be a small rounding loss here. This is acceptable.
    $.pendingAccumulatedAllocatorIssuance += (totalIssuance * (MILLION -
$.totalSelfMintingPPM)) / MILLION;
    $.lastAccumulationBlock = toBlockNumber;
}
```

The pending distribution logic is also provided:

```
for (uint256 i = 0; i < $.targetAddresses.length; ++i) {
    address target = $.targetAddresses[i];
    AllocationTarget storage targetData = $.allocationTargets[target];
    if (0 < targetData.allocatorMintingPPM) {
        // There can be a small rounding loss here. This is acceptable.
        // Pending issuance is distributed in proportion to allocator-
minting portion of total available allocation.
        uint256 targetIssuance = (pendingAmount *
targetData.allocatorMintingPPM) /
            (MILLION - $.totalSelfMintingPPM);
        GRAPH_TOKEN.mint(target, targetIssuance);
        emit IssuanceDistributed(target, targetIssuance);
    }
}
```

The key detail to note is that the pending amount increases by **totalIssuance** adjusted by the ratio of all allocations that are not self-minting, out of all allocations. The amount is later divided by the same ratio during distribution. The behavior results in two side effects:

- In case there is non-zero percentage of unused issuance, which is reduced or nullified during the configuration change, a particular issuance would retroactively grow during the pause period, coming out of the unused issuance pool. This happens despite the allocation percentage not being touched for that target.
- Any changes of **totalSelfMintingPPM** (even if unused percentage is untouched) will have the target receive a different value than their allocation percentage would suggest.

When governance is not aware of the intricacies and does not carefully simulate every configuration change, it could result in a permanent misallocation of GRT.

Recommended mitigation

For the first effect, changing the accounting to divide by the occupied allocation amount, rather than the remainder from **totalSelfMintingPPM**, would avoid over-allocating retroactively.

For the complex accounting issue, consider either documenting the formulas to be followed, or to adjust the accounting in code to maintain the desired property.

Team response

The code has been refactored so that at all times, the total allocation across targets remains 100%, simplifying calculations.

Mitigation review

The root cause of the issue has been addressed correctly. Other issues with the implementation are noted as separate items.

TRST-M-2 Change of the default allocator does not distribute leading to loss for previous allocator

- **Category:** Accounting issues
- **Source:** IssuanceAllocator.sol
- **Status:** Fixed

Description

When *setDefaultAllocationAddress()* is used, there is no distribution that occurs, regardless of whether the contract is paused. This means that in the non-pause case, the assumption that any change does not affect addresses retroactively is broken, and the new allocator will receive all the previous registered address's capacity.

Recommended mitigation

Trigger a distribution flow similarly to when setting a non-default allocation.

Team response

Fixed.

Mitigation review

Issue has been addressed with a *_handleDistributionBeforeAllocation()* call. The contract will still allow changing the address and apply it retroactively, but only if done by intention and during a pause period.

Low severity findings

TRST-L-1 Reward reclaiming efficiency is significantly limited

- **Category:** Logical flaws
- **Source:** RewardManager.sol
- **Status:** Open

Description

PR #1967 introduces reclaiming of rewards lost due to ineligibility. Without it, any rewards which cannot be claimed by an unauthorized indexer are permanently lost for everyone. The method where rewards are reclaimed through the *takeRewards()* flow is flawed, since a rational indexer who would not receive rewards would not trigger the rewarding logic in the first place.

Recommended mitigation

Consider adding calls to *takeRewards()* from StakingExtension and/or SubgraphService, so that any entity can force a reclaim of particular rewards if they are denied.

Team response

TBD

TRST-L-2 lastChangeNotifyBlock of default allocation target will be incorrect on some default address changes

- **Category:** Memory corruption
- **Source:** IssuanceAllocator.sol
- **Status:** Fixed

Description

When setting a default allocation target, the following code is executed:

```
// Notify both old and new addresses of the allocation change
_notifyTarget(oldAddress);
_notifyTarget(newAddress);
// Update the default allocation address at index 0
$.targetAddresses[0] = newAddress;
$.allocationTargets[newAddress] = $.allocationTargets[oldAddress];
delete $.allocationTargets[oldAddress];
```

Notification logic sets the **lastChangeNotifiedBlock** to the current block number if the **target** is not zero. The highlighted line copies the AllocationTarget values from the old to the new address, with the intention of copying the **allocatorMintingPPM** for the default target. However, this also copies the **lastChangeNotifiedBlock** from the **oldAddress**, which for the zero address, is never updated. The correct value should be the block.number, which it currently has already set from the new address *_notifyTarget()* call. Similarly, the logic would lead to zero-value default to have an updated **lastChangeNotifiedBlock**, despite that never being possible otherwise, or constituting coherent logic.

Higher-level impacts have not been achieved from the issue, so it remains an invariant violation.

Recommended mitigation

Instead of the affected line, directly copy only the **allocatorMintingPPM** from the old address.

Team response

Fixed.

Mitigation review

Issue has been addressed by copying the notification block for the new address after the struct overwrite.

TRST-L-3 Inaccurate view functions for the zero-address target

- **Category:** View-related issues
- **Source:** IssuanceAllocator.sol
- **Status:** Open

Description

The Allocator exposes several view functions to observe state. In *getTotalAllocation()*, the case of a zero-address for the default target is handled correctly. However, *getTargetAllocation()* and *getTargetIssuancePerBlock()* do not have customized logic for that case. Since the effective minting for it would be zero, the placeholder **allocatorMintingPPM** should be replaced by 0 in any calculations. This should only lead to concrete impacts under unusual integration scenarios.

Recommended mitigation

Address the calculations for the zero-address as described above.

Team response

TBD

Additional recommendations

TRST-R-1 Follow naming conventions for non-public functions

The codebase incorporates several private and internal functions, however they are not preceded by an underscore, for example *accumulatePendingIssuance()*. Consider adding those for clarity and to make the code less error-prone.

TRST-R-2 Emit events for all types of mints for visibility

In *_distributePendingIssuance()*, there are only events for allocator-minted amounts. Consider adding logging for self-mints as well, with a different event name.

TRST-R-3 Document security assumptions

- In *_distributePendingIssuance()*, division by zero is impossible since **allocatorMintingPPM** > 0 and so **totalSelfMintingPPM** must be at least 1 less than MILLION.
- The first issuance will actually execute for all blocks from 0 to block.number, which is safe because at that stage **issuancePerBlock** is guaranteed to be 0.
- In *_distributeIssuance()*, the formula for overflow should account for **allocatorMintingPPM**, which is up to MILLION, so could be of **type(uint236).max**.

TRST-R-4 Improve reentrancy protection for IssuanceAllocator

The IssuanceAllocator operates with notifications to targets, and notably CEI is not always respected in the contract code (for implementation reasons). While no concrete exploits have been identified, additional protection of a reentrancy guard would be significant in reducing likelihood of any issues arising. Note that functions initiated by governor role could still be reentered by malicious bots, if the role is fulfilled by a multi-sig and all signatures are known at that stage, for an example exploit weaponizing such a vector, the reader is referred to the Optimism report.

TRST-R-5 Clarify reclaim precedence

The code takes an opinionated step when rewards are dropped, in case both the subgraph deny list and the indexer eligibility test fail. It should be documented that the **subgraphDeniedReclaimAddress** is prioritized over **indexerEligibilityReclaimAddress** in that scenario.

TRST-R-6 Account for default allocator in _removeTargetFromList()

The code using pop-and-swap algorithm for removing a target from the list in *_removeTargetFromList()*. However, the loop begins from *i=0*, meaning if a target were to match in the first iteration, it would replace the default allocator due to the swapping logic. It is recommended to start the loop from *i=1*. The issue is only theoretical because **target** can never be the first address due to a previous validation.

TRST-R-7 Fix wrong documentation

In the docs, behavior of setters under pause is explained below:

"Governance functions like setIssuancePerBlock() and setTargetAllocation() still work. However, unlike changes made while unpaused, changes made will be applied from **lastIssuanceDistributionBlock** *rather than the current block."*

In fact, when changing issuance, or changing a target's self-mint allocation, this only applies from the current block, and previous rates are accumulated.

Centralization risks

TRST-CR-1 Governor is fully trusted

The Governor should be considered a fully trusted entity. They can assign any other roles, or permanently remove all roles from the system. Furthermore, they can trigger issuance of any amount of GRT tokens, to any address.

TRST-CR-2 Pauser is trusted

In case of a malicious PAUSE_ROLE, issuance for allocator-minted addresses would be suspended. This could eventually be addressed by removing the malicious pauser and nominating a new one.

TRST-CR-3 Operator is trusted

In case of a malicious operator, the DirectAllocation contract could be drained of any GRT tokens accrued