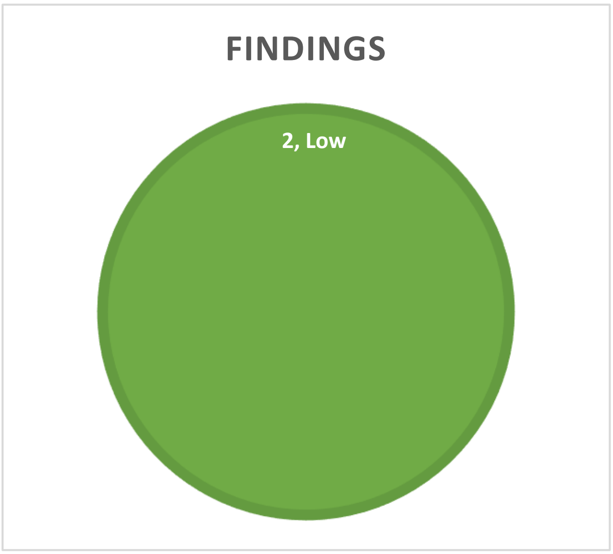
Trust Security

Smart Contract Audit

The Graph – RewardsEligibilityOracle

13/12/25

Executive summary

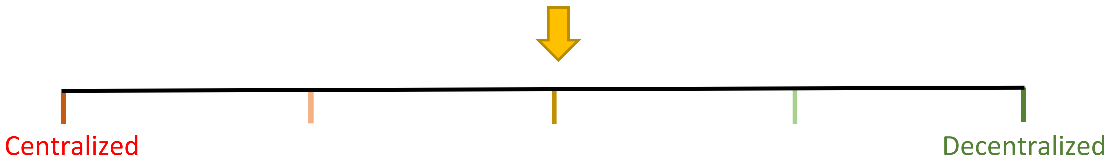


Category	Indexing
Audited file count	1
Lines of Code	116
Auditor	Trust

Findings

Severity	Total	Fixed	Acknowledged
High	0	-	-
Medium	0	-	-
Low	2	2	-

Centralization score



Signature

Digitally signed by Trust
Date: 2025.12.13 21:40:46 +02'00'

Document properties

Versioning

Version	Date	Description
0.1	05/12/25	Client report
0.2	13/12/25	Mitigation review

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

- RewardsEligibilityOracle.sol

Repository details

- **Repository URL:** https://github.com/graphprotocol/contracts
- **PR #1256 commit hash**: fec6aa724a9e8891d30254255a241d81596a8e29
- **Mitigation review commit hash:** 1bdf837f7a3de1407894d392e02239d75b0c89fa

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

Low severity findings

TRST-L-1 Usage of zero-initialized memory may lead to incorrect eligibility check in edge cases

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

Description

The *isEligible()* view function of the Eligibility oracle is the key interface to derive whether an address is eligible for rewards. The following check is implemented:

```
return block.timestamp < $.indexerEligibilityTimestamps[indexer] +
$.eligibilityPeriod;
```

If an indexer has never been registered, the expected behavior is that *isEligible()* would return false at all times, which is supported by the documentation:

"All indexers are initially ineligible for rewards until their eligibility is explicitly renewed by an authorized oracle."

Consider the case where the eligibility period is set to a very large value (for example, to signify that eligibility is permanent once set). The check could return **true** for a non-registered indexer, if the inequality **block.timestamp < eligibilityPeriod** holds.

Recommended mitigation

One of the following fixes is recommended:

- If the timestamp value for an **indexer** is 0, return **false** immediately.
- When setting an **eligibilityPeriod**, ensure it is smaller than current block timestamp.

Team response

Addressed by adding explicit documentation of the behavior.

Mitigation review

Document covers the discussed case comprehensively, so if the issue occurs responsibility would fall on the operator.

TRST-L-2 Race condition between configuration changes and reward claiming could lead to users permanently losing rewards

- **Category:** Race conditions
- **Source:** RewardsEligibilityOracle.sol, RewardsManager.sol
- **Status:** Fixed

Description

In the RewardsManager, if an **indexer** is considered not eligible for rewards, the rewards are marked as spent yet are not dispatched. This behavior is only safe when the caller certainly intends to lose the rewards, but that is not necessarily the case:

- From the time a user's transaction is queued, until it is executed, a configuration change could for example reduce the eligibility window, invalidating the user's claim.
- Delays due to gas costs or congestion could make the TX execute when the user is no longer eligible.

Recommended mitigation

Ideally, the *takeRewards()* call would accept a **forfeitRewards** flag to denote accepting the loss, but that is not backwards compatible. Consider documenting the risks above, and ensuring users are fully aware of any configuration changes by implementing a two-step process in the RewardsEligibilityOracle.

Team response

Addressed by adding explicit documentation of the behavior.

Mitigation review

Document covers the discussed case comprehensively, including remediation plan for operators.

Centralization risks

TRST-CR-1 Operator is trusted

In case of a malicious operator, permanent damage could be caused to the RewardManager:

- Any *takeReward()* call can be frontrun by setting a zero eligibility period, forcing lost rewards as detailed under L-2 but not limited by conditions.
- Any number of malicious **ORACLE_ROLE** accounts could be nominated.
- Allowlisting any otherwise ineligible indexer.

TRST-CR-2 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.

It should be mentioned that the RewardManager can still assign a fresh RewardsEligibilityOracle in emergencies.

Systemic risks

TRST-SR-1 Malicious indexers cannot be invalidated for entire eligibility period

The contract operates by marking a window of size **eligibilityPeriod** from the indexer's timestamp as valid. It represents a commitment to allow claiming, even if the indexer is being malicious for the entire period. It is therefore important to configure a reasonable period to limit the effect of such abuse.

TRST-SR-2 Eligibility design commits to potentially significant gas expenditures

The RewardsEligibilityOracle requires periodic maintenance by the **ORACLE_ROLE**. For each **indexer** the timestamp must be updated for each successive period. In case new indexers become eligible or ineligible during a window, fragmentation will occur and batching would be limited. Without an upper cap on eligible indexers, the design commits to a significant gas burden, which could be aggravated by artificial indexers.

TRST-SR-3 Indexers don't have flexibility on reward claiming

In most designs of reward claiming schemes, once a user has accrued certain rewards, they can claim them at any time. In the EligibilityOracle, a user must maintain a valid *isEligible()* pass to receive rewards. So, an **indexer** wishing to retire has a specific claiming window, and if it elapses, they will silently lose the rewards (as highlighted in L-2). If the current design is kept, ensure users are aware of the time sensitivity of claiming as to not lose out on rewards.

TRST-SR-4 Capital efficiency of rewards is reduced by ineligible indexers

The RewardManager operates by withholding the rewards of ineligible users. It performs no accounting of the unspent amount, so all ineligible rewards are forever lost, instead of shared by the eligible indexers. While this simplifies the engineering aspects, it should be understood that it makes griefing attacks on indexers more effective, allowing unlimited dilution of rewards by malicious actors (still subject to economic costs of receiving the proportional allocations).