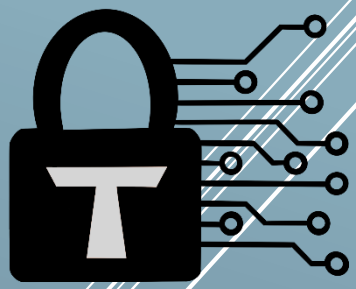
Trust Security

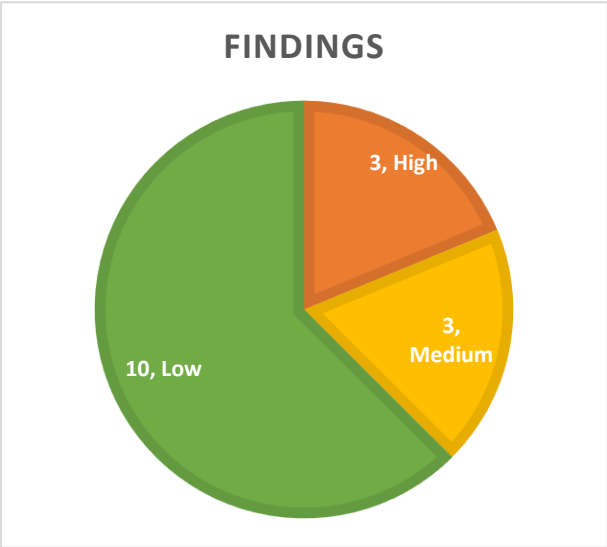


Smart Contract Audit

The Graph – Horizon Update

13/08/25

Executive summary

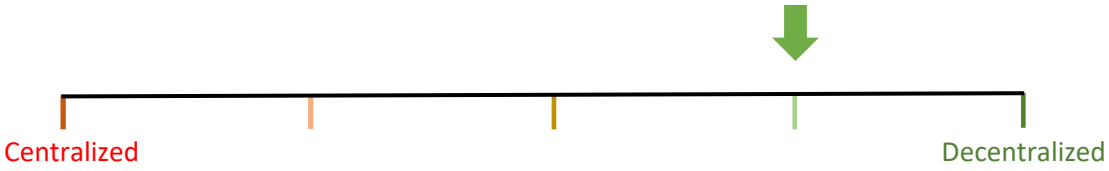


Category	Service Platform
Auditor	Trust
Time period	09/06/25-26/06/25

Findings

Severity	Total	Fixed	Acknowledged
High	3	3	-
Medium	3	3	-
Low	10	4	6

Centralization score



Signature

Digitally signed by Trust
Date: 2025.08.13 12:49:39 +03'00'

Document properties

Versioning

Version	Date	Description
0.1	26/06/25	Client report
0.2	08/08/25	Mitigation review
0.3	13/08/25	Mitigation review #2

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

Non-test files under:

- packages/horizon/contracts/**
- packages/subgraph-service/**

Repository details

- **Repository URL:** https://github.com/graphprotocol/contracts
- **Commit hash:** a7fb8758ffccad6a6f80dadcc68b12306ac0f615
- **Mitigation review:**
  - b53ca01e3837392d80cc66050443dfd418e51eba
  - 7695c9ec5f03ed265f6f78fc80e2a192d83db823
  - 8048c4cbb45d3cb6c40444beb140e3882365eaeb
  - 29dfdccadf74dce4b7a52ae658328ad026f59c9c
  - 345cfc8d6331e19e4e16900bde3b9348624b123c
  - 8b2e93a342fd1b5e22b3b314927849699e108c33
  - aac9f8b7d9db82d854b73dd3c2c140e256ba13d4
  - 836c0c2ec01551a4cc09cd5143f88eab62e8ea9b
  - e3d2787b6d123b53ff87cebdc5e735403f5157a9
  - 308d6e6d07e6dfcf499494f47d06f8f9580bf1a7
  - 17b794e49c8144558210a04e93230761cfd7161f
  - b492251395565ab97869b9e3e34b5840c1e6eb18
- **2nd Mitigation review:**
  - 0e469beeba0ec433e313be8c9129bcf99acdaac6

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	**Good**	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 does not introduce significant unnecessary centralization risks.

Findings

High severity findings

TRST-H-1 Service providers would not be able to collect indexing fees when agreement is cancelled by payer

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

Description

The RecurringCollector intends for the service provider to be able to collect payments until the contract is cancelled by a payer.

```
require(
    agreement.state == AgreementState.Accepted || agreement.state
== AgreementState.CanceledByPayer,

RecurringCollectorAgreementIncorrectState(_params.agreementId,
agreement.state)
);
```

However, in IndexingAgreement *collect()* it is ensured the agreement state is **Accepted**.

```
function _isActive(AgreementWrapper memory wrapper) private view
returns (bool) {
    return
        wrapper.collectorAgreement.dataService == address(this)
&&
        wrapper.collectorAgreement.state ==
IRecurringCollector.AgreementState.Accepted &&
        wrapper.agreement.allocationId != address(0);
}
```

Service providers would not be able to collect the rewards for that time period.

Recommended mitigation

Accept an agreement in **CanceledByPayer** state in IndexingAgreement *collect()*.

Team response

Fixed.

Mitigation review

The issue has been addressed by the IndexingAgreement having synced collection criteria with the RecurringCollector.

TRST-H-2 Anyone may collect the indexing fees of an indexer

- **Category:** Access control issues
- **Source:** SubgraphService.sol
- **Status:** Fixed

Description

When collecting fees the SubgraphService *collect()* is called with an **agreementId** which reaches *_collectIndexingFees()* and then IndexingAgreement *collect()*. It ensures the matching **allocationId** belongs to the **serviceProvider** from the agreement. However, it is missing authentication that the indexer calling *collect()* is the owner of the **allocationId**. Thus, an attacker could simply register as an indexer and call *collect()* on another indexer's agreement and receive their rewards. The payment destination is passed down from SubgraphService *collect()* and is controlled by the attacker.

Recommended mitigation

Ensure the allocation indexer matches the caller of *collect()*.

Team response

Issue has been addressed as suggested.

TRST-H-3 Attacker can avoid payment for services by crafting a malicious agreement

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

Description

The RecurringCollector is susceptible to an escrow bypass exploit similar to issue TRST-H-1 of the previous Horizon audit, previously affecting the TAPCollector.

The *collect()* function of RecurringCollector does not validate there is a trust relationship between the service provider and the data service of the collected agreement, which is the msg.sender. An agreement can be signed by an attacker with a victim service provider, nominating a malicious data service. They could then call *collect()* and pass a data service cut close to 100%. As a result, almost all escrow funds can be exfiltrated, while a legitimate agreement for the (payer,provider,collector) tuple will not collectable.

Recommended mitigation

The issue can be solved similar to the TallyGraphCollector's data service check:

```
{
    uint256 tokensAvailable =
_graphStaking().getProviderTokensAvailable(
        signedRAV.rav.serviceProvider,
        signedRAV.rav.dataService
    );
    require(tokensAvailable > 0,
```

```
GraphTallyCollectorUnauthorizedDataService(signedRAV.rav.dataServ
ice));
}
```

Team response

Fixed.

Mitigation review

Fixed as recommended.

Medium severity findings

TRST-M-1 Wrong TYPEHASH string is used for agreement updates, limiting functionality

- **Category:** Typo errors
- **Source:** RecurringCollector.sol
- **Status:** Fixed

Description

The RecurringCollector uses the following structure for an agreement update:

```
struct RecurringCollectionAgreementUpdate {
    bytes16 agreementId;
    uint64 deadline;
    uint64 endsAt;
    uint256 maxInitialTokens;
    uint256 maxOngoingTokensPerSecond;
    uint32 minSecondsPerCollection;
    uint32 maxSecondsPerCollection;
    bytes metadata;
}
```

However, the structure EIP-712 TYPEHASH is defined below:

```
bytes32 public constant EIP712_RCAU_TYPEHASH =
    keccak256(
        "RecurringCollectionAgreementUpdate(bytes16
agreementId,uint256 deadline,uint256 endsAt,…
    );
```

The type mismatch would cause parties producing an agreement update hash from the correct structure to fail.

Recommended mitigation

Use the same types as the struct definition.

Team response

Fixed.

Mitigation review

Issue has been addressed for both RCAU and RCA typehashes.

TRST-M-2 Collection for an elapsed or canceled agreement could be wrong

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

Description

The indexing agreement calculates the amount of tokens to collect in *_tokensToCollect()*:

```
function _tokensToCollect(
    StorageManager storage _manager,
    bytes16 _agreementId,
    IRecurringCollector.AgreementData memory _agreement,
    uint256 _entities
) private view returns (uint256) {
    IndexingAgreementTermsV1 memory termsV1 =
_manager.termsV1[_agreementId];
    uint256 collectionSeconds = block.timestamp;
    collectionSeconds -= _agreement.lastCollectionAt > 0 ?
_agreement.lastCollectionAt : _agreement.acceptedAt;
    return collectionSeconds * (termsV1.tokensPerSecond +
termsV1.tokensPerEntityPerSecond * _entities);
}
```

Note that the end time is assumed to be block.timestamp, but the correct time could be earlier if the agreement is canceled or elapsed, as calculated in the RecurringCollector:

```
uint256 collectionEnd = canceledOrElapsed ?
Math.min(canceledOrNow, _agreement.endsAt) : block.timestamp;
```

It is intended for the RecurringCollector to narrow the collection token total if needed, but this would still result in a wrong calculation of the RecurringCollector amount is larger than the IndexingAgreement amount. That could well be the case because the indexing-layer **tokensPerSecond** could be lower than the collector-layer **maxOngoingTokensPerSecond**.

Recommended mitigation

Use the same duration calculation in IndexingAgreement.

Team response

Fixed.

Mitigation review

The core issue has been addressed, but during refactoring a minor issue surfaced.

```
/**
 * @notice Get collection info for an agreement
 * @param agreement The agreement data
 * @return isCollectable Whether the agreement is in a valid
state that allows collection attempts,
 * not that there are necessarily funds available to collect.
 * @return collectionSeconds The valid collection duration in
seconds (0 if not collectable)
 */
function getCollectionInfo(
    AgreementData memory agreement
) external view returns (bool isCollectable, uint256
collectionSeconds);
```

The new *getCollectionInfo()* function returns **isCollectable** which is documented to mean the state allows collection. However, in case the starting and ending collection time coincide, the function will return true:

```
if (collectionEnd < collectionStart) {
    return (false, 0);
}
collectionSeconds = collectionEnd - collectionStart;
return (isCollectable, collectionSeconds);
```

While the RecurringCollector would in fact be blocking such a collection in *collect()*:

```
require(
    collectionSeconds > 0,
    RecurringCollectorZeroCollectionSeconds(_params.agreementId,
block.timestamp, agreement.lastCollectionAt)
);
```

It is recommended to have the same logic in both locations to avoid integration issues.

Team response

Fixed.

Mitigation review

The locations in code are now aligned and a separate reason variable has been added for context.

TRST-M-3 Updates could be submitted in unexpected order leading to deficiencies in payment

- **Category:** Signature reuse attacks
- **Source:** RecurringCollector.sol
- **Status:** Fixed

Description

In the RecurringCollector, each agreement by design can be updated multiple times. However the structure lacks a nonce, and the same update can be submitted again, for example the sequence (Update-1, Update-2, Update-1) would be accepted. A payer would expect Update-2 payment structure to be active, while in fact they would be paying under the Update-1 plan.

Recommended mitigation

Introduce a nonce to the Update structure.

Team response

Fixed.

Mitigation review

Fixed as recommended.

Low severity findings

TRST-L-1 If voteTimeLimit is set to a large value, votes would not be able to be passed

- **Category:** Underflow issues
- **Source:** SubgraphAvailabilityManager.sol
- **Status:** Acknowledged

Description

In *checkVotes()*, the starting timestamp from which a vote can be considered valid is calculated below:

```
// timeframe for a vote to be valid
uint256 voteTimeValidity = block.timestamp - voteTimeLimit;
```

In case **voteTimeLimit** is configured to be very large (greater than block.timestamp), the line above will underflow and it will not be possible to pass a vote. The expected behavior for a large limit is to count a vote from *any* timestamp.

Recommended mitigation

In case the limit is larger than the current timestamp, set **voteTimeValidity** to zero.

Team response

Acknowledged, will address these after the initial Horizon deployment. Will at least document that **voteTimeLimit** should not be set to such high value.

TRST-L-2 The indexer may not be able to collect fees if they are not programmed to release the allocations

- **Category:** Out-of-gas issues
- **Source:** SubgraphService.sol
- **Status:** Acknowledged

Description

In *collectQueryFees()*, *_releaseStake()* is called to release matured stakes. This could lead to an expensive loop which reverts due to out of gas. For that reason, a safety hatch was designed where *releaseStake()* of the DataServiceFees parent contract could be called with a controlled iteration count. However, this can only be called by the indexer. That breaks the pattern where indexer functions on the Subgraph can be called by an authorized party. Note that in HorizonStaking, there is a similar safety hatch in *deprovision()*, which is callable by any authorized user. In the worst case, the indexer is a smart contract which does not have a programmed call to *releaseStake()*, and thus cannot clear the funds.

Recommended mitigation

In the SubgraphService wrap the *releaseStake()* function in a way that allows an authorized account to call it.

Team response

Acknowledged, will address these after the initial Horizon deployment

TRST-L-3 Attacker can block new agreements from being created

- **Category:** Frontrunning attacks
- **Source:** IndexingAgreement.sol, RecurringCollector.sol
- **Status:** Fixed

Description

Agreements are accepted through *IndexingAgreement.accept()*, which passes an **agreementID** that has to be unused. An attacker observing the mempool or otherwise capable of predicting an **agreementID** pattern is able to preemptively register placeholder agreements between two attacker entities. At this point, the honest *accept()* call would fail. The issue also exists at the RecurringCollector layer and may be abused in other data services.

Recommended mitigation

Do not allow arbitrary agreementIDs, instead generate them using strong identifiers for the payer, data service, provider, and nonces of parties.

Team response

The issue has been addressed as suggested.

TRST-L-4 A disputed transaction may not be fully identifiable by the given parameters

- **Category:** Logical flaws
- **Source:** DisputeManager.sol
- **Status:** Acknowledged

Description

In the DisputeManager a dispute with a **poi** uses block.number as an identifier for the **disputeID** generation. The intention is to determine a unique instance which can be disputed, however at the same block the same **poi** could be submitted multiple times. The intention is to identify a unique slashable transaction, so another identifier should be used.

Recommended mitigation

Add the hash of the offending transaction, or the transaction index in the encompassing block.

Team response

WONT FIX - Reasoning:

- For Indexing Payments, only the first of those POIs will have collected funds, so we can safely assume that'sthe one being disputed.
- Does this issue affect other disputes as well?

Mitigation Review

For legacy disputes, those are trusted and select an allocation ID, so there is no case for confusion. For attestation-based disputes, a request-response CID pair should be specific enough to resolve the response in question, but this type involves a more elaborate off-chain process which is out of scope for the smart contract review.

TRST-L-5 The RecurringCollector could narrow collection value by more than intended

- **Category:** Slippage issues
- **Source:** RecurringCollector.sol
- **Status:** Fixed

Description

The collection mechanism is designed so that an IndexingAgreement calculates the indexing fees, which are passed to the RecurringCollector. The Collector would narrow down the fees if they exceed the maximum allowable at the collector level. Note that regardless of the narrowing, the last collected timestamp is updated to the current time, and any difference between the expected and actual collected amount is forgotten in the contract state. While this may be intended in some cases, in others the difference, which depends on external factors like elapsed time until execution, is greater than expected. In essence there is unlimited slippage between the two contracts responsible for collection.

In other agreements integrating with RecurringCollector this could have more significant impact.

Recommended mitigation

A slippage parameter could be provided by the user. The RecurringCollector could check the difference does not exceed this parameter or it would revert the transaction.

Team response

Fixed.

Mitigation review

The issue has been addressed as suggested.

TRST-L-6 The Agreement metadata version is not checked properly

- **Category:** Typo issues
- **Source:** IndexingAgreement.sol
- **Status:** Fixed

Description

In *update()*, the wrapper agreement version is set to the passed metadata version.

```
wrapper.agreement.version = metadata.version;
```

The line above should have been a validation that the values are the same. Setting the wrapper value doesn't achieve anything as it's a memory variable.

Recommended mitigation

Change the line above to a **require** statement.

Team response

Fixed.

Mitigation review

Issue has been addressed as suggested.

TRST-L-7 The indexer can time update() and collect() calls to change historic payments

- **Category:** Time sensitivity issues
- **Source:** RecurringCollector.sol
- **Status:** Acknowledged

Description

An agreement represents a commitment of payer to pay fees until the **endsAt** timestamp. At regular intervals fees are collected through *collect()*, while *update()* can be used to update **endsAt**. There are several race-conditions between *update()* and *collect()*. Before **endsAt**:

- If *update()* is called before *collect()*, the duration between the previous collection time and the current block.timestamp is paid with the new pricing.
- If *collect()* is called before *update()*, that same duration is paid with the old pricing.

If an agreement is updated after **endsAt**:

- If *update()* is called before *collect()*, the duration between **endsAt** and the current block.timestamp is paid with the new pricing.
- If *collect()* is called before *update()*, the same duration is paid with the old pricing.

The indexer can therefore choose to maximize between the options above, while the payer has no control. However, the opportunity for gain is limited to the last collection period, as if the maximum duration is crossed no payment can be collected at all.

Recommended mitigation

The *update()* function could make the last collected time the current timestamp. Then at no point is pricing ambiguous, and the indexer cannot gain by ordering the transactions.

Team response

The behavior is documented as expected.

TRST-L-8 RecurringCollector agreements cannot be revoked leading to functionality limitations

- **Category:** Logical flaws
- **Source:** RecurringAgreement.sol
- **Status:** Acknowledged

Description

Once a RecurringCollector agreement is signed, it cannot be revoked, and the provider is the one that submits it. Therefore, a payer would have to wait until the deadline to sign another agreement in case it is withheld by the provider, or risk paying for multiple indexing agreements at the same time.

Recommended mitigation

Consider adding a nonce to the RCA structure and make it incrementable by the payer, or to introduce custom revoke functionality.

Team response

WONT FIX - Reasoning:

- Implementation would add complexity
- Existing mechanisms (deadlines, cancellation) provides sufficient mitigation

TRST-L-9 An agreement may not be cancelled when the matching allocation is closed

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

Description

In the SubgraphService, any time an allocation is closed, the code calls *_onCloseAllocation()* to cancel an agreement relating to the closed allocation. However, in case *_collectIndexingRewards()* is called and *presentPOI()* closes an allocation due to *_isOverAllocated()* check, it won't cancel the matching agreement.

Recommended mitigation

Add the *_onCloseAllocation()* call in this case as well. It is not possible to directly call it from the AllocationHandler library, so it may need to be checked explicitly in SubgraphService after *presentPOI()* completes.

Team response

Fixed.

Mitigation review

Issue has been addressed as suggested.

TRST-L-10 Rewards will be distributed for zero-epoch allocations

- **Category:** Logical flaws
- **Source:** HorizonStakingExtension.sol
- **Status:** Acknowledged

Description

In *closeAllocation()* of HorizonStakingExtension, it is now allowed to close a zero-epoch allocation. However, there is no limitation to reward distribution in this scenario:

```
// Process non-zero-allocation rewards tracking
if (alloc.tokens > 0) {
    // Distribute rewards if proof of indexing was presented by
the indexer or operator
    if (isIndexerOrOperator && _poi != 0) {
        _distributeRewards(_allocationID, alloc.indexer);
    } else {
```

A bot can get rewards since last collection without waiting a complete epoch, by immediately withdrawing their allocation after reward accrual.

The impact is limited as during the Horizon period new legacy allocations cannot be opened.

Recommended mitigation

In the case **epochs == 0**, skip reward distribution.

Team response

Acknowledged, will address these after the initial Horizon deployment.

Additional recommendations

TRST-R-1 Documentation errors

- RewardsManager *getRewards()* – Should add documentation of rewardIssuer.
- IndexingAgreementDecoder *decodeIndexingAgreementTermsV1()* – wrong revert string.
- RecurringCollector *accept()/update()/cancel()* – mentions indexing agreement but this relates to a generic RCA agreement.

TRST-R-2 Collection may revert due to external conditions

In GraphPayments *collect()*, if **receiverDestination** is zero, it will attempt staking the rewards using *stakeTo()*. However, if HorizonStaking is paused, it would cause the collection to revert. Consider either documenting the risk, or adding safe handling in this scenario.

TRST-R-3 Reusing of legacy disputes could overwrite storage

In legacy disputes, a second dispute of the same legacy **allocationId** would result in the same **disputeId**. It is recommended to ensure those are unique to avoid overwriting values of previous slashes. Note that the function requires high permissions.

TRST-R-4 Avoid CEI violations in RecurringCollector

In RecurringCollector *_collect()*, the effect of setting **lastCollectionAt** to the current time is done after various interactions in PaymentEscrow *collect()*. If the contract could be reentered, it would collect the same time period multiple times. Consider setting it before the *collect()* call.

TRST-R-5 Improve validation of IndexingAgreement parameters

The IndexingAgreement configuration is handled both in IndexingAgreement and the RecurringCollector. The contract could introduce sanity checks to make sure the **tokensPerSecond**, **tokensPerEntityPerSecond** values are in line with the **maxOngoingTokensPerSecond** value of RecurringCollector.

TRST-R-6 Allow for future non-zero service cuts of IndexingAgreement

The *collect()* function in IndexingAgreement calls RecurringCollector *collect()* passing a fixed **dataServiceCut** of 0. It may be useful to allow this value to be configured by Governance to avoid redeployment and user intervention of provisioning a new data service in case that is desired in the future.

TRST-R-7 Allow better control for authorized user of indexer

Most functions in SubgraphService allow an authorized user of the Indexer to perform various operations on its behalf. However, the *setPaymentsDestination()* function is used by the indexer and operates on msg.sender. Consider refactoring it to allow an authorized user.

TRST-R-8 Dispute identifiers are not protected from collisions between different dispute types

Each dispute creation function constructs an identifier using different parameters:

```
bytes32 disputeId = keccak256(abi.encodePacked(allocationId,
"legacy"));
```

```
bytes32 disputeId = keccak256(
    abi.encodePacked(
        _attestation.requestCID,
        _attestation.responseCID,
        _attestation.subgraphDeploymentId,
        indexer,
        _fisherman
    )
);
```

```
bytes32 disputeId = keccak256(abi.encodePacked(_allocationId,
_poi, _blockNumber));
```

```
// Create a disputeId
bytes32 disputeId = keccak256(
    abi.encodePacked("IndexingFeeDisputeWithAgreement",
_agreementId, _poi, _entities, _blockNumber)
);
```

It should be enforced via construction that the preimages from different methods cannot collide, for example by prefixing with a dispute type code.

TRST-R-9 The SubgraphService register() function is error-prone

In SubgraphService, the *register()* operation sets the payment destination unless the parameter is zero:

```
if (paymentsDestination_ != address(0)) {
    _setPaymentsDestination(indexer, paymentsDestination_);
}
```

Note that the previous value may be non-zero, and it is intended to now set it to zero (if the previous value is non-zero and the new value is non-zero, it indeed changes the state). A zero payment address is used to choose the staking option in GraphPayments. Consider either documenting the intended behavior, or removing the condition above.

TRST-R-10 Redundant check in PaymentsEscrow introduces risks

In PaymentsEscrow, there is a pre/post balance check for the *collect()* call.

```
uint256 escrowBalanceBefore =
_graphToken().balanceOf(address(this));

_graphToken().approve(address(_graphPayments()), tokens);
_graphPayments().collect(paymentType, receiver, tokens,
dataService, dataServiceCut, receiverDestination);

// Verify that the escrow balance is consistent with the
collected tokens
uint256 escrowBalanceAfter =
_graphToken().balanceOf(address(this));
require(
    escrowBalanceBefore == tokens + escrowBalanceAfter,
    PaymentsEscrowInconsistentCollection(escrowBalanceBefore,
escrowBalanceAfter, tokens)
);
```

In case the GraphPayments.collect() call changes the GRT balance of PaymentsEscrow except the exact token amount, it would cause the *collect()* call to revert. This could happen if the Escrow is a registered receiver for some reason, or if somehow the contract receives a donation. Consider removing the balance check as it is not necessary, the GraphPayments contract is trusted.