Soroban RPC

Security Assessment

December 20th, 2024 — Prepared by OtterSec

Robert Chen r@osec.io

Tuyết Duong tuyet@osec.io

Table of Contents

01 — Executive Summary

Overview

Stellar engaged OtterSec to assess the `soroban-rpc` program. This assessment was conducted between November 27th and December 11th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 3 findings throughout this audit engagement.

In particular, we identified a vulnerability concerning the overflow of cursor (OS-RPC-ADV-00). Additionally, there is a lack of proper validation of the end ledger in the getter functions for retrieving the events and transactions, allowing it to be set to a value less than that of the start ledger or outside the valid ledger range (OS-RPC-ADV-01).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/stellar/stellar-rpc. This audit was performed against commit a60044e.

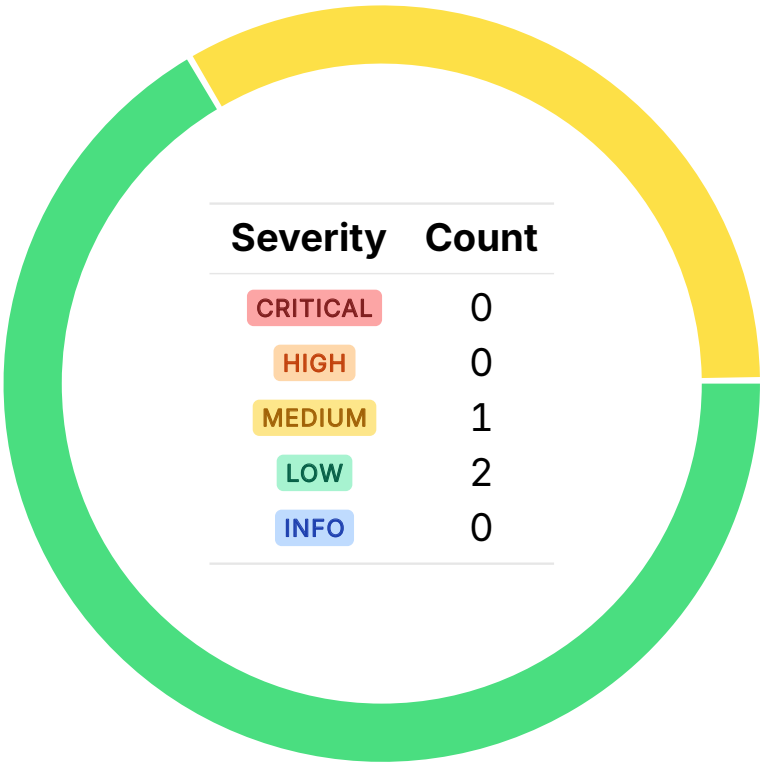
A brief description of the program is as follows:

Name	Description
soroban-rpc	The Soroban RPC program serves as a lightweight gateway for interacting with the Stellar Network.

03 — Findings

Overall, we reported 3 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	0
MEDIUM	1
LOW	2
INFO	0

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-RPC-ADV-00	MEDIUM	RESOLVED	`MaxCursor.String` results in a transaction order overflow due to exceeding the allowable range for `toid`. Also, there is an inconsistency in the definition of the maximum cursor value in `cursor` and `getEvents`.
OS-RPC-ADV-01	LOW	RESOLVED	There is a lack of proper validation of `EndLedger` in `getEvents` and `get_transactions`, allowing it to be set to a value less than `StartLedger` or outside the valid ledger range.
OS-RPC-ADV-02	LOW	RESOLVED	`BatchGetLedgers` may encounter an underflow if both `sequence` and `batchSize` are set to zero.

Inconsistency in the MaxCursor Definition and Overflow MEDIUM OS-RPC-ADV-00

Description

In the current implementation, there is a possible overflow scenario when representing the maximum possible cursor value as a string. The `MaxCursor` is the largest possible cursor. However, calling `MaxCursor.String` results in an overflow due to a `"transaction order overflow"` issue. When `MaxCursor.String` is called, the values of `Ledger`, `Tx`, and `Op` exceed the capacity of a signed 64-bit integer during the bit-shifting operations in `toid.New`. The combined value would require more than 64 bits, resulting in an overflow in `toid.New`.

>_ *stellar-rpc/internal/methods/get_events.go* GO

```go
func (h eventsRPCHandler) getEvents(ctx context.Context, request GetEventsRequest)
    ↪  (GetEventsResponse, error) {
    [...]
    else {
                // cursor represents end of the search window if events does not reach limit
                // here endLedger is always exclusive when fetching events
                // so search window is max Cursor value with endLedger - 1
                cursor = db.Cursor{Ledger: endLedger - 1, Tx: math.MaxUint32, Event:
                    ↪  math.MaxUint32 - 1}.String()
    }
    [...]
}
```

Additionally, there is an inconsistency in how `MaxCursor` is defined in `cursor` and `getEvents`. In `cursor`, it utilizes `math.MaxInt32`, but the max cursor returned in `getEvents` utilizes `math.MaxUint32` instead, as highlighted above.

Remediation

Add a check to ensure that the combined values of `Ledger`, `Tx`, and `Op` do not exceed 64 bits before shifting and combining. Also, ensure consistency by using the same maximum values for cursor fields throughout all parts of the code.

Patch

Fixed in PR#343

Inadequate Ledger Range Validation LOW OS-RPC-ADV-01

Description

The vulnerability arises from the fact that `getEvents` does not enforce that `EndLedger` should be greater than or equal to `StartLedger`, and both should fall within the bounds of the valid ledger range (`ledgerRange`). In `getEvents`, the `StartLedger` value is validated to ensure it is positive. However, `EndLedger` may be less than `StartLedger` or outside the `ledgerRange` boundaries, resulting in attempts to access invalid or non-existent data.

>_ stellar-rpc/internal/methods/get_transactions.go GO

```
// isValid checks the validity of the request parameters.
func (req GetTransactionsRequest) isValid(maxLimit uint, ledgerRange
    ↪  ledgerbucketwindow.LedgerRange) error {
    if req.Pagination != nil && req.Pagination.Cursor != "" {
        if req.StartLedger != 0 {
                return errors.New("startLedger and cursor cannot both be set")
        }
    } else if req.StartLedger < ledgerRange.FirstLedger.Sequence || req.StartLedger >
        ↪  ledgerRange.LastLedger.Sequence {
        return fmt.Errorf(
           "start ledger must be between the oldest ledger: %d and the latest ledger: %d
           for this rpc instance",
                     ledgerRange.FirstLedger.Sequence,
                       ledgerRange.LastLedger.Sequence,
        )
    }
    [...]
}
```

Furthermore, in `get_transactions::isValid` (shown above), when pagination is enabled (`req.Pagination` is not `nil` and `req.Pagination.Cursor` is set), the function skips validating whether the `StartLedger` falls within the acceptable range defined by `ledgerRange.FirstLedger.Sequence` and `ledgerRange.LastLedger.Sequence`. As a result, the system may attempt to query a ledger that does not exist or is invalid.

Remediation

Add validation to ensure that `EndLedger` is greater than or equal to `StartLedger`, and that it falls within the valid `ledgerRange` boundaries. Additionally, validate the ledger range even when utilizing pagination.

Patch

Fixed in PR#343

Batch Size Underflow Risk LOW OS-RPC-ADV-02

Description

There is a potential underflow in `ledger::BatchGetLedgers`, specifically when both `sequence` and `batchSize` are set to zero. In that case, in the following line: `sq.LtOrEq"sequence": sequence + uint32(batchSize) - 1`, the calculation of `sequence + uint32(batchSize) - 1` becomes `0 + 0 - 1 = -1`, resulting in an underflow.

>_ *stellar-rpc/internal/db/ledger.go* GO

```go
func (l ledgerReaderTx) BatchGetLedgers(ctx context.Context, sequence uint32,
        batchSize uint,
) ([]xdr.LedgerCloseMeta, error) {
    sql := sq.Select("meta").
        From(ledgerCloseMetaTableName).
        Where(sq.And{
                sq.GtOrEq{"sequence": sequence},
                sq.LtOrEq{"sequence": sequence + uint32(batchSize) - 1},
        })
    [...]
    return results, nil
}
```

Remediation

Verify the inputs to ensure that `batchSize` is greater than zero before performing the calculations.

Patch

Fixed in PR#343.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.