RTRTR User Manual

NLnet Labs

Oct 25, 2021

CONTENTS

RTRTR is an RPKI data proxy, designed to collect Validated ROA Payloads from one or more sources in multiple formats and dispatch it onwards. It provides the means to implement multiple distribution architectures for RPKI such as centralised RPKI validators that dispatch data to local caching RTR servers.

RTRTR can read RPKI data from multiple RPKI Relying Party packages via RTR and JSON and, in turn, provide an RTR service for routers to connect to. The HTTP server provides the validated data set in JSON format, as well as a monitoring endpoint in plain text and Prometheus format.

If you run into a problem with RTRTR or you have a feature request, please create an issue on Github. We are also happy to accept your pull requests. For general discussion and exchanging operational experiences we provide a mailing list and a Discord server.

CHAPTER

ONE

INTRODUCTION

For larger networks, RTRTR is an ideal companion to Routinator. For example, it is possible to centralise validation performed by Routinator and have RTRTR running in various locations around the world to which routers can connect.

RTRTR can read RPKI data from multiple RPKI Relying Party instances via RTR and JSON and, in turn, provide an RTR service for routers to connect to. The HTTP server provides the validated data set in JSON format, as well as a monitoring endpoint in plain text and Prometheus format. RTRTR also supports SLURM, so you can add your local exceptions to any instances you're pulling data from.

CHAPTER

TWO

INSTALLATION

2.1 System Requirements

When choosing a system to run RTRTR on, make sure you have 1GB of available memory and 1GB of disk space.

2.2 Quick Start

Getting started with RTRTR is really easy by either installing a binary package for Debian and Ubuntu or for Red Hat Enterprise Linux and CentOS. You can also run with Docker or build from Cargo, Rust's build system and package manager.

Deb Packages

RPM Packages

Docker

Cargo

If you have a machine with an amd64/x86_64 architecture running a recent Debian or Ubuntu distribution, you can install RTRTR from our software package repository.

To use this repository, add the line below that corresponds to your operating system to your `/etc/apt/sources.list` or `/etc/apt/sources.list.d/`:

```
deb [arch=amd64] https://packages.nlnetlabs.nl/linux/debian/ stretch main
deb [arch=amd64] https://packages.nlnetlabs.nl/linux/debian/ buster main
deb [arch=amd64] https://packages.nlnetlabs.nl/linux/debian/ bullseye main
deb [arch=amd64] https://packages.nlnetlabs.nl/linux/ubuntu/ xenial main
deb [arch=amd64] https://packages.nlnetlabs.nl/linux/ubuntu/ bionic main
deb [arch=amd64] https://packages.nlnetlabs.nl/linux/ubuntu/ focal main
```

Then run the following commands to add the public key and update the repository list:

```
wget -qO- https://packages.nlnetlabs.nl/aptkey.asc | sudo apt-key add -
sudo apt update
```

You can then install RTRTR by running:

```
sudo apt install rtrtr
```

You can now configure RTRTR by editing `/etc/rtrtr.conf` and start it with `sudo systemctl enable --now rtrtr`.

You can check the status of RTRTR with:

```
sudo systemctl status rtrtr
```

You can view the logs with:

```
sudo journalctl --unit=rtrtr
```

If you have a machine with an amd64/x86_64 architecture running a RHEL (Red Hat Enterprise Linux)/CentOS 7 or 8 distribution, you can install RTRTR from our software package repository.

To use this repository, create a file named `/etc/yum.repos.d/nlnetlabs.repo`, enter this configuration and save it:

```
[nlnetlabs]
name=NLnet Labs
baseurl=https://packages.nlnetlabs.nl/linux/centos/$releasever/main/$basearch
enabled=1
```

Then run the following command to add the public key:

```
sudo rpm --import https://packages.nlnetlabs.nl/aptkey.asc
```

You can then install RTRTR by running:

```
sudo yum install -y rtrtr
```

You can now configure RTRTR by editing `/etc/rtrtr.conf` and start it with `sudo systemctl enable --now rtrtr`.

You can check the status of RTRTR with:

```
sudo systemctl status rtrtr
```

You can view the logs with:

```
sudo journalctl --unit=rtrtr
```

To run RTRTR with Docker you will first need to create an `rtrtr.conf` file somewhere on your host computer and make that available to the Docker container when you run it. For example if your config file is in `/etc/rtrtr.conf` on the host computer:

```
docker run -v /etc/rtrtr.conf:/etc/rtrtr.conf nlnetlabs/rtrtr -c /etc/rtrtr.conf
```

RTRTR will need network access to fetch and publish data according to the configured units and targets respectively. Explaining Docker networking is beyond the scope of this quick start, however below are a couple of examples to get you started.

If you need an RTRTR unit to fetch data from a source port on the host you will also need to give the Docker container access to the host network. For example one way to do this is with `--net=host`, where `...` represents the rest of the arguments to pass to Docker and RTRTR:

```
docker run --net=host ...
```

If you're not using `--net=host` you will need to tell Docker to expoese the RTRTR target ports, either one by one using `-p`, or you can publish the default ports exposed by the Docker container (and at the same time remap them to high numbered ports) using `-P`:

```
docker run -p 8080:8080/tcp -p 9001:9001/tcp ...
```

Or:

```
docker run -P ...
```

Assuming you have a newly installed Debian or Ubuntu machine, you will need to install rsync, the C toolchain and Rust. You can then install RTRTR:

```
apt install curl rsync build-essential
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source ~/.cargo/env
cargo install --locked rtrtr
```

Once RTRTR is installed, you need to create a *configuration file* that suits your needs. The config file to use needs to be passed to RTRTR via the `-c` option, i.e.:

```
rtrtr -c rtrtr.conf
```

If you have an older version of Rust and RTRTR, you can update via:

```
rustup update
cargo install --locked --force rtrtr
```

2.3 Installing Specific Versions

Before every new release of RTRTR, one or more release candidates are provided for testing through every installation method. You can also install a specific version, if needed.

Deb Packages

RPM Packages

Docker

Cargo

To install release candidates of RTRTR, add the line below that corresponds to your operating system to your `/etc/apt/sources.list` or `/etc/apt/sources.list.d/`:

```
deb [arch=amd64] https://packages.nlnetlabs.nl/linux/debian/ stretch-proposed main
deb [arch=amd64] https://packages.nlnetlabs.nl/linux/debian/ buster-proposed main
deb [arch=amd64] https://packages.nlnetlabs.nl/linux/debian/ bullseye-proposed main
deb [arch=amd64] https://packages.nlnetlabs.nl/linux/ubuntu/ xenial-proposed main
deb [arch=amd64] https://packages.nlnetlabs.nl/linux/ubuntu/ bionic-proposed main
deb [arch=amd64] https://packages.nlnetlabs.nl/linux/ubuntu/ focal-proposed main
```

You can use this command to get an overview of the available versions:

```
sudo apt policy rtrtr
```

You can install a specific version using `<package name>=<version>`, e.g.:

```
sudo apt install rtrtr=0.1.1
```

To install release candidates of RTRTR, create an additional repo file named `/etc/yum.repos.d/nlnetlabs-testing.repo`, enter this configuration and save it:

```
[nlnetlabs-testing]
name=NLnet Labs Testing
baseurl=https://packages.nlnetlabs.nl/linux/centos/$releasever/proposed/$basearch
enabled=1
```

You can use this command to get an overview of the available versions:

```
sudo yum --showduplicates list rtrtr
```

You can install a specific version using `<package name>-<version info>`, e.g.:

```
sudo yum install -y rtrtr-0.1.1
```

All release versions of RTRTR, as well as release candidates and builds based on the latest main branch are available on Docker Hub.

For example, installing RTRTR 0.1.2 is as simple as:

```
docker run -it nlnetlabs/rtrtr:v0.1.2
```

All release versions of RTRTR, as well as release candidates, are available on crates.io, the Rust package registry. If you want to install a specific version of RTRTR using Cargo, explicitly use the `--version` option. If needed, use the `--force` option to overwrite an existing version:

```
cargo install --locked --force rtrtr --version 0.1.2
```

All new features of RTRTR are built on a branch and merged via a pull request, allowing you to easily try them out using Cargo. If you want to try the a specific branch from the repository you can use the `--git` and `--branch` options:

```
cargo install --git https://github.com/NLnetLabs/rtrtr.git --branch main
```

For more installation options refer to the Cargo book.

2.4 Installing From Source

You need a C toolchain and Rust to install and run RTRTR. You can install RTRTR on any system where you can fulfil these requirements.

2.4.1 C Toolchain

Some of the libraries RTRTR depends on require a C toolchain to be present. Your system probably has some easy way to install the minimum set of packages to build from C sources. For example, this command will install everything you need on Debian/Ubuntu:

```
apt install build-essential
```

If you are unsure, try to run **cc** on a command line. If there is a complaint about missing input files, you are probably good to go.

2.4.2 Rust

The Rust compiler runs on, and compiles to, a great number of platforms, though not all of them are equally supported. The official Rust Platform Support page provides an overview of the various support levels.

While some system distributions include Rust as system packages, RTRTR relies on a relatively new version of Rust, currently 1.52 or newer. We therefore suggest to use the canonical Rust installation via a tool called **`rustup`**.

To install **`rustup`** and Rust, simply do:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Alternatively, visit the official Rust website for other installation methods.

You can update your Rust installation later by running:

```
rustup update
```

2.4.3 Building

The easiest way to get RTRTR is to leave it to Cargo by saying:

```
cargo install --locked rtrtr
```

The command will build RTRTR and install it in the same directory that Cargo itself lives in, likely `$HOME/.cargo/bin`. This means RTRTR will be in your path, too.

CHAPTER

THREE

CONFIGURATION

RTRTR uses two classes of components: *units* and *targets*. Units take data from somewhere and produce a single, constantly updated data set. Targets take the data set from exactly one other unit and serve it in some specific way.

Both units and targets have a name — so that we can refer to them — and a type that defines which particular kind of unit or target this is. For each type, additional arguments need to be provided. Which these are and what they mean depends on the type.

Units and targets can be wired together in any way to achieve your specific goal. This is done in a configuration file, which also specifies several general parameters for logging, as well as status and Prometheus metrics endpoints via the built-in HTTP server.

Note: The configuration file is in TOML (Tom's Obvious Minimal Language) format, which is somewhat similar to INI files. You can find more information on the TOML website.

3.1 General Parameters

The configuration file starts out with a number of optional parameters to specify logging. The built-in HTTP server provides status information at the **`/status`** path and Prometheus metrics at the **`/metrics`** path. Note that details are provided for each unit and each target.

```
# The minimum log level to consider.
log_level = "debug"

# The target for logging. This can be "syslog", "stderr", "file", or "default".
log_target = "stderr"

# If syslog is used, the syslog facility can be given.
log_facility = "daemon"

# If file logging is used, the log file must be given.
log_file = "/var/log/rtrtr.log"

# Where should the HTTP server listen on?
http-listen = ["127.0.0.1:8080"]
```

3.2 Units

RTRTR currently has four types of units. Each unit gets its own section in the configuration. The name of the section, given in square brackets, starts with `units.` and is followed by a descriptive name you set, which you can later refer to from other units, or a target.

3.2.1 RTR Unit

The unit of the type `rtr` takes a feed of Validated ROA Payloads (VRPs) from a Relying Party software instance via the RTR protocol. Along with a unique name, the only required argument is the IP or hostname of the instance to connect to, along with the port. Because the RTR protocol uses sessions and state, we don't need to specify a refresh interval for this unit.

```
[units.rtr-unit-name]
type = "rtr"
remote = "validator.example.net:3323"
```

3.2.2 JSON Unit

Most Relying Party software packages can produce the Validated ROA Payload set in JSON format as well, either as a file on disk or at an HTTP endpoint. RTRTR can use this format as a data source too, using units of the type `json`. Along with specifying a name, you must specify the URI to fetch the VRP set from, as well as the refresh interval in seconds.

```
[units.json-unit-name]
type = "json"
uri = "http://validator.example.net/vrps.json"
refresh = 60
```

3.2.3 Any Unit

The `any` unit type is given any number of *other* units and picks the data set from one of them. Units can signal that they currently don't have an up-to-date data set available, allowing the `any` unit to skip those. This ensures there is always an up-to-date data set available.

Important: The `any` unit uses a single data source at a time. RTRTR does **not** attempt to make a union or intersection of multiple VRPs sets, to avoid the risk of making a route *invalid* that would otherwise be *unknown*.

To configure this unit, specify a name, set the type to `any` and list the sources that should be used. Lastly, specify if a random unit should be selected every time it needs to switch or whether it should go through the list in order.

```
[units.any-unit-name]
type = "any"
sources = [ "unit-1", "unit-2", "unit-3" ]
random = false
```

3.2.4 SLURM Unit

In some cases, you may want to override the global RPKI data set with your own local exceptions. You can do this by specifying route origins that should be filtered out of the output, as well as origins that should be added, in a file using JSON notation according to the SLURM (Simplified Local Internet Number Resource Management with the RPKI) standard specified in **RFC 8416**.

You can refer to the JSON file you created with a unit of the type `slurm`. As the source to which the exceptions should be applied, you must specify any of the other units you have created. Note that the `files` attribute is an array and can take multiple values as input.

```
[units.slurm]
type = "slurm"
source = "source-unit-name"
files = [ "/var/lib/rtrtr/local-expections.json" ]
```

3.3 Targets

RTRTR currently has two types of targets. As with units, each unit gets its own section in the configuration. And also here, the name of the section starts with `targets.` and is followed by a descriptive name you set, all enclosed in square brackets.

3.3.1 RTR Target

Targets of the type `rtr` let you serve the data you collected with your units via the RPKI-to-Router (RTR) protocol. You must give your target a name and specify the host name or IP address it should listen on, along with the port. As the RTR target can listen on multiple addresses, the listen argument is a list. Lastly, you must specify the name of the unit the target should receive its data from.

```
[targets.rtr-target-name]
type = "rtr"
listen = [ "127.0.0.1:9001" ]
unit = "source-unit-name"
```

3.3.2 HTTP Target

Targets of the type `http` let you serve the collected data via HTTP, which is currently only possible in `json` format. You can us this data stream for monitoring, provisioning, your IP address management, or any other purpose that you require. To use this target, specify a name and a path, as well as the name of the unit the target should receive its data from.

```
[targets.http-target-name]
type = "http"
path = "/json"
format = "json"
unit = "source-unit-name"
```

CHAPTER

FOUR

EXAMPLE SCENARIO

To make it clearer how you can deploy RTRTR, below is an example scenario. This flow may not be entirely realistic, but it intends to show all the different ways you can wire units and targets together using a visual representation and the configuration file needed to accomplish it.

In this example, there is routing infrastructure in a data centre labeled as `dc1`. To ensure redundancy, it gets Validated ROA Payloads (VRPs) primarily from relying party software running in the `eu-west-3` location, using the RTR protocol. There are two backups configured: a validator serving RTR in `ap-south-1` and an instance from another vendor offering a feed in JSON format in `us-east-2`. A unit of the type `any` is configured to get a feed from all three and, should the first one fail, do a round robin to the next available one.

To make the management of some statically configured routes for this location easy, the `slurm` unit gets its data from the `any` unit so only a single file has to be kept up-to-date.

Finally, an `http` target is configured to get the VRPs without the SLURM exceptions, to be fed into internal tooling and an `rtr` unit is defined to serve the routing infrastructure.

4.1 Configuration File

```
log_level = "debug"
log_target = "stderr"
log_facility = "daemon"
log_file = "/var/log/rtrtr.log"

http-listen = ["dc1.http.example.net:8080"]

# RTR UNITS

[units.eu-west-3]
type = "rtr"
remote = "paris.validator.example.net:3323"

[units.ap-south-1]
type = "rtr"
remote = "mumbai.validator.example.net:3323"

# JSON UNIT

[units.us-east-2]
type = "json"
uri = "https://ohio.validator.example.net/rpki.json"
refresh = 60

```

(continues on next page)

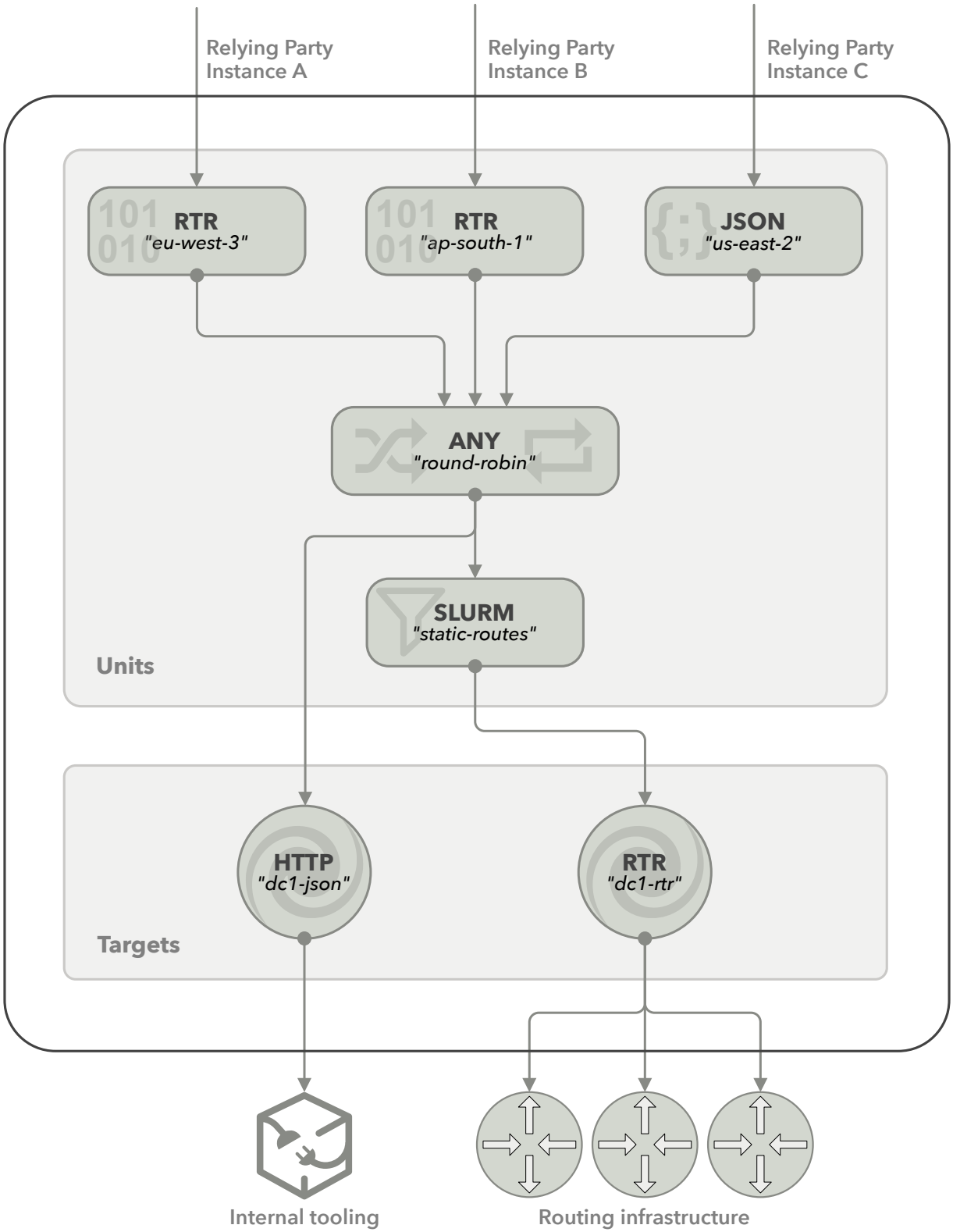

Fig. 1: Example of an RTRTR data flow

(continued from previous page)

```
# ANY UNIT

[units.round-robin]
type = "any"
sources = [ "eu-west-3", "ap-south-1", "us-east-2" ]
random = false

# SLURM

[units.static-routes]
type = "slurm"
source = "round-robin"
files = [ "/var/lib/rtrtr/local-expections.json" ]

# RTR TARGET

[targets.dc1-rtr]
type = "rtr"
listen = [ "dc1.rtr.example.net:9001" ]
unit = "static-routes"

# JSON TARGET

[targets.dc1-json]
type = "http"
path = "/json"
format = "json"
unit = "round-robin"
```

INDEX

R