RPKI Hints, Top Tips, and FAQs

On this page I'm collecting how to do various RPKI bits and pieces. Usually because the supplied documentation is incomplete, or just plain useless.

These are my build notes, and I keep them current in so far as I use them for the validators I maintain for the NSRC Global R&E Routing Table report and for RouteViews. YMMV.

NB: always use the most up to date/current versions of the validators - things are moving pretty quickly, and the older versions can often be quite buggy, if not unstable.

Here is the list (so far):

- AS0 TALs
- Routinator 3000 validator from NLnetLabs
- FORT validator from NIC Mexico
- RPKI-client validator
- StayRTR
- Cisco IOS-XE
- Cisco IOS-XR
- Juniper
- BIRD
- FRR

The tips and tricks for the validator builds discussed below all are for Ubuntu 24.04. They should also work on Ubuntu 18.04 and Ubuntu 20.04 (neither of which is supported), and Ubuntu 22.04 (which is supported until April 2027). If you are installing a fresh container or VM, please use Ubuntu 24.04 (if you are wedded to the Ubuntu family).

AS0 TALs

Two of the Regional Internet Registries have supplied Trust Anchor Locators (TALs) for unassigned IP address space that they hold.

If you want to use these TALs, you can read more:

- APNIC's AS0 TAL
- LACNIC's AS0 TAL

Generally, to use these TALs, place each in a separate file (eg place APNIC's one in **apnic-as0.tal**) in the usual place where you keep your TALs - it depends on your validator of course.

NLnetLabs Routinator

Nothing to say here, the instructions just work, the validator installs sweetly, and just runs. As long as the instructions are followed. The current version of Routinator is 0.15.2, at time of writing.

If using Debian/Ubuntu as I do, then just use the supplied package and your favourite package manager. The instructions for how to install are in NLnetLabs excellent documentation.

The installer will set up the necessary **systemd** file so that Routinator starts automatically on boot. Remember to modify the **/etc/routinator/routinator.config** file so that Routinator listens on the IPv4 (and IPv6) ports of the system - and you can enable the default statistics pages which listen on port 8323. A working configuration file would look like this:

```
repository-dir = "/var/lib/routinator/rpki-cache"
rtr-listen = ["x.x.x.x:3323", "[x:x:x:x::x]:3323"]
http-listen = ["x.x.x.x:8323", "[x:x:x:x::x]:8323"]
```

As from Routinator 0.12.0, the 5 RIR TALs are distributed built-in to the validator, so there is no longer any need to read and agree to the ARIN RPA.

At time of writing, though, if you want to use the AS0 TALs from LACNIC and APNIC (see above), the “extra-tals-dir” option mentioned in the configuration guide only works on the command line and not in the Routinator configuration file that ships as part of the package. This means creating an user modification to the systemd setup **/etc/systemd/system/routinator.service** like this:

```
[Service]
ExecStart=
ExecStart=/usr/bin/routinator --config=/etc/routinator/routinator.conf --
extra-tals-dir="/var/lib/routinator/tals" --syslog server
```

where the **/var/lib/routinator/tals** folder contains the two AS0 TALs you want to use. The first blank **ExecStart** is needed to overwrite the one that is shipped in **/lib/system/system/routinator.service**. Once this bug is fixed you can remove this user defined systemd setting and put the **extra-tals-dir** definition in the configuration file, like this:

```
repository-dir = "/var/lib/routinator/rpki-cache"
extra-tals-dir = "/var/lib/routinator/tals"
rtr-listen = ["x.x.x.x:3323", "[x:x:x:x::x]:3323"]
http-listen = ["x.x.x.x:8323", "[x:x:x:x::x]:8323"]
```

FORT

FORT is the validator developed by NIC Mexico. More about it is on the Project page. At time of writing, version 1.6.7 has been released and fixes many issues present in previous versions. However from version 1.6.3, FORT requires Ubuntu 24.04 as it requires libjansson4 (>= 2.14). Ubuntu 22.04 only comes with libjansson4 2.13.1-1.1build3 will only support FORT version 1.6.2.

FORT is available as part of Ubuntu 22.04 packaging, but it is an older version (1.5.3-1). Likewise for Ubuntu 24.04, the FORT shipped is version 1.6.1-1build3. For this reason we use the latest NIC Mexico produced package.

FORT is not quite so easy to install, but still relatively simple as long as you follow the instructions on their Github repo closely.

First step is to grab the **.deb** file from their archive:

```
wget
https://github.com/NICMx/FORT-validator/releases/download/1.6.8/fort_1.6.8-1
_amd64.deb
```

and then install it:

```
sudo apt install ./fort_1.6.8-1_amd64.deb
```

Note that the **apt install** installs a **systemd** file and starts FORT running automatically. FORT uses TCP/323 as the listener port - you may want to customise this, and to do that, edit the configuration file **/etc/fort/config.json**. This is the configuration file that I use:

```
{
        "tal": "/etc/fort/tal",
        "local-repository": "/var/lib/fort",
        "slurm": "/etc/fort/slurm/",
        "server": {
                "port": "3323"
        },
        "log": {
                "output": "syslog"
        }
}
```

All I have done is modify the port that the server listens on.

The package ships with 4 of the 5 Trust Anchor Locators, so to get the missing one (ARIN's), you will need to run:

```
sudo fort --init-tals --tal=/etc/fort/tal
```

You will be asked to confirm that you have read the Terms and Conditions regarding ARIN's TAL:

```
...
Jan 26 03:50:46 DBG: Done. Total bytes transferred: 466
Jan 26 03:50:46 DBG: HTTP result code: 200
Successfully fetched '/etc/fort/tal/apnic.tal'!

Attention: ARIN requires you to agree to their Relying Party Agreement (RPA)
before you can download and use their TAL.
Please download and read https://www.arin.net/resources/manage/rpki/rpa.pdf
If you agree to the terms, type 'yes' and hit Enter: yes
Jan 26 03:50:51 DBG: HTTP GET:
https://www.arin.net/resources/manage/rpki/arin.tal
Jan 26 03:50:51 DBG: Done. Total bytes transferred: 487
Jan 26 03:50:51 DBG: HTTP result code: 200
Successfully fetched '/etc/fort/tal/arin.tal'!

Jan 26 03:50:51 DBG: HTTP GET:
https://www.lacnic.net/innovaportal/file/4983/1/lacnic.tal
```

```
...
```

One thing that I found is that FORT crashes on start up following the above installation instructions to the letter. The issue is that the **/var/lib/fort** folder is owned by **root**, not by the **fort** user. Easy to fix:

```
sudo chown fort:fort /var/lib/fort
```

Then restart FORT:

```
sudo systemctl restart fort
```

and it should run successfully. You should see something like this when you run **systemctl status fort**:

```
● fort.service - FORT RPKI validator
     Loaded: loaded (/usr/lib/systemd/system/fort.service; enabled; preset:
enabled)
     Active: active (running) since Mon 2024-10-07 22:58:03 AEST; 29s ago
       Docs: man:fort(8)
             https://nicmx.github.io/FORT-validator/
   Main PID: 148150 (fort)
      Tasks: 27 (limit: 38225)
     Memory: 680.6M (peak: 680.7M)
        CPU: 27.801s
     CGroup: /system.slice/fort.service
             └─148150 /usr/bin/fort --configuration-file
/etc/fort/config.json

Oct 07 22:58:03 fort systemd[1]: Started fort.service - FORT RPKI validator.
```

You can check by using **ps ax** to get:

```
  195 ?        Ssl   95:13 /usr/bin/fort --configuration-file
/etc/fort/config.json
```

and **netstat -an** (upto Ubuntu 18.04) or **ss -an** (on Ubuntu 20.04 onwards) to get:

```
tcp     LISTEN   0        128                     0.0.0.0:3323
0.0.0.0:*
```

The **systemd** that ships with FORT doesn't have a “restart on failure” setting so it is a good idea that we do this so that if/when FORT crashes, it will automatically restart.

Create a new **/etc/systemd/system/fort.service** (which is where to create user-defined extras for **systemd** go), and in it put:

```
[Service]
Restart=
Restart=on-failure
```

(If you edit the **/lib/systemd/system/fort.service** it will be overwritten on the next upgrade of FORT, so it is best create the user entry instead.)

And your new FORT installation is now ready for service!

RPKI-client

rpki-client is just a validator - it does not have the functionality to accept connections from a router. We'll come to that later on (we'll need to use StayRTR, which is a fork of Cloudflare's now unmaintained GoRTR).

rpki-client has now been packaged and is available across most mainstream Linux/Unix-based platforms. Including as part of the Ubuntu 22.04 and later distributions. However, the packaged version in Ubuntu is old (version 7.6 on 22.04, 9.0 on 24.04). At the time of writing, the current release of **rpki-client** is version 9.8. There is a version of **rpki-client** on the Ubuntu Snap Store, but it is unclear which version of **rpki-client** this is.

My preference is to stay up to date on Ubuntu, and so we have to build it ourselves. A pity that the **rpki-client** maintainers don't build their own deb package, or pre-build packages like NLnetLabs do with Routinator, given that Ubuntu maintainers seem to be unable to keep the software current. Oh well.

Initial Preparation

Before you attempt to download and build it, the **rpki-client** instructions note that you need a few other packages in place. These include **automake**, **autoconf**, **make**, **git** itself, **libtool** and **expat**. This is all quite easy using the Ubuntu package manager.

```
sudo apt install automake autoconf make git libtool libexpat1-dev
```

The other required package noted in the instructions is **tls** from LibreSSL. LibreSSL is a branch of OpenSSL and is used on OpenBSD - not found on Linux, but seems to be appearing in the latest Debian/Ubuntu beta builds. So we need to download the bits we need and install. The **rpki-client** instructions don't say anything about how to do that.

First we go to https://ftp.openbsd.org/pub/OpenBSD/LibreSSL/ and select the latest package, which is libressl-4.3.2.tar.gz at time of writing

```
wget https://ftp.openbsd.org/pub/OpenBSD/LibreSSL/libressl-4.3.2.tar.gz
```

We then unpack it:

```
tar zxf libressl-4.3.2.tar.gz
```

and then build it:

```
cd libressl-4.3.2
./configure --enable-libtls-only
```

```
make
sudo make install
```

Note the option to only build **libtls** - we don't need the rest of LibreSSL and it could well interfere with OpenSSL which will already be on the system. Now that **libtls** is built, the **install** action will put the libraries in **/usr/local/lib** like this:

```
-rw-r--r-- 1 root root 18854748 Jul 10 04:03 libtls.a
-rw-r--r-- 1 root root      923 Jul 10 04:03 libtls.la
lrwxrwxrwx 1 root root       16 Jul 10 04:03 libtls.so -> libtls.so.33.0.2
lrwxrwxrwx 1 root root       16 Jul 10 04:03 libtls.so.33 ->
libtls.so.33.0.2
-rw-r--r-- 1 root root  8843992 Jul 10 04:03 libtls.so.33.0.2
```

Run **sudo ldconfig** so that the system knows about the new libraries.

Next we need to get some packages that **rpki-client** needs. These are **libssl-dev**, **rsync** and **zlib1g-dev**.

```
sudo apt install libssl-dev libtls-dev zlib1g-dev
```

And now we are ready to build **rpki-client**.

Building rpki-client

Easiest is just to install **rpki-client** from the Github repository:

```
git clone https://github.com/rpki-client/rpki-client-portable.git
```

and then:

```
cd rpki-client-portable
./autogen.sh
./configure --with-tal-dir=/etc/rpki \
   --with-base-dir=/var/lib/rpki-client \
   --with-output-dir=/var/db/rpki-client
make
```

The **autogen.sh** script fixes the config set up, ready to run **configure** which will produce the **Makefile**. Note that we are specifying where our TALs go, where the temporary files go (following Ubuntu norms), and where the output file storing the VRPs goes (again following Ubuntu norms).

Hidden in the official instructions is a comment that **rpki-client** runs as a normal user if started as root. So we need to create that normal user:

```
sudo groupadd _rpki-client
sudo useradd –g _rpki-client –s /sbin/nologin –d /nonexistent –c "rpki-
client user" _rpki-client
```

Now we can install RPKI-client:

```
sudo make install
```

which will install the client in **/usr/local/sbin** and the 5 TALs in **/etc/rpki**, as well as create the cache and output directories needed. It will also copy the 5 RIR "constraints" files into **/etc/rpki**; these prevent overclaiming of resources by the 5 RIRs.

Now the client can be run. There is no daemon option, it simply runs at the command line, and when it has finished downloading all the VRPs (around 10-15 minutes depending on bandwidth) it exits. But that's okay. Try running the client:

```
sudo /usr/local/sbin/rpki-client
```

You'll see errors about various CAs or files not being accessible - that's their problem, not yours. If you check in the **/var/db/rpki-client** folder you will see an **openbgpd** file once the above run of **rpki-client** completes. This is the configuration you'd need if you run **openbgp**. However, we are going to run a standalone RtR client instead, so we will need JSON output instead.

Once **rpki-client** completes, we can now set it up to run automatically. To do this, we create a file in **/etc/cron.hourly** called **rpki-client**, and in it we put:

```
#!/bin/bash

# run RPKI-client every hour
# - default output location is /var/db/rpki-client
# - -j option means json output, suitable for stayrtr

/usr/local/sbin/rpki-client -j > /tmp/rpki-client.log 2>&1
```

and that's it. Every hour, cron will run **rpki-client** which will produce JSON output of all the VRPs it has collected. As noted above, JSON output is what is used by StayRTR and GoRTR as their input sources. Make sure that the **/etc/cron.hourly/rpki-client** is executable, otherwise it will not run.

If you would like to include the AS0 TALs from APNIC and LACNIC it is not sufficient to just place them in your chosen TAL directory. You will also need to include the **-0** option in the command line, like this:

```
/usr/local/sbin/rpki-client -0j > /tmp/rpki-client.log 2>&1
```

It's a good idea to check the log file in case **rpki-client** reports issues trying to write local files etc. But mostly what you'll see there are all the transactions with the various CAs, and the problems encountered (there will be lots, unfortunately).

StayRTR

StayRTR is a hard fork of GoRTR (which is no longer maintained by Cloudflare and is badly out of date). For this reason, I **strongly** recommend you use StayRTR rather than GoRTR. If you have an existing GoRTR install, simply replace it with StayRTR.

StayRTR has now been packaged and is available as part of the Ubuntu 22.04 distribution (packaged version is 0.3.0) and the Ubuntu 24.04 distribution (packaged version is 0.5.1). At the time of writing, the current release of StayRTR is version 0.6.2, and I much prefer to have the latest version of a critical piece of software like a validator.

So for this reason, and to stay up to date, at least on Ubuntu, we have to build it ourselves. A pity that the **StayRTR** maintainers don't build their own deb package, or pre-build packages like NLnetLabs do with Routinator.

Installing Go

First you will need a working Go environment. Full instructions are at https://go.dev/doc/install, and I've reproduced the key pieces here to make it easy for installers.

First off, download the latest Go package (1.24.4 at time of writing):

```
wget https://go.dev/dl/go1.24.4.linux-amd64.tar.gz
```

If you have an existing Go environment, perhaps save it in case something goes wrong with the new version:

```
sudo mv /usr/local/go /usr/local/go.old
```

and then you can unpack the new version:

```
cd /usr/local
sudo chmod 777 .
tar xzf ~/go1.24.4.linux-amd64.tar.gz
sudo chmod 755 .
```

Next add **/usr/local/go/bin** to the **PATH** environment variable. If you use **bash**, this would be in the **.profile** in your home directory, and just add:

```
if [ -d "/usr/local/go/bin" ] ; then
    PATH="$PATH:/usr/local/go/bin"
fi
```

Log off. And then log in again. (Easiest way of activating the updated **PATH**.)

And now check you have a working Go environment:

```
go version
```

If the version shows what you installed, you are set!

Building StayRTR

Easiest way to install StayRTR is build from the Github repo.

```
git clone https://github.com/bgp/stayrtr.git
cd stayrtr
make build-all
```

which builds **stayrtr** as well as **rtrmon** and **rtrdump** (the latter used for testing purposes).

Copy the resulting binaries to **/usr/local/bin**:

```
cd dist
sudo cp -p stayrtr-v0.6.4-1-g4bad963-linux-x86_64 /usr/local/bin/stayrtr
sudo cp -p rtrdump-v0.6.4-1-g4bad963-linux-x86_64 /usr/local/bin/rtrdump
sudo cp -p rtrmon-v0.6.4-1-g4bad963-linux-x86_64 /usr/local/bin/rtrmon
```

StayRTR has lots of options, but the ones we need are these:

```
 -bind string
       Bind address (default ":8282")
 -cache string
       URL of the cached JSON data (default
"https://console.rpki-client.org/vrps.json")
```

We don't need to use the public RPKI-client JSON source, given we have our own from the newly created RPKI-client.

We run StayRTR like this:

```
/usr/local/bin/stayrtr -bind :3323 -cache /var/db/rpki-client/json
```

which will at least let us test that it works. Run it and see what happens - you should see output at the command line looking like this:

```
INFO[0000] new cache file: Updating sha256 hash  ->
e0a14ea955e183e2719dcfbee0e9429b34581972c6ad5f6e9e064ee1396caf60
INFO[0001] New update (307007 uniques, 307007 total prefixes).
INFO[0002] Updated added, new serial 0
INFO[0002] StayRTR Server started (sessionID:60037, refresh:3600, retry:600,
expire:7200)
```

And if you check the ports that are listening (**ss -an**) you will see:

```
tcp     LISTEN   0        128          *:9847              *:*
tcp     LISTEN   0        128          *:3323              *:*
```

Port 3323 is the listening port for Router connections. And Port 9847 is the metrics port, for monitoring systems to connect to.

But perhaps this isn't good for long term operations as you'd prefer to have this start running automatically when the system starts. And for that we'd need to set up a suitable **systemd** entry.

First off, let's create a user for StayRTR (it does not have to run as root):

```
sudo groupadd _stayrtr
sudo useradd -g _stayrtr -s /sbin/nologin -d /nonexistent -c "StayRTR user"
_stayrtr
```

Next we create a file **/etc/default/stayrtr** with the following contents:

```
# Settings for StayRTR. Consult https://github.com/bgp/stayrtr for
# more discussion and other available options

STAYRTR_ARGS=-bind :3323 -cache /var/db/rpki-client/json
#
```

Then we go to the **/lib/systemd/system/** folder and create the **systemd** entry - call it **stayrtr.service**. Here is a simple one that should work:

```
[Unit]
Description=StayRTR RPKI to Router Server
Documentation=https://github.com/bgp/stayrtr
After=network.target

[Service]
EnvironmentFile=/etc/default/stayrtr
ExecStart=/usr/local/bin/stayrtr $STAYRTR_ARGS
Type=exec
User=_stayrtr
Group=_stayrtr
AmbientCapabilities=CAP_NET_BIND_SERVICE
CapabilityBoundingSet=CAP_NET_BIND_SERVICE

[Install]
WantedBy=multi-user.target
```

We then need to enable it:

```
sudo systemctl enable stayrtr
```

which then displays:

```
Created symlink /etc/systemd/system/multi-user.target.wants/stayrtr.service
→ /lib/systemd/system/stayrtr.service.
```

and then we can run StayRTR, like this:

```
sudo systemctl start stayrtr
```

Once it is running, check that it is working by running:

```
sudo systemctl status stayrtr
```

and you should see something like this:

```
● stayrtr.service - StayRTR RPKI to Router Server
     Loaded: loaded (/usr/lib/systemd/system/stayrtr.service; enabled;
preset: enabled)
     Active: active (running) since Thu 2024-08-15 16:57:07 +06; 31s ago
       Docs: https://github.com/bgp/stayrtr
   Main PID: 44045 (stayrtr)
      Tasks: 5 (limit: 4614)
     Memory: 379.9M (peak: 459.4M)
        CPU: 3.805s
     CGroup: /system.slice/stayrtr.service
             └─44045 /usr/local/bin/stayrtr -bind :3323 -cache /var/db/rpki-
client/json

Aug 15 16:57:06 rpki systemd[1]: Starting stayrtr.service - StayRTR RPKI to
Router Server...
Aug 15 16:57:07 rpki systemd[1]: Started stayrtr.service - StayRTR RPKI to
Router Server.
Aug 15 16:57:10 rpki stayrtr[44045]: time="2024-08-15T16:57:10+06:00"
level=info msg="New update (602077 uniques, 602151 total prefixes, 71 vaps,
3 router>
Aug 15 16:57:11 rpki stayrtr[44045]: time="2024-08-15T16:57:11+06:00"
level=info msg="Update added, new serial 0"
Aug 15 16:57:11 rpki stayrtr[44045]: time="2024-08-15T16:57:11+06:00"
level=info msg="StayRTR Server started (sessionID:61374, refresh:3600,
retry:600, ex>
```

and you can also run the more traditional **ps ax** to see something like:

```
  44045 ?        Ssl    0:03 /usr/local/bin/stayrtr -bind :3323 -cache
/var/db/rpki-client/json
```

And that's it. Enjoy your new StayRTR installation.

Cisco IOS-XE Hints

This section shows the basic configuration needed to get route origin validation up and running on Cisco IOS-XE platforms.

Most commentary is for IOS-XE 16.x. Older versions will also likely work with the following examples, but their RPKI implementation is somewhat old and buggy.

IOS-XE Configuration with Validator

Setting up a Cisco IOS-XE router to talk with a validator is a one line configuration:

```
router bgp <ASN>
 bgp rpki server tcp <ip address> port <port> refresh 3600
```

where <ip address> is the IP address (IPv4 or IPv6) of the validator, <port> is the TCP port the validator listens on, and 3600 is the RFC8210 recommended refresh interval (the period which the router will use to ask the validator if there is new/updated validation information available).

Doing this will download the VRPs from the specified validator(s), and then update the BGP table (BGP RIB) with the prefixes validation state (whether valid, invalid, or not-found), and drop invalids. (**NB: See caveats below.**)

To find out what is in the validation database (IPv4 and IPv6 commands shown):

```
show ip bgp rpki table
show bgp ipv6 rpki table
```

and to find out the status of the connection to the validator:

```
show ip bgp rpki servers
```

Cisco IOS-XE Caveats

Cisco IOS-XE has many defaults which are non-standard and will be potentially frustrating for operators:

- Cannot specify a source-interface for the router to validator connection
- Automatically activates route origin validation (can be turned off!)
- Automatically drops invalids (can be turned off!)
- Locally originated prefixes are always marked as valid (cannot be turned off!) - fixed in most recent IOS-XE 17.x releases
- Automatically prefers Valid path over Invalid/NotFound, even if latter has higher local-preference (BGP Best Path Selection over-ride) (cannot be turned off!)
- If validator disappears, router validation database is flushed within a few minutes - fixed in most recent IOS-XE 16.6 releases

To turn off the checking of the RPKI validation database (**IOS-XE 15.5 onwards**):

```
router bgp <ASN>
 address-family ipv4
  bgp bestpath prefix-validate disable
 address-family ipv6
  bgp bestpath prefix-validate disable
```

The ROAs are still listed in the RPKI table but the router will not use them. (This should be the default, as per RFC.)

To turn off the automatic dropping of invalids:

```
router bgp <ASN>
 address-family ipv4
  bgp bestpath prefix-validate allow-invalid
 address-family ipv6
  bgp bestpath prefix-validate allow-invalid
```

A new set up of RPKI in a Cisco IOS-XE network should start with “monitoring” only. Which is only downloading the validation database, not implementing any checking. So the savvy operator would combine the above like this:

```
router bgp <ASN>
 address-family ipv4
  bgp bestpath prefix-validate disable
  bgp bestpath prefix-validate allow-invalid
 address-family ipv6
  bgp bestpath prefix-validate disable
  bgp bestpath prefix-validate allow-invalid
```

Once they are ready to implement RPKI, first remove the `prefix-validate disable`, and monitor. And once ready to do ROV, removing the `prefix-validate allow-invalid` would be the last step.

The major show-stopper for an IOS-XE based network is the insertion of validation check in the BGP path selection process, over-riding `local-preference`. There is no way of turning this mis-feature off and it is a fundamental impediment to implementing ROV in a Cisco IOS-XE based network.

To propagate the validation state in IBGP, both BGP speakers need:

```
neighbor x.x.x.x announce rpki state
```

Please do **NOT** do this, as there are operational consequences, especially if validators become unreachable (specifically that IOS-XE has added an undocumented feature in the path selection process whereby a prefix marked as *valid* from one IBGP neighbour is preferred over *invalid*/*notfound* from another IBGP neighbour regardless of *local-preference* setting).

Summary: Cisco has tried to make it easy to deploy ROV, but unfortunately this “assistance” ignores standards and best practices. Any implementation must never take control away from the operator about what they can and cannot do, especially if there is no way of turning off mis-features.

Cisco IOS-XR Hints

This section shows the basic configuration needed to get route origin validation up and running on Cisco IOS-XR platforms.

Most commentary is for IOS-XR 7.5 onwards. Older versions will also likely work with the following examples, but their RPKI implementation is likely to be more buggy.

IOS-XR Configuration with Validator

Setting up a Cisco IOS-XR router to talk with a validator is done like this:

```
router bgp <ASN>
 rpki server <ip address>
  bind-source interface Loopback0
```

```
  transport tcp port <port>
  refresh-time 3600
  response-time 600
```

where <ip address> is the IP address (IPv4 or IPv6) of the validator, <port> is the TCP port the validator listens on, and 3600 is the RFC8210 recommended refresh interval (the period which the router will use to ask the validator if there is new/updated validation information available). Binding to the Loopback address makes ACLs for the validator simpler to manage. Note that IOS-XE does not offer this valuable feature.

To find out what is in the validation database (IPv4 and IPv6 commands shown):

```
show ip bgp rpki table
show bgp ipv6 rpki table
```

and to find out the status of the connection to the validator:

```
show ip bgp rpki server summary
```

To turn on validation for prefixes on the router, we need to activate the functionality per address family (as per best practice - the operator needs to choose!).

```
router bgp <ASN>
 address-family ipv4 unicast
  bgp origin-as validation enable
  bgp bestpath origin-as use validity
  bgp bestpath origin-as allow invalid
 !
 address-family ipv6 unicast
  bgp origin-as validation enable
  bgp bestpath origin-as use validity
  bgp bestpath origin-as allow invalid
```

The above enables origin validation, and still allows invalid prefixes in the BGP table.

Once you are ready to drop invalids, as per recommended best practices:

```
router bgp <ASN>
  address-family ipv4 unicast
    no bgp bestpath origin-as allow invalid
  address-family ipv6 unicast
    no bgp bestpath origin-as allow invalid
```

To display the validation state of prefixes, you can use the following command:

```
RP/0/RP0/CPU0:cr1#show bgp origin-as validity ?
  invalid    filter routes with invalid origin-as
  not-found  filter routes with unknown (not found) origin-as
  standby    Display Standby BGP information
  valid      filter routes with valid origin-as
  |          Output Modifiers
```

```
  <cr>
```

The sub-options will display all the prefixes fitting into each category.

Juniper Hints

This section shows the basic configuration needed to get route origin validation up and running on a JunOS platform. JunOS follows standards, as far as I can tell. All policy has to be explicitly configured by the operator.

Configuration with Validator

The configuration needed for JunOS to talk with two validators looks like this:

```
routing-options {
  validation {
    group ISP {
      /* validatorA */
      session <ip address validatorA> {
        refresh-time 600;
        hold-time 3600;
        preference 1;
        port <port>;
        local-address <router loopback>;
      }
      /* validatorB */
      session <ip address validatorB> {
        refresh-time 600;
        hold-time 3600;
        preference 2;
        port <port>;
        local-address <router loopback>;
      }
    }
  }
}
```

where <ip address> is the IP address (IPv4 or IPv6) of the validator, <port> is the TCP port the validator listens on, and 3600 is the RFC8210 recommended refresh interval (the period which the router will use to ask the validator if there is new/updated validation information available).

Two validators are recommended - if you only have one, then edit the above to suit.

This simply creates a validation database on the router - it does not touch the BGP RIB. To find out what is in the validation database:

```
show validation replication database
```

and to find out the status of the connection to the validator:

```
show validation session detail
```

Flagging validation status in BGP RIB

To indicate validation status in the BGP RIB, the operator needs to implement a policy statement to do that. Here is an example:

```
policy-options {
  policy-statement RPKI-validation {
    term VALID {
      from {
        protocol bgp;
        validation-database valid;
      }
      then {
        validation-state valid;
        next policy;
      }
    }
    term INVALID {
      from {
        protocol bgp;
        validation-database invalid;
      }
      then {
        validation-state invalid;
        next policy;
      }
    }
    term UNKNOWN {
      from {
        protocol bgp;
        validation-database unknown;
      }
      then {
        validation-state unknown;
        next policy;
      }
    }
  }
}
protocols {
  bgp {
    group EBGP {
      type external;
      local-address <local-router>;
      neighbor <ebgp-peer> {
```

```
        description "Upstream";
        import [ RPKI-validation Upstream-in ];
        export LocalAS-out;
        peer-as <their-ASN>;
      }
    }
  }
}
```

This is intended to be used as an **inbound** policy on an EBGP peer. The *validation-database* configuration refers to the database created by the router's session with the validator. The *validation-state* configuration enters a flag in the BGP RIB to indicate the validation state of the prefix. Note that the *Upstream-in* policy is other inbound policy that is on the router, and not documented here.

And then the operator can do things like:

```
show route protocol bgp validation-state valid
```

to show the **valid** prefixes as in this example. Other options are available for **invalid** and **notfound**.

Implementing Route Origin Validation

The final step in JunOS is to implement Route Origin Validation. This is can be simply achieved by replacing the:

```
next policy
```

line in the INVALID term with:

```
reject
```

This will discard invalid prefixes as they arrive on an EBGP speaking router. If you want to find out what has been dropped, you can run:

```
show route protocol bgp validation invalid hidden
```

The discarded prefixes are still in Adj-RIB-In and can be seen with the **hidden** keyword option.

BIRD Hints

This section shows the basic configuration needed to get route origin validation up and running on a BIRD platform. This will be of most interest to IXPs, as BIRD is the mostly widely used Route Server implementation today. The configuration here is for BIRDv2 (2.14 at time of writing).

Configuration with Validator

The configuration needed for BIRD to talk with a validator is:

```
roa4 table r4;
roa6 table r6;

protocol rpki validator1 {
    roa4 { table rpki4; };
    roa6 { table rpki6; };
    remote <ip address1> port <port1>;
    retry keep 600;
    refresh keep 3600;
    expire keep 7200;
}

protocol rpki validator2 {
    roa4 { table rpki4; };
    roa6 { table rpki6; };
    remote <ip address2> port <port2>;
    retry keep 600;
    refresh keep 3600;
    expire keep 7200;
}
```

where <ip address> is the IP address (IPv4 or IPv6) of the validator, <port> is the TCP port the validator listens on, and 3600 is the RFC8210 recommended refresh interval (the period which the router will use to ask the validator if there is new/updated validation information available). Two validators are shown in this example.

This simply creates a validation database in BIRD - it does not touch the BGP RIB. To find out what is in the validation database (IPv4 and IPv6 shown):

```
show route table rpki4
show route table rpki6
```

and to find out the status of the connection to *validator1*:

```
show protocols validator1
```

Implementing Route Origin Validation

The final step in BIRD is to implement Route Origin Validation. This needs a policy statement, to be applied as *outbound* policy on all BGP sessions (internal and external). We build those up using BIRD functions, like below (as used on route-server implementations of BIRD).

```
# v4 function to check if prefix valid
function is_v4_rpki_invalid () {
  return roa_check(rpki4, net, bgp_path.last_nonaggregated) = ROA_INVALID;
}
```

```
# v6 function to check if prefix valid
function is_v6_rpki_invalid () {
  return roa_check(rpki6, net, bgp_path.last_nonaggregated) = ROA_INVALID;
}

# return true if invalid, false if not
function prefix_is_invalid()
{
    if net.type = NET_IP4 then
        if is_v4_rpki_invalid() then return true;
    if net.type = NET_IP6 then
        if is_v6_rpki_invalid() then return true;
    return false;
}
```

And then the *prefix_is_valid* function will be called as part of a larger outbound policy function, for example:

```
filter EXPORT
{
  if (prefix_is_bogon()) then reject "[Rejected Prefix: Bogon] ", net;
  if (prefix_is_invalid()) then reject "[Rejected Prefix: Invalid] ", net;
  if (as_path_contains_bogons()) then reject "[Rejected Prefix: Bogon AS] ",
net;
  if (bgp_path.len > 32) then reject "[Rejected Prefix: Long AS] ", net;
  if (bgp_path.len < 1) then reject "[Rejected Prefix: No AS] ", net;
  accept "[Exported Prefix] ", net;
}
```

FRrouting Hints

This section shows the basic configuration needed to get route origin validation up and running on implementations using FRrouting (FRR). The commentary below assumes FRR 8.2.2. Older versions of FRR (8.1 and 7.5) have slightly different CLI and may not have all the features shown here.

Configuration with Validator

The configuration needed for FRR to talk with a validator is:

```
rpki
 rpki polling_period 3600
 rpki cache <ip address1> <port1> preference 1
 rpki cache <ip address2> <port2> preference 2
exit
```

Two validators are configured in this example, the preferred one has preference 1. The backup validator (used when the primary has become unreachable) is preference 2. More can be added

following this principle.

FRR automatically populates the BGP RIB with validation status of each prefix. No operator intervention is needed. To find out what is in the validation database:

```
show rpki prefix-table
```

and to find out the status of the connection to the active validator:

```
show rpki cache-connection
```

To show the BGP RIB with the validation information now flagged, just use the standard:

```
show bgp
```

command set. To show prefixes that meet any of the validation states:

```
show bgp rpki valid
```

will show all the entries that are *valid*. There is also a command option for *invalid* and *notfound*.

Implementing Route Origin Validation

The final step in FRR is to implement Route Origin Validation. This needs a policy statement, to be applied as *outbound* policy on all BGP sessions (internal and external). The following example shows a route-map exporting IPv4 routes. Similar can be done for IPv6.

```
route-map EXPORT deny 5
 description Don't send RPKI Invalids
 match rpki invalid
exit
!
route-map EXPORT deny 15
 description Drop the IPv4 bogons
 match ip address prefix-list v4bogon
exit
!
route-map EXPORT permit 20
 description Everything else is good
exit
!
```

Back to Home page

From:
https://www.bgp4all.com.au/pfs/ - **Philip Smith's Internet Development Site**

Permanent link:
https://www.bgp4all.com.au/pfs/hints/rpki?rev=1784558691

Last update: **2026/07/20 14:44**