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.

Here is the list (so far):

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

The tips and tricks for the validator builds discussed below all are for Ubuntu 20.04. They should also work just fine on Ubuntu 18.04 (which is supported until April 2023) and I'll note if I experience otherwise. I've not tested anything on the Ubuntu interims since 20.04, although I plan to test 22.04 which has just been released.

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.

If using Debian/Ubuntu as I do, then just use the supplied package and your favourite package manager. Described in NLnetLabs's Github repo.

You could build from source if you really want to, but why bother. If the link to the supplied package is added to your package manager, for example **apt** on Ubuntu, then create an entry in **/etc/apt/sources.list.d** called **routinator.list** and put this in it (which is for Ubuntu 20.04):

```
deb [arch=amd64] https://packages.nlnetlabs.nl/linux/ubuntu/ focal main
```

Easy!

FORT

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

The installation instructions are on Github.

(Actually, Marco d'Itri has created a Debian package which you can use to install from. But the FORT team note that it may be a release or two behind their own packaging.)

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

```
wget
https://github.com/NICMx/FORT-validator/releases/download/1.5.3/fort_1.5.3-1
_amd64.deb
```

and then install it:

```
sudo apt install ./fort_1.5.3-1_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 (/lib/systemd/system/fort.service; enabled; vendor
preset: enabled)
    Drop-In: /run/systemd/system/service.d
             └─zzz-lxc-service.conf
     Active: active (running) since Wed 2022-01-26 03:54:05 UTC; 4s ago
       Docs: man:fort(8)
             https://nicmx.github.io/FORT-validator/
   Main PID: 3100 (fort)
      Tasks: 37 (limit: 28794)
     Memory: 12.0M
     CGroup: /system.slice/fort.service
             └─3100 /usr/bin/fort --configuration-file /etc/fort/config.json
```

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:*
```

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 Cloudflare's GoRTR or its fork, StayRTR).

rpki-client has no package, although Marco d'Itri is working on one as you can find from the Debian package tracker.

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-3.4.2.tar.gz at time of writing

```
wget https://ftp.openbsd.org/pub/OpenBSD/LibreSSL/libressl-3.4.2.tar.gz
```

We then unpack it:

```
tar zxf libressl-3.4.2.tar.gz
```

and then build it:

```
cd libressl-3.4.2
./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  27392794 Jan 20 15:17 libtls.a
-rw-r--r--  1 root root       933 Jan 20 15:17 libtls.la
lrwxrwxrwx  1 root root        16 Jan 20 15:17 libtls.so -> libtls.so.22.0.0
lrwxrwxrwx  1 root root        16 Jan 20 15:17 libtls.so.22 ->
libtls.so.22.0.0
```

```
-rw-r--r--  1 root root  13146784 Jan 20 15:17 libtls.so.22.0.0
```

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** and **rsync**.

```
sudo apt install libssl-dev rsync
```

And now we are ready to build **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 4 TALs in **/etc/rpki**, as well as create the cache and output directories needed. Note that the ARIN TAL requires users to read the disclaimer first so is not provided by default. So you need to do this manually:

```
wget https://www.arin.net/resources/manage/rpki/arin.tal
sudo mv arin.tal /etc/rpki
```

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.

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).

GoRTR

I've included GoRTR here though it is no longer maintained by Cloudflare as the maintainer has moved on to pastures new. All development work is now being carried out on StayRTR which is a hard fork of GoRTR.

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 (1.18.1 at time of writing):

```
wget https://go.dev/dl/go1.18.1.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 ~/go1.18.1.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!

Installing GoRTR

Easiest way to do this is to build from the Github repo.

Note1: You could download and use the provided binaries if you wish.

Note2: You could even download the Debian package if you wish, and install that. It needs the **adduser** package, and a **libc** from 2.4 onwards (most modern Ubuntu releases). Bonus with the .deb package is that it comes with a **systemd** configuration.

But we will focus on building from the source.

```
git clone https://github.com/cloudflare/gortr.git
cd gortr
make build-gortr build-rtrmon build-rtrdump
```

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

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

```
cd dist
sudo -s
cp -p gortr-v0.14.7-1-g2125744-linux-x86_64 /usr/local/bin/gortr
cp -p rtrdump-v0.14.7-1-g2125744-linux-x86_64 /usr/local/bin/rtrdump
cp -p rtrmon-v0.14.7-1-g2125744-linux-x86_64 /usr/local/bin/rtrmon
```

GoRTR 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://rpki.cloudflare.com/rpki.json")
 -checktime
       Check if file is still valid (default true)
 -verify
       Check signature using provided public key (disable by passing -
verify=false)
```

We don't need to use the Cloudflare JSON source, given we have our own from the newly created RPKI-client. RPKI-client doesn't insert a timestamp in the way that GoRTR wants, nor is there a signature on it, so we need to disable that too.

We run GoRTR like this:

```
/usr/local/bin/gortr -bind :3323 -verify=false -cache /var/db/rpki-
client/json -checktime=false
```

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[0001] New update (304138 uniques, 304138 total prefixes). 0 bytes.
Updating sha256 hash  ->
0592ddc6e9a82666f8ddc5eda8cad76cb61f22640f17199b1bff06b5928b9718
INFO[0002] Updated added, new serial 0
INFO[0002] GoRTR Server started (sessionID:33094, refresh:3600, retry:600,
expire:7200)
```

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

```
tcp    LISTEN     0       128          *:3323              *:*
tcp    LISTEN     0       128          *:8080              *:*
```

Port 3323 is the listening port for Router connections. And Port 8080 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 GoRTR (it does not have to run as root):

```
sudo groupadd _gortr
sudo useradd –g _gortr –s /sbin/nologin –d /nonexistent –c "GoRTR user"
_gortr
```

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

```
# Settings for GoRTR. Consult https://github.com/cloudflare/gortr for
# more discussion and other available options

STAYRTR_ARGS=-bind :3323 -verify=false -cache /var/db/rpki-client/json -
checktime=false
```

```
#
```

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

```
[Unit]
Description=GoRTR RPKI to Router Server
Documentation=https://github.com/cloudflare/gortr
After=network.target

[Service]
EnvironmentFile=/etc/default/gortr
ExecStart=/usr/local/bin/gortr $GORTR_ARGS
Type=exec
User=_gortr
Group=_gortr
AmbientCapabilities=CAP_NET_BIND_SERVICE
CapabilityBoundingSet=CAP_NET_BIND_SERVICE

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

We then need to enable it:

```
sudo systemctl enable gortr
```

which then displays:

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

and then we can run GoRTR, like this:

```
sudo systemctl start gortr
```

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

```
sudo systemctl status gortr
```

and you should see something like this:

```
* gortr.service - GoRTR RPKI to Router Server
     Loaded: loaded (/lib/systemd/system/gortr.service; enabled; vendor
preset: e
     Active: active (running) since Fri 2022-01-28 15:52:17 AEST; 25s ago
       Docs: https://github.com/cloudflare/gortr   Main PID: 17490 (gortr)
      Tasks: 11 (limit: 4915)     Memory: 221.1M
     CGroup: /system.slice/gortr.service
             └─17490 /usr/local/bin/stayrtr -bind :3323 -verify=false -cache
/var
```

```
Jan 28 15:52:27 gortr systemd[1]: Starting StayRTR RPKI to Router Server...
Jan 28 15:52:27 gortr systemd[1]: Started StayRTR RPKI to Router Server.
Jan 28 15:52:27 gortr gortr[17490]: time="2022-01-28T15:52:27+10:00"
level=info m
Jan 28 15:52:28 gortr gortr[17490]: time="2022-01-28T15:52:28+10:00"
level=info m
Jan 28 15:52:29 gortr gortr[17490]: time="2022-01-28T15:52:29+10:00"
level=info m
Jan 28 15:52:29 gortr gortr[17490]: time="2022-01-28T15:52:29+10:00"
level=info m
```

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

```
17490 ?        Ssl    0:04 /usr/local/bin/gortr -bind :3323 -verify=false -
cache /var/db/rpki-client/json -checktime=false
```

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

StayRTR

StayRTR is a hard fork of GoRTR which is no longer maintained by Cloudflare. For this reason, I recommend you use StayRTR rather than GoRTR.

As of writing this, there are no packages you can just download and run. So we'll download the source and build from there.

Note: there is an experimental .deb package by Marco d'Itri, but that needs at least Ubuntu 21.04 to support it, as it requires a libc which is 2.32 or newer (Ubuntu 20.04 uses libc version 2.31).

Given it's parentage in GoRTR, the install process is very similar. First off, make sure you have a working Go environment - consult the instructions in the GoRTR section above.

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** (note if you have GoRTR installed too, as I do, you may want to rename its **rtrdump** and **rtrmon** binaries appropriately):

```
cd dist
sudo cp -p stayrtr-0.1-91-gc4ac625-linux-x86_64 /usr/local/bin/stayrtr
sudo cp -p rtrdump-0.1-91-gc4ac625-linux-x86_64 /usr/local/bin/rtrdump
sudo cp -p rtrmon-0.1-91-gc4ac625-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 (/lib/systemd/system/stayrtr.service; enabled; vendor
preset: ena
     Active: active (running) since Fri 2022-01-28 15:50:27 AEST; 25s ago
       Docs: https://github.com/bgp/stayrtr   Main PID: 17390 (stayrtr)
      Tasks: 11 (limit: 4915)     Memory: 241.8M
     CGroup: /system.slice/stayrtr.service
             └─17390 /usr/local/bin/stayrtr -bind :3323 -cache /var/db/rpki-
client/js
Jan 28 15:50:27 stayrtr systemd[1]: Starting StayRTR RPKI to Router
Server...
Jan 28 15:50:27 stayrtr systemd[1]: Started StayRTR RPKI to Router Server.
Jan 28 15:50:27 stayrtr stayrtr[17390]: time="2022-01-28T15:50:27+10:00"
level=info m
```

```
Jan 28 15:50:28 stayrtr stayrtr[17390]: time="2022-01-28T15:50:28+10:00"
level=info m
Jan 28 15:50:29 stayrtr stayrtr[17390]: time="2022-01-28T15:50:29+10:00"
level=info m
Jan 28 15:50:29 stayrtr stayrtr[17390]: time="2022-01-28T15:50:29+10:00"
level=info m
```

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

```
17390 ?        Ssl    0:04 /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 a Cisco IOS-XE platform. (Cisco IOS-XR is not covered here and will likely be different.)

Most commentary is for IOS-XE 16.x onwards. IOS 15.5S (and IOS-XE equivalent) will also likely work with the following examples, but their RPKI implementation is somewhat old and buggy.

Configuration with Validator

Setting up a Cisco router to talk with a validator is simple:

```
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).

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
```

Caveats

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

- Automatically activates route origin validation (cannot 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!) - only relevant if propagating validation information in IBGP (not recommended)
- If validator disappears, router validation database is flushed within a few minutes - fixed in most recent IOS-XE 16.x releases

To turn off the automatic dropping of invalids:

```
router bgp <ASN>
 bgp bestpath prefix-validate allow-invalid
```

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.

Implementing Route Origin Validation

The final step in Cisco IOS-XE is to implement Route Origin Validation. This is achieved simply by turning off the knob we noted above that automatically drops *invalid* prefixes.

```
router bgp <ASN>
 no bgp bestpath prefix-validate allow-invalid
```

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 a validator is:

```
routing-options {
  validation {
    group ISP {
      session <ip address>;
      port <port>;
      refresh-time 600;
      hold-time 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).

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 again needs a policy statement, to be applied as *outbound* policy on all BGP sessions (internal and external).

```
to be done
```

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 (2.0.8 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 300;
}

protocol rpki validator2 {
    roa4 { table rpki4; };
    roa6 { table rpki6; };
    remote <ip address2> port <port2>;
    retry 300;
}
```

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