Cloudflare and RPKI at scale

Louis Poinsignon

Martin J. Levy
Introduction

Louis Poinsignon:

Network Engineer at Cloudflare in San Francisco

Open-source projects including flows and RPKI

Network data collection (BGP, flows, peering-portal)

Featured in RIPE’s video!

https://www.youtube.com/watch?v=Y9vbbxr-Gbl
How did it start?

BGP leaks and cryptocurrencies
24 Apr 2018 by Louis Piarroux

Over the few last hours, a dozen news stories have broken about how an attacker attempted (and perhaps managed) to steal cryptocurrencies using a BGP leak.

The Initial Story

Authoritative DNS route hijack in April 2018
DNS route announced via peering session (in Chicago)
This affected our network, hence our DNS Resolver
What should we do?

The Initial Story

At the time...

150+ PoPs
26,000 BGP sessions
IP space from five RIRs

Just the RIPE Validator [1]

How to distribute a prefix list efficiently?

[1] Cloudflare is very grateful for the RIPE Validator s/w
The Initial Story

July: started deploying internally GoRTR.

August: open-source release.

https://github.com/cloudflare/gortr

September → December:

- Turn up RTR sessions
- Signing prefixes`
Behind the scene (until January 2019)

**RIPE Validator** providing list of prefixes

Running in a Mesos cluster

With a cronjob:
- Fetching the data
- Filtering (remove > /24 and > /48 and duplicates)
- Signing it
- Making it available to our edge

https://rpki.cloudflare.com/rpki.json was born.
Effects

The question everyone asked us:

**How much traffic was affected?**

Many invalids. Little traffic in practice

(we had a default or valid less specific)

Except in one place: Few gigabits per seconds displaced due to geographical more specific
Accounting

Using flows, we see at least 30% of our traffic being valid.

Very little/none invalid.

We use GoFlow for accounting. (Other tools compatible with flows: pmacct, Kentik, etc.)

Traffic with a no ROA
Traffic with a valid ROA
Signing the routes
Signing the routes

Cloudflare has IP space from five RIRs

(no space from twnic/jpnic/cnnic)

<table>
<thead>
<tr>
<th>RIR</th>
<th>Features</th>
<th>Ease of use</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFRINIC</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
</tr>
<tr>
<td>APNIC</td>
<td>⭐⭐</td>
<td>⭐⭐</td>
<td>⭐</td>
</tr>
<tr>
<td>ARIN</td>
<td>⭐⭐</td>
<td>⭐⭐</td>
<td>⭐⭐</td>
</tr>
<tr>
<td>LACNIC</td>
<td>⭐️</td>
<td>⭐⭐⭐</td>
<td>⭐</td>
</tr>
<tr>
<td>RIPE</td>
<td>⭐⭐⭐</td>
<td>⭐⭐⭐</td>
<td>⭐⭐⭐</td>
</tr>
</tbody>
</table>
Rankings

Features: RRDP, 2 factors, extra info, CA

Ease of use: steps to sign a ROA, multi user

API: functional, complete and accessible
Comparison - AFRINIC

Hard to set up: client TLS certificate to create (BPKI) in order to do RPKI.

Buggy.

No RRDP.

No API.

No auto-renew.

Hosted CA possible.

Extensive certificate informations.
Comparison - APNIC

Two factors or client certificate.
   RRDP.
   Auto-renew.
   Allow BGP batch signing.
   (slight bugs with large amount of prefixes).
   Hosted CA possible.

Draft for API:
https://www.apnic.net/manage-ip/apnic-services/services-roadmap/public-api-draft-for-members/
Comparison - ARIN

Two factors. Separate signing key.
No RRDP.
No auto-renew.
Semi-functional API (add).
Dashboard not easy to find.
Hosted CA possible.
Slow rsync update (4 times a day).

Some certificate information.
Comparison - LACNIC

No two factors. Single user.
No RRDP.
No API.
Auto-renew opt-in.
Allow BGP batch signing.
Based off RIPE.
No Hosted CA.
Some extra info (revoked, path).
Incorrect certificate encoding (BER). High turnover of certificate (few days).
Comparison - RIPE

Two factors.
RRDP.
Auto-renew.
Nice API.
Allow BGP batch signing.
No Hosted CA (theoretically).
No extra information. But history.
Incorrect certificate encoding (BER).
Automation

We automated prefixes adding on **ARIN and RIPE** with a **Salt state**.

Two secrets to store (API key and signing key).

Cannot delete or list via API for ARIN: very prone to mistakes if user wants to reduce the amount of ROA files.

```python
def _format_payload(roas, signature):
    template = "\"\"\"-----BEGIN ROA REQUEST-----
    {roas}
    -----END ROA REQUEST-----
    -----BEGIN SIGNATURE-----
    {signature}
    -----END SIGNATURE-----
    "\"\"\"
    payload = template.format(
        roas=roas, signature="\n".join(textwrap.wrap(signature, width=64))
    )
    return payload

def _make_roa(name, asn, t, start_val, end_val, prefix, length, maxlength):
    template = ('\1(time)|\1(name)|\1(asn)|\1(start_val)|\1(end_val)|\1(prefix)|\1(length)|\1(maxlength)\1'
    time_str = calendar.timegm(t.timetuple())
    start_val_str = start_val.strftime(_TIME_FORMAT)
    end_val_str = end_val.strftime(_TIME_FORMAT)
    roa = template.format(
        time=time_str,
        name=name,
        asn=asn,
        start_val=start_val_str,
        end_val=end_val_str,
        prefix=prefix,
        length=length,
        max_length=maxlength,
    )
    return roa

def _sign(pkey, roas):
    signature = pkey.sign(roas.encode('utf-8'), padding.PKCS1v15(), hashes.SHA256())
    return base64.b64encode(signature).decode('utf-8')
```
Why write a new validator?

November 2018: First release of NLnet Labs Routinator 3000 [1]

We were still using RIPE Validator

We wanted something more custom: with monitoring and RRDP

By building it in Golang:

● Many APIs and easy concurrency
● Community doing cryptography
● Cloudflare uses Golang a lot (cfssl, sidh, etc.)

[1] https://github.com/NLnetLabs/routinator
Challenges

Juniper bugs: Routing Validation disabled

Difficulties: rsync, BER encoded instead of DER, conditions in cryptography

The TAL is an ordered sequence of:

1) a URI section,
2) a <CRLF> or <LF> line break,
3) a subjectPublicKeyInfo [RFC5280] in DER format [X.509], encoded in Base64 (see Section 4 of [RFC4648]). To avoid long lines, <CRLF> or <LF> line breaks MAY be inserted into the Base64-encoded string.

Cloudflare’s RPKI Toolkit

Sets of libraries and tools written in Go

Including OctoRPKI 🦅

https://github.com/cloudflare/cfrpki
Cloudflare’s RPKI Toolkit

Libraries

- CER/ROA/MFT decoder
- PKI manager (exploring, validating)
- RRDP/rsync fetcher
- Validation of prefixes

Software

- Local validator (without RRDP/rsync)
- API tools for a distributed version without filesystem
- OctoRPKI
- Certificate Transparency tool
OctoRPKI - Features (1/2)

- Decodes TAL/CER/ROA/MFT
- Explore via Manifest or directory.
- RRDP support (and failover to rsync)
- Monitoring (Prometheus and JSON API which includes logs)
- Dockerizeable
- Handle stability (generate file when done)

https://hub.docker.com/r/cloudflare/octorpki
OctoRPKI - Features (2/2)

- Full compatibility with GoRTR (including signing the JSON file)
- Server + caching options for generated file (CDN friendly)
- Configuration options
  - Disable/Enable components
  - Modes (server, one-off)
- ~5-15 minutes for a full cold-start sync
OctoRPKI - Compute footprint

CPU

125.00%
100.00%
75.00%
50.00%
25.00%
0%

20:30 21:00 21:30 22:00 22:30 23:00

RAM

1.0 GB
0.8 GB
0.6 GB
0.4 GB
0.2 GB
0 B

20:30 21:00 21:30 22:00 22:30 23:00

avg

rpki-benchmark-octorpki 428 MB
rpki-benchmark-ripe 906 MB
rpki-benchmark-routinator 511 MB

OctoRPKI v1.1.3
RIPE Validator v2.25
Routinator v3.3.0
INFO[0842] RRDP: sync https://rrdp.aonic.net/notification.xml
INFO[0844] RRDP: Downloading root notification https://rrdp.aonic.net/notification.xml
INFO[0844] RRDP: finished downloading https://rrdp.aonic.net/notification.xml. Last serial 83980
INFO[0844] RRDP: sync https://rrdp.ripe.net/notification.xml
INFO[0845] RRDP: Downloading root notification https://rrdp.ripe.net/notification.xml
INFO[0845] RRDP: finished downloading https://rrdp.ripe.net/notification.xml. Last serial 83981
INFO[0845] RRDP: sync https://rrdp.rpki.net/notification.xml
INFO[0845] RRDP: Downloading root notification https://rrdp.rpki.net/notification.xml
INFO[0845] RRDP: finished downloading https://rrdp.rpki.net/notification.xml. Last serial 8688
INFO[0847] RRDP: sync https://rpki.cnic.cn/rrdp.notify.xml
INFO[0847] RRDP: Downloading root notification https://rpki.cnic.cn/rrdp.notify.xml
INFO[0847] RRDP: sync https://rpki.arin.net/repository
INFO[0847] RRDP: sync rsync://rpki.arin.net/repository
INFO[0847] RRDP: sync rsync://rpki.arin.net/repository
INFO[0848] RRDP: sync rsync://rpki.arin.net/repository
INFO[0851] RRDP: sync rsync://rpki.arin.net/repository
INFO[0851] RRDP: sync rsync://rpki.arin.net/repository
INFO[0857] RRDP: sync rsync://rpki.ripe.net/ta/ripe-ncc-to-ter
INFO[0858] RRDP: sync rsync://rpki.ripe.net/ta/ripe-ncc-to-ter
INFO[0858] RRDP: sync rsync://rpki.ripe.net/ta/ripe-ncc-to-ter
INFO[0858] RRDP: sync rsync://rpki.ripe.net/ta/ripe-ncc-to-ter
INFO[0858] Stable state. Revalidating in 200ms
OctoRPKI - Run it yourself

$ docker run -ti \
  -p 8080:8080 \
  -v $PWD/cache:/cache \
  -v $PWD/tals/arin.tal:/tals/arin.tal \n  cloudflare/octorpki

Container image

Adding ARIN TAL

Use cache folder on host

Open port 8080 on host

https://hub.docker.com/r/cloudflare/octorpki
GoRTR

OctoRPKI does not embed a RTR server. Modular and independence!
Fully compatible with GoRTR [https://github.com/cloudflare/gortr](https://github.com/cloudflare/gortr)
Signs the prefix list to ensure a safe distribution of the file.
Can run natively on Juniper!
GoRTR

The only software to support plaintext, SSH and TLS as transports

**Compatibility matrix**

A simple comparison between software and devices. Implementations on versions may vary.

<table>
<thead>
<tr>
<th>Device/software</th>
<th>Plaintext</th>
<th>TLS</th>
<th>SSH</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTRdump</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Juniper</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Cisco</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Only SSH password</td>
</tr>
<tr>
<td>Alcatel</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Arista</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>FRRouting</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Only SSH password</td>
</tr>
<tr>
<td>Bird</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Only SSH key</td>
</tr>
<tr>
<td>Quagga</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

https://github.com/cloudflare/gortr#configurations
GoRTR without installing anything

SSH:

rtr.rpki.cloudflare.com:8283 (user: rpki / pass: rpki)

Plaintext:

rtr.rpki.cloudflare.com:8282

Just configure into your router and go!

```
router bgp 65001
  rpki server 192.168.1.100
  transport tcp port 8282
!  
```

https://github.com/cloudflare/gortr#configurations
Cloudflare’s Internal Version

rpki.cloudflare.com
Cloudflare’s Internal Version

Provides https://rpki.cloudflare.com/rpki.json

Also a GraphQL API for the dashboard
Certificate Transparency

Historical records of certificates

Contains a chain (root → ROA)

Sent by our validator

https://ct.cloudflare.com/logs/cirrus
Other data
Other data - so how fresh are those ROAs?

ARIN uses ten year expire

LACNIC random expires

RIPE regenerates certificates!
Future projects
Future projects or ideas

Certificate encoder, ASPA.

More toolings and visualizations around RPKI (BGP collection):

- Integration in our portal [https://peering.cloudflare.com/](https://peering.cloudflare.com/) *(ask for your free access)*
Who validates?
The Probing Project

Could we probe the entire Internet \[1\] to see who is doing validation?

\[1\] v4 IPs that are public and open to probing, around 3.9 billions
Who?

Involved in this project and special thanks:

- Job Snijders: NTT, NLNOG
- Jérôme Fleury, Vasco Asturiano: Cloudflare
Methodology

1. Run two tests with zmap (~2hrs/test):
   one test with behind an RPKI valid prefix and;
   one test behind an RPKI invalid prefix
2. Sum the IPs that replied by range for each test
3. Visualize the ratio of replies between the two tests per prefix
Methodology
Hilbert

1024x1024 → 1 pixel per /20

Ratio

Received with source as invalid prefix
Received with source as valid prefix

Good!
Hilbert

AT&T Announced IP addresses
Learnings

AT&T has ~400k /24.

Easiest to visualize because big blocks.

Could we **automatically** identify networks?

*Going from “0 to 1” to “0 and 1” (0 good and 1 bad)*
Machine learning

Simplify the identification of “validating prefixes”. Requires **training** instead of defining specific ratios.

*Little particles of artificial intelligence*

*Your network*
Training

Take an easy sample to manually identify.
~6,500 /20 prefixes

Assisted by script: (with ASN mapping)

```
$ ./create_training.py \
  --input stripped-data.csv \
  --asn-valids 7018 \
  --asn-invalids 701 \
  --subnets 96.0.0.0/8 97.0.0.0/8 98.0.0.0/8 \
  --output training_data.csv
```
Classification

The SVM model will take a few minutes to run on 433k /20’s.

Model improvements: adds more variables (proximity to other low-ratio prefixes...).
Results

Detected validating:

\(/20\) detected: 28,199

\(/24\) detected: 320k

less than AT&T total prefixes assumed validating:

not everything is responding to ICMP

Work in progress!
Recent Leaks And Conclusions
Summary of Amazon Route Hijack

An attacker announces Amazon Authority DNS prefixes.

Cloudflare and Google accept them in specific locations.

Cloudflare and Google DNS resolvers use this route when clients request the website, the attacker’s server is returned.

The server has a phishing website for the client.

Attacker gather credentials and steals Bitcoins.

Resolver, what is the IP for myetherwallet.com?

Let me ask the authority

Client, its 192.168.1.xx

Hello 192.168.1.xx, This is my user/password

192.168.1.xx

Resolver, its 192.168.1.xx

Amazon, what is the IP for myetherwallet.com?

"TOTALLY REAL"
205.251.195.x

Attacker

205.251.195.x

myetherwallet.com

54.192.146.xx
Summary of Amazon Route Hijack

Amazon did not have signed routes

Cloudflare did not do RPKI validation + route filtering

If RPKI was deployed:

Route would have been rejected because wrong origin

Summary of Verizon Route Leak

A company has two Internet accesses: Verizon and another ISP. The ISP has a BGP optimizer which feeds more-specific routes. Unfortunately, the ISP sends the routes to the company which end up being sent to Verizon. Verizon did not filter them and re-announces them to its peers and clients. Cloudflare loses traffic.

Summary of Verizon Route leak

Cloudflare had signed routes.

Verizon did not filter. Many networks accepted the leak.

*Cloudflare filtering routes did not matter here.*

**If basic filtering was deployed:**

Peering sessions would have been removed when going above prefix threshold.
AS-Path filtering could have avoided accepting routes.

**If RPKI was deployed:**

Routes would have been rejected because wrong length.

What we learned

*RPKI will not be the solution to everything. But in our stories...*

**Filtering** solves Amazon being hijacked

**Signing** helps your network not being leaked
Deploy RPKI now

*Because tomorrow is already too late*

With filtering

Without filtering
Thank you

Questions?

louis@cloudflare.com
@lpoinsig (twitter)