RFC 8210 : The Resource Public Key Infrastructure (RPKI) to Router Protocol, Version 1

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 28 septembre 2017

Date de publication du RFC : Septembre 2017

https://www.bortzmeyer.org/8210.html

Le protocole décrit dans ce RFC fait partie de la grande famille des RFC sur la RPKI <https://www.bortzmeyer.org/securite-routage-bgp-rpki-roa.html>, un ensemble de mécanismes permettant de sécuriser le routage sur l'Internet. Il traite un problème bien particulier : comme la validation des annonces de route, potentiellement complexe en cryptographie, se fait typiquement sur une machine spécialisée, le **cache validateur**, comment communiquer le résultat de ces validations au routeur, qui va devoir accepter ou rejeter des routes ? C'est le rôle du protocole RTR ("*RPKI/Router Protocol*"), un protocole très simple. Ce RFC décrit une nouvelle version de RTR, la version 1, la première (version 0) était dans le RFC 6810 [1].

L'opérateur Internet a donc deux acteurs qui font du RTR entre eux, le cache validateur qui récupère les ROA ("*Route Origin Authorization*", cf. RFC 6482) par rsync, les valide cryptographiquement et garde en mémoire le résultat de cette validation (pour des exemples d'utilisation de deux programmes mettant en œuvre cette fonction, voir mon article sur les logiciels de la RPKI <https://www.bortzmeyer.org/rpki-tests.html>), et le routeur (Cisco, Juniper, Unix avec Quagga, etc). Ce dernier joue un rôle critique et on ne souhaite pas trop charger la barque en lui imposant les opérations assez lourdes du validateur. Dans la plupart des déploiements de la RPKI, le routeur ne fait donc que récupérer des résultats simples (« ce préfixe IP peut être annoncé par cet AS »). Le jeu de données que connait le cache validateur a un numéro de série, qui est incrémenté dès qu'il y a une modification (RFC 1982). Le cache peut donc facilement savoir si un de ses clients routeur est à jour en comparant son numéro de série avec celui du routeur. Le numéro de série est spécifique à un cache donné, il n'a pas de signification globale. Le routeur ne validant pas, le cache qu'il utilise doit donc être une machine de confiance.

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc6810.txt

Un autre acteur, externe à l'opérateur, est la RPKI, l'ensemble des informations de routage publiées (RFC 6481), et qui sont transportées en général en rsync et validées par les signatures cryptographiques qu'elles portent.

Le routeur va ouvrir une connexion avec un ou plusieurs caches et transmettre le numéro de série de ses données. Le cache va lui envoyer les données plus récentes que ce numéro (ou bien rien, si le routeur est à jour). Le cache dispose aussi d'un mécanisme de notification pour dire à tous les routeurs connectés « j'ai du nouveau ». La section 10 décrit plus en détail le processus avec plusieurs caches.

La section 5 décrit le format des paquets : chacun, outre le classique champ version (1 dans ce RFC, c'était 0 dans le RFC précédent), a un type, indiqué par un entier de 8 bits (requête, réponse, etc), le numéro de série de l'envoyeur et un identifiant de session, qui identifie le serveur du cache (pour détecter une remise à zéro d'un cache, par exemple), la longueur du préfixe annoncé et bien sûr le préfixe lui-même.

Les types de paquets les plus utiles (la liste complète est dans un registre IANA <https://www.iana.org/assignments/rpki/rpki.xml#rpki-rtr-pdu>) :

— "*Serial Notify*" (type 0) qui sert au cache à envoyer une notification non sollicitée par les routeurs « J'ai du nouveau ».
— "*Serial Query*" (type 1) où le routeur demande au cache validateur quelles informations il possède, depuis le numéro de série indiqué par le routeur.
— "*Cache Response*" (type 3), la réponse à la requête précédente, composée de :
— "*IPv4 Prefix*" (type 4) et "*IPv6 Prefix*" (type 6) qui indiquent un préfixe IP avec les numéros de systèmes autonomes autorisés à l'annoncer.
— "*End of Data*" (type 7) : comme son nom l'indique.
— "*Error*" (type 10), envoyé lorsque le précédent message avait déclenché un problème.

Le dialogue typique entre un routeur et un cache validateur est décrit en section 8 (le RFC détaille aussi le dialogue de démarrage, ici, je ne montre que celui de fonctionnement régulier) :

— Le routeur demande des données avec un "*Serial Query*" ou "*Reset Query*"),
— Le validateur envoie successivement un "*Cache Response*", plusieurs "*IPvX Prefix*", puis un "*End of Data*".

À noter que puisqu'il y a désormais deux versions de RTR dans la nature, il a fallu mettre en place un système de négociation, décrit dans la section 7. Lorsqu'un cache/validateur connait la version 1 (celle de notre nouveau RFC), et reçoit une requête de version 0 du routeur, il doit se rabattre sur la version 0 (s'il la connait, et veut bien la parler), **ou bien** envoyer un message d'erreur (code 4 « "*Unsupported Protocol Version*" »).

Si c'est l'inverse (routeur récent parlant à un vieux cache/validateur), deux cas, qui dépendent de la politique du validateur. S'il répond avec une erreur, cela peut être intéressant de réessayer avec le protocole 0, si le routeur le connait. Si, par contre, le validateur répond avec des messages de version 0, le routeur peut décider de continuer en version 0.

Les codes d'erreur possibles sont décrits dans la section 12 et font l'objet d'un registre IANA <https://www.iana.org/assignments/rpki/rpki.xml#rpki-rtr-error>. On y trouve par exemple 0 (données corrompues), 1 (erreur interne dans le client ou le serveur), 2 (pas de nouvelles données ; ce n'est pas à proprement parler une erreur).

La question du transport des données entre le cache et le routeur est traitée dans la section 9. Le principal problème est le suivant : comme le but de l'opération RPKI+ROA <https://www.bortzmeyer.org/securite-routage-bgp-rpki-roa.html> est d'augmenter la sécurité du routage, il faut que chaque maillon de la chaîne soit sécurisé. Il ne servirait pas à grand'chose de déployer une telle usine

à gaz de validation si la liaison entre le routeur et le cache permettait à un intermédiaire de changer les données en cours de route. Il était donc tentant de normaliser une technique de sécurité particulière pour le transport, comme par exemple TLS. Malheureusement, il n'existe pas à l'heure actuelle <http://www.ietf.org/mail-archive/web/sidr/current/msg02899.html> de technique accessible sur **tous** les routeurs. L'IETF ayant toujours soin de normaliser des protocoles réalistes, qui puissent être mis en œuvre et déployés, le choix a été fait de ne **pas** imposer une technique de sécurisation particulière. L'interopérabilité en souffre (un cache et un routeur peuvent ne pas avoir de protocole de sécurité en commun) mais c'était la seule option réaliste. (J'ajoute que, si le but de la séparation du routeur et du validateur était de dispenser ce dernier des calculs cryptographiques, il ne serait pas logique de lui imposer ensuite une session protégée cryptographiquement avec le validateur.)

Le RFC se contente donc de donner une préférence à AO ("*TCP Authentication Option*", cf. RFC 5925) en signalant qu'elle sera la technique préférée, dès lors qu'elle sera disponible sur tous les routeurs (ce qui est très loin d'être le cas en 2017, les progrès sont faibles depuis le précédent RFC sur RTR). Les algorithmes du RFC 5926 doivent être acceptés.

En attendant AO, et pour s'assurer qu'il y aura au moins un protocole commun, le RFC spécifie que routeurs et caches doivent pouvoir se parler en utilisant du TCP nu, sans sécurité, vers le port 323. Dans ce cas, le RFC spécifie que routeur et cache validateur doivent être sur le même réseau (idéalement, sur le même segment, cf. section 13) et que des mécanismes, par exemple de contrôle d'accès physiques, empêchent les méchants de jouer avec ce réseau.

Au cas où AO ne soit pas disponible (la quasi-totalité des routeurs aujourd'hui) et où le transport non sécurisé soit jugé nettement trop faible, le RFC présente plusieurs transports sécurisés possibles. Par exemple, SSH (RFC 4252), avec une sécurisation des clés obtenue par un moyen externe (entrée manuelle des clés, par exemple). Le sous-système SSH à utiliser est nommé `rpki-rtr`. Notez que, si SSH n'a pas été rendu obligatoire, alors que quasiment tous les routeurs ont un programme SSH, c'est parce que ce programme n'est pas toujours disponible sous forme d'une bibliothèque, accessible aux applications comme RTR.

Autre alternative de sécurisation évidente, TLS. Le RFC impose une authentification par un certificat client, comportant l'extension `subjectAltName` du RFC 5280, avec une adresse IP (celle du routeur qui se connecte, et que le serveur RTR, la cache validateur, doit vérifier). Le client RTR, lui (le routeur), utilisera le nom de domaine du serveur RTR (le cache) comme entrée pour l'authentification (cf. RFC 6125). N'importe quelle AC peut être utilisée mais notre RFC estime qu'en pratique, les opérateurs créeront leur propre AC, et mettront son certificat dans tous leurs routeurs.

Notez que ce problème de sécurité ne concerne que le transport entre le routeur et le cache validateur. Entre les caches, ou bien entre les caches et les serveurs publics de la RPKI, il n'y a pas besoin d'une forte sécurité, l'intégrité des données est assurée par la validation cryptographique des signatures sur les objets.

Ce RFC spécifie un protocole, pas une politique. Il pourra y avoir plusieurs façons de déployer RTR. La section 11 donne un exemple partiel de la variété des scénarios de déploiement, notamment :

— Le petit site (« petit » est relatif, il fait quand même du BGP) qui sous-traite la validation à un de ses opérateurs. Ses routeurs se connectent aux caches validateurs de l'opérateur.
— Le grand site, qui aura sans doute plusieurs caches en interne, avec peut-être un repli vers les caches de l'opérateur en cas de panne.
— L'opérateur qui aura sans doute plusieurs caches dans chaque POP (pour la redondance).

Pour limiter la charge sur les serveurs rsync de la RPKI (voir RFC 6480), il est recommandé d'organiser les caches de manière hiérarchique, avec peu de caches tapant sur les dépôts rsync publics, et les autres caches se nourrissant (et pas tous en même temps) à partir de ces caches ayant un accès public.

Bien que ce protocole RTR ait été conçu dans le cadre de la RPKI, il peut parfaitement être utilisé avec d'autres mécanismes de sécurisation du routage, puisque RTR se contente de distribuer des autorisations, indépendemment de la manière dont elles ont été obtenues.

Et puis un petit point opérationnel : sur le validateur/cache, n'oubliez pas de bien synchroniser l'horloge, puisqu'il faudra vérifier des certificats, qui ont une date d'expiration…

Quels sont les changements par rapport au RFC original, le RFC 6810, qui normalisait la version 0 de RTR ? Ces changements sont peu nombreux, mais, du fait du changement de version, les versions 0 et 1 sont incompatibles. La section 1.2 résume ce qui a été modifié entre les deux RFC :

— Un nouveau type de paquet, "*Router Key*" (type 9),
— Des paramètres temporels explicites (durée de vie des informations, etc),
— Spécification de la négociation de la version du protocole, puisque les versions 0 et 1 vont coexister pendant un certain temps.

Aujourd'hui, qui met en œuvre RTR ? Il existe en logiciel libre une bibliothèque en C, RTRlib <http://rpki.realmv6.org/>. Elle a fait l'objet d'un bon article de présentation <https://labs.ripe.net/Members/waehlisch/beta-version-of-the-rpki-rtr-client-c-library-released/view> (avec schéma pour expliquer RTR). Elle peut tourner sur TCP nu mais aussi sur SSH. RTRlib permet d'écrire, très simplement, des clients RTR en C, pour déboguer un cache/validateur, ou bien pour extraire des statistiques. Un client simple en ligne de commande est fourni avec, rtrclient. Il peut servir d'exemple (son code est court et très compréhensible) mais aussi d'outil de test simple. rtrclient se connecte et affiche simplement les préfixes reçus. Voici (avec une version légèrement modifiée pour afficher la date) son comportement (en face, le RPKI Validator du RIPE-NCC, dont il faut noter qu'il ne marche pas <https://github.com/RIPE-NCC/rpki-validator/issues/24> avec Java 9, il faut reculer vers une version plus vieille, et qu'il ne gère pas encore la version 1 de RTR <https://github.com/RIPE-NCC/rpki-validator/issues/30>):

```
% rtrclient tcp localhost 8282
...
2012-05-14T19:27:42Z + 195.159.0.0         16-16        5381
2012-05-14T19:27:42Z + 193.247.205.0       24-24       15623
2012-05-14T19:27:42Z + 37.130.128.0        20-24       51906
2012-05-14T19:27:42Z + 2001:1388::         32-32        6147
2012-05-14T19:27:42Z + 2001:67c:2544::     48-48       44574
2012-05-14T19:27:42Z + 178.252.36.0        22-22        6714
2012-05-14T19:27:42Z + 217.67.224.0        19-24       16131
2012-05-14T19:27:42Z + 77.233.224.0        19-19       31027
2012-05-14T19:27:42Z + 46.226.56.0         21-21        5524
2012-05-14T19:27:42Z + 193.135.240.0       21-24         559
2012-05-14T19:27:42Z + 193.247.95.0        24-24       31592
...
```

Autre bibliothèque pour développer des clients RTR, cette fois en Go, ma GoRTR <https://github.com/bortzmeyer/GoRTR>. Le programmeur doit fournir une fonction de traitement des données, qui sera appelée chaque fois qu'il y aura du nouveau. GoRTR est fournie avec deux programmes d'exemple, un qui affiche simplement les préfixes (comme rtrclient plus haut), l'autre qui les stocke dans une base de données, pour étude ultérieure. Par exemple, on voit ici dans cette base de données que la majorité des préfixes annoncés autorisent une longueur de préfixe égale à la leur (pas de tolérance) :

```
essais=> SELECT count(id) FROM Prefixes WHERE serialno=(SELECT max(serialno) FROM Prefixes) AND
   maxlength=masklen(prefix);
 count
-------
  2586
(1 row)

essais=> SELECT count(id) FROM Prefixes WHERE serialno=(SELECT max(serialno) FROM Prefixes) AND
   maxlength>masklen(prefix);
 count
-------
  1110
(1 row)
```

Et voici une partie de évenements d'une session RTR, stockée dans cette base :

```
 tests=> select * from events;
 id |            time            |             server              |              event               | serialno
----+----------------------------+---------------------------------+----------------------------------+----------
 ...
 12 | 2017-06-24 13:31:01.445709 | rpki-validator.realmv6.org:8282 | (Temporary) End of Data          |    33099
 13 | 2017-06-24 13:31:01.447959 | rpki-validator.realmv6.org:8282 | Cache Response, session is 4911 |    33099
 14 | 2017-06-24 13:31:25.932909 | rpki-validator.realmv6.org:8282 | (Temporary) End of Data          |    33099
 15 | 2017-06-24 13:31:38.64284  | rpki-validator.realmv6.org:8282 | Serial Notify #33099 -> #33100  |    33099
 16 | 2017-06-24 13:31:38.741937 | rpki-validator.realmv6.org:8282 | Cache reset                      |    33099
 17 | 2017-06-24 13:31:38.899752 | rpki-validator.realmv6.org:8282 | Cache Response, session is 4911 |    33099
 18 | 2017-06-24 13:32:03.072801 | rpki-validator.realmv6.org:8282 | (Temporary) End of Data          |    33100
 ...
```

Notez que GoRTR gère désormais les deux versions du protocole mais ne connait pas la négociation de version de la section 7 du RFC : il faut indiquer explicitement quelle version on utilise.

Autrement, Cisco et Juniper ont tous les deux annoncé que leurs routeurs avaient un client RTR, et qu'il a passé des tests d'interopérabilité avec trois mises en œuvre du serveur (celles de BBN, du RIPE et de l'ISC). Ces tests d'interopérabilité ont été documentés dans le RFC 7128. Quant aux serveurs, un exemple est fourni par le logiciel RPKI du RIPE-NCC qui fournit un serveur RTR sur le port 8282 par défaut.

En 2013, un client RTR récupèrait 3696 préfixes (ce qui est plus que le nombre de ROA <https://www.ripe.net/lir-services/resource-management/certification/rir-trust-anchor-statistics> puisqu'un ROA peut comporter plusieurs préfixes). C'est encore très loin de la taille de la table de routage globale et il reste donc à voir si l'infrastructure suivra <http://techreports.verisignlabs.com/tr-lookup.cgi?trid=1120005>. (Sur ma station de travail, le validateur, écrit en Java, rame sérieusement.)

Pour les curieux, pourquoi est-ce que l'IETF n'a pas utilisé Netconf (RFC 6241) plutôt que de créer un nouveau protocole ? Les explications sont dans une discussion en décembre 2011 <http://www.mail-archive.com/sidr@ietf.org/msg03845.html>. Le fond de la discussion était « les informations de la RPKI sont des données ou de la configuration » ? La validation est relativement statique mais reste quand même plus proche (fréquence de mise à jour, notamment) des données que de la configuration.

Si vous voulez tester un client RTR, il existe un serveur public rpki-validator.realmv6.org:8282. Voici le genre de choses qu'affiche rtrclient (test fait avec un vieux serveur, qui n'est plus disponible) :

```
% rtrclient tcp rtr-test.bbn.com 12712
...
2012-05-14T19:36:27Z + 236.198.160.184     18-24  4292108787
2012-05-14T19:36:27Z + 9144:8d7d:89b6:e3b8:dff1:dc2b:d864:d49d 105-124 4292268291
2012-05-14T19:36:27Z + 204.108.12.160       8-29  4292339151
2012-05-14T19:36:27Z + 165.13.118.106      27-28  4293698907
2012-05-14T19:36:27Z + 646:938e:20d7:4db3:dafb:6844:f58c:82d5   8-82  4294213839
2012-05-14T19:36:27Z + fd47:efa8:e209:6c35:5e96:50f7:4359:35ba  20-31  4294900047
...
```

Question mise en œuvre, il y a apparemment plusieurs logiciels de validateurs/cache qui mettent en œuvre cette version 1, mais je n'ai pas trouvé lesquels. (Le RPKI Validator <https://www.ripe.net/manage-ips-and-asns/resource-management/certification/tools-and-resources> du RIPE-NCC ne le fait pas, par exemple, ni le serveur public indiqué plus haut, tous les deux ne connaissent que la version 0.)

Wireshark ne semble pas décoder ce protocole. Si vous voulez vous y mettre, un pcap est disponible en (en ligne sur https://www.bortzmeyer.org/files/rtr.pcap).