RFC 8915 : Network Time Security for the Network Time Protocol

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 1 octobre 2020

Date de publication du RFC : Septembre 2020

https://www.bortzmeyer.org/8915.html

Ce nouveau RFC spécifie un mécanisme de sécurité pour le protocole de synchronisation d'horloges NTP. Une heure incorrecte peut empêcher, par exemple, de valider des certificats cryptographiques. Ou fausser les informations enregistrées dans les journaux. Ce mécanisme de sécurité NTS ("*Network Time Security*") permettra de sécuriser le mode client-serveur de NTP, en utilisant TLS pour négocier les clés cryptographiques qui serviront à chiffrer le trafic NTP ultérieur..

La sécurité n'était pas jugée trop importante au début de NTP : après tout, l'heure qu'il est n'est pas un secret. Et il n'y avait pas d'application cruciale dépendant de l'heure. Aujourd'hui, les choses sont différentes. NTP est un protocole critique pour la sécurité de l'Internet. NTP a quand même un mécanisme de sécurité, fondé sur des clés secrètes partagées entre les deux pairs qui communiquent. Comme tout mécanisme à clé partagée, il ne passe pas à l'échelle, d'où le développement du mécanisme « "*Autokey*" » dans le RFC 5906[1]. Ce mécanisme n'a pas été un grand succès et un RFC a été écrit pour préciser le cahier des charges d'une meilleure solution, le RFC 7384. La sécurité de l'Internet repose de plus en plus que l'exactitude de l'horloge de la machine. Plusieurs systèmes de sécurité, comme DNSSEC ou X.509 doivent vérifier des dates et doivent donc pouvoir se fier à l'horloge de la machine. Même criticité de l'horloge quand on veut coordonner une "*smart grid*", ou tout autre processus industriel, analyser des journaux après une attaque, assurer la traçabilité d'opérations financières soumises à régulation <https://eur-lex.europa.eu/legal-content/FR/TXT/?uri=CELEX%3A32014L0065>, etc. Or, comme l'explique bien le RFC 7384, les protocoles de synchronisation d'horloge existants, comme NTP sont peu ou pas sécurisés. (On consultera également avec profit le RFC 8633 qui parle entre autre de sécurité.)

Sécuriser la synchronisation d'horloge est donc crucial aujourd'hui. Cela peut se faire par un protocole extérieur au service de synchronisation, comme IPsec, ou bien par des mesures dans le service

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc5906.txt

de synchronisation. Les deux services les plus populaires sont NTP (RFC 5905) et PTP. Aucun des deux n'offre actuellement toutes les fonctions de sécurité souhaitées par le RFC 7384. Mais les protocoles externes ont un inconvénient : ils peuvent affecter le service de synchronisation. Par exemple, si l'envoi d'un paquet NTP est retardé car IKE cherche des clés, les mesures temporelles vont être faussées. Le paquet arrivera intact mais n'aura pas la même utilité.

Les menaces contre les protocoles de synchronisation d'horloge sont décrites dans le RFC 7384. Les objectifs de notre RFC sont d'y répondre, notamment :

— Permettre à un client d'authentifier son maître, via X.509 tel que l'utilise TLS,
— Protéger ensuite l'intégrité des paquets (via un MAC).
— En prime, mais optionnel, assurer la confidentialité du contenu des paquets.
— Empêcher la traçabilité entre les requêtes d'un même client (si un client NTP change d'adresse IP, un observateur passif ne doit pas pouvoir s'apercevoir qu'il s'agit du même client).
— Et le tout sans permettre les attaques par réflexion <https://www.bortzmeyer.org/attaques-reflexio html> qui ont souvent été un problème avec NTP <https://www.bortzmeyer.org/ntp-reflexion. html>.
— Passage à l'échelle car un serveur peut avoir de très nombreux clients,
— Et enfin performances (RFC 7384, section 5.7).

Rappelons que NTP dispose de plusieurs modes de fonctionnement (RFC 5905, section 3). Le plus connu et sans doute le plus utilisé est le mode client-serveur (si vous faites sur Unix un `ntpdate ntp.nic.fr`, c'est ce mode que vous utilisez). Mais il y aussi un mode symétrique, un mode à diffusion et le « mode » de contrôle (dont beaucoup de fonctions ne sont pas normalisées). Le mode symétrique est sans doute celui qui a le plus d'exigences de sécurité (les modes symétriques et de contrôle de NTP nécessitent notamment une protection réciproque contre le rejeu, qui nécessite de maintenir un état de chaque côté) mais le mode client-serveur est plus répandu, et pose des difficultés particulières (un serveur peut avoir beaucoup de clients, et ne pas souhaiter maintenir un état par client). Notre RFC ne couvre que le mode client-serveur. Pour ce mode, la solution décrite dans notre RFC 8915 est d'utiliser TLS afin que le client authentifie le serveur, puis d'utiliser cette session TLS pour générer la clé qui servira à sécuriser du NTP classique, sur UDP. Les serveurs NTP devront donc désormais également écouter en TCP, ce qu'ils ne font pas en général pas actuellement. La partie TLS de NTS ("*Network Time Security*", normalisé dans ce RFC) se nomme NTS-KE (pour "*Network Time Security - Key Exchange*"). La solution à deux protocoles peut sembler compliquée mais elle permet d'avoir les avantages de TLS (protocole éprouvé) et d'UDP (protocole rapide).

Une fois qu'on a la clé (et d'autres informations utiles, dont les "*cookies*"), on ferme la session TLS, et on va utiliser la cryptographie pour sécuriser les paquets. Les clés sont dérivées à partir de la session TLS, suivant l'algorithme du RFC 5705. Quatre nouveaux champs NTP sont utilisés pour les paquets suivants (ils sont présentés en section 5), non TLS. Dans ces paquets se trouve un "*cookie*" qui va permettre au serveur de vérifier le client, et de récupérer la bonne clé. Les "*cookies*" envoyés du client vers le serveur et en sens inverse sont changés à chaque fois, pour éviter toute traçabilité, car ils ne sont pas dans la partie chiffrée du paquet. Notez que le format des "*cookies*" n'est pas spécifié par ce RFC (bien qu'un format soit suggéré en section 6). Pour le client, le "*cookie*" est opaque : on le traite comme une chaîne de bits, sans structure interne.

On a dit que NTS ("*Network Time Security*") utilisait TLS. En fait, il se restreint à un profil spécifique de TLS, décrit en section 3. Comme NTS est un protocole récent, il n'y a pas besoin d'interagir avec les vieilles versions de TLS et notre RFC impose donc TLS 1.3 (RFC 8446) au minimum, ALPN (RFC 7301), les bonnes pratiques TLS du RFC 7525, et les règles des RFC 5280 et RFC 6125 pour l'authentification du serveur.

Passons ensuite au détail du protocole NTS-KE ("*Network Time Security - Key Establishment*") qui va permettre, en utilisant TLS, d'obtenir les clés qui serviront au chiffrement symétrique ultérieur. Il est spécifié dans la section 4 du RFC. Le serveur doit écouter sur le port ntske (4460 par défaut) <https://

www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xml>. Le client utilise ALPN (RFC 7301) pour annoncer qu'il veut l'application ntske/1. On établit la session TLS, on échange des messages et on raccroche (l'idée est de ne pas obliger le serveur à garder une session TLS ouverte pour chacun de ses clients, qui peuvent être nombreux).

Le format de ces messages est composé d'un champ indiquant le type du message, de la longueur du message, des données du message, et d'un bit indiquant la criticité du type de message. Si ce bit est à 1 et que le récepteur ne connait pas ce type, il raccroche. Les types de message actuels sont notamment :

- "*Next Protocol Negotiation*" (valeur 1) qui indique les protocoles acceptés (il n'y en a qu'un à l'heure actuelle, NTP, mais peut-être d'autres protocoles comme PTP suivront, la liste est dans un registre IANA <https://www.iana.org/assignments/nts/nts.xml#nts-next-protocols>),
- "*Error*" (valeur 2) qui indique...une erreur (encore un registre IANA de ces codes d'erreur <https://www.iana.org/assignments/nts/nts.xml#nts-error-codes>, qui seront sans doute moins connus que ceux de HTTP <https://www.bortzmeyer.org/http-code-emoji.html>),
- "*AEAD Algorithm Negotiation*" (valeur 3) ; NTS impose l'utilisation du chiffrement intègre (AEAD pour "*Authenticated Encryption with Associated Data*") et ce type de message permet de choisir un algorithme de chiffrement intègre parmi ceux enregistrés <https://www.iana.org/assignments/aead-parameters/aead-parameters.xml#aead-parameters-2>,
- "*New Cookie*" (valeur 5), miam, le serveur envoie un "*cookie*", que le client devra renvoyer (sans chercher à la comprendre), ce qui permettra au serveur de reconnaitre un client légitime ; le RFC recommande d'envoyer plusieurs messages de ce type, pour que le client ait une provision de "*cookies*" suffisante,
- "*Server Negotiation*" (valeur 6) (et "*Port Negotiation*", valeur 7), pour indiquer le serveur NTP avec lequel parler de façon sécurisée, serveur qui n'est pas forcément le serveur NTS-KE.

D'autres types de message pourront venir dans le futur, cf. le registre <https://www.iana.org/assignments/nts/nts.xml#nts-key-establishment-record-types>.

Une fois la négociation faite avec le protocole NTS-KE, tournant sur TLS, on peut passer au NTP normal, avec le serveur indiqué. Comme indiqué plus haut, quatre champs supplémentaires ont été ajoutés à NTP pour gérer la sécurité. Ils sont présentés dans la section 5 de notre RFC. Les clés cryptographiques sont obtenues par la fonction de dérivation (HKDF) du RFC 8446, section 7.5 (voir aussi le RFC 5705). Les clés du client vers le serveur sont différentes de celles du serveur vers le client (ce sont les clés c2s et s2c dans l'exemple avec ntsclient montré plus loin).

Les paquets NTP échangés ensuite, et sécurisés avec NTS, comporteront l'en-tête NTP classique (RFC 5905, section 7.3), qui ne sera pas chiffré (mais sera authentifié), et le paquet original chiffré dans un champ d'extension (cf. RFC 5905, section 7.5, et RFC 7822). Les quatre nouveaux champs <https://www.iana.org/assignments/ntp-parameters/ntp-parameters.xml#ntp-parameters-3> seront :

- L'identifiant du paquet (qui sert à détecter le rejeu), il sera indiqué (cf. section 5.7) dans la réponse du serveur, permetant au client de détecter des attaques menées en aveugle par des malfaisants qui ne sont pas situés sur le chemin (certaines mises en œuvre de NTP utilisaient les estampilles temporelles pour cela, mais elles sont courtes, et en partie prévisibles),
- Le "*cookie*",
- La demande de "*cookies*" supplémentaires,
- Et le principal, le champ contenant le contenu chiffré du paquet NTP original.

On a vu que le format des "*cookies*" n'était pas imposé. Cela n'affecte pas l'interopérabilité puisque, de toute façon, le client n'est pas censé comprendre les "*cookies*" qu'il reçoit, il doit juste les renvoyer tels quels. La section 6 décrit quand même un format suggéré. Elle rappelle que les "*cookies*" de NTS sont utilisés à peu près comme les "*cookies*" de session de TLS (RFC 5077). Comme le serveur doit pouvoir reconnaitre les bons "*cookies*" des mauvais, et comme il est préférable qu'il ne conserve pas un état différent par client, le format suggéré permet au serveur de fabriquer des "*cookies*" qu'il pourra reconnaitre, sans qu'un attaquant n'arrive à en fabriquer. Le serveur part pour cela d'une clé secrète, changée de temps en temps. Pour le cas un peu plus compliqué où un ou plusieurs serveurs NTP assureraient

le service avec des "*cookies*" distribués par un serveur NTS-KE séparé, le RFC recommande que les clés suivantes soient dérivées de la première, par un cliquet et la fonction de dérivation HKDF du RFC 5869.

Quelles sont les mises en œuvre de NTS à l'heure actuelle ? La principale est dans chrony (mais dans le dépôt git seulement, la version 3.5 n'a pas encore NTS). chrony est écrit en C et comprend un client et un serveur. NTS est compilé par défaut (cela peut être débrayé avec l'option `--disable-nts` du script `./configure`), si et seulement si les bibliothèques sur lesquelles s'appuie chrony (comme la bibliothèque cryptographique nettle) ont tout ce qu'il faut. Ainsi, sur une Ubuntu stable, `./configure` n'active pas l'option NTS alors que ça marche sur une Debian instable (sur cette plate-forme, pensez à installer les paquetages `bison` et `asciidoctor`, `./configure` ne vérifie pas leur présence). Cela se voit dans cette ligne émise par `./configure` (le `+NTS`) :

```
% ./configure
   ...
   Checking for SIV in nettle : No
   Checking for SIV in gnutls : Yes
   Features : +CMDMON +NTP +REFCLOCK +RTC -PRIVDROP -SCFILTER -SIGND +ASYNCDNS +NTS +READLINE +SECHASH +IPV
```

Autre serveur ayant NTS, NTPsec <https://www.ntpsec.org/>. (Développement sur Github <https://gitlab.com/NTPsec/ntpsec>.) Écrit en C. C'est ce code qui est utilisé pour deux serveurs NTS publics, `ntp1.glypnod.com:123` et `ntp2.glypnod.com:123` (exemples plus loin).

Il y a également une mise en œuvre de NTS <https://press.netnod.se/posts/news/netnod-provides- en FPGA (sur Github <https://github.com/Netnod/FPGA_NTP_SERVER/>). La même organisation gère deux serveurs NTP publics, `nts.sth1.ntp.se:4460` et `nts.sth2.ntp.se:4460`. Elle a publié une bonne synthèse de ses travaux <https://labs.ripe.net/author/christer-weinigel/implementing-network-time-security-at-the-hardware-level/> et un article plus détaillé <https://www.netnod.se/sites/default/files/2021-01/Netnod_NTS_Whitepaper_2020.pdf>.

Dernier serveur public ayant NTS, celui de Cloudflare <https://blog.cloudflare.com/nts-is-now-rfc/>, `time.cloudflare.com`. Il utilise sans doute nts-rust <https://github.com/wbl/nts-rust>, écrit par Cloudflare (en Rust).

Les autres mises en œuvre de NTS semblent assez expérimentales, plutôt de la preuve de concept pas très maintenue. Il y a :

— , écrit en C++.
— , écrit en Python.
— , écrit en Go, mais client seulement, voir l'exemple plus loin.

Voici un exemple avec ntsclient <https://gitlab.com/hacklunch/ntsclient> et les serveurs publics mentionnés plus haut. Vérifiez que vous avez un compilateur Go puis :

```
% git clone https://gitlab.com/hacklunch/ntsclient.git
% make
% ./ntsclient --server=ntp1.glypnod.com:123 -n
Network time on ntp1.glypnod.com:123 2020-07-14 09:01:08.684729607 +0000 UTC. Local clock off by -73.844479
```

(Le serveur devrait migrer bientôt vers le port 4460.) Si on veut voir plus de détails et toute la machinerie NTS (le type de message <https://www.iana.org/assignments/nts/nts.xml#nts-key-establishm 4 est AEAD, le type 5 le "*cookie*", l'algorithme <https://www.iana.org/assignments/aead-parameters/aead-parameters.xml#aead-parameters-2> 15 est `AEAD_AES_SIV_CMAC_256`) :

```
% ./ntsclient --server=ntp1.glypnod.com:123 -n --debug
Conf: &main.Config{Server:"ntp1.glypnod.com:123", CACert:"", Interval:1000}
Connecting to KE server ntp1.glypnod.com:123
Record type 1
Critical set
Record type 4
Record type 5
Record type 5
Record type 5
Record type 5
Record type 5
Record type 5
Record type 5
Record type 5
Record type 0
Critical set
NTSKE exchange yielded:
  c2s: ece2b86a7e86611e6431313b1e45b02a8665f732ad9813087f7fc773bd7f2ff9
  s2c: 818effb93856caaf17e296656a900a9b17229e2f79e69f43f9834d3c08194c06
  server: ntp1.glypnod.com
  port: 123
  algo: 15
  8 cookies:
  [puis les cookies]
```

Notez que les messages de type 5 ont été répétés huit fois, car, conformément aux recommandations du RFC, le serveur envoie huit "*cookies*". Notez aussi que si vous voulez analyser avec Wireshark, il va par défaut interpréter ce trafic sur le port 123 comme étant du NTP et donc afficher n'importe quoi. Il faut lui dire explicitement de l'interpréter comme du TLS ("*Decode as...*"). On voit le trafic NTS-KE en TCP puis du trafic NTP plus classique en UDP.

Enfin, la section 8 du RFC détaille quelques points de sécurité qui avaient pu être traités un peu rapidement auparavant. D'abord, le risque de dDoS. NTS, décrit dans notre RFC, apporte une nouveauté dans le monde NTP, la cryptographie asymétrique. Nécessaire pour l'authentification du serveur, elle est bien plus lente que la symétrique et, donc, potentiellement, un "*botnet*" pourrait écrouler le serveur sous la charge, en le forçant à faire beaucoup d'opérations de cryptographie asymétrique. Pour se protéger, NTS sépare conceptuellement l'authentification (faite par NTS-KE) et le service NTP à proprement parler. Ils peuvent même être assurés par des serveurs différents, limitant ainsi le risque qu'une attaque ne perturbe le service NTP.

Lorsqu'on parle de NTP et d'attaques par déni de service, on pense aussi à l'utilisation de NTP dans les attaques par réflexion et amplification <https://www.bortzmeyer.org/ntp-reflexion.html>. Notez qu'elles utilisent en général des fonctions non-standard des serveurs NTP. Le protocole lui-même n'a pas forcément de défauts. En tout cas, NTS a été conçu pour ne pas ajouter de nouvelles possibilités d'amplification. Tous les champs envoyés par le serveur ont une taille qui est au maximum celle des champs correspondants dans la requête du client. C'est par exemple pour cela que le client doit envoyer un champ de demande de "*cookie*", rempli avec des zéros, pour **chaque** cookie supplémentaire qu'il réclame. Cela peut sembler inutile, mais c'est pour décourager toute tentative d'amplification.

Cette section 8 discute également la vérification du certificat du serveur. Bon, on suit les règles des RFC 5280 et RFC 6125, d'accord. Mais cela laisse un problème amusant : la date d'expiration du certificat. Regardons celui des serveurs publics cités plus haut, avec `gnutls-cli` :

```
% gnutls-cli ntp1.glypnod.com:123
Processed 126 CA certificate(s).
Resolving 'ntp1.glypnod.com:123'...
Connecting to '104.131.155.175:123'...
```

```
- Certificate type: X.509
- Got a certificate list of 2 certificates.
- Certificate[0] info:
 - subject 'CN=ntp1.glypnod.com', issuer 'CN=Let's Encrypt Authority X3,O=Let's Encrypt,C=US', serial 0x04f3
Public Key ID:
sha1:726426063ea3c388ebcc23f913b41a15d4ef38b0
sha256:94b8f942c2c7f0cf0f8cb4967ba48d957bf87f1540c88e947a9f7d456b24bce5
Public Key PIN:
pin-sha256:lLj5QsLH8M8PjLSWe6SNlXv4fxVAyI6Uep99RWskvOU=

- Certificate[1] info:
 - subject 'CN=Let's Encrypt Authority X3,O=Let's Encrypt,C=US', issuer 'CN=DST Root CA X3,O=Digital Signatu
- Status: The certificate is trusted.
- Description: (TLS1.3-X.509)-(ECDHE-SECP256R1)-(RSA-PSS-RSAE-SHA256)-(AES-256-GCM)
- Options:
- Handshake was completed
```

On a un classique certificat Let's Encrypt, qui expire le 18 août 2020. Cette date figure dans les certificats pour éviter qu'un malveillant qui aurait mis la main sur la clé privée correspondant à un certificat puisse profiter de son forfait éternellement. Même si la révocation du certificat ne marche pas, le malveillant n'aura qu'un temps limité pour utiliser le certificat. Sauf que la vérification que cette date d'expiration n'est pas atteinte dépend de l'horloge de la machine. Et que le but de NTP est justement de mettre cette horloge à l'heure…On a donc un problème d'œuf et de poule : faire du NTP sécurisé nécessite NTS, or utiliser NTS nécessite de vérifier un certificat, mais vérifier un certificat nécessite une horloge à l'heure, donc que NTP fonctionne. Il n'y a pas de solution idéale à ce problème. Notre RFC suggère quelques trucs utiles, comme :

— Prendre un risque délibéré en ne vérifiant pas la date d'expiration du certificat,
— Chercher à savoir si l'horloge est correcte ou pas ; sur un système d'exploitation qui dispose de fonctions comme `ntp_adjtime`, le résultat de cette fonction (dans ce cas, une valeur autre que `TIME_ERROR`) permet de savoir si l'horloge est correcte,
— Permettre à l'administrateur système de configurer la validation des dates des certificats ; s'il sait que la machine a une horloge matérielle sauvegardée, par exemple via une pile, il peut demander une validation stricte,
— Utiliser plusieurs serveurs NTP et les comparer, dans l'espoir qu'ils ne soient pas tous compromis (cf. RFC 5905, section 11.2.1).

Comme toujours avec la cryptographie, lorsqu'il existe une version non sécurisée d'un protocole (ici, le NTP traditionnel), il y a le risque d'une attaque par repli, qui consiste à faire croire à une des deux parties que l'autre partie ne sait pas faire de la sécurité (ici, du NTS). Ce "*NTP stripping*" est possible si, par exemple, le client NTP se rabat en NTP classique en cas de problème de connexion au serveur NTS-KE. Le RFC recommande donc que le repli ne se fasse pas par défaut, mais uniquement si le logiciel a été explicitement configuré pour prendre ce risque.

Enfin, si on veut faire de la synchronisation d'horloges sécurisée, un des problèmes les plus difficiles est l'attaque par retard (section 11.5 du RFC). Le MAC empêche un attaquant actif de modifier les messages mais il peut les retarder, donnant ainsi au client une fausse idée de l'heure qu'il est réellement (RFC 7384, section 3.2.6 et Mizrahi, T., « "*A game theoretic analysis of delay attacks against time synchronization protocols*" », dans les "*Proceedings of Precision Clock Synchronization for Measurement Control and Communication*" à ISPCS 2012). Ce dernier article n'est pas en ligne mais, heureusement, il y a Sci-Hub (`DOI 10.1109/ISPCS.2012.6336612`).

Les contre-mesures possibles ? Utiliser plusieurs serveurs, ce que fait déjà NTP. Faire attention à ce qu'ils soient joignables par des chemins différents (l'attaquant devra alors contrôler plusieurs chemins). Tout chiffrer avec IPsec, ce qui n'empêche pas l'attaque mais rend plus difficile l'identification des seuls paquets de synchronisation.

Revenons au mode symétrique de NTP, qui n'est pas traité par notre RFC. Vous vous demandez peut-être comment le sécuriser. Lors des discussions à l'IETF sur NTS, il avait été envisagé d'encapsuler tous les paquets dans DTLS mais cette option n'a finalement pas été retenue. Donc, pour le symétrique, la méthode recommandée est d'utiliser le MAC des RFC 5905 (section 7.3) et RFC 8573. Et le mode à diffusion ? Il n'y a pas pour l'instant de solution à ce difficile problème.

Et la vie privée (section 9) ? Il y a d'abord la question de la non-traçabilité. On ne veut pas qu'un observateur puisse savoir que deux requêtes NTP sont dues au même client. D'où l'idée d'envoyer plusieurs "*cookies*" (et de les renouveler), puisque, autrement, ces identificateurs envoyés en clair trahiraient le client, même s'il a changé d'adresse IP.

Et la confidentialité ? Le chiffrement de l'essentiel du paquet NTP fournit ce service, les en-têtes qui restent en clair, comme le "*cookie*", ne posent pas de problèmes de confidentialité particuliers.

Un petit mot maintenant sur l'historique de ce RFC. Le projet initial était plus ambitieux (cf. l'"*Internet-Draft*" `draft-ietf-ntp-network-time-security`). Il s'agissait de développer un mécanisme abstrait, commun à NTP et PTP/IEEE 1588 (alors que ce RFC 8915 est spécifique à NTP). Un autre groupe de travail IETF TICTOC <https://datatracker.ietf.org/wg/tictoc> continue de son côté mais ne fait plus de sécurité.

Si vous voulez des bonnes lectures sur NTS, autres que le RFC, il y a un article sur la sécurité NTP à la conférence ISPCS 17 <https://www.internetsociety.org/wp-content/uploads/2017/10/ispcs-2017-time-security-final-copyright.pdf>, « "*New Security Mechanisms for Network Time Synchronization Protocols*" ») (un des auteurs de l'article est un des auteurs du RFC). C'est un exposé général des techniques de sécurité donc il inclut PTP mais la partie NTP est presque à jour, à part la suggestion de DTLS pour les autres modes, qui n'a finalement pas été retenue). Les supports du même article à la conférence sont sur Slideshare <https://www.slideshare.net/ISOCtech/new-security-mechanisms-for-network-time-synchronization-protocols>. Et cet article est résumé dans l'IETF journal de novembre 2017, volume 13 issue 2 <https://www.ietfjournal.org/a-new-security-mechanism-for-the-network-time-protocol/>. Il y a aussi l'article de l'ISOC <https://www.internetsociety.org/blog/2020/10/nts-rfc-published-new-standard-to-en>, qui résume bien les enjeux.