RFC 9421 : HTTP Message Signatures

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 17 janvier 2025

Date de publication du RFC : Février 2024

https://www.bortzmeyer.org/9421.html

Ce RFC spécifie comment on peut signer cryptographiquement et vérifier des messages HTTP, afin de s'assurer de leur authenticité. Des signatures HTTP conceptuellement proches, mais syntaxiquement différentes sont notamment utilisées par le fédivers. (Notez que le RFC, délibérément, ne spécifie pas un protocole complet mais une des briques, qui devra impérativement être complétée avec d'autres règles.)

Alors, attention, vous allez me dire « mais tout le monde utilise HTTPS, de toute façon » mais ce n'est pas tout à fait la même chose. Comme l'explique la section 1 de notre RFC, TLS protège une connexion mais ne fonctionne pas de bout en bout, s'il y a sur le trajet des relais qui terminent une session TLS et ouvrent une autre connexion (la section 7.1.2 détaille les limites de TLS pour les scénarios envisagés). Et TLS ne permet pas d'authentifier le client, sauf à utiliser des certificats client, que beaucoup trouvent peu pratique, et qui dépendent d'un système centralisé, les AC. En prime, dans un contexte d'utilisation serveur-serveur, comme dans le fédivers, on n'a pas envie de confier sa clé privée à son serveur. Donc, il y a un besoin pour une signature des messages (TLS n'est pas inutile pour autant, il fournit notamment la confidentialité, cf. section 8.2 de notre RFC). Voilà pourquoi Mastodon, par exemple, a commencé à utiliser des signatures HTTP pour la sécurité. Le format ActivityPub ne prévoyait en effet aucune sécurité, d'où la nécessité de développer des techniques "*ad hoc*". Mais notez que la technique de Mastodon, non normalisée et même pas spécifiée quelque part, sauf dans le code source, n'est pas compatible avec celle du RFC <https://github.com/mastodon/mastodon/issues/29905>. Un travail est en cours pour documenter cela <https://swicg.github.io/activitypub-http-signature/>.

Notons qu'il existe aussi des solutions pour signer le corps d'un message HTTP, comme JWS (RFC 7515 [1]) si le corps est en JSON (RFC 8259). Mais les signatures HTTP de notre RFC 9421 protègent non seulement le corps (quel que soit son format) mais aussi une partie des champs de l'en-tête. Il n'y a pas de solution générale de protection des contenus en HTTP, juste une protection du canal de communication, avec HTTPS.

Donc, le principe est simple : le signeur canonicalise certains champs de l'en-tête (et, indirectement, le corps, via la condensation du RFC 9530), les signe et met la signature dans l'en-tête. Par exemple, avec ce message HTTP (une simple demande de lecture, qui n'a donc probablement pas besoin d'être authentifiée mais c'est juste un exemple) :

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc7515.txt

```
GET /9421.html HTTP/1.1
Host: www.bortzmeyer.org
Date:  Fri, 17 Jan 2025 10:53:17 +0100
Content-Type: text/plain
Content-Digest: sha-256=:591b6607e9e257e26808e2ccf3984c23a5742b78defad9ec7b2966ddcef29909=:

Hello, HTTP
```

La signature ajoutera ces deux champs :

```
Signature-Input: sig=("date" "host";sf "content-type";sf "content-digest" "@method" "@target-uri" "@authorit
Signature: sig=:0dnCVeCDOJ9fQiLqUXLE8hwBDNpAVlO3TJZgqm/FBJ1x2h5/g5qU20UM87BkJt0iK9vpRRgPF9fmWobf6Y5iYQ==:
```

(≪ sig ≫ est un identificateur arbitraire, qui sert à faire la correspondance entre une signature et ses paramètres, dans le cas où il y a plusieurs signatures.) Le vérificateur devra récupérer la clé publique (notez que le RFC ne précise pas comment : chaque application utilisant les signatures HTTP devra décrire sa méthode de récupération des clés), canonicaliser et refaire les calculs cryptographiques, validant (ou non) la signature.

Voilà, vous connaissez l'essentiel du RFC, maintenant, place aux détails. D'abord, un peu de terminologie : en dépit de leur nom, les signatures HTTP peuvent être aussi bien de ≪ vraies ≫ signatures, faites avec de la cryptographie asymétrique que des MAC incluant une clé secrète (avec de la cryptographie symétrique, donc). Ensuite, relisez bien le RFC 9651 : les champs structurés dans l'en-tête sont un cas particulier, en raison de la canonicalisation qu'ils vont subir. (Vous avez remarqué le `sf` dans le champ `Signature-Input:` plus haut ? Il veut dire "*Structured Field*" et indique un traitement particulier.)

Ensuite, la question évidemment compliquée de la canonicalisation. HTTP permet et même parfois impose des transformations du message faites par des intermédiaires (la section 1.3 donne une liste partielle des transformations possibles). Il est très difficile, voire impossible, de définir quels champs de l'en-tête vont être respectés et lesquels vont être modifiés. Il n'y a donc pas de liste officielle des champs à signer, chaque application va indiquer les siens. Dans l'exemple ci-dessus, on signe les quatre champs présents, et on l'indique dans le `Signature-Input:`. Si un intermédiaire ajoute un champ quelconque (par exemple le `Forwarded:` du RFC 7239), la signature restera valide. Le récepteur ne devra donc pas considérer ce champ comme authentifié. Même chose pour le corps si on ne signe pas son condensat (RFC 9530). La section 7.2.8 du RFC rappelle l'importance de signer le corps, pour la plupart des cas. Notez également que le `Signature-Input:` plus haut incluait aussi des composants qui ne sont pas des champs de l'en-tête mais qu'on veut signer, comme la méthode (`GET`) ou les composants de l'URL. Leur nom est précédé d'un arobase.

Autre problème du monde réel : les programmes n'ont pas un contrôle complet du message HTTP produit. Par exemple, le programme va utiliser une bibliothèque qui va formater les champs à sa manière, en ajouter, etc. C'est pour cela qu'il est important de laisser ouverte la question de la liste des champs à signer. Chaque application choisira.

Les signatures HTTP ne sont pas une solution complète prête à l'emploi : bien des points sont délibérément laissés vides dans le RFC. Une solution complète de sécurité doit donc spécifier :

— quels sont les champs qui doivent être signés, la section 7.2.3 fournissant des indications sur ce choix (par exemple, Mastodon impose que `Date:` soit présent et signé, pour détecter les messages trop anciens que quelqu'un essaierait de rejouer, cf. la section 7.2.2),

— comment récupérer la clé publique d'un correspondant (sur le fédivers, c'est fait en utilisant WebFinger, système normalisé dans le RFC 7033),
— et les autres exigences diverses spécifiques à cette application (la section 3.2.1 donne une bonne idée de ce qu'elles peuvent être).

Les noms des composants signés (champs de l'en-tête et autres) sont décrits dans la section 2 de notre RFC. Pour un champ, c'est simplement son nom en minuscules (comme `"date"` dans le `Signature-Input:` plus haut). Autrement, les noms commencent par un arobase par exemple `@method` indique la méthode HTTP utilisée. Les méthodes de canonicalisation (complexes, surtout pour les champs structurés !) sont dans la même section.

Les différents paramètres de la signature, comme un identificateur pour la clé utilisée, ou comme la date de signature, peuvent également être inclus dans le `Signature-Input:`.

Tous ces paramètres forment ce qu'on appelle la base (section 2.5). Celle du message donné comme exemple plus haut est :

```
"date": Fri, 17 Jan 2025 10:53:17 +0100
"host";sf: www.bortzmeyer.org
"content-type";sf: text/plain
"content-digest": sha-256=:591b6607e9e257e26808e2ccf3984c23a5742b78defad9ec7b2966ddcef29909=:
"@method": GET
"@target-uri": https://www.bortzmeyer.org/9421.html
"@authority": www.bortzmeyer.org
"@scheme": https
"@request-target": /9421.html
"@path": /9421.html
"@query": ?
"@signature-params": ("date" "host";sf "content-type";sf "content-digest" "@method" "@target-uri" "@authority" "
```

Ici, elle est relativement courte, la plupart des métadonnées n'étant pas mentionnée.

Une fois la signature générée, le signeur ajoute deux champs à l'en-tête (section 4 du RFC) :

— `Signature-Input:` qui indique les paramètres de la signature (liste des champs signés, et optionnellement métadonnées sur la signature),
— Et `Signature:`, qui contient la signature elle-même.

Les deux champs sont structurés selon la syntaxe du RFC 9651. (Notez que ces deux champs sont une des différences avec l'ancienne version des signatures HTTP, utilisée dans le fédivers. Cette ancienne version n'avait qu'un champ, `Signature:`. Un vérificateur qui veut gérer les deux versions peut donc utiliser la présence du champ `Signature-Input:` comme indication que la version utilisée est celle du RFC. L'annexe A expose cette heuristique, qui figure également dans le projet d'intégration avec ActivityPub `<https://swicg.github.io/activitypub-http-signature/#how-to-upgrade-supported-versions>`.)

Voici un exemple de signature avec des métadonnées (date de signature et identifiant de la clé :

```
Signature-Input: reqres=("@status" "content-digest" "content-type" \
     "@authority";req "@method";req "@path";req "content-digest";req)\
     ;created=1618884479;keyid="test-key-ecc-p256"
Signature: reqres=:dMT/A/76ehrdBTD/2Xx8QuKV6FoyzEP/I9hdzKN8LQJLNgzU\
     4W767HK05rx1i8meNQQgQPgQp8wq2ive3tV5Ag==:
```

Notez que notre RFC décrit aussi une méthode pour demander qu'un correspondant signe ses messages : inclure un champ `Accept-Signature:` (section 5 du RFC).

Les signatures HTTP nécessitent la modification ou la création de plusieurs registres IANA :

— Les champs `Accept-Signature:`, `Signature:` et `Signature-Input:` ont été ajoutés au registre des champs d'en-tête HTTP <https://www.iana.org/assignments/http-fields/http-fields.xml#field-names>.
— Un registre des algorithmes de signature <https://www.iana.org/assignments/http-message-signn http-message-signature.xml#signature-algorithms> a été créé. On va y trouver par exemple `ecdsa-p256-sha256`, que j'ai utilisé dans mon exemple (algorithme ECDSA). On peut y ajouter des algorithmes via la politique « Spécification nécessaire » du RFC 8126.
— Un registre des métadonnées des signatures <https://www.iana.org/assignments/http-message-si http-message-signature.xml#signature-metadata-parameters> a été créé. On y trouve par exemple `created` (la date de signature) ou `keyid` (l'identificateur de la clé, selon un schéma de nommage spécifique à l'application). On peut y ajouter des valeurs via la politique « Examen par un expert » du RFC 8126.
— Un registre des noms de composants <https://www.iana.org/assignments/http-message-signatu http-message-signature.xml#signature-derived-component-names> a été créé, pour mettre ces noms commençant par « @ », comme `@method`. On peut y ajouter des noms via la politique « Examen par un expert » du RFC 8126.
— Un registre des noms des paramètres des composants <https://www.iana.org/assignments/http-message-signature/http-message-signature.xml#component-parameters> a été créé. Vous avez déjà rencontré `sf`, par exemple, qui indique qu'un champ de l'en-tête est un champ structuré et doit donc être traité de manière spéciale. On peut y ajouter des noms via la politique « Examen par un expert » du RFC 8126.

Si vous voulez un article d'introduction :

— Un bon article de synthèse <https://blog.valerauko.net/2024/12/08/http-signatures-are-rf >.

Questions mises en œuvre, on dispose désormais de bibliothèques pour de nombreux langages de programmation (je ne les ai pas testées) :

— Pour Go, il y a ,
— Pour PHP, ,
— Pour Rust, regardez ,
— Pour Python, .

Une discussion est en cours <https://github.com/curl/curl/discussions/13376> pour ajouter ces signatures à curl. Si vous programmez, pour tester votre code, je recommande fortement le service en ligne . Notez qu'il ne vérifie pas le condensat du corps <https://github.com/bspk/httpsig-org/issues/20> du message. Pour fabriquer les clés pour ce service, si vous voulez faire de l'ECDSA, vous pouvez utiliser les commandes OpenSSL suivantes (oui, il doit y avoir une version plus simple) :

```
openssl ecparam -out server.pem -name prime256v1 -genkey
openssl req -new -key server.pem  -nodes -days 1000 -out server.csr
openssl x509 -in server.csr -out server.crt -req -signkey server.pem -days 2001
```

La clé privée sera dans `server.pem` et la publique dans `server.crt`.

Concernant l'ancienne version des signatures HTTP, vous pouvez consulter :

— Eugen Rochko a fait deux bons articles d'introduction : et .
— Dans Mobilizon, le code de signature est `Mobilizon.Federation.HTTPSignatures.Signature` et il vient à l'origine de Pleroma.

— Dans PeerTube (le code vient, je crois, de Misskey, à moins que ce ne soit le contraire) : c'est ici <https://www.npmjs.com/package/@peertube/http-signature> (et pour Misskey, ici <https://github.com/misskey-dev/node-http-message-signatures>).
— J'ai écrit (avec Chloé Baut) une mise en œuvre des protocoles du fédivers <https://framagit.org/bortzmeyer/tonola>, en Python, purement expérimentale mais qui peut être intéressante à lire pour comprendre comment tout cela marche.

Cette technique des signatures HTTP a eu une histoire longue et compliquée. Née chez Amazon, elle avait d'abord été décrite dans un "*Internet draft*", draft-cavage-http-signatures en 2013, et déployée, notamment dans le secteur financier. En 2017, Mastodon avait utilisé la technique telle que décrite dans ce document (forçant le reste du fédivers à s'aligner sur « ce que fait Mastodon »). C'est après l'adoption par le groupe de travail httpbis <https://datatracker.ietf.org/wg/httpbis/> que le projet avait pris sa forme finale. (Regardez les supports de présentation à l'IETF en 2019 <https://datatracker.ietf.org/meeting/106/materials/slides-106-secdispatch-http-signing> et la vidéo de la réunion <https://youtu.be/CYBhLQ0-fwE?t=3006>.) L'auteur du projet initial avait écrit un bon résumé <https://lists.w3.org/Archives/Public/ietf-http-wg/2020JanMar/0029.html> en 2020, décrivant de l'intérieur comment se passe la normalisation dans l'Internet. Les différences principales avec ce que fait le fédivers :

— Deux champs au lieu d'un, Signature-Input: et Signature:,
— Les champs utilisent la syntaxe des champs structurés du RFC 9651,
— Les métadonnées sont également signées.

Et pour terminer, voilà comment ça se fait dans le fédivers (qui, je le rappelle, n'utilise **pas** la syntaxe de ce RFC mais des concepts proches) :

— le client annonce son identité (pour moi, bortzmeyer@mastodon.gougere.fr),
— le serveur fait alors une requête WebFinger vers l'instance indiquée (comme un wget https://mastodon.gougere
— puis il va suivre suivre l'URL indiquée dans le membre self (comme un curl -o bortzmeyer.json -H 'Accept: application/activity+json' "https://mastodon.gougere.fr/users/bortzmeyer" et la clé publique est dans le membre publicKey,
— et le serveur peut alors vérifier les signatures et voir que c'était bien bortzmeyer@mastodon.gougere.fr qui écrivait (en toute rigueur, on vérifie juste que c'est bien mastodon.gougere.fr qui écrit et on lui fait confiance pour avoir authentifié ses propres utilisateurs).