RFC 5280 : Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 15 septembre 2013

Date de publication du RFC : Mai 2008

https://www.bortzmeyer.org/5280.html

L'une des rares normes de l'UIT encore utilisées sur l'Internet est X.509, une norme décrivant un format de certificats électroniques pour prouver son identité. X.509, comme toutes les normes UIT, est très complexe et probablement impossible à mettre en œuvre correctement et complètement. C'est pourquoi les applications sur l'Internet utilisent en général un **sous-ensemble** de X.509, décrit par ce RFC.

Ce sous-ensemble de X.509 (ou, plus exactement, de la version 3 de X.509) est souvent nommé **PKIX**, pour « "*Public-Key Infrastructure using X.509*" ». C'est lui qui est derrière toutes les utilisations des certificats X.509 qu'on voit sur l'Internet, par exemple le fameux petit cadenas de HTTPS. La section 2 décrit les motivations pour avoir créé ce sous-ensemble. Il s'agit notamment de faciliter l'usage des certificats en mettant l'accent sur l'interopérabilité des programmes (n'importe quel certificat PKIX doit être compris par n'importe quel programme PKIX, ce qui n'est pas du tout garanti avec le X.509 original, où deux mises en œuvre de X.509 peuvent être parfaitement conformes au standard... et néanmoins incapables d'interopérer) et le déterminisme (savoir comment le certificat va être compris, ce qui est difficile avec le X.509 original, souvent trop vague). Tous les utilisateurs de X.509 (pas seulement l'IETF avec son PKIX) doivent donc définir un sous-ensemble de X.509, le "*profile*" (dans la langue de Whitfield Diffie). L'importance de cette notion de sous-ensemble est bien résumée par le RFC : « "*unbounded choices greatly complicate the software*" ». Au contraire, l'UIT travaille autour du principe « satisfaire tout le monde » et toute demande est donc prise en compte dans la norme, la rendant ainsi trop riche pour être programmée correctement.

Le format de certificat X.509 en est à sa troisième version, la première ayant été publiée par l'UIT et l'ISO en 1988. Plusieurs protocoles Internet se servent de ce format, le premier ayant été le projet PEM du RFC 1422 [1] en 1993. C'est entre autres l'expérience de ce projet qui avait mené aux versions 2, puis 3 de X.509.

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc1422.txt

Ce RFC est uniquement une norme technique : il ne donne aucune règle sur les questions juridiques ou politiques, il ne dit pas, par exemple, comment une AC doit vérifier une demande de certificat, ni ce que doit faire un navigateur Web lorsqu'un certificat est invalide pour une raison ou pour une autre.

La section 3 pose l'architecture et les rôles de la PKIX, notamment les notions d'utilisateurs ("*end entity*"), qui demandent des certificats, et d'Autorités de Certification, qui les signent.

En 3.1, elle explique ce qu'est un certificat : lorsqu'on utilise la cryptographie asymétrique, on utilise la clé publique de son correspondant pour chiffrer les messages qu'on lui envoie. Mais comment savoir que c'est bien sa clé publique à lui, et pas celle d'un méchant qui écoute les communications, et pourra les déchiffrer si on utilise la clé qu'il nous a fournie ? Un certificat est l'affirmation, signée par une Autorité de Certification, qu'une clé publique est bien celle d'un utilisateur donné ("*subject*", dans le jargon X.509). On nomme cette affirmation le lien ("*binding*") entre la clé et le sujet.

Le certificat contient aussi diverses informations comme sa date d'expiration. Un certificat est donc juste une structure de données dont les champs sont la clé publique, la signature de l'AC, la date d'expiration, etc. Rendons cela plus concret avec un exemple sur un vrai certificat X.509, celui du site <https://app.capitainetrain.com/>. Pour simplifier les operations ultérieures, commençons par copier le certificat en local, avec openssl :

```
% openssl s_client -connect app.capitainetrain.com:443 > capitainetrain.pem < /dev/null
depth=1 C = US, O = "GeoTrust, Inc.", CN = RapidSSL CA
verify error:num=20:unable to get local issuer certificate
verify return:0
DONE
```

On aurait aussi pu utiliser GnuTLS pour récupérer le certificat :

```
% gnutls-cli --print-cert app.capitainetrain.com < /dev/null > capitainetrain.pem
```

On peut maintenant regarder le certificat et y découvrir les informations mentionnées plus haut :

```
% openssl x509 -text -in capitainetrain.pem
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 555296 (0x87920)
    Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=US, O=GeoTrust, Inc., CN=RapidSSL CA
        Validity
            Not Before: Sep 30 02:22:13 2012 GMT
            Not After : Oct  2 09:04:29 2015 GMT
        Subject: serialNumber=HjJWka0qtZ2MV9EYsbjRf51Mbcd/LYOQ, OU=GT04045327, OU=See www.rapidssl.com/resou
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:cd:af:c5:f8:ac:db:3b:10:fc:af:05:2f:ec:e5:
                    ...
           X509v3 CRL Distribution Points:

                Full Name:
                  URI:http://rapidssl-crl.geotrust.com/crls/rapidssl.crl
           ...
   Signature Algorithm: sha1WithRSAEncryption
         95:d4:fe:88:bc:9c:f8:b7:d5:72:52:c1:c8:98:04:ee:82:f7:
    ...
```

On y trouve le numéro de série du certificat, 555296, le nom de l'AC émettrice (`C=US, O=GeoTrust, Inc., CN=RapidSSL CA`, à savoir RapidSSL, une AC "*low-cost*"), la période de validité (jusqu'au 2 octobre 2015), le titulaire (`app.capitainetrain.com`), la clé publique, l'endroit où récupérer les CRL, et le tout avec la signature de l'AC.

Il y a aussi d'autres informations qui ne sont pas affichées par défaut par openssl. Par exemple, les usages permis (merci à Laurent Frigault pour son aide à ce sujet). Il faut utiliser l'option `-purpose` :

```
% openssl x509 -text -purpose -in capitainetrain.pem
...
Certificate purposes:
SSL client : Yes
SSL client CA : No
SSL server : Yes
SSL server CA : No
...
```

Ce certificat peut être utilisé pour un serveur TLS (anciennement nommé SSL), l'usage le plus courant des certificats, sur l'Internet.

Dans les normes, cette structure de données est décrite en ASN.1 et encodée en DER, suivant la norme X.690.

La section 4 de notre RFC décrit rigoureusement le sous-ensemble de X.509 utilisé dans l'Internet. Voici une partie de cette description d'un certificat, montrant les champs les plus importants :

```
   Certificate  ::=  SEQUENCE  {
        tbsCertificate       TBSCertificate,
        signatureAlgorithm   AlgorithmIdentifier,
        signatureValue       BIT STRING  }

   TBSCertificate  ::=  SEQUENCE  {
        version         [0]  EXPLICIT Version DEFAULT v1,
        serialNumber         CertificateSerialNumber,
        issuer               Name,
        validity             Validity,
        subject              Name,
        subjectPublicKeyInfo SubjectPublicKeyInfo,
        ...
        }
```

Ici, `signatureValue` est la signature apposée par l'AC. Elle s'applique à un certificat, le `tbsCertificate`. Celui-ci a un numéro de série unique, un émetteur (l'AC, identifiée par un nom complet - "*Distinguished Name*" ou DN), une période de validité (entre telle date et telle date), un titulaire ("*subject*"), la clé publique dudit titulaire, etc. Vous pouvez voir tous ces champs dans l'exemple de `app.capitainetrain.com` montré plus haut.

Comme toujours en bonne cryptographie, plusieurs algorithmes de signature sont possibles, afin de faire face aux progrès de la cryptanalyse. Les RFC 3279, RFC 4055 et RFC 4491 donnent la liste des algorithmes standard dont bien sûr le fameux RSA. Évidemment, il faut utiliser des algorithmes de cryptographie forts, à la sécurité connue et éprouvée. Les AC ne devraient pas, par exemple, accepter de signer des certificats utilisant des algorithmes aux faiblesses identifiées (comme plusieurs l'avaient fait avec le

vieil MD5 <http://www.veracode.com/blog/2008/12/major-break-in-md5-signed-x509-certifica
>).

Comment identifier un titulaire (champ `subject`) ? C'est un des points délicats de X.509, qui permet une grande variété de noms. Dans l'Internet, le plus fréquent est d'utiliser un nom de domaine, soit comme spécifié par le RFC 4519 (section 2.4), soit en utilisant l'extension X.509 « noms alternatifs » (section 4.2.1.6 de notre RFC). Cette dernière extension permet aussi d'utiliser comme identificateur une adresse IP, un URI, etc. La grande variété des types de « noms » dans X.509, et la difficulté de la comparaison de ces noms créent d'ailleurs pas mal de risques de sécurité, et compliquent la vie du programmeur <https://www.bortzmeyer.org/openssl-hostname-check.html>.

L'extension « noms alternatifs » sert aussi si, derrière une même adresse IP, on a plusieurs sites <https://www.bortzmeyer.org/auth-x509-plusieurs-noms.html>. On voit ainsi quatre noms possibles pour `www.digicert.com` :

```
% openssl x509 -text -in digicert.pem
...
X509v3 Subject Alternative Name:
   DNS:www.digicert.com, DNS:digicert.com, DNS:content.digicert.com, DNS:www.origin.digicert.com
```

Ce champ `subject` est évidemment d'une importance vitale. Le but de la signature du certificat est précisément de lier une clé publique à un identificateur ("*subject*"). L'AC **doit** donc vérifier le contenu de ce champ. Par exemple, CAcert <https://www.bortzmeyer.org/cacert.html> n'inclut dans les certificats signés que les champs que cette AC a vérifié. Pour un nom de domaine, CAcert teste qu'il se termine par un des noms dont l'utilisateur a prouvé la possession (preuve typiquement apportée en répondant à un courrier envoyé au titulaire du domaine).

Les noms ("*subjects*") ne sont pas forcément limités à l'ASCII, ils peuvent inclure de l'Unicode et la section 7 de ce RFC décrit les règles spécifiques à suivre dans ce cas. Notamment, les noms doivent être canonicalisés en suivant le RFC 4518. Pour les noms de domaine, le type utilisé par X.509, `IA5String`, ne permet hélas que l'ASCII et les IDN devront donc être représentés sous forme d'un "*A-label*" (`www.xn--potamochr`
au lieu de `www.potamochère.fr`, par exemple). Même chose pour les IRI du RFC 3987 qui devront être convertis en URI. (Notez que les règles sur l'internationalisation ont été légèrement changées par la suite avec le RFC 8399.)

Une possibilité peu connue de X.509 est l'extension « "*Name constraints*" » (section 4.2.1.10) qui permet de signer des certificats d'AC avec une limite : l'AC qui a ainsi reçu une délégation ne peut signer que des certificats obéissant à cette contrainte (si le titulaire du certificat est un URI ou une adresse de courrier, la contrainte s'applique au nom de domaine compris dans l'URI ou l'adresse). On peut ainsi déléguer à une AC qui ne pourra signer que si le nom de domaine du titulaire se termine en `.fr`, par exemple. Cela résoud une faille de sécurité importante de X.509 (celle qui a motivé le RFC 6698) : le fait que n'importe quelle AC puisse signer un nom de n'importe quel domaine, que le titulaire de ce nom de domaine soit le client de l'AC ou pas. Mais, en pratique, cette extension semble peu utilisée.

Enfin, la section 6 du RFC décrit le mécanisme de validation d'un certificat, par exemple lorsqu'un navigateur Web se connecte en HTTPS à un serveur et reçoit un certificat prouvant que ce serveur est bien celui demandé.

La section 8 étudie en détails toutes les questions de sécurité liées à X.509. D'abord, comme tous les éléments décrits dans ce RFC (les certificats et les CRL) sont signés, leur distribution ne nécessite pas de

précautions particulières : à l'arrivée, et quel que soit le mécanisme de transport utilisé, le destinataire pourra vérifier l'intégrité de ces éléments.

Plus délicate est la question des procédures que suit l'AC avant de signer un certificat : c'est là que se situent l'essentiel des faiblesses de X.509. Par exemple, est-ce qu'une AC doit vérifier que le nom de domaine n'est pas proche d'un autre <http://news.netcraft.com/archives/2013/09/06/deceptive-domain-and-ssl-certificate-issued-by-network-solutions.html> ?

Le RFC oublie, semble t-il, de dire que prendre soin de choisir une « bonne » AC n'est pas très utile puisque n'importe quelle AC de confiance peut émettre un certificat pour n'importe quel nom. Si Google est client de GeoTrust, qui signe donc les certificats de `gmail.com`, rien n'empêche DigiNotar, avec qui Google n'a aucun lien, de signer aussi des certificats pour `gmail.com`. Le point critique n'est donc pas l'AC qu'on choisit pour signer ses certificats, mais les AC auxquelles nos clients choisissent de faire confiance. Qu'une seule de ces AC trahisse ou soit piratée et tout le système s'écroule. Or, justement, comment sont choisies ces AC de confiance ? Très peu d'utilisateurs examinent la liste (le « magasin ») d'AC de leur logiciel. Ils font une confiance aveugle à l'éditeur du logiciel, qui décide selon ses propres critères, d'inclure telle ou telle AC. Il y a donc des AC qui ne sont que dans certains navigateurs Web (rendant les certificats signés par ces AC difficiles à utiliser). Pour limiter ces difficultés, les éditeurs de navigateurs Web incluent **beaucoup** d'AC (des centaines, dans un navigateur typique), rendant difficile de valider leur sécurité. Petit exercice : cherchez dans votre navigateur la liste des AC acceptées (sur Firefox, "*Edit -¿ Preferences*", puis "*Advanced*" puis "*View certificates*") et lisez-la. Vous découvrirez plein d'organisations inconnues et lointaines.

Au cas où il faille ajouter ou retirer une AC, on ne peut pas demander à l'utilisateur de le faire. Il faut donc une mise à jour du logiciel, ou bien une distribution, par un autre canal, d'une liste d'AC (distribution sécurisée, par exemple, par une signature avec la clé privée de l'éditeur du logiciel).

L'AC doit évidemment gérer avec le plus grand soin la clé privée de ses propres certificats. Si un méchant copie cette clé, il pourra se faire passer pour l'AC et émettre des faux certificats. À une moins échelle, c'est aussi vrai pour l'utilisateur : si sa clé privée est compromise, il sera possible de se faire passer pour lui, et la signature de la clé publique par l'AC ne changera rien.

Notons que le RFC ne semble pas parler du cas où la clé privée de l'AC est bien gardée et inaccessible **mais** où le système d'information de l'AC a été piraté, permettant à l'attaquant de faire générer des « vrais/faux certificats ». C'est pourtant la menace principale, comme l'avaient illustrés les cas de Comodo et DigiNotar.

Cela nous amène à un autre point faible de X.509, la révocation. Des clés privées vont forcément tôt ou tard être copiées illégalement par un attaquant, qui va pouvoir alors se faire passer pour le titulaire du certificat. Si une clé privée est copiée à l'extérieur, ou qu'on soupçonne sérieusement qu'elle est copiée, comment empêcher l'attaquant de jouir des fruits de son forfait ? Ou, sans chercher un cas aussi dramatique, on peut aussi imaginer qu'un employé qui avait accès à la clé privée quitte l'organisation et qu'on souhaite, par précaution, qu'il ne puisse pas utiliser cette connaissance. Une solution est d'attendre l'expiration (rappelez-vous que les certificats X.509 ont une période de validité, entre deux dates, celle de début de validité et celle d'expiration). Mais cela peut prendre longtemps. Il faut donc émettre tout de suite une révocation, une annulation du certificat, signée par l'AC. X.509 permet aux AC d'émettre des CRL ("*Certificate Revocation List*"), qui sont des listes de certificats révoqués, signées par l'AC qui avait émis le certificat original. Étant signées, les CRL n'ont même pas besoin (comme les certificats, d'ailleurs) d'être distribuées par un canal sûr.

Le problème est de distribuer celles-ci efficacement. Comme l'ont montré plusieurs études <https://www.imperialviolet.org/2011/03/18/revocation.html>, la révocation marche mal car il

est difficile de s'assurer que tous les clients ont reçu l'information. Il y a plusieurs raisons à cela, le RFC n'en note qu'une : la révocation n'est pas temps réel. Entre le moment où la CRL est émise et celui où elle est traitée par tous les consommateurs, un certain temps peut s'écouler. (Une solution possible est le protocole OCSP, dans le RFC 6960.)

Voici une CRL de RapidSSL (celle annoncée dans le certificat de `app.capitainetrain.com` vu plus haut) :

```
% openssl crl -text -inform DER -in rapidssl.crl
Certificate Revocation List (CRL):
        Version 2 (0x1)
    Signature Algorithm: sha1WithRSAEncryption
        Issuer: /C=US/O=GeoTrust, Inc./CN=RapidSSL CA
        Last Update: Sep 13 12:03:00 2013 GMT
        Next Update: Sep 23 12:03:00 2013 GMT
...
   Serial Number: 070B6B
        Revocation Date: Aug 27 10:56:39 2013 GMT
...
```

On voit ainsi que le certificat de numéro de série 461675 (070B6B en hexadécimal) a été révoqué le 27 août 2013. Les raisons de la révocation sont également indiquées par certaines AC. Dans ce cas, on verrait :

```
    Serial Number: 18F...
        Revocation Date: Sep  3 07:58:20 2012 GMT
        CRL entry extensions:
            X509v3 CRL Reason Code:
                Key Compromise
    Serial Number: 5FD...
        Revocation Date: Aug 12 12:59:20 2012 GMT
        CRL entry extensions:
            X509v3 CRL Reason Code:
                Key Compromise
...
```

Enfin, trois annexes du RFC seront particulièrement utiles pour les programmeurs. Ceux-ci doivent d'abord noter que X.509 est très complexe et très difficile à implémenter. La première lecture recommandée, pour avoir une idée du travail, est le « "*X.509 Style Guide*" <http://www.cs.auckland.ac.nz/~pgut001/pubs/x509guide.txt> » de Peter Gutmann. L'annexe A liste les structures de données de X.509, en syntaxe ASN.1. On y trouve même des héritages d'un lointain passé comme :

```
poste-restante-address INTEGER ::= 19
```

L'annexe B donne des conseils aux programmeurs sur la mise en œuvre de X.509. L'annexe C donne des exemples de certificats et de CRL. Le programme dumpasn1 <http://www.cs.auckland.ac.nz/~pgut001/> est utilisé pour les afficher. Attention, cet outil ne lit que le DER, pas le PEM. Si on a un certificat au format PEM, il faut d'abord le convertir :

```
% openssl x509 -inform pem -outform der -in capitainetrain.pem -out capitainetrain.der
```

Et on peut alors l'analyser :

```
% dumpasn1  capitainetrain.der|more

   0 1330: SEQUENCE {
   4 1050:   SEQUENCE {
   8    3:     [0] {
  10    1:       INTEGER 2
        :       }
  13    3:     INTEGER 555296
  18   13:     SEQUENCE {
  20    9:       OBJECT IDENTIFIER sha1WithRSAEncryption (1 2 840 113549 1 1 5)
...
  95   30:     SEQUENCE {
  97   13:       UTCTime 30/09/2012 02:22:13 GMT
 112   13:       UTCTime 02/10/2015 09:04:29 GMT
...
 294   31:       SET {
 296   29:         SEQUENCE {
 298    3:           OBJECT IDENTIFIER commonName (2 5 4 3)
 303   22:           PrintableString 'app.capitainetrain.com'
        :           }
...
```

Si vous préférez un truc moderne et qui brille, l'outil interactif en ligne ASN.1 JavaScript decoder <http://lapo.it/asn1js/> est remarquable (et, lui, il accepte le PEM).

Ce sous-ensemble (« *"profile"* ») de X.509 avait à l'origine été normalisé dans le RFC 3280, que ce RFC met à jour. La liste des changements est décrite en section 1. Rien de crucial, mais on notera que la nouvelle norme fait une part plus grande à l'internationalisation.

Merci à Manuel Pégourié-Gonnard pour la correction d'erreurs.