Vérifier le nom dans un certificat : pas trivial

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 15 décembre 2019

https://www.bortzmeyer.org/openssl-hostname-check.html

J'ai récemment eu à écrire un programme qui se connecte en TLS à un serveur et devait donc vérifier le certificat. La bibliothèque TLS utilisée ne vérifie pas que le nom dans le certificat correspond au nom demandé, c'est au programmeur de le faire, et ce n'est pas trivial, avec de nombreux pièges.

Un peu de contexte pour comprendre : le programme était un client DoT (DNS sur TLS, normalisé dans le RFC 7858 [1]). Il ne s'agit pas d'un client HTTP (qui ont tous des mécanismes de vérification du certificat). Au début, je me suis dit « pas de problème, je ne vais pas programmer TLS moi-même, de toute façon, cela serait très imprudent, vu mes compétences, je vais utiliser une bibliothèque toute faite, et, avantage en prime, j'aurais moins de travail ». Le client étant écrit en Python, j'utilise la bibliothèque pyOpenSSL <https://www.pyopenssl.org/>, qui repose sur la bien connue OpenSSL.

Je prends bien soin d'activer la vérification du certificat et, en effet, les certificats signés par une AC inconnue, ou bien les certificats expirés, sont rejetés. Mais, surprise (pour moi), si le nom dans le certificat ne correspond pas au nom demandé, le certificat est quand même accepté. Voici un exemple avec le client OpenSSL en ligne de commande, où j'ai mis `badname.example` dans mon `/etc/hosts` pour tester (une autre solution aurait été d'utiliser l'adresse IP et pas le nom, pour se connecter, ou bien un alias du nom) :

```
% openssl s_client -connect badname.example:853  -x509_strict
...
Certificate chain
 0 s:CN = dot.bortzmeyer.fr
   i:C = US, O = Let's Encrypt, CN = Let's Encrypt Authority X3
...
```

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc7858.txt

Et aucun message d'erreur, tout se passe bien, alors que le certificat ne contenait pas du tout `badname.example`.

Notez que c'est précisé. La documentation d'OpenSSL nous dit <https://wiki.openssl.org/index.php/SSL/TLS_Client#Verification> « "*You must confirm a match between the hostname you contacted and the hostnames listed in the certificate. OpenSSL prior to 1.1.0 does not perform hostname verification, so you will have to perform the checking yourself.*" ». Pourquoi cette absence de vérification ? Et comment faire la vérification ?

En fait, le problème n'est pas spécifique à OpenSSL. Le système de sécurité autour des certificats est un empilement complexe de normes. À la base, se trouve une norme UIT nommée X.509. Elle était disponible en ligne <https://www.itu.int/ITU-T/recommendations/rec.aspx?rec=14033> (depuis, l'UIT l'a fait disparaitre, où irions-nous si tout le monde pouvait lire les normes) et fait, aujourd'hui, 236 pages. Elle décrit le format des certificats, le rôle des AC, et plein d'autres choses, mais ne parle pas de la vérification des noms (on dit « sujets » et pas « noms », dans la terminologie X.509), et pour cause : X.509 permet de très nombreuses façons d'indiquer le sujet, ce n'est pas forcément un nom de domaine. Comme le dit sa section 9.3.1, « "*public-key certificates need to be usable by applications that employ a variety of name forms*" ». Une convention aussi banale que d'utiliser une astérisque comme joker n'est même pas mentionnée. Bref, X.509 ne va pas répondre à nos questions. Mais l'Internet n'utilise pas X.509. Il utilise la plupart du temps le protocole TLS, normalisé dans le RFC 8446. TLS utilise comme méthode d'authentification principale un certificat, décrit par un **profil**, une variation de X.509, ne gardant pas toutes les posssibilités de X.509, et en ajoutant d'autres. Ce profil est souvent nommé PKIX, pour "*Public-Key Infrastructure X.509*" et est normalisé dans le RFC 5280. PKIX est plus précis que X.509 mais reste encore loin de tout spécifier, renvoyant souvent aux applications telle ou telle question. Par exemple, les jokers, déjà mentionnés, sont laissés à l'appréciation des applications. On comprend donc que les développeurs de OpenSSL n'aient pas inclus une vérification par défaut.

Pour compléter cette revue des normes, il faut citer surtout le RFC 6125, qui, lui, décrit précisément la vérification des sujets, quand ce sont des noms de domaine. C'est la principale source d'information quand on veut programmer une vérification du nom.

Ah, et, quand on teste, bien penser à séparer bibliothèque et programme utilisant cette bibliothèque. Ce n'est pas parce que la commande `openssl` peut tester le nom que la bibliothèque le fait par défaut. L'option pertinente se nomme `-verify_hostname` :

```
% openssl s_client -connect badname.example:853  -x509_strict -verify_hostname badname.example
...
verify error:num=62:Hostname mismatch
...
```

Si on avait utilisé le bon nom (celui présent dans le certificat,ici `dot.bortzmeyer.fr`), on aurait eu :

```
Verification: OK
Verified peername: dot.bortzmeyer.fr
```

Notez bien que c'est une option. Si elle est mise sur la ligne de commande, `openssl` demandera à la bibliothèque OpenSSL de faire la vérification, mais cela ne sera pas fait autrement. OpenSSL dispose d'un sous-programme `X509_check_host` qui fait la vérification, mais il n'est pas appelé par défaut, et il n'est pas présent dans pyOpenSSL <https://github.com/pyca/pyopenssl/issues/795>, la bibliothèque Python.

Par contre, avec la bibliothèque GnuTLS, le programme en ligne de commande fait cette vérification, par défaut :

```
% gnutls-cli --port 853 badname.example
...
Connecting to '2001:41d0:302:2200::180:853'...
...
- Certificate[0] info:
 - subject 'CN=dot.bortzmeyer.fr', issuer 'CN=Let's Encrypt Authority X3,O=Let's Encrypt,C=US', serial 0x042ab81
...
- Status: The certificate is NOT trusted. The name in the certificate does not match the expected.
*** PKI verification of server certificate failed...
*** Fatal error: Error in the certificate.
```

Le sous-programme dans GnuTLS qui fait la vérification se nomme `gnutls_x509_crt_check_hostname`.

Et si vous voulez faire cette vérification vous-même (ce qui n'est pas prudent <https://framagit.org/bortzmeyer/homer/issues/11> mais on va dire qu'on veut vivre dangereusement) ? Il faut penser à plein de choses. Prenons l'exemple d'un programme en Python. Le test de base, où on compare le nom de serveur donné au « "*common name*" » (CN) dans le certificat :

```
if hostname == cert.get_subject().commonName:
```

est non seulement insuffisant (il ne tient pas compte des jokers, ou des « "*subject alternative name*" », ou SAN) mais il est en outre incorrect, le RFC 6125 disant bien, dans sa section 6.4.4, qu'il ne faut tester le « "*common name*" » que si tous les autres tests ont échoué. Il faut en fait tenir compte des SAN ("*Subject Alternative Names*") contenus dans le certificat. Il faut examiner tous ces SAN :

```
for alt_name in get_certificate_san(cert).split(", "):
    if alt_name.startswith("DNS:"):
        (start, base) = alt_name.split("DNS:")
        if hostname == base:
   ...
```

Rappelez-vous qu'un sujet, en X.509, n'est pas forcément un nom de domaine. Cela peut être une adresse IP, un URL, ou même du texte libre. Traitons le cas des adresses IP. Le certificat de Quad9 <https://www.bortzmeyer.org/quad9.html> utilise de telles adresses :

```
% openssl s_client -connect 9.9.9.9:853 -showcerts | openssl x509 -text
...
    X509v3 Subject Alternative Name:
       DNS:*.quad9.net, DNS:quad9.net, IP Address:9.9.9.9, IP Address:9.9.9.10, IP Address:9.9.9.11, IP Address:
```

Voici une première tentative pour les tester :

```
    elif alt_name.startswith("IP Address:"):
          (start, base) = alt_name.split("IP Address:")
          if hostname == base:
     ...
```

Mais ce code n'est pas correct, car il compare les adresses IP comme si c'était du texte. Or, les adresses peuvent apparaitre sous différentes formes. Par exemple, pour IPv6, le RFC 5952 spécifie un format canonique mais les certificats dans le monde réel ne le suivent pas forcément. Le certificat de Quad9 ci-dessus utilise par exemple `2620:FE:0:0:0:0:FE:10` alors que cela devrait être `2620:fe::fe:10` pour satisfaire aux exigences du RFC 5952. Il faut donc convertir les adresses IP en binaire et comparer ces binaires, ce que je fais ici en Python avec la bibliothèque netaddr <https://github.com/drkjam/netaddr/> :

```
    elif alt_name.startswith("IP Address:"):
         (start, base) = alt_name.split("IP Address:")
         host_i = netaddr.IPAddress(hostname)
         base_i = netaddr.IPAddress(base)
         if host_i == base_i:
    ...
```

Ce problème du format des adresses IP illustre un piège général lorsqu'on fait des comparaisons avec les certificats, celui de la canonicalisation. Les noms (les sujets) peuvent être écrits de plusieurs façons différentes, et doivent donc être canonicalisés avant comparaison. Pour les noms de domaine, cela implique par exemple de les convertir dans une casse uniforme. Plus subtil, il faut également tenir compte des IDN (RFC 5891). Le RFC 6125, section 6.4.2, dit qu'il faut comparer les « *"A-labels"* » (la forme en Punycode), on passe donc tout en Punycode (RFC 3492) :

```
def canonicalize(hostname):
    result = hostname.lower()
    result = result.encode('idna').decode()
    return result
```

Vous pouvez tester avec `www.potamochère.fr`, qui a un certificat pour le nom de domaine en Unicode.

Ensuite, il faut tenir compte du fait que le certificat peut contenir des jokers, indiqués par l'astérisque. Ainsi, `rdap.nic.bzh` présente un certificat avec un joker :

```
% gnutls-cli  rdap.nic.bzh
...
- Certificate[0] info:
 - subject `CN=*.nic.bzh', issuer `CN=RapidSSL TLS RSA CA G1,OU=www.digicert.com,O=DigiCert Inc,C=US', seria
...
```

Il faut donc traiter ces jokers. Attention, comme le précise le RFC 6125 (rappelez-vous que la norme X.509 ne parle pas des jokers), il ne faut accepter les jokers que sur le premier composant du nom de domaine (`*.nic.bzh` marche mais `rdap.*.bzh` non), et le joker ne peut correspondre qu'à un seul composant (`*.nic.bzh` accepte `rdap.nic.bzh` mais pas `foo.bar.kenavo.nic.bzh`.) Cela se traduit par :

```
    if possibleMatch.startswith("*."): # Wildcard
        base = possibleMatch[1:] # Skip the star
        (first, rest) = hostname.split(".", maxsplit=1)
        if rest == base[1:]:
            return True
        if hostname == base[1:]:
            return True
        return False
    else:
        return hostname == possibleMatch
```

Un point amusant : le RFC 6125 accepte explicitement des astérisques au **milieu** d'un composant, par exemple un certificat pour `r*p.nic.bzh` accepterait `rdap.nic.bzh`. Mon code ne gère pas ce cas vraiment tordu, mais les développeurs d'OpenSSL l'ont prévu (si l'option `X509_CHECK_FLAG_NO_PARTIAL_WILDCARDS` n'est pas activée) :

```
/* Only full-label '*.example.com' wildcards? */
     if ((flags & X509_CHECK_FLAG_NO_PARTIAL_WILDCARDS)
                && (!atstart || !atend))
                return NULL;
           /* No 'foo*bar' wildcards */
```

Autre cas amusant, le code de curl, quand il vérifie que le nom du serveur correspond au contenu du certificat, refuse les jokers si le nom dans le certificat n'a que deux composants. Le but est sans doute d'éviter qu'un certificat pour `*.com` ne permette de se faire passer pour n'importe quel nom en `.com` mais cette règle, qui ne semble pas être dans le RFC 6125, n'est sans doute pas adaptée aux récents TLD `.BRAND`, limités à une entreprise.

Au passage, j'avais dit au début que mon but initial était de tester un serveur DoT ("*DNS-over-TLS*", RFC 7858). Le RFC original sur DoT ne proposait pas de tester le nom, estimant qu'un résolveur DoT serait en général configuré via son adresse IP, pour éviter un problème d'œuf et de poule (on a besoin du résolveur pour trouver l'adresse IP du résolveur…) La solution était d'authentifier via la clé publique du résolveur (l'idée a été développée dans le RFC 8310.)

Voilà, vous avez un exemple de programme Python qui met en œuvre les techniques données ici (il est plus complexe, car il y a des pièges que je n'ai pas mentionnés), en (en ligne sur `https://www.bortzmeyer.org/files/tls-check-host.py`). Une autre solution, n'utilisant que la bibliothèque standard de Python, est (en ligne sur `https://www.bortzmeyer.org/files/tls-check-host-std-lib.py`), mais, attention, ladite bibliothèque standard ne gère pas les IDN, donc ça ne marchera pas avec des noms non-ASCII. (Mais son utilisation évite de recoder la vérification.)

En conclusion, j'invite les programmeurs qui utilisent TLS à bien s'assurer que la bibliothèque TLS qu'ils ou elles utilisent vérifie bien que le nom de serveur donné corresponde au contenu du certificat. Et, si ce n'est pas le cas et que vous le faites vous-même, attention : beaucoup d'exemples de code source de validation d'un nom qu'on trouve au hasard de recherches sur le Web sont faux, par exemple parce qu'ils oublient les jokers, ou bien les traitent mal `<https://framagit.org/bortzmeyer/homer/issues/11>`. Lisez bien le RFC 6125, notamment sa section 6, ainsi que cette page OpenSSL `<https://wiki.openssl.org/index.php/Hostname_validation>`.

Merci à Patrick Mevzek et KMK pour une intéressante discussion sur cette question. Merci à Pierre Beyssac pour le rappel de ce que peut faire la bibliothèque standard de Python.