TP7 - TP SSL/TLS

1 Gestion de certificats x509

Le système de clé publique/privée, c'est pratique dans une certaine mesure, mais ça ne passe pas à l'échelle si l'on veut qu'un grand nombre de clients ou serveurs vérifient l'identité d'un grand nombre d'autres clients ou serveurs. Typiquement, que le navigateur de n'importe qui puisse vérifier l'identité d'un serveur web lorsqu'il utilise le protocole `https`. On utilise pour cela une tierce personne, une Autorité de Certification, qui délivre des certificats, que tout un chacun peut alors vérifier à l'aide du certificat racine de cette autorité. Mettons cela en œuvre à l'aide de la commande *certtool* (GnuTLS), en jouant les trois rôles du mécanisme : autorité, serveur et client. Fabriquons d'abord les trois répertoires associés à ces rôles :

```
mkdir autorite
mkdir serveur
mkdir client
```

1.1 Autorité de certification

A priori il est rare d'avoir à jouer ce rôle de tierce personne, on préfère utiliser les services de Verisign, Let's Encrypt, CAcert, ... Il reste intéressant de comprendre ce qui se passe. Prenons donc d'abord le rôle de l'autorité.

```
cd autorite/
```

Commençons par créer la clé privée de notre autorité.

```
certtool --generate-privkey --outfile ca.key
```

Il s'agit maintenant de créer un certificat racine :

```
certtool --generate-self-signed --load-privkey ca.key --outfile ca.crt
```

Des questions sont posées, pour enregistrer dans le certificat des informations utiles pour se souvenir de qui gère cette autorité. Voici les informations importantes à saisir. Pour les autres questions, vous pouvez garder les réponses par défaut, en laissant la réponse vide et en tapant 'enter'.

— Common name : maCA
— The certificate will expire in (days) : 1000
— Does the certificate belong to an authority ? (y/N) : y
— Will the certificate be used to sign other certificates ? (y/N) : y
— Is the above information ok ? (y/N) : y

Note : le common name permet d'identifier l'autorité de certification, pour que le client sache plus tard à qui il a envie de faire confiance ou non.

On peut revoir le contenu du certificat à l'aide de :

```
certtool --certificate-info --infile ca.crt
```

ou bien juste son empreinte (fingerprint) :

```
certtool --infile ca.crt --fingerprint
```

Le fichier `ca.key` est la partie privée du certificat, à ne pas divulguer à qui que ce soit d'autre. C'est justement là que s'appuie l'autorité : seul celui qui connait le contenu de `ca.key` pourra créer des certificats. `ca.crt` est par contre le certificat racine, à diffuser aux clients, on en reparle plus loin.

1.2 Requête du serveur : demande d'un certificat

Jouons maintenant le rôle d'un administrateur de serveur qui veut s'authentifier auprès d'un client. Dans le cadre de ce TP, notre serveur s'exécutera sur la machine locale d'adresse IP `127.0.0.1`. Nous utiliserons donc cette adresse comme identifiant (ou CN = Common Name) pour le certificat. Mais, en pratique, c'est le nom DNS du serveur web qu'il faudrait indiquer comme CN, comme par exemple `www.google.com`.

```
cd ../serveur/
```

Commençons par créer la clé privée de notre serveur.

```
certtool --generate-privkey --outfile server.key
```

Pour obtenir un certificat en bonne et due forme, notre serveur a besoin d'effectuer une requête auprès d'une autorité de certification, afin que ce dernier lui délivre le précieux certificat. Effectuons une telle requête auprès de l'autorité CA.

```
certtool --generate-request --load-privkey server.key --outfile server.csr
```

Voici les informations importantes à saisir. Pour le reste, conserver les réponses par défaut.
— Common name : 127.0.0.1
— Will the certificate be used for signing (DHE ciphersuites) ? (Y/n) : y
— Will the certificate be used for encryption (RSA ciphersuites) ? (Y/n) : y

Le fichier `server.key` est la partie privé du certificat, à ne pas divulguer à qui que ce soit. C'est justement là que s'appuie la certification : seul celui qui connait le contenu de `server.key` pourra être certifié. Il faut par contre transmettre `server.csr` à l'autorité de certification.

```
cp server.csr ../autorite/
```

Changeons de casquette, nous sommes de nouveau l'autorité de certification.

```
cd ../autorite/
```

Nous regardons la demande de certificat :

```
certtool --crq-info --infile server.csr
```

Nous vérifions alors l'identité de l'administrateur du serveur par un moyen « classique » (registre d'industrie, carte d'identité, connaissance personnelle, ...), et que cela correspond bien à ce qui est écrit dans le certificat demandé, et nous vérifions le common name (CN) dont nous nous assurons par un moyen « classique » qu'il appartient bien à celui qui fait la demande de certificat.

Une fois les vérifications faites, nous pouvons valider alors la requête de la manière suivante :

```
certtool --generate-certificate --load-request server.csr --load-ca-privkey ca.key
         --load-ca-certificate ca.crt --outfile server.crt
```

On gardera la plupart des réponses par défaut, en faisant attention en particulier aux réponses suivantes :

— The certificate will expire in (days) : 255
— Will the certificate be used for signing (required for TLS) ? (Y/n) : y
— Will the certificate be used for encryption (not required for TLS) ? (Y/n) y
— Is the above information ok ? (y/N) : y

Vérifions maintenant le certificat du serveur, fraîchement créé, grâce au certificat de l'autorité :

```
certtool --verify --load-ca-certificate ca.crt --infile server.crt
```

Si tout est OK, nous pouvons transmettre ce certificat (`server.crt`) à l'administrateur du serveur :

```
cp server.crt ../serveur/
```

1.3 Mise en oeuvre du certificat

Reprenons donc la casquette d'administrateur du serveur web.

```
cd ../serveur/
```

Nous pourrions par exemple configurer un serveur web comme Apache pour utiliser ce certificat. Il lui faudrait à la fois la partie publique (`server.crt`) et la clé privée (`server.key`), nécessaire pour s'authentifier vraiment. Typiquement il faudrait ajouter ceci dans /etc/apache2/sites-available/monsite-ssl :

```
SSLCertificateFile server.crt
SSLCertificateKeyFile server.key
```

Mais Apache est long à installer, faisons plutôt le test de notre certificat avec l'outil GNU TLS qui peut jouer à la fois le rôle d'un serveur web et d'un client.

Lançons le serveur web :

```
gnutls-serv --http --x509keyfile=server.key --x509certfile=server.crt --port=1234
```

1.4 Client

Jouons maintenant le rôle du client. Ouvrez un nouveau terminal.

```
cd client/
cp ../autorite/ca.crt .
```

Connectons nous avec le client GNU TLS :

```
gnutls-cli --x509cafile ca.crt -p 1234 127.0.0.1
```

Si tout se passe bien :

1. Le client ouvre une connexion TCP/IP classique.
2. Le client entame la négociation SSL/TLS au cours de laquelle il récupère le certificat du serveur web : *Got a certificate...* avec comme info `CN=127.0.0.1` et `Issuer CN=CA`.
3. Puis le client vérifie que le certificat du serveur est conforme, c'est-à-dire qu'il est signée par une autorité de certification connue dans /etc/ssl/certs/. Ici, il s'agit du fichier `ca.crt` passé en argument.
4. Le client vérifie ensuite que le nom du serveur passé en ligne de commande `127.0.0.1` correspond bien au nom indiqué sur le CN du certificat. Alors il peut afficher la ligne : *The certificate is trusted.*
5. A partir de maintenant, le client a authentifié le serveur et la connection est sécurisée. Notons en revanche, que le client reste *anonyme* pour le serveur.

Si tout est OK, le client peut taper la requête HTTP à la main :

```
GET / HTTP/1.0
```

Tapez deux fois 'enter' pour valider la requête GET et l'envoyer au serveur (en cryptée). La réponse HTTP est décryptée par le client, qui l'affiche en clair dans votre terminal.

Ouvrez maintenant le navigateur web comme *Firefox* ou *Chrome* et consultez la page `https://127.0.0.1:1234/`. Pourquoi le navigateur web affiche-t-il un avertissement de sécurité ?

Si un navigateur web ne possède pas déjà le certificat racine de l'autorité de certification, il affiche un avertissement de sécurité et propose d'utiliser le certificat tout de même. Demandez à voir le certificat, constatez que c'est bien celui que vous avez émis en comparant l'empreinte. Quel risque peut-il y avoir à accepter le certificat sans vérifier l'empreinte ?

Ajoutez l'exception au navigateur en décochant la case "permanent", constatez que l'on obtient bien la page web du serveur. Fermez complètement le navigateur, relancez-le. Que constate-t-on ? On peut cocher la case "permanent" pour installer le certificat du serveur.

Il est également possible d'importer définitivement notre autorité de certification manuellement dans le navigateur, ainsi tous les serveurs ayant un certificat signé par notre autorité seront reconnus. Cherchez comment faire pour votre navigateur.

1.5 Bilan

Au final, il suffit que :

— Les clients récupèrent auprès de l'autorité de certification son certificat racine.

— Les administrateurs de serveurs obtiennent auprès de l'autorité des certificats.

Alors, les clients peuvent authentifier les serveurs web, sans avoir à interagir directement avec eux. Mais, où sont les maillons faibles ?

2 Progammation Socket SSL en Python

Prenons le code Python3 d'un client/serveur Echo (comme étudié au TP5) :

— `http://dept-info.labri.fr/~thibault/Reseau/sslsocket/server.py`

— `http://dept-info.labri.fr/~thibault/Reseau/sslsocket/client.py`

Lancez cet exemple : `./server.py` ; puis `./client.py`.

En vous aidant de la documentation `https://docs.python.org/3/library/ssl.html`, complétez le code du client et du serveur pour utiliser le protocole SSL/TLS avec les certificats générés dans l'exercice précédent.

On commencera par faire un `import ssl`. Puis, le serveur doit créer un contexte SSL et y charger son certificat et sa clé privée :

```
context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
context.load_cert_chain(...)
```

De même, le client va créer un contexte SSL et y charger le certificat de l'autorité :

```
context = ssl.create_default_context(ssl.Purpose.SERVER_AUTH)
context.load_verify_locations(...)
```

A partir d'une connection TCP/IP classique (socket *conn*) et d'un contexte SSL, il est possible d'obtenir une socket sécurisée *sslconn* de la manière suivante. Il faudra alors appeler cette méthode de manière appropriée à la fois du côté client et serveur. En particulier, il faut renseigner correctement côté client l'argument `server_hostname` avec le *Common Name* du certificat de notre serveur.

```
sslconn = context.wrap_socket(conn, server_side=???, ...)
```

On pourra alors utiliser `sslconn` à la place de `conn` pour tous les `send` et `recv`.