université de BORDEAUX

TP7 - Sécurité

1 Socket Sécurisé

Le but de cet exercice est de mettre en oeuvre les protocoles SSL/TLS à l'aide l'API Socket en Python3. Nous allons reprendre le code d'un client & serveur Echo en TCP afin de le sécuriser !

Vous pouvez effectuer cette partie sur votre propre Linux (ou WSL), il vous faut juste installer les packages : *openssl, gnutls-bin*.

1.1 Certificat X509

Afin de sécuriser un service sur Internet, il est tout d'abord nécessaire de disposer d'un certificat pour la machine qui va héberger ce service ! Dans la vraie vie, il serait nécessaire de le demander aux administrateurs de notre réseau, ou bien directement à une autorité de certification si on administre nous même le serveur.

Afin de réaliser cet exercice sur votre machine locale, nous vous fournissons un certificat *auto-signé* pour la machine *localhost* (ou 127.0.0.1). Bien évidemment, ce certificat n'a aucune légitimité sur Internet.

- le certificat de l'autorité de certification (ou CA) : ca.pem
- la clé privé de l'autorité de certification : ca.key
- le certificat du serveur : server.pem
- la clé privé du serveur : server.key

Précisez le rôle de chacun de ces fichiers ? Affichez les informations de ces deux certificats à l'aide de la commande `certtool` :

```
certtool --certificate-info --infile <file.pem>
```

En particulier, recherchez dans le certificat du serveur les informations suivantes, et précisez leur signification :

- la version du certificat,
- le numéro du série de certificat,
- le nom du propriétaire du certificat,
- le nom de l'autorité de certification,
- la date d'expiration du certificat,
- le *key id*,
- la signature,
- l'empreinte (ou *fingerprint*),
- la valeur de la clé publique.

Remarquez que le champs *Authority Key Identifier* du certificat serveur est bien le même que *Subject Key Identifier* de l'autorité... Ouf, tout va bien !

On peut aussi récupérer le *fingerprint* et la *clé publique* directement avec ces commandes :

```
$ certtool --fingerprint --infile <file.pem>
$ certtool --pubkey-info --infile <file.pem>
```

A l'aide de votre navigateur web préféré, affichez le certificat d'un serveur web de votre choix et analysez ces différents champs. Par exemple : `www.google.com`, `www.wikipedia.org`, `www.perdu.com`.

1.2 Mise en œuvre du certificat

Nous pourrions par exemple configurer un serveur web comme Apache pour utiliser ce certificat. 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.pem --port=1234
```

Jouons maintenant le rôle du client. Ouvrez un nouveau terminal et connectez-vous avec le client GNU TLS en utilisant le certificat de l'autorité pour vérifier celui du serveur (attention, il faut mettre exactement `127.0.0.1` et non pas `localhost` car c'est la valeur du CN) :

```
$ gnutls-cli --x509cafile ca.pem -p 1234 127.0.0.1
```

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 chiffré). La réponse HTTP est déchiffrée par le client, qui l'affiche en clair dans votre terminal.
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.pem` passé en argument.
4. Le client vérifie alors 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.

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 du serveur en comparant l'empreinte SHA1. Quel risque peut-il y avoir à accepter le certificat sans vérifier l'empreinte ?

Ajoutez l'exception au navigateur d'une façon ou d'une autre, et constatez que l'on obtient bien la page web du serveur.

1.3 Progammation Socket SSL en Python

Prenons le code Python3 d'un client/serveur `Echo`, comme celui déjà étudié :

— Serveur Echo TCP : server.py

— Client Echo TCP : client.py

Lisez le code, puis testez cet exemple : `./server.py` ; puis `./client.py`.

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

Commencez par faire un `import ssl` côté client & serveur. 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 doit 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 de la socket `s` et du contexte SSL `context` créé auparavant, il est possible d'obtenir une socket sécurisée `sslsock`. Il faut pour cela appeler la méthode `wrap_context` 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.

```
sslsock = context.wrap_socket(s, server_side=???, ...)
```

On peut alors utiliser `sslsock` à la place de `s` pour tous les `send` et `recv`.