université de BORDEAUX

TP6 - chiffrement, ssh/ssl

1 Introduction et premiers pas avec SSH

1.1 Rappels de quelques notions sur le chiffrement

Le principe d'une *paire de clés publique/privée* est que toute donnée chiffrée avec la clé publique ne peut être déchiffrée qu'avec la clé privée, et toute donnée chiffrée avec la clé privée peut être déchifrée avec la clé publique avec la garantie que c'est bien celui qui possède la clé privée qui a effectué le chiffrement.

Le principe d'une *clé symétrique* est que l'on partage en commun une clé qui permet à la fois de chiffrer et de déchiffrer. Les algorithmes utilisés sont en général bien plus légers que ceux utilisant une paire de clés publique/privée.

La difficulté de l'utilisation d'une clé symétrique est de s'échanger cette clé. En général on utilise à la fois des paires de clés publique/privée et des clés symétriques.

1.2 Le protocole SSH

Brièvement, le protocole SSH (Secure SHell) permet à des utilisateurs d'accéder à une machine distante à travers une communication chiffrée (appelée tunnel). Il utilise pour son fonctionnement à la fois une clé symétrique appelée *clé de session* et des paires de clés publique/privée.

Notamment, on suppose au départ que le serveur possède une clé privée appelée *host key* (qui ne change que lorsqu'on réinstalle entièrement la machine) et le client a une copie de la clé publique correspondante. L'établissement d'une liaison passe alors par plusieurs étapes.

— Le client se connecte en TCP au serveur.
— Le serveur envoie sa clé publique, le client vérifie que c'est bien la même que celle qu'il a déjà.
— Le client crée une clé symétrique de session.
— Le client utilise la clé publique du serveur pour chiffrer la clé de session. Il l'envoie au serveur, qui est donc le seul à pouvoir déchiffrer cette clé de session. Dès lors, à la fois le client et le serveur peuvent poursuivre tous leurs échange de manière chiffrée en utilisant la clé de session.
— Le client peut alors notamment envoyer login/mot de passe au serveur pour authentifier l'utilisateur de manière chiffrée.

Cette petite négociation est le point de départ de toute la sécurité de SSH.

— Faites un schéma pour expliquer l'établissement d'une connexion décrite ci-dessus.

— Expliquez brièvement les garanties de sécurité fournies par SSH : à la fois la confidentialité, l'authentification (dans les deux sens !) et l'intégrité des données échangées.
— Expliquez pourquoi lorsque vous vous connectez pour la première fois à un serveur donné votre client vous demande si la clé publique récupérée est bien celle du serveur. Pourquoi la clé privée du serveur doit-elle rester secrète ?
— ssh est basé sur une architecture client/serveur. Sur quel port écoute le serveur ?
— Essayez de vous connecter sur la machine de votre voisin à l'aide de la commande ssh.

2 Assez des mots de passe ?

Lors de la connexion vers une machine distante, ssh demande à l'utilisateur son mot de passe. Imaginez un administrateur système qui doit exécuter cette opération plusieurs fois par jour... Heureusement, de façon analogue à l'authentification du serveur par le client, un utilisateur peut utiliser un couple de clés privée/publique pour s'authentifier auprès du serveur.

2.1 On oublie les mots de passe

Voici la marche à suivre :

1. Créez un couple de clés : `ssh-keygen`, utilisez pour l'instant le choix par défaut pour les 3 questions posées, ne tapez donc rien et validez simplement le choix par défaut avec Entrée (et donc pas de passphrase pour le moment). La clé privée se trouve ainsi dans `~/.ssh/id_rsa` et votre clé publique est dans `~/.ssh/id_rsa.pub`.
2. Transférez votre clé publique sur le serveur :

   `ssh-copy-id monserveur`

   Ce que cette commande effectue pour vous, c'est en gros ceci :

   `ssh monserveur "cat >> ~/.ssh/authorized_keys" < ~/.ssh/id_rsa.pub`

   (`authorized_keys` s'appelle selon les versions `authorized_keys2`). Bien sûr, puisque les *homes* sont les mêmes sur les différentes machines du CREMI, il suffirait de copier/coller directement les fichiers, mais entre votre propre ordinateur et `jaguar`, par exemple, il faut réellement effectuer un transfert via ssh, clé USB, voire à la main !
3. Vous pouvez vous connecter sur la machine de votre voisin sans mot de passe !

Ajoutez sur votre dessin expliquant le protocole ce qui se passe ici : au lieu d'envoyer le mot de passe, le client répond à un *challenge* envoyé par le serveur, que seul le client (qui seul possède la partie secrète de sa clé) peut résoudre. Le serveur est ainsi sûr que c'est bien l'utilisateur qui essaie de se connecter.

Nous allons maintenant vérifier que ssh se préoccupe de la sécurité de vos clés :
— Donnez à `~` les droits 711. Cela ne pose pas de problème particulier.
— Donnez à `~/.ssh` les droits 711. Pas de problème non plus

— Modifiez les droits du fichier `~/.ssh/id_rsa` (qui contient votre clé privée) en 644. Connectez-vous sur le serveur distant qui contient votre clé publique. Que se passe-t-il ? Pourquoi ?
— Maintenant que votre clé privée est potentiellement divulguée, il faut la recréer... Supprimez le fichier `~/.ssh/id_rsa` , relancez le `ssh-keygen` , et re-transférez vers le serveur avec `ssh-copy-id`
— Maintenant, sur le serveur distant, modifiez les droits de `~/.ssh/authorized_keys` en 666. Déconnectez-vous du serveur puis reconnectez-vous au serveur. Expliquez (N'oubliez pas de rétablir les droits d'origine 644 ensuite, videz le fichier `authorized_keys` entièrement, et recréez de nouveaux vos clés de zéro). L'administrateur du serveur pourrait voir un warning s'afficher dans `/var/log/auth.log`.
— Ainsi donc, comment l'administrateur du serveur s'assure-t-il que le contenu de `authorized_keys` est bien de votre responsabilité ?

Je veux fournir à un collègue un accès `ssh` à un serveur dont je suis administrateur. Quelle est la manière la plus sûre de procéder ?

2.2 Mode paranoïaque

Pour protéger votre clé privée et s'assurer que c'est bien la bonne personne qui utilise la clé, il est possible lors de la création des clés de saisir une passphrase servant à chiffrer la clé privée, et qui sera donc demandée à chaque connexion pour pouvoir l'utiliser. Recommencez la procédure de création des clés en utilisant cette fois-ci une passphrase. N'oubliez pas de déposer la nouvelle clé publique dans votre `authorized_keys` à l'aide de `ssh-copy-id`

On avait évité d'avoir à taper un mot de passe à l'aide d'une paire de clés, mais maintenant on doit taper une passphrase ! Mais on a quand même amélioré la sécurité. Pourquoi ? (Quelles données passent par le réseau ? Pourquoi est-ce notamment intéressant pour des ordinateurs portables ?)

Pour éviter d'avoir à retaper la passphrase à chaque connexion, on peut utiliser un agent ssh qui va conserver dans un cache votre clé privée déchiffrée durant toute la durée de votre session sur la machine cliente.

Si votre environnement s'occupe déjà de lancer un agent ssh pour vous, une fenêtre s'ouvre pour vous demander votre passphrase, c'est l'agent qui vous demande la passphrase. Sinon, on peut le faire à la main comme ci-dessous. Note : si vous fermez par mégarde votre terminal, recommencer la procédure depuis le début pour être sûr d'avoir un agent fonctionnel.

1. Assurez-vous d'être bien sur votre propre machine
2. Tuez d'abord tout agent que gnome ou autre aurait lancé pour vous :
   `killall ssh-agent ; killall gnome-keyring-daemon`
3. Initialisez l'agent ssh : `eval $(ssh-agent)` Constatez avec `printenv | grep SSH` que cela a défini deux nouvelles variables d'environnement.
4. Ajoutez votre clé au cache de l'agent : `ssh-add`
5. Essayez de vous connecter sur une autre machine. Alors ?

6. Ouvrez une autre fenêtre de terminal. Essayez de vous connecter sur une autre machine. Expliquez pourquoi il n'arrive pas à profiter de l'agent. D'habitude, on démarre l'agent avec la session graphique, et tous les processus héritent des deux variables d'environnement.
7. Essayez de vous connecter en cascade (*i.e.* un `ssh` à l'intérieur d'un autre `ssh`) sur différentes machines. Est-ce que cela fonctionne toujours ?

   Remarquez l'option `-A` (il faut l'utiliser sur le *premier* ssh)

3 Transferts de fichiers

Les commandes `scp` permettent de transférer des fichiers par l'intermédiaire de `ssh`. Par exemple,

```
scp monfichier lamachinedistante:unrepertoire/
scp lamachinedistante:unrepertoire/unautrefichier .
```

Testez, par exemple avec le répertoire `/tmp`

À noter que le répertoire de base pour les chemins distants est `$HOME`

Essayez aussi de transférer un fichier entre deux ordinateurs distants tout en étant sur un troisième. Pour éviter d'avoir à taper son mot de passe, il faut par contre penser à forwarder l'agent avec `-o ForwardAgent=yes` (équivalent à l'option `-A` de ssh) pour que la source puisse se connecter à la destination.

En pratique, `scp` n'est pas très efficace pour transmettre de nombreux fichiers (option `-r`). Expliquez comment la commande suivante fonctionne et l'importance de l'antislash devant le `;` :

```
tar c myproject | ssh monserveur cd foobar \; tar x
```

4 Affichage X distant

Comme vu au TP2, le protocole `ssh` permet aussi d'exporter un affichage X11 vers votre machine cliente. Quelle option faut-il ajouter à la commande `ssh` ? Vous pouvez utiliser `xeyes` pour vérifier.

Essayez. Connectez-vous sur plusieurs machines en cascade. Est-ce que l'export de l'affichage suit ?

De façon plus générale, le comportement de `ssh` peut être paramétré en créant un fichier `config` dans le répertoire `.ssh` de votre home (`man 5 ssh_config`). Refaites les dernières manipulations en utilisant cette fois le fichier de configuration (cherchez l'option `X11Forwarding`).

5 Redirection de ports

Ssh peut aussi servir à transférer des données pour des services TCP/IP en utilisant une redirection de port. Cela est le plus souvent utile pour accéder à un service qui n'est disponible que depuis certaines machines *internes*.

Par exemple, un serveur tourne sur le port TCP 12345 de `mcgonagall`. Essayez de vous y connecter en lançant depuis votre machine

`telnet mcgonagall 12345`

Essayez directement depuis mcgonagall en utilisant

`telnet localhost 12345`

(si cela ne fonctionne toujours pas, appelez le chargé de TD pour qu'il relance le serveur)

Observez à l'aide de `netstat -tuan` ou `ss -tuan` que l'écoute sur le port `12345` n'est effectivement faite que pour l'adresse `localhost` . Au passage, remarquez qu'avec l'option `-n` on obtient les adresses IP sous forme numérique, et que `0.0.0.0` signifie `*` .

L'option `-L` de `ssh` permet de rediriger un port de votre machine vers le port local `12345` de `mcgonagall` (on appelle cela "pont SSH"). L'idée est qu'en ajoutant au lancement de ssh l'option `-L port:host:hostport`, toutes les connexions que vous faites vers le port `port` de votre propre machine sont redirigées par `ssh`, pour être effectuées depuis la machine à laquelle `ssh` est connecté, vers la machine `host` et le port `hostport`. Vous pouvez utiliser n'importe quelle valeur pour `port`, dès le moment qu'elle est plus grande que 1024, et inoccupée. On utilise souvent en plus les options `-N` et `-f`.

Créez donc un tel pont depuis votre machine :

`ssh mcgonagall -L 1111:localhost:12345`

Notez qu'ici `localhost` est interprété sur mcgonagall : c'est donc bien vers le port 12345 de mcgonagall que le pont est dirigé, mais en passant par son IP 127.0.0.1. Vous pouvez le voir avec netstat.

Essayez maintenant de vous connecter dans un autre terminal avec `telnet localhost 1111`

Refaites la même chose mais en mettant plusieurs machines entre `mcgonagall` et votre machine.

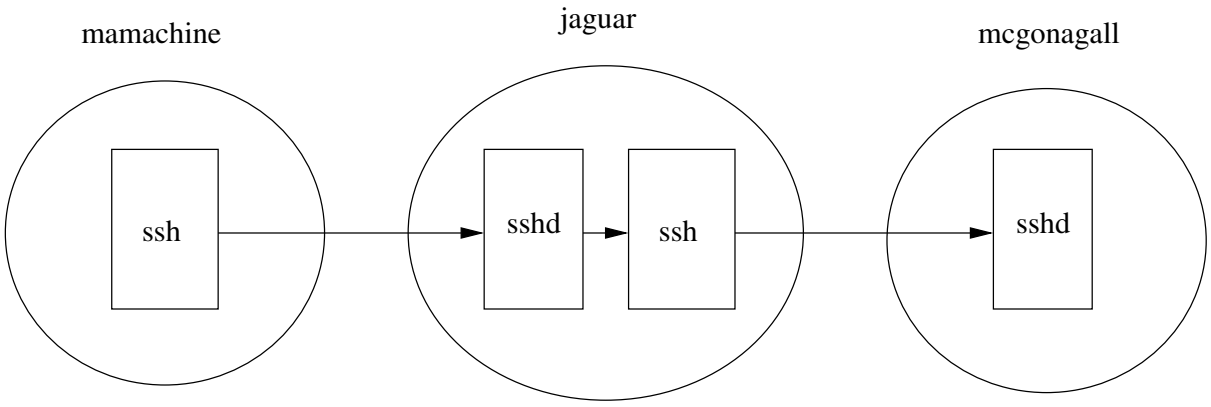
L'option `-R` est le *dual* de `-L`. Que fait-elle ? Faites un schéma.

Notez également que pendant une connexion ssh, on peut ajouter à la volée des redirections de ports : tapez `<Entrée>~C` pour récupérer l'invite `ssh>`, où vous pouvez taper `?` pour la syntaxe.

6 Comment cascader du `ssh`

Supposons que vous êtes à l'extérieur de l'université et vous souhaitez vous connecter à `mcgonagall`, mais comme seule `jaguar` est visible depuis l'extérieur, il est nécessaire de passer par elle. Il y a quatre solutions.

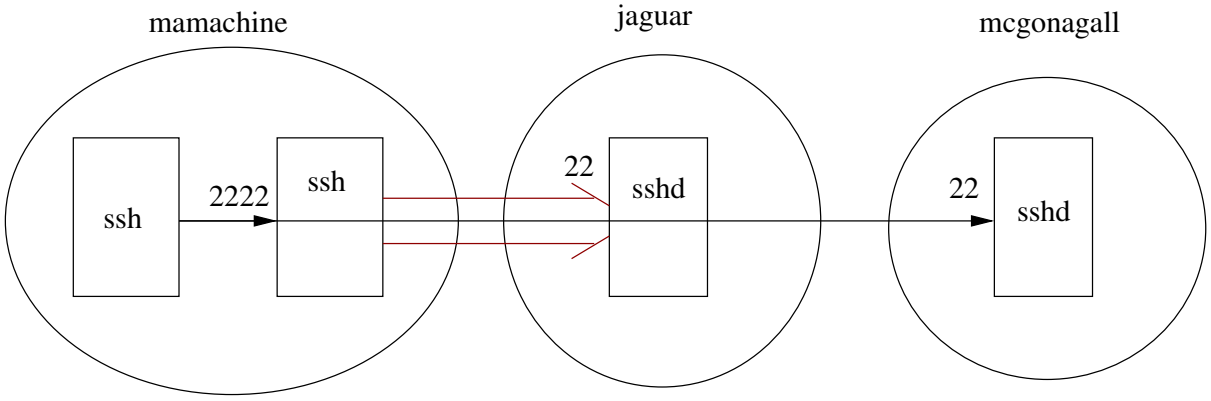
La plus simple est de lancer `ssh jaguar` et à l'intérieur, de lancer `ssh mcgonagall`. C'est un peu pénible, et pour les redirections de ports c'est encore plus pénible.



Une deuxième solution est d'effectuer une redirection de port, en lançant un premier ssh vers `jaguar` qui ouvre sur votre machine une redirection (disons sur le port 2222) vers le port ssh de `mcgonagall` :

```
ssh -L 2222:mcgonagall:22 jaguar
```

Laissez-le tourner. Un simple `ssh localhost -p 2222` depuis votre machine dans un autre terminal suffit alors à se connecter au `sshd` de `mcgonagall` via `jaguar`.



Une troisième solution est d'utiliser l'option `-J` :

```
ssh -J jaguar mcgonagall
```

qui effectue automatiquement la deuxième solution :)

On peut systématiser cette dernière solution en ajoutant à `~/.ssh/config` les lignes :

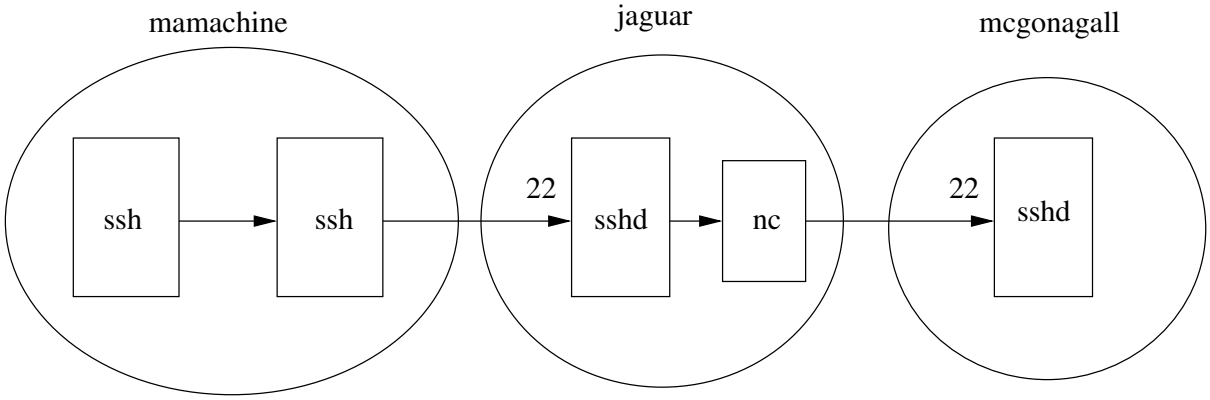
```
Host jaguar
  User monlogin
  Hostname jaguar.emi.u-bordeaux.fr
Host mcgonagall
  User monlogin
  Hostname mcgonagall.emi.u-bordeaux.fr
  ProxyJump jaguar
```

Une quatrième solution est d'utiliser `ProxyCommand` : dans votre `~/.ssh/config`, ajoutez les lignes suivantes :

```
Host monmcgonagall
   ProxyCommand ssh jaguar.emi.u-bordeaux.fr -W mcgonagall:22
```

Ces deux lignes indiquent que lorsque l'on lance `ssh monmcgonagall`, plutôt qu'ouvrir une connection TCP, `ssh` va lancer un autre `ssh` vers `jaguar`, le faire se connecter au port `ssh` de `mcgonagall`, et utiliser ce canal pour établir une session `ssh` avec `mcgonagall`

Mettez en oeuvre ces quatre solutions.



7 (Bonus) 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, on veut que le navigateur de n'importe qui puisse vérifier l'identité d'un serveur web lorsqu'il utilise le protocole `https`, sans qu'un message d'avertissement s'affiche la première fois qu'on s'y connecte (ce qu'on a vu avec ssh). 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 oeuvre 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
```

7.1 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 : ma.CA.a.moi.fr
— 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
```

et notamment ses empreintes (*Fingerprint*) SHA1 et SHA256 et son *Subject Key Identifier* SHA1

On peut aussi récupérer les fingerprints directement avec :

```
certtool --hash=SHA1 --fingerprint --infile ca.crt
certtool --hash=SHA256 --fingerprint --infile ca.crt
```

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 pour qu'ils puissent vérifier les certificats créés par l'autorité, on en reparle plus loin.

7.2 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`, de nom de domaine `localhost`. Nous utiliserons donc ce nom de domaine 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. Créons une telle requête.

```
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 : localhost
— 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 privée 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/
```

7.3 Certification par l'autorité

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

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

On peut aussi vérifier quelle autorité a certifié le certificat du serveur :

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

On y remarque *Authority Key Identifier*, c'est bien le même que le *Subject Key Identifier* de l'autorité que l'on avait vu précédemment, tout va bien.

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

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

7.4 Mise en œuvre 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
```

7.5 Client

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

```
cd client/
```

On commence par récupérer le certificat de l'autorité de certification. On peut regarder ce qui est indiqué dans le certificat. C'est l'étape cruciale : il faut vérifier que l'on a bien reçu le certificat en mains propres et que ce qui est indiqué correspond bien à l'identité de celui qui nous fournit ce certificat d'autorité.

```
cp ../autorite/ca.crt .
certtool --certificate-info --infile ca.crt
```

Dans le cas d'un navigateur web, les certificats des autorités de certification bien connues sont installés en même temps que le navigateur.

Connectons-nous avec le client GNU TLS en utilisant le certificat de l'autorité pour vérifier celui du serveur :

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

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=localhost` 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 `localhost` 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 chiffré). La réponse HTTP est déchiffrée 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://localhost: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 les empreintes sha-256 et sha-1. 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 dans les préférences de sécurité 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.

7.6 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 ?

8 (Bonus) Progammation Socket SSL en Python

Prenons le code Python3 d'un client/serveur Echo (comme étudié au TP3) :

— 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, 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`. 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 d'une connection TCP/IP classique (la socket déjà connectée `conn`) et d'un contexte SSL tel que créé ci-dessus, il est possible d'obtenir une socket sécurisée `sslconn` de la manière ci-dessous. Il faut 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 peut alors utiliser `sslconn` à la place de `conn` pour tous les `send` et `recv`.