

## TP7: chiffrement, ssh/ssl

Dans le tutoriel sur le télétravail au CREMI, nous avons vu l'utilisation de ssh, mais en passant assez allègrement sur les questions de sécurité pour permettre d'être opérationnels le plus vite possible.

Dans ce TP nous revenons sur les question de sécurité, l'utilisation de chiffrement, de clés, etc.

### 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 privé, et toute donnée chiffrée avec la clé privée peut être déchiffrée avec la clé publique avec la garantie que c'est bien celui qui possède la clé privé 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 léger 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 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?

## 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.

Dans le tutoriel pour le télétravail au CREMI on a vu la procédure : utiliser **ssh-keygen** pour créer une paire de clé, et **ssh-copy-id** pour envoyer la clé publique. Revoyons cela en vérifiant les détails.

### 2.1 On oublie les mots de passe

Voici la marche à suivre :

1. Connectez-vous d'abord à un des serveurs du CREMI (pas jaguar, puma ou leopard, elles sont trop limitées; disons jolicoeur).
2. Essayez de vous connecter à un autre serveur (par exemple boursouf). Si vous n'avez pas encore mis de clé ssh en place, on vous demande votre mot de passe.
3. Créez un couple de clés : **ssh-keygen -t rsa**, utilisez pour l'instant le choix par défaut pour les 3 questions posées, validez donc simplement avec Entrée. La clé privée se trouve ainsi dans `~/.ssh/id_rsa` et votre clé publique est dans `~/.ssh/id_rsa.pub`.
4. Transférez votre clé publique sur le serveur :

```
ssh-copy-id machine
```

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

```
ssh machine "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 !

5. Réessayez de vous connecter à un autre serveur, vous n'avez plus besoin de taper de 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 ?
- Rétablissez les droits de `~/.ssh/id_rsa`. Maintenant, sur le serveur distant, modifiez les droits de `~/.ssh/authorized_keys` en 666. Déconnectez-vous puis reconnectez-vous. Expliquez (N'oubliez pas de rétablir les droits d'origine ensuite).
- Ainsi donc, comment l'administrateur du serveur s'assure-t-il que la clé publique installée dans votre *home* est bien la vôtre ?

Je veux fournir à un collègue un accès `ssh` à ma machine. 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.

1. Tuez d'abord tout agent que gnome ou autre aurait lancé pour vous : `killall ssh-agent`
2. Initialisez l'agent ssh : `eval $(ssh-agent)` Constatez avec `printenv | grep SSH` que cela a défini deux nouvelles variables d'environnement.
3. Ajoutez votre clé au cache de l'agent : `ssh-add`
4. Essayez de vous connecter sur une autre machine. Alors ?
5. Ouvrez un autre terminal sur votre ordinateur et connectez-vous de nouveau à jolicoeur. 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.

6. 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`.

## 3 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.

Vous pouvez effectuer cette partie sur votre propre Linux ou WSL, il vous faut juste installer le paquet `gnutls-bin`.

Fabriquons d'abord les trois répertoires associés à ces rôles :

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

### 3.1 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 --infile ca.crt --certificate-info
```

et notamment son *Fingerprint* et son *Subject Key Identifier* SHA1.

On peut aussi récupérer le fingerprint directement avec :

```
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 connaît 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 parle plus loin.

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

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

### 3.3 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/
```

### 3.4 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
```

(Puisque vous êtes potentiellement plusieurs à lancer des serveurs web sur la même machine au CREMI pour pouvoir lancer ces commandes, mettez votre propre choix de port)

### 3.5 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 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 signé 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 chiffré). La réponse HTTP est déchiffrée par le client, qui l'affiche en clair dans votre terminal.

Si vous avez travaillé au CREMI, faites une redirection de ce port pour pouvoir vous y connecter depuis chez vous (cf le tutoriel de télétravail).

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.

## 3.6 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 ?



## 4 Programmation 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`.