# "ssh" : connexion à distance sécurisée

Il existe de nombreux moyens de se connecter à un ordinateur distant, qu'il fasse partie de votre réseau local ou à travers le réseau Internet. La principale préoccupation dans ce contexte est le niveau de sécurité fourni par ces connexions distantes, surtout si elles doivent transiter par le réseau Internet. Le programme incontournable dans ce domaine est sans aucun doute “**ssh**” et les utilitaires de la même famille comme “**sshfs**” (monter un système de fichiers distants), “**scp**” et “**sftp**” (copie/transfert de fichiers distants). Cette famille de programmes a la particularité d'offrir un haut niveau de sécurité, toutes les données échangées étant cryptées. De plus “ssh” peut fournir un service “d'encapsulation” des connexions pour d'autres programmes non-sécurisés, on parle alors de “tunnel ssh”.

Un signe $ précède les commandes qui ne nécessitent pas de droits administrateur ; un signe # précède celles qui nécessitent des droits administrateur (ces signes ne font **PAS** partie des commandes). Les lignes qui ne commencent pas par un signe $ ou # correspondent au résultat de la commande précédente.

## Installation

L'installation est très simple, le nom du paquet concerné peut varier d'une distribution à l'autre, sur Debian il s'agit de “openssh-server” et “openssh-client”, souvent réunis par un paquet virtuel “ssh”. Comme l'indiquent les noms des paquets, “ssh” peut être envisagé de deux façons :

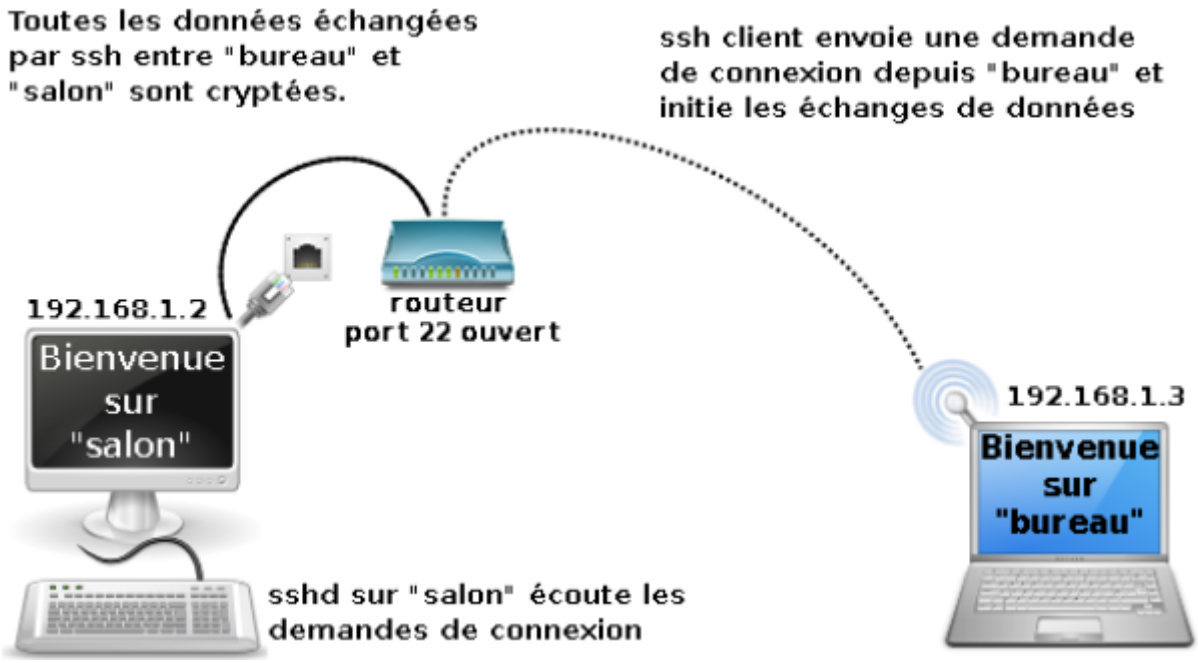
- Un “serveur ssh” fonctionne comme démon (daemon) et démarre en général en même temps que le système (script “init”). Il “écoute” les connexions sur un port particulier (le 22 par défaut), et répond à une demande en autorisant la connexion selon les règles fixées dans sa configuration. Tout ce qui se réfère à cet aspect “serveur” contient le nom “**sshd**”, le “d” final étant là pour “daemon”. La configuration est par exemple dans /etc/ssh/sshd_config (ou /etc/sshd_config). Le fonctionnement de ce démon n'est pas interactif, on le configure et on le laisse faire son travail discrètement.
- Un “client ssh” ne fait rien d'autre que d'essayer de se connecter à une machine qui possède un “serveur sshd”. La demande de connexion a lieu en général sur demande, et vers une adresse ip indiquée par l'utilisateur. Le “client ssh” est un programme interactif, il va interagir avec l'utilisateur (demande de mot de passe ...etc), et recevoir des instructions de celui-ci (adresses ip à interroger, commandes à exécuter).

Pour clarifier la chose, imaginons un réseau local composé de deux ordinateurs, appelons les “salon” et “bureau”. Ces ordinateurs ont les caractéristiques suivantes :

- “salon” : adresse ip local 192.168.1.2 , utilisateur “famille” ayant un compte sur l'ordinateur. Connecté par câble réseau Ethernet à un modem/router qui lui donne un accès à Internet et au réseau local, et joue le rôle de pare-feu.
- “bureau” : adresse ip local 192.168.1.3 , utilisateur “geek” ayant un compte sur l'ordinateur. Connecté par wifi au même modem/routeur que “salon”.

Si “geek” veut depuis l'ordinateur “bureau” être en mesure de faire les mises à jour sur “salon”, y puiser ou déposer des fichiers, et dépanner l'utilisateur “famille” qui y est connecté, le “serveur ssh”

devra être installé sur “salon”, “geek” utilisera un “client ssh” depuis l'ordinateur “bureau” pour s'y connecter.
Si on veut que les possibilités de connexion soient symétriques, il faut qu'un démon sshd fonctionne sur les deux machines.
C'est clair ? Un dessin illustrant cette situation :




Dans ce contexte on peut se demander l'utilité d'une connexion sécurisée, cependant en milieu urbain les connexions wifi sont assez facile à “écouter”, la connexion cryptée garantie donc un peu plus de respect de la vie privée. Mais c'est lors de connexion à travers le réseau Internet que cette sécurité prend tout son sens.

## Utilisation basique

Une séquence de connexion se passe comme ceci du côté client (“geek” sur “bureau” dans notre exemple) :

```
$ ssh famille@192.168.1.2
famille@192.168.1.2 passwd:
```

Vous dites à “ssh” de se connecter au compte “famille” sur l'ordinateur dont l'adresse ip est 192.168.1.2 (“salon”). “ssh” va donc envoyer la demande de connexion qui va être reçue par “sshd” sur l'ordinateur “salon”, qui en retour demande le mot de passe du compte “famille”.
Lors de la première connexion uniquement vous recevrez un avertissement car le serveur auquel vous vous connectez n'est pas “connu”, “ssh” vous demandera si vous acceptez de l'ajouter à la liste des serveurs de confiance, répondez “oui”.

```
$ ssh famille@192.168.1.2
famille@192.168.1.2 passwd:

The authenticity of host '[192.168.1.2]:22 ([192.168.1.2]:22)' can't be
```

```
established.
RSA key fingerprint is 32:fe:a4:27:c3:d7:5f:52:bb:re:ee:a6:25:14:6a:a3.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[192.168.1.2]:22' (RSA) to the list of known
hosts.
[...]
```

Une fois ces formalités passées vous êtes connecté sur l'ordinateur cible (ici “salon”) sous l'identité du compte dont vous avez donné le mot de passe (ici “famille”). Toutes les commandes que vous utilisez à partir de maintenant sont exécutées sur et depuis l'ordinateur cible (“salon”). Par exemple si vous lancez une installation de logiciel celui-ci sera installé sur “salon”, et pas sur “bureau”.
Pour illustrer ce basculement le texte qui précède les commandes dans une console linux, qu'on appel le “prompt”, va changer pour indiquer la nouvelle origine du shell. Dans notre exemple on passera de :

```
geek@bureau:~$
```

à

```
famille@salon:~$
```

Pour quitter la session “ssh” et fermer la connexion, vous pouvez taper “**exit**” ou utiliser la séquence d'échappement “**~.**” (sans les guillemets).

Ce fonctionnement basique peut tout à fait suffire dans un contexte d'usage personnel, mais on peut faire mieux. Nous allons faire un tour dans la configuration.

Si l'utilisateur du poste client dispose d'un compte sur la machine serveur, il peut se connecter en indiquant simplement l'adresse ip du serveur, sans préciser de nom de login (la partie “user@”).

## Configuration

Le point le plus important de la configuration de “ssh” est l'authentification des connexions. En bref il s'agit de déterminer qui peut se connecter, à partir d'où, et selon quelles modalités.

Les fichiers de configurations se trouvent dans le répertoire /etc/ssh, il s'agit de “sshd_config” pour la partie “serveur”, et “ssh_config” pour la partie “client”. À cela s'ajoute dans le répertoire personnel (noté **~**) de l'utilisateur un dossier caché ~/.ssh/. Il contient les clés d'identification lorsqu'elles ont été créées, ainsi que “l'empreinte” des serveurs auxquels l'utilisateur s'est déjà connecté, et dont il a accepté la clé d'identification (fichier “known_hosts”). Sur le poste qui fait office de serveur on trouvera également un fichier “~/.ssh/authorized_keys” qui contient la signature “publique” des clés de sécurité des clients autorisés, si le mode l'authentification par clé est utilisé.
En plus des fichiers de configuration globale on peut créer des fichiers additionnels par utilisateur “~/.ssh/config” pour les réglages personnalisés.

Par défaut sshd sera probablement configuré pour accepter les connexions de n'importe quel client, avec pour authentification le “login” (son mot de passe) de l'utilisateur sur le compte duquel “sshd”

fonctionne. Dans notre exemple sshd est lancé au démarrage sur “salon” où l'utilisateur “famille” est connecté. Le client sur “bureau” pourra donc initier une connexion vers “salon” et s'identifier en donnant le mot de passe du compte “famille”, s'il le connaît.

Pour améliorer la sécurité on pourra commencer par ces options dans sshd_config :

Après tout changement de configuration de “sshd” vous devez redémarrer le service, sinon la nouvelle configuration ne sera pas prise en compte. Un moyen simple est d'appeler le script d'initialisation comme ceci :

```
# /etc/init.d/ssh restart
```

- Retirer l'autorisation de se connecter depuis le compte du super-utilisateur “root”. C'est généralement inutile et cela peu représenter un risque pour la sécurité. Une fois connecté on peut si nécessaire de toute façon obtenir des droits root avec "su" ou "sudo". On s'assurera d'avoir la ligne :

```
PermitRootLogin no
```

- Déterminer si on doit faire suivre la connexion du serveur graphique via “ssh”, c'est à dire permettre au client de lancer une application graphique sur le serveur, et de la voir s'afficher sur son ordinateur. Cette possibilité peut être pratique dans le cadre d'une utilisation personnelle, et certains programmes ne sont pas utilisables en ligne de commande :

```
X11Forwarding yes
```

Pour tirer partie du déport du serveur graphique vous pouvez utiliser l'option “**-X**” lors de la connexion ssh :

```
$ ssh -X famille@192.168.1.2
```

De cette façon si vous lancez une application graphique pendant la session ssh elle sera affichée sur votre écran et non sur celui de la machine distante. Attention avec une connexion lente, les temps d'affichage et de rafraîchissement de l'affichage peuvent être longs.
Si vous voulez systématiquement utiliser le déport du serveur graphique vous pouvez ajouter dans le fichier “/etc/ssh/ssh_config” du(des) poste(s) client(s) :

```
ForwardX11 yes
```

Cette ligne rend inutile l'option “**-X**” en autorisant le déport d'affichage systématiquement par défaut, à partir du moment où le serveur l'autorise également bien sûr. On peut annuler pour une session le déport du serveur graphique avec l'option “**-x**” (“x” minuscule cette fois). Le déport du serveur graphique peut constituer un risque pour la sécurité, un utilisateur malveillant ou un attaquant pouvant récupérer les frappes au clavier par l'intermédiaire du serveur graphique Xorg, et donc d'éventuels mots de passe. Ce risque est totalement négligeable dans le cadre d'un usage personnel sur un réseau local.

- Changer le port de communication par défaut. “ssh” utilise par défaut le port 22, mais “ssh” étant un service très utilisé et sensible au niveau de la sécurité les attaques contre ce port sont devenus très fréquentes. Si vous utilisez ssh à travers le réseau Internet c'est une bonne idée de changer ce port pour quelque chose de moins “classique”, cela vous épargnera les assauts de groupe de machines “robots” (bots) qui testent de manière aléatoire des machines sur le port 22, et tentent ensuite de forcer le mot de passe de connexion.

Pour que cela fonctionne il faut modifier le port utilisé avec la directive (on peut utiliser plusieurs ports):

```
port 62500
```

Vous devez vous assurer que le(s) port(s) choisi(s) (ici 62500) soit accessible à travers votre routeur (NAT) et votre pare-feu, à la fois sur la machine “serveur” et sur les machines clients. Pour que “ssh-client” utilise ce nouveau port au lieu du 22 il faut le lui indiquer avec l'option “**-p**” :

```
$ ssh -p 62500 famille@192.168.1.2
```

Si vous souhaitez que ce changement soit permanent, modifiez la directive “port” de la même façon que pour “sshd_config” dans “/etc/ssh_config” sur le(s) poste(s) client(s). Dans ce cas vous n'aurez plus à utiliser l'option “**-p**”.

Avant de choisir un port arbitrairement, consultez la listes des ports réservés et ne choisissez pas un de ceux qui sont employés de manière standard pour d'autres services. D'une manière générale n'employez jamais un port inférieur à 1024.

- Forcer ssh à n'écouter que sur une interface. Si vous avez plusieurs interfaces réseau, vous pouvez restreindre l'écoute de sshd sur l'une d'elles avec :

```
ListenAddress 192.168.0.2
```

Ici seule l'interface avec l'adresse ip 192.168.0.2 sera utilisée par sshd. Bien sûr n'utilisez pas cette option si votre adresse ip est attribuée par dhcp et change régulièrement.

- Restreindre l'accès à un utilisateur particulier, ou un réseau, ou les deux. Si vous utilisez l'identification par le login des utilisateurs, vous pouvez restreindre l'usage de ssh à un seul utilisateur avec :

```
AllowUsers famille
```

Ici seul le compte “famille” pourra être utilisé. Pour spécifier un réseau, par exemple le réseau local, utilisez :

```
AllowUsers *@192.168.0.*
```

Ici tous les utilisateurs sont autorisés à se connecter depuis le réseau “192.168.0.*” (“*” est un joker qui autorise n'importe quelle valeur, par exemple 192.168.0.2), c'est à dire toutes les adresses qui appartiennent à ce réseau local. Pour coupler les deux, utilisez la syntaxe “utilisateur@adresse”, par exemple “famille@192.168.0.*” pour l'utilisateur “famille” sur le réseau local “192.168.0.*”.

Alternativement on peut indiquer un ou plusieurs groupe(s) avec “**AllowGroups**”.

- Empêcher que la connexion ssh n'expire. Par défaut si une connexion ssh ouverte n'est pas utilisée, elle sera automatiquement coupée par le serveur. Si vous n'avez pas d'impératif de sécurité strict et que vous voulez maintenir les sessions en cours ouvertes, utilisez :

```
TCPKeepAlive yes
```

- Améliorer la sécurité sur le serveur en séparant les droits, et en vérifiant les permissions des fichiers de configuration (à placer dans “/etc/ssh/sshd_config”) :

```
UsePrivilegeSeparation yes
StrictModes yes
```

La première ligne permet de forcer ssh à n'exécuter que le strict minimum de code en root, la seconde provoque une vérification des permissions sur les fichiers de configuration (clés d'identification etc...).

- On peut empêcher les connexions vers des comptes sans mot de passe :

```
PermitEmptyPasswords no
```

Accepter les connexions vers des comptes sans mot de passe est un risque pour la sécurité, mais ça peut être nécessaire pour certaines applications de sauvegarde à distance, qui se connectent automatiquement avec un compte spécifique sans mot de passe. Dans ces cas il vaudra bien mieux utiliser une identification par clés sans mot de passe (voir “Authentification par clés” ci-dessous).

- Désactiver le mode d'identification “Challenge-Response”. Ce mode d'identification a déjà fait l'objet d'attaques sur des failles de sécurité par le passé, et il n'y a pas de raison de le laisser activé si on utilise une identification par mot de passe ou par clés.

```
ChallengeResponseAuthentication no
```

- On peut utiliser un mode d'encryption différent de celui par défaut (3des) pour crypter les connexions. L'algorithme “blowfish” est par exemple réputé aussi sûr que “3des”, mais plus rapide. Ajoutez au fichier “/etc/ssh/ssh_config” du client :

```
Cipher blowfish-cbc
```

Si vous voulez une liste, qui sera testée dans l'ordre en cas de non compatibilité d'un algorithme avec le serveur, mettez :

```
Cipher blowfish-cbc,aes256-cbc,aes192-cbc,3des-cbc
```

Pour une liste complète des algorithmes supportés, voyez le site de “Openssh” (voir “Liens” en fin de page).

Si vous voulez plus d'information sur une option donnée, vous pouvez consulter la documentation (voir “Liens” en fin de page). Coller simplement l'option en question dans un moteur de recherche devrait vous renvoyer de nombreux résultats pertinents également. Bien sur la page de manuel est un passage obligé :

```
$ man sshd_config
```

## Authentification par clés

La première étape consiste à générer la paire de clés, qui se compose d'une clé “publique”, que nous communiquerons au serveur (“salon”), et d'une clé “privée”, qui restera sur l'ordinateur client (“bureau”). La paire de clés générée sera cryptée selon un algorithme choisi au moment de la création, “rsa” ou “dsa” généralement. Peu importe ce que vous choisirez dans un cadre familiale, si rien n'est précisé c'est le choix par défaut qui s'appliquera.

La paire de clés peut être protégée par mot de passe ou pas, à votre guise. Si vous choisissez un mot de passe l'authentification se déroulera en deux étapes, la comparaison des clés publique et privée, ainsi qu'une demande de mot de passe. Cette méthode procure un niveau de sécurité élevé (si le mot de passe est bien choisi), mais nécessite de ne pas oublier le mot de passe, de le taper à chaque connexion (des programmes permettent d'alléger cette contrainte en stockant temporairement le mot de passe), et peut bloquer l'utilisation de certains services à distance qui requièrent un accès ssh sans mot de passe (programme de sauvegarde à distance, montage automatique d'un répertoire distant avec “sshfs”, etc...).

À vous de choisir ce qui est le plus adapté à votre cas, si vous optez pour une paire de clés sans mot de passe, veillez simplement à ne pas “égarer” une copie de votre clé privée, et soyez conscient que quiconque a accès au poste client aura également accès au poste qui fait tourner “sshd”.

Pour créer les clés on utilise :

```
$ ssh-keygen
```

Choisissez un mot de passe “fort”, du type de ceux que l'on réserve au compte root, avec des lettres, chiffres, différentes casses et des symboles éventuellement. Au minimum 10 caractères, et ni de noms propres, dates de naissances ou mot existant dans un dictionnaire (peu importe la langue), ces précautions vous prémuniront contre les attaques qui consistent à tester un grand nombre de mots de passes courants à l'aide de programmes spécialisés. Cela n'est important que si votre utilisation de ssh passe par le réseau Internet, si elle ne concerne que votre réseau local comme dans notre exemple le nom du chat fera l'affaire ! Si vous voulez générer des clés sans mot de passe, laisser blanc.

Si pour des raisons de compatibilité vous devez créer des clés compatibles avec le protocole “sshv1” (l'ancien protocole, aujourd'hui remplacé par la version 2), vous pouvez indiquer le type de clé à créer avec l'option “**-t**”. Pour une clé rsa version 1 vous indiquerez “**ssh-keygen -t rsa1**”. Vous pouvez également choisir une clé “dsa” au lieux du choix par défaut “rsa” (“dsa” et “rsa” sont uniquement compatibles avec la version 2 de ssh)

L'étape suivante consiste à copier la clé publique sur le(s) serveur(s) auxquels vous souhaitez pouvoir vous connecter. “ssh” installe un script qui peut se charger de cette étape pour vous, si vous posséder déjà un accès ssh à la machine serveur (ici “famille@192.168.1.2”). Il suffit de taper sur le client :

```
$ ssh-copy-id famille@192.168.1.2
```

Si vous souhaitez indiquer une clé particulière (par exemple une clé “dsa”, utilisez l'option “**-i**” comme ceci :

```
$ ssh-copy-id -i ~/.ssh/id_dsa.pub famille@192.168.1.2
```

Cette commande va effectuer plusieurs tâches importantes, à commencer par la création d'un répertoire “~/.ssh” sur le serveur, avec des permissions adaptées de “700” (rwx------). À l'intérieur de ce répertoire sera créé un fichier “authorized_keys” avec des permissions de “600” (rw-------) qui contiendra la clé publique créée auparavant avec “ssh-keygen” sur le poste client.

Ces opérations peuvent être effectuées manuellement, en copiant la clé publique sur le serveur, à condition que le répertoire “~/.ssh” y existe déjà. Un exemple pour effectuer ces opérations depuis le poste client via “ssh” (utilisé de manière simple le temps de l'opération), vers le poste serveur (ici “famille@192.168.1.2”) :

```
$ cat ~/.ssh/id_rsa.pub | ssh famille@192.168.1.2 tee -a
~/.ssh/authorized_keys
```

<wrap info>Certaines versions de “ssh” n'autorisent pas la connexion en “strict mode” si la clé publique est toujours présente dans “~/.ssh/” sur le client. Il faut donc la copier ailleurs et l'effacer.</wrap>

Si le répertoire “~/.ssh” n'existe pas encore il faut le créer, pour cela on se connecte avec “ssh” sur le serveur, et on y effectue les opérations nécessaires :

```
$ ssh famille@192.168.1.2
mkdir ~/.ssh
chmod 700 ~/.ssh
touch ~/.ssh/authorized_keys
chmod 600 ~/.ssh/authorized_keys
```

Vous avez saisi le but, peu importe la méthode, vous pouvez également copier la clé publique “id_rsa.pub” sur le serveur avec une clé usb, créer les répertoires nécessaires sur le serveur et copier le contenu de la clé dans le fichier “~/.ssh/authorized_keys”.

Une fois que les clés sont en place il reste à modifier la configuration du serveur pour ne plus accepter les connexions par login, mais uniquement celles par clés. Ne vous trompez pas si vous effectuez les modifications par ssh et que vous n'avez pas d'accès physique à la machine (hébergement distant etc…), sinon vous risquez de ne plus pouvoir vous connecter !

Autoriser l'identification par clés :

```
PubkeyAuthentication yes
```

Interdire l'identification par le login d'un utilisateur :

```
PasswordAuthentication no
ChallengeResponseAuthentication no
```

Une fois ces changements appliqués, et “sshd” redémarré, il ne sera plus possible de se connecter au serveur sans y avoir enregistré sa clé publique.

### Changement d'identité du serveur ?

ssh vérifie l'identité des serveurs auxquels vous vous êtes déjà connecté (empreinte présente dans le fichier “know_hosts”). Cette précaution permet d'éviter des attaques de type “man in the middle”, où votre connexion est interceptée et redirigée (souvent à travers un proxy) vers un serveur ssh usurpant l'identité du serveur légitime auquel vous essayez de vous connecter. Voici un aperçu de l'avertissement reçu en cas de non vérification de la signature, c'est assez explicite :

```
$ ssh famille@192.168.1.2
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
e4:a0:73:8d:8a:f1:26:a1:49:0c:69:bb:bz:24:36:a1.
Please contact your system administrator.
Add correct host key in /home/geek/.ssh/known_hosts to get rid of this
message.
Offending key in /home/geek/.ssh/known_hosts:2
RSA host key for [192.168.1.2]:22 has changed and you have requested strict
checking.
Host key verification failed.
```

Vous recevrez cet avertissement dans deux cas, si vous êtes victime d'une attaque de type “man in the middle”, peu probable si vous utilisez ssh sur votre réseau local, à moins que votre fils soit un fan de “hacking”. Vous recevrez également cet avertissement si vous avez réinstallé le système, ou simplement “ssh” en recréant des clés sur le serveur. Dans ce cas il faudra supprimer l'ancienne signature du serveur de votre fichier ~/.ssh/known_hosts. La signature en question est indiquée dans le message d'erreur, c'est important si vous avez activé le “hachage” des signatures car vous n'avez aucun moyen de savoir laquelle correspond au serveur en question.

```
Add correct host key in /home/geek/.ssh/known_hosts to get rid of this
message.
Offending key in /home/geek/.ssh/known_hosts:2
```

On peut traduire par :
“Ajoutez la clé correcte dans ”/home/geek/.ssh/known_hosts“ pour ne plus voir ce message. La clé incriminée est ”/home/geek/.ssh/known_hosts:2“
où le chiffre “**2**” vous indique la ligne à supprimer dans le fichier “known_hosts”.
À la prochaine connexion il vous sera demandé d'accepter la nouvelle signature du serveur, n'acceptez évidemment pas si vous pensez être victime d'une manœuvre de piraterie !!

**"ssh-agent", garder en mémoire les mots de passe de clés**

Si vous utilisez un identification par clé protégée par mot de passe, et que vous avez un usage “intensif” de “ssh” avec plusieurs connexions simultanées, il est intéressant d'utiliser une programme qui garde en mémoire le mot de passe associé à la clé le temps de la session.

“ssh-agent” fait justement ça, il stocke le mot de passe associé aux clés de sécurité le temps de la session, et permet de ne pas avoir à le retaper à chaque nouvelle connexion.

“ssh-agent” est en général installé avec le paquet “ssh-client”, et configuré pour démarrer en même temps qu'une session est ouverte. Pour savoir s'il fonctionne :

```
$ pgrep -l ssh-agent
```

Si ça n'est pas le cas, vous pouvez lancer “ssh-agent” depuis la ligne de commande, mais comme “ssh-agent” a besoin de passer des variables à votre environnement shell au lancement on ne peut pas simplement appeler la commande (sinon les variables seront simplement écrites sur la sortie de la console, ce sera à vous de les exporter). Vous pouvez utiliser :

```
$ eval `ssh-agent`
```

Pour mettre le mot de passe associé à votre (vos) clé(s) en mémoire pour la session il suffit donc d'appeler “**ssh-add**” sans argument :

```
$ ssh-add
```

Le mot de passe associé à la clé vous sera demandé une fois, et vous pourrez ensuite vous connecter sans devoir le taper à chaque nouvelle connexion.

Quelques astuces, pour lister les clés actuellement en mémoire utilisez l'option “**-l**”, pour effacer toutes les clés enregistrées utilisez l'option “**-D**”.
Pour afficher l'intégralité des clés stockées en mémoire, et pas seulement leurs empreintes, vous pouvez utiliser l'option “**-L**”. C'est pratique pour ajouter toutes les clés enregistrée à un fichier “authorized_keys” prêt à être copié sur un serveur. Pour ça faite :

```
$ ssh-add -L > authorized_keys
```

Vous allez créer un fichier “authorized_keys” dans le répertoire courant.

Si vous quitter votre poste et qu'il risque d'être utilisé en votre absence, verrouillez “ssh-agent” avec un mot de passe en utilisant l'option “**-x**” :

```
$ ssh-add -x
```

Pour le déverrouiller utilisez l'option “**-X**” (majuscule cette fois), et donner le mot de passe que vous avez choisi.

Un mot sur la sécurité, “ssh-agent” stocke les clés de manière sûre, seul un utilisateur local avec des droits root pourrait accéder aux clés stockées et les utiliser pour se connecter. En revanche “ssh-agent” possède une fonction de “forwarding” (suivi) des clés qui peut constituer un risque pour la

sécurité. Cette fonction est destinée à exporter les clés enregistrées sur votre machine vers une seconde à laquelle vous vous connectez, et d'où les clés stockées par ssh-agent seront utilisables également (pour se connecter vers une troisième machine depuis la seconde par exemple). Les clés seront lisibles par un utilisateur disposant de droits root sur la machine où seront exportées les clés, et ce mécanisme est assez facilement exploitable. Pour désactiver cette fonction de “forwarding” si vous ne l'utilisez pas, mettez dans votre fichier ”/etc/ssh/ssh_config“ :

```
Host *
ForwardAgent no
```

## Tunnel ssh

“ssh” est fréquemment utilisé pour “encapsuler” un autre service non sécurisé, on parle alors de “tunnel ssh”. Le procédé consiste à établir une connexion ssh entre deux ordinateurs sur des ports déterminé, et ensuite d'utiliser cette connexion “ssh” pour connecter un autre comme “vnc”, notre exemple ici.
“vnc” est une programme de connexion à distance graphique, une sorte de bureau distant à la manière du programme “remote desktop” intégré à Windows. “vnc” est très pratique et efficace, mais transmet toutes les informations en clair, c'est donc un bon candidat pour l'encapsulation “ssh”.

On suppose que le serveur “vnc” fonctionne sur le poste serveur (“famille@192.168.1.2” dans notre exemple), par exemple le programme “vino” qui fait partie de Gnome. Par défaut le serveur “vnc” va attendre les connexions sur le port 5900 (paramétrable dans les préférences du serveur vnc).

- Établir le tunnel sur le port 5900, à taper sur le client qui veut accéder au serveur “vnc” (“geek@192.168.1.3” dans notre exemple) :

```
$ ssh -L 5901:localhost:5900 famille@192.168.1.2
```

L'option “**-L**” (pour “Local forward”) indique l'établissement d'un tunnel, depuis le port local 5901, vers le port 5900 de l'ordinateur distant. Le choix des ports est libre, on aurait pu choisir n'importe quel port libre sur la machine locale, la convention fait que vnc utilise les ports 590*.
Une fois le tunnel établie, il ne reste qu'à lancer un client “vnc” (comme “vinagre” du bureau Gnome, ou “KRDC” pour KDE), et à lui indiquer de se connecter à “**localhost:5901**” (sans guillemets). La connexion “vnc” se déroule ensuite normalement, le client “vnc” envoie ses requêtes sur “localhost” mais c'est bel et bien l'ordinateur distant qui les reçoit, le passage par le tunnel “ssh” est totalement transparent, mais toutes les données échangées seront cryptés (sur le réseau du moins, elles transitent en clair sur les machines elles mêmes entre le tunnel “ssh” et les applications “vnc”). La connexion à travers le tunnel peut juste s'avérer plus lente, particulièrement en wifi.
Pour avoir une “image” plus clair de ce qui se passe, vous pouvez lancer la commande “nc” pour surveiller l'activité sur le port 5901 de la machine locale, avant et après l'établissement du tunnel. Vous verrez le serveur “vnc” se manifester sur le port après l'ouverture du tunnel.

```
$ nc localhost 5901
```

L'autre avantage du tunnel est que vous n'avez pas besoin d'ouvrir de ports supplémentaires sur la machine locale ou le routeur, ni sur la machine distante, tout le trafic passant par le port utilisé par “ssh”.

Si la connexion n'est destinée qu'à établir un tunnel, on peut veiller à ce qu'elle ne soit pas fermée pendant une période d'inactivité avec l'option “TCPKeepAlive yes” dans le fichier de configuration de “ssh”. Si ça n'est pas suffisant on peut utiliser le programme “autossh” qui maintient la connexion ouverte, et la rétablie si elle vient à être fermée malgré tout. Pour améliorer la sécurité on peut préciser à “ssh” que la connexion n'est destinée qu'à établir un tunnel, et pas à exécuter des commandes sur l'ordinateur de destination, avec l'option “**-N**”.

Une tentative de représentation “visuelle” du “tunnel ssh” :

```
application <--> boucle locale, port 5901       boucle locale, port 5900 <-
-> application
Client vnc     |                                                ^
Serveur vnc
           |              Réseau local               |
               V              ou Internet                |
     SSH (port 22 par défaut) --------------------> SSH (port 22 par défaut)
                         tunnel cryptée
```

On peut créer un alias pour le tunnel, à l'aide d'un fichier de configuration ”~/.ssh/config“ (à créer). Ce fichier aura par exemple le contenu suivant dans notre exemple :

```
Host             tunnel_vnc
Hostname         192.168.1.2
LocalForward     localhost:5901 localhost:5900
User             famille
```

Vous pourrez alors lancer le tunnel avec la commande :

```
$ ssh tunnel_vnc
```

Pour en savoir plus, “man ssh_config” bien sûr !

#### Tunnel inversé

Le “tunnel inversé”, ou “Remote forward” fait juste l'inverse du “Local forward” vu précédemment, il “exporte” un port de la machine locale vers un autre port d'une machine.
On utilise l'option “**-R**” de “ssh” depuis la machine dont on veut “exporter” un port, la syntaxe est l'inverse de celle du tunnel local, on précise d'abord le port d'écoute sur la machine distante, puis celui sur la machine locale. Un exemple dans lequel on exporte le port 631 (port de l'interface d'administration du serveur d'impression “CUPS”) vers le port 8888 de la machine distante (ici “famille@192.168.1.2” sur “salon”) :

```
$ ssh -R 8888:localhost:631 famille@192.168.1.2
```

Si on laisse tourner cette commande sur “bureau”, et qu'on se rend sur “salon” où un ouvre un navigateur Internet, en pointant celui-ci vers l'adresse “localhost:8888” (sans guillemets) on arrivera sur la page d'administration de “CUPS” (il s'agit évidement de celle de la machine “bureau” que nous venons “d'exporter”).

Si vous utilisez :

```
$ ssh -R 8888:www.linuxpedia.fr:80 -N famille@192.168.1.2
```

vous permettez à l'utilisateur “famille” sur l'ordinateur “salon” (192.168.0.2) de se connecter au site “www.linuxpedia.fr”, et ses sous-rubriques, en pointant un navigateur Internet sur l'url ”localhost:8888“ (sans guillemets). Si vous cliquez ensuite sur une sous-rubrique du site, par exemple “ligne de commande”, vous verrez s'afficher la page avec dans la barre d'état ou d'url l'adresse ”http://localhost:8888/doku.php/commande/commande“. Vous êtes bien sur la page dédiée à la ligne de commande de “linuxpedia.fr”, mais pour votre navigateur vous naviguez toujours l'url local “localhost” sur le port 8888.
Bien sûr cela fonctionne même si “salon” ne dispose normalement pas d'un accès Internet, mais simplement d'un accès au réseau local, alors que l'utilisateur “geek” sur “bureau” dispose lui d'un accès Internet. On comprend bien que ces mécanismes de “tunnel” peuvent facilement servir à contourner un pare-feu, un routeur utilisant le NAT ou même un système de filtrage basé sur un proxy (type “protection parentale”). Le “tunnel ssh” est très pratique dans certains usages, il peut être un cauchemar pour un administrateur réseau en permettant aux utilisateurs de “s'évader” du réseau local, et de créer par la même occasion des failles de sécurité… Tout dépend de l'usage qu'on en fait.

<WRAP info>Le port 80 que nous avons “exporté” dans le dernier exemple est celui qui sert aux serveurs Internet comme “apache”, et celui qu'utilisent les navigateurs Internet pour se connecter. Pour que “l'export” avec un “tunnel ssh” fonctionne il faut que ce port soit libre, c'est à dire qu'un navigateur Internet ou un autre service ne soit pas en fonction sur ce port.</WRAP>

<WRAP danger>Si vous utilisez les “tunnel ssh” pour contourner un pare-feu ou un proxy à votre travail, dans votre université ou autres cas similaires, vous vous exposez à des sanctions extrêmement graves. En utilisant cette technique vous ouvrez l'ordinateur au réseau Internet, offrant une porte d'entrée potentielle à des pirates que les administrateurs réseau s'efforcent de tenir à l'écart. Si un piratage résulte de votre “bidouillage” vous pourrez être condamné à payer des dommages et intérêts colossaux, en plus d'être renvoyé bien sûr… À bon entendeur.</WRAP>

### Tunnel ssh comme proxy "SOCKS"

Dans l'exemple précédent nous avons “partagé” un domaine Internet avec un tunnel inversé, mais “ssh” peut faire bien mieux, il peut jouer le rôle de proxy de connexion vers l'ensemble du réseau Internet, par exemple pour des machines ne possédant pas d'accès directe à Internet. Pour cela on utilise l'option “**-D**” (pour “dynamic”) :

```
$ ssh -D 6969 famille@192.168.1.2
```

Étant donné que la connexion “ssh” n'est pas destiné à obtenir un shell sur la machine distante on utilisera l'option “**-N**” (No execute), et on préférera “autossh” (à installer séparément) à “ssh” pour garder le tunnel ouvert, et le reconnecter en cas de coupure de la connexion. Si vous utilisez une connexion à faible débit, vous pouvez également activer la compression des données avec l'option “**-C**” de “ssh” :

```
$ autossh -NCD 6969 famille@192.168.1.2
```

Une fois le tunnel dynamique lancé “ssh” va se comporter comme un proxy SOCKS sécurisé, pour

l'utiliser avec Firefox ouvrez les préférences > avancées > réseau > paramètres, et configurez manuellement comme dans la capture ci-dessous le proxy SOCKS v5 pour qu'il pointe vers “localhost:6969”.

À partir de là tout votre trafic Internet est routé à travers le port local 6969, le tunnel “ssh” (qui utilise le port de “ssh) crypté entre les deux machines, puis vers le réseau Internet à partir de la seconde machine. Quel peut être l'intérêt ?

- Si la première machine n'a pas de connexion vers Internet, c'est un moyen simple de partager la connexion de la seconde, de façon sécurisée y compris à travers une connexion wifi.
- Si vous êtes derrière un pare-feu qui interdit ou filtre les connexions vers Internet, mais autorise ssh, vous pouvez utiliser votre ordinateur personnel comme proxy.
- À l'inverse vous pouvez utiliser ce proxy pour rendre accessible l'intranet de votre bureau depuis votre domicile.
- Si vous êtes sur un réseau ouvert publique (cybercafé, “hotspot” wifi), vous pouvez utiliser ce mécanisme pour vous connecter à travers votre ordinateur personnel, à travers le proxy sécurisé, plutôt que de surfer “en clair” sur un réseau ouvert.

Soyez conscient que c'est l'adresse ip publique de l'ordinateur qui sert de proxy qui sera visible pour les sites Internet, et le propriétaire de cet ordinateur sera tenu responsable de l'usage qui en sera fait.

Il existe des limitations à cette technique, vous pouvez utiliser une extension à Firefox ”Foxy-Proxy” pour gérer plus finement comment le proxy est utilisé, et pour quel(s) site(s). Si le port ssh n'est pas ouvert là où vous vous trouvez vous pouvez utiliser un autre port, ou utiliser une application spécialisée comme “proxytunnel” pour “déguiser” le tunnel ssh en simple connexion http.
Soyez également conscient que certaines connexions peuvent échapper au proxy, en particulier les requêtes dns qui peuvent en dire long sur les sites visités. Pour remédier à cela il faut configurer la connexion Internet de manière globale (au niveau des paramètres système) et pas seulement au niveau du navigateur Internet.
Attention à ces points si vous voulez offrir un proxy sécurisé à des amis habitant dans des pays où l'accès Internet est sévèrement filtré, dans ce contexte la moindre erreur de réglage peut leur coûter très cher... Pour des configurations de proxy avancées il faudra coupler ssh à d'autres proxy comme “squid”.

- Pour voir ce qui se passe lors d'une connexion, utilisez le mode “verbose” ou “debug” de “ssh” avec l'option “**-v**”. Un exemple sur une connexion qui établie un tunnel, la clé est stockée par “ssh-agent”, et “ssh” utilise le port 69222 :

```
$ ssh -v -L 5901:localhost:5900 famille@192.168.1.2
OpenSSH_5.1p1 Debian-5+b1, OpenSSL 0.9.8g 19 Oct 2007
debug1: Reading configuration data /etc/ssh/ssh_config
debug1: Applying options for *
debug1: Connecting to 192.168.1.2 [192.168.1.2] port 69222.
debug1: Connection established.
debug1: identity file /home/geek/.ssh/id_rsa type -1
debug1: identity file /home/geek/.ssh/id_dsa type -1
debug1: Remote protocol version 2.0, remote software version OpenSSH_4.7p1
```

```
Debian-8ubuntu1.2
debug1: match: OpenSSH_4.7p1 Debian-8ubuntu1.2 pat OpenSSH_4*
debug1: Enabling compatibility mode for protocol 2.0
debug1: Local version string SSH-2.0-OpenSSH_5.1p1 Debian-5+b1
debug1: SSH2_MSG_KEXINIT sent
debug1: SSH2_MSG_KEXINIT received
debug1: kex: server->client aes128-cbc hmac-md5 none
debug1: kex: client->server aes128-cbc hmac-md5 none
debug1: SSH2_MSG_KEX_DH_GEX_REQUEST(1024<1024<8192) sent
debug1: expecting SSH2_MSG_KEX_DH_GEX_GROUP
debug1: SSH2_MSG_KEX_DH_GEX_INIT sent
debug1: expecting SSH2_MSG_KEX_DH_GEX_REPLY
debug1: Host '[192.168.1.2]:69222' is known and matches the RSA host key.
debug1: Found key in /home/geek/.ssh/known_hosts:3
debug1: ssh_rsa_verify: signature correct
debug1: SSH2_MSG_NEWKEYS sent
debug1: expecting SSH2_MSG_NEWKEYS
debug1: SSH2_MSG_NEWKEYS received
debug1: SSH2_MSG_SERVICE_REQUEST sent
debug1: SSH2_MSG_SERVICE_ACCEPT received
debug1: Authentications that can continue: publickey
debug1: Next authentication method: publickey
debug1: Offering public key: /home/geek/.ssh/id_rsa
debug1: Server accepts key: pkalg ssh-rsa blen 277
debug1: Authentication succeeded (publickey).
debug1: Local connections to LOCALHOST:5901 forwarded to remote address
localhost:5900
debug1: Local forwarding listening on 127.0.0.1 port 5901.
debug1: channel 0: new [port listener]
socket: Address family not supported by protocol
debug1: channel 1: new [client-session]
debug1: Entering interactive session.
debug1: Sending environment.
debug1: Sending env LANG = fr_FR.UTF-8
debug1: Sending env LC_CTYPE = fr_FR.UTF-8
Last login: Mon Jun  1 21:05:21 2009 from bureau
```

## Transfert de fichiers sécurisé : scp, sftp et sshfs

Si “ssh” permet d'obtenir un shell et exécuter des commandes sur un ordinateur distant, il permet également de transférer des fichiers entre deux machines, toujours de manière sécurisée.

### scp

Le moyen le plus simple et rudimentaire est la commande “**scp**”, fournie avec “Openssh-client” :

```
$ scp ~/Documents/fichier.txt famille@192.168.1.2:Documents/
```

Ici on envoie “fichier.txt” vers la machine distante 192.168.1.2, le fichier sera copié dans le répertoire “/home/famille/Documents”. Attention à la syntaxe de la cible, les deux point “:” qui suivent l'adresse indique le répertoire personnel de l'utilisateur. Si vous copiez vers/depuis un sous-répertoire du répertoire personnel vous devez l'indiquer sans “/” au début, comme dans notre exemple. Si vous voulez copier par exemple dans /etc/ssh/ il faudra écrire “famille@192.168.1.2:/etc/ssh/”.

Pour faire des copies récursives “scp” supporte l'option “**-r**” (les liens symboliques sont alors suivis par “scp”), “**-P**” permettra d'indiquer un port de connexion.

Avec l'option “-r” le comportement sera différent si le répertoire cible existe ou non lors de la copie, par exemple :

```
$ scp -r dossier famille@192.168.1.2:Documents
```

Si “Documents” existe sur la machine distante, “dossier” sera créé et son contenu copié à l'intérieur de “Documents”. Si “Documents” n'existe pas, la copie de “dossier” sera renommée en “Documents” et son contenu copié à l'intérieur de ce nouveau répertoire.

Pour copier plusieurs fichiers de la machine locale à la machine distante, il suffit de les mettre à la suite, séparés par un espace :

```
$ scp fichier1 fichier2 fichier3 famille@192.168.1.2:
```

N'oubliez pas les deux points “**:**” après l'adresse de la machine distante, vous pouvez bien sûr indiquer des chemins différents.

Vous pouvez utiliser des expressions rationnelles avec “scp”, par exemple pour copier nos trois fichiers 1,2 et 3 de la machine distante à la machine locale (l'inverse de précédemment) :

```
$ scp famille@192.168.1.2:fichier[1-3] .
```

La notation “[1-3]” correspond aux entiers compris entre 1 et 3, cela correspond donc à nos “fichier1”, “fichier2” et “fichier3”. Le point “**.**” en fin de commande indique le répertoire courant sur la machine locale comme cible de la copie.

#### sftp

Plus évolué, sftp est un programme de transfert de fichier “ftp” sécurisé, fournie avec “openssh-client”, pour lancer une session “sftp” interactive on procède comme pour “ssh” :

```
$ sftp famille@192.168.1.2
sftp>put Documents/fichier.txt Desktop/
Uploading Documents/fichier.txt to /home/famille/Desktop/fichier.txt
Documents/fichier.txt                  100% 1681     1.6KB/s   00:00
```

Dans cet exemple on lance un esession sftp vers “famille@192.168.1.2”, puis on utilise la commande “**put**” pour envoyer le fichier ~/Desktop/fichier.txt vers /home/famille/Desktop/ . Vous remarquerez que l'origine des chemins est ici encore le répertoire personnel des utilisateurs. “sftp” donne des indications pendant les tranferts sur la vitesse de la transaction, sa progression et sa durée.

Pour récupérer un fichier on utilisera cette fois la commande “**get**” de la même façon que pour “put” :

```
$ sftp famille@192.168.1.2
sftp>get Desktop/fichier.txt Documents/
Fetching /home/famille/Desktop/fichier.txt to Documents/fichier.txt
/home/famille/Desktop/fichier.txt     100% 1681     1.6KB/s   00:00
```

Vous remarquez que la syntaxe s'inverse en fonction de la commande :

```
sftp>get source(distante)   destination(locale)

sftp>put source(locale)   destination(distante)
```

“sftp” admet l'option “**-P**” permettra de conserver les permissions et attributs des fichiers copiés. On peut également utiliser la plupart des commandes habituelles pour effectuer des opérations sur les fichiers comme “mkdir” (créer un répertoire), “chgrp” (change groupe), “chown” (change propriétaire), “chmod” (change permissions), “rm” et “rmdir” (effacement), “rename” (renommer) etc... Ces commandes s'appliquent par défaut aux fichiers distants. On peut également obtenir des informations sur le remplissage du système de fichier distant avec “df”, mais cette comande n'est pas supporté par toutes les versions de “ssh”.
Enfin les commandes “quit ou “bye” terminent la session “sftp”.

### "lftp" comme client "sftp"

“sftp” a quelque lacune par rapport à d'autres programmes de transfert ftp, un des plus gênant est l'absence de complétion des chemins. La complétion consiste à compléter un chemin ou nom de fichier que vous êtes en train de taper, en fonction des fichiers et répertoires qui existent réellement sur ce chemin, cela limite les fautes de frappe. Pour compenser cette lacune il est courant d'utiliser le logiciel ”lftp“ (à installer séparément) comme client “sftp” afin de cumuler les avantages des deux, en pratique on bénéficie des fonctions plus avancés de “lftp”, comme la complétion des noms et chemins, le support de commandes parallèles, la reprise de transferts en cas d'interruption etc... mais on conserve le gain de sécurité apporté par “sftp”. Pour ouvrir une session on utilise simplement la directive “**open**” dans une session “lftp” :

```
$ lftp
lftp :~> open sftp://famille@192.168.1.2
Mot de passe :

lftp famille@192.168.1.2:~>ls
drwx------    2 famille famille     4096 Jun  3 09:04 .aptitude
-rw-------    1 famille famille    10185 Jun  4 01:12 .bash_history
-rw-r--r--    1 famille famille      220 Jan 29 16:43 .bash_logout
-rw-r--r--    1 famille famille     2928 Jan 29 16:43 .bashrc
drwxr-xr-x   14 famille famille     4096 Feb 15 18:19 .config
drwx------    2 famille famille     4096 Jan 31 11:43 .cups
drwx------    3 famille famille     4096 Jun  1 17:45 .dbus
-rw-------    1 famille famille       28 Jun  4 07:35 .dmrc
drwxr-xr-x   32 famille famille     4096 May 25 19:03 .dvdcss
[...]
```

```
drwxr-xr-x    2 famille famille     4096 Jun  3 21:29 Desktop
drwxr-xr-x    3 famille famille     4096 May 24 18:03 Documents
drwxr-xr-x    5 famille famille     4096 Jun  2 16:22 Downloads
drwxr-xr-x    2 famille famille     4096 Jan 29 17:09 Music
drwxr-xr-x    3 famille famille     4096 May 31 17:01 My GCompris
drwx------    2 famille famille     4096 May 28 14:08 PDF
drwxr-xr-x    3 famille famille     4096 Jan 29 18:46 Pictures
drwxr-x---    4 famille famille     4096 Feb  1 12:31 Podcasts
drwxr-xr-x    2 famille famille     4096 Jan 29 17:09 Public
drwxr-xr-x    2 famille famille     4096 Jan 29 17:09 Templates
drwxr-xr-x    2 famille famille     4096 Jan 29 17:09 Videos

lftp famille@192.168.1.5:~> !ls -l
total 3316328
drwxr-xr-x   4 geek geek       4096 mai 18 15:57 Audio Projects
drwxr-xr-x  11 geek geek       4096 mai 18 10:55 bordel in progress
drwxr-xr-x   2 geek geek       4096 jun  3 19:42 Desktop
-rw-r--r--   1 geek geek     113902 avr  7 23:38 Disk_layout.xcf
drwxr-xr-x  13 geek geek       4096 jun  3 21:41 Documents
drwxr-xr-x  26 geek geek       4096 jun  3 12:14 Downloads
[...]
```

Le listing de la commande “ls” a été coupé en raison de sa longueur, il permet juste de s'assurer qu'on a bien ouvert une session sur le compte de “famille”, et de voir que la commande “ls” de “lftp” liste également les fichiers cachés sur la machine distante.
La seconde commande est précédée du signe “**!**”, ce qui provoque sont exécution sur le compte local. On voit également que les commandes “lftp” se comportent comme des commandes Bash, et admettent des options comme ici “ls -l”. Sans surprise le listing renvoie le contenu du répertoire personnel de “geek”, celui qui a lancé la session “lftp>sftp” vers “famille”.

“lftp” nécessite un tutoriel propre, en attendant vous pouvez utiliser le manuel pour connaître les commandes de base, sachez que “get” et “put” ont la même fonction que pour “sftp”, et “exit” ou “quit” termineront la session. “lftp” supporte également le protocole “**fish**” basé sur “ssh” également, vous pouvez ouvrir une session “fish” de la même façon que pour “sftp” :

```
$ lftp
lftp :~> open fish://famille@192.168.1.2
Mot de passe :
```

Tout le trafic est crypté, et passe par le port utilisé par “ssh”, aucun besoin d'ouvrir des ports supplémentaires.

Sur certaines distributions il est possible que “lftp” ne soit pas compilé avec le support de “ssl”ou “tls” comme moyen d'identification, auquel cas il ne sera pas utilisable avec “sftp”. Pour vérifier si votre version est compatible utilisez la commande **ldd `which lftp` | egrep '(ssl|tls)'** , si celle-ci renvoie quelque chose c'est bon, sinon il faut recompiler lftp avec le support adéquat, et/ou faire un rapport de bug à votre distribution.

**sshfs**

Vous connaissez sans doute “samba” et “nfs” qui permettent de partager des répertoires en les montant sur une machine distante, et bien “sshfs” fait la même chose en bénéficiant de la sécurité de “ssh”.

Pour utiliser “sshfs” il est en général nécessaire de l'installer séparément. “sshfs” étant dépendant du système de fichier en espace utilisateur “fuse” il faut que votre utilisateur soit membre du groupe “fuse” (sur certaines distribution ce groupe peut s'appeler “fuseuser”). Pour savoir si votre utilisateur appartient au groupe “fuse” :

```
$ groups
```

Si “fuse” n'est pas dans la liste ajoutez votre utilisateur à ce groupe, et activer le changement avec :

```
# usermod -a -G fuse geek
$ newgrp fuse
```

Remplacez l”utilisateur “geek” par votre nom d'utilisateur.

L'utilisation de sshfs est ensuite très simple, elle s'apparente à un montage simple, par exemple pour monter le répertoire “Documents” sur le poste “famille@192.168.1.2” on fera :

```
$ mkdir sshfs_Documents

$ ls -l sshfs_Documents
total 0

$ sshfs famille@192.168.1.2:Documents ~/sshfs_Documents/

$ ls -l sshfs_Documents
rwxr-xr-x 4 famille famille    4096 déc  9 21:56 articles
drwxr-xr-x 7 famille famille    4096 déc  9 21:56 Boulot
-rw-r--r-- 1 famille famille   75974 déc 24 18:53 COPY INVOICE I10306044.pdf
-rw-r--r-- 1 famille famille   75989 déc 24 18:52 COPY INVOICE I10316756.pdf
drwxr-xr-x 4 famille famille    4096 mai  4 10:57 divers écrits
drwxr-xr-x 3 famille famille    4096 déc  9 21:56 doc international
drwxr-xr-x 2 famille famille    4096 déc  9 21:56 KENYA
[...]

$ fusermount -u sshfs_Documents

$ ls -l sshfs_Documents
total 0
```

Les commandes ont été séparées par un espace pour plus de lisibilité, nous avons fait dans l'ordre :

- Création d'un répertoire ~/sshfs_Documents/ qui servira de point de montage.
- Listing du contenu du nouveau répertoire, il est vide (“total 0”).
- Montage du répertoire distant ~/Documents sur “famille@192.168.1.2”, avec pour point de

montage local le répertoire “~/sshfs_Documents” créé auparavent.
- Listing du contenu du répertoire local ~/sshfs_Documents/, il contient maintenant la même chose que “famille@192.168.1.2:Documents” qui est monté par “sshfs”. On peut modifier les fichiers (en fonction de leurs permissions), en créer de nouveaux, les modifications sont appliquées sur le répertoire distant.
- Démontage avec la commande “**fusermount -u**”, qui prend en argument le point de montage.
- Listing du contenu du répertoire local ~/sshfs_Documents/, il est à nouveau vide (“total 0”).

Voilà un moyen simple et rapide de monter à distance un répertoire, sans ouvrir de ports supplémentaires, sans configurer “nfs” ou “samba”, et de manière sécurisée.

## Quelques astuces en vrac

Vous êtes maintenant à même d'utiliser “ssh”, au moins pour les bases, voici quelques “trucs” qui peuvent vous être utiles :

- En cas de problème de connexion, vérifiez d'abord les ouvertures de ports dans le(s) pare-feu, sur la machine locale ET distante, ainsi que la configuration du routeur (utilisation du “NAT” et nécessité de faire suivre le port de “ssh”).
- Si vous vous connectez à travers le réseau Internet, vous devez connaître l'adresse IP publique (externe) du poste distant. On peut connaître son IP publique en consultant http://whatismyip.org ou http://checkip.dyndns.com/ . Si l'adresse IP publique est attribuée dynamiquement par le fournisseur d'accès Internet, on peut créer un compte gratuit sur un service comme http://www.dyndns.org pour obtenir un nom de domaine fixe (à utiliser en conjonction avec un programme comme “ddclient”, voir documentation sur le site de dyndns.org).
- Utilisez le programme “autossh” pour vos tunnel, ça évite de voir un tunnel “mourir” en raison d'une mauvaise connexion ou d'une inactivité prolongée.
- Pour vous protéger des attaques “brute de force” contre votre démon sshd, surtout si vous utilisez le port par défaut 22, vous pouvez utiliser le programme “**sshguard**”. Celui-ci considère comme une attaque les tentatives de connexions répétées (comportement configurable) et bloque les ip des attaquant à l'aide de règles de pare-feu “iptables”. Dans de tels situations les paquets “**fail2ban**” ou “**denyhosts**” pourront être considérés également car ils sont plus souples et sophistiqués. Sur les noyaux récents le module “**recent**” de “netfilter” (le pare-feu intégré au noyau) est une excellente solution. (voir lien “protection attaques brute de force” ci-dessous).
- “ssh” fonctionne en cascade, vous pouvez vous loguer depuis un ordinateur “A” sur un ordinateur “B”, puis depuis l'ordinateur “B” vous loguer sur une machine “C”. De la même manière vous pouvez enchaîner des tunnels “ssh” en faisant correspondre les ports d'entrée/sortie des tunnels sur les machine successives.
- Sur Windows le logiciel “**putty**” peut agir comme un client ssh, avec une interface graphique.
- Certains gestionnaires de fichiers, comme Konqueror ou Midnight Commander, supportent le protocole “**fish**” basé sur ssh, tapez juste dans la barre d'url “fish://famille@192.168.1.2” (sans guillemets) et vous naviguerez dans l'arborescence de fichiers distante via ssh. Mais la ligne de commande est tellement plus simple…

- Si vous testez des configurations hasardeuses, qui nécessitent un redémarrage du démon “sshd” et risquent de vous faire perdre l'accès au serveur “ssh” en cas d'erreur, vous pouvez utiliser “webmin” comme “assurance”. “webmin” est une interface “web” de configuration de différents services, elle est accessible à distance avec un simple navigateur Internet, et couvre la configuration de ssh. Bien que “webmin” n'ait pas toujours une bonne réputation quant à sa sécurité, cela peut constituer une bonne “assurance” le temps d'être sûr de votre configuration “ssh”.

### Liens

- Site officiel de Openssh
- Sécuriser ssh, manuel Debian
- "Protection attaques brute de force" (“denyhosts”, “fail2ban”, “recent”)(eng)

Envoyer un message au mainteneur

From:
http://chanterie37.fr/fablab37110/ - **Castel'Lab le Fablab MJC de Château-Renault**

Permanent link:
**http://chanterie37.fr/fablab37110/doku.php?id=start:raspberry:ssh:doc&rev=1668269830**

Last update: **2023/01/27 16:08**