Le protocole SSH est surtout connu comme étant un moyen sécurisé d'obtenir un shell sur une machine distante. Cette vision n'est pas fausse, mais elle est en revanche terriblement limitée. Dans cet article, nous allons couvrir l'utilisation basique de SSH ainsi que quelques-unes de ses utilisations moins connues, comme le tunneling ou le montage de systèmes de fichiers distants. Nous verrons ainsi que SSH est bien plus qu'un simple remplaçant de telnet !

Historique

SH (pour Secure SHell) est un protocole réseau sécurisé développé à l'université de Helsinki. Son but premier était de fournir un remplaçant robuste aux vieillissants ~~telnet~~, ~~rlogin~~ et ~~rsh~~. ~~telnet~~ est un vieux protocole (1969 !) qui permet d'établir une connexion TCP sur un hôte pour y envoyer et recevoir des données.
Typiquement, il était utilisé pour se connecter à une machine distante et utiliser son propre ordinateur comme un terminal de cette machine.
Les caractères tapés au clavier étaient envoyés au serveur, et ce dernier renvoyait les caractères à afficher à l'écran de l'utilisateur. ~~rlogin~~ et ~~rsh~~ utilisent deux protocoles proches de celui de ~~telnet~~. Dans la même veine, ~~rcp~~ permettait de copier des fichiers d'une machine à une autre.
Ces programmes étaient certes très pratiques, mais transmettaient leurs données sur le réseau en clair.
Ce n'était pas un problème à une époque où les réseaux étaient fermés, mais dès lors qu'ils sont devenus accessibles au grand public (notamment dans les universités), un problème s'est vite posé : n'importe quelle personne ayant accès aux câbles pouvait écouter ce qui passait sur le réseau et récupérer ainsi fichiers, logins et mots de passe !
Quand on sait qu'il y a encore quelques années, les machines d'un réseau Ethernet étaient souvent reliées entre elles par un hub (équipement renvoyant les données émises par l'un des postes branchés dessus à tous les autres), on comprend la réalité de la menace. Et ne parlons même pas d'internet.
SSH offre un service relativement similaire à ~~telnet~~ et ses petits frères, mais l'encapsule dans une connexion sécurisée basée sur une cryptographie à clé publique.
Il existe deux versions du protocole SSH.
La version 1 est considérée comme obsolète et trop vulnérable, et ne devrait donc plus être utilisée. La version 2, la seule que nous considérerons ici, fait encore ses preuves de nos jours. La plupart des clients SSH supportent les deux protocoles.
Les exemples de cet article sont prévus pour Openssh, l'implémentation libre du client SSH que l'on devrait retrouver sur toutes les distributions Linux.

Utilisation de base

Voici la configuration réseau que nous allons assumer pour les exemples de cet article : ~~lancelot~~ est la machine client sur laquelle nous travaillons. Nous voulons nous connecter à ~~percival~~, qui dispose donc d'un serveur SSH (Figure 1).

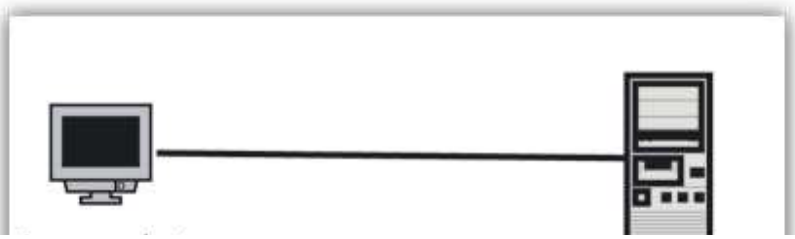


Fig. 1 : Configuration réseau de lancelot et percival.

Utiliser SSH comme telnet

L'exemple le plus simple d'utilisation de SSH est de se connecter directement à un serveur afin d'obtenir un shell :

```
moi@lancelot:~$ ssh percival
Password:
moi@percival:~$
```

Ici, nous avons simplement ouvert une session pour l'utilisateur moi (qui lance la connexion) sur percival en donnant son mot de passe sur cette machine. Un autre utilisateur peut être précisé en utilisant la syntaxe suivante :

```
moi@lancelot:~$ ssh lui@percival
Password:
lui@percival:~$
```

Openssh supporte de nombreuses options. Voici quelques-unes des plus fréquentes :

- -X permet de rediriger l'affichage des programmes Xwindow vers le serveur X du client. En d'autres termes, si vous lancez un programme X tel que xeyes sur le serveur, celui-ci s'exécutera sur ce dernier mais s'affichera sur l'écran du client.
- -C permet de compresser les données transférées. Cette option est souvent utilisée conjointement à -X pour soulager les lignes à bas débit.
- -v, -vv et -vvv seront utiles si le serveur SSH vous refuse l'accès alors que vous pensez qu'il ne devrait pas. Ces options rendent votre client plus bavard sur ce qu'il fait.

Copie de fichiers distants

SSH offre également un remplaçant à rcp. scp permet de copier un ou plusieurs fichiers en utilisant une syntaxe similaire à celle de cp :

```
$ scp monfichier.txt lui@percival:~/
```

Ici, monfichier.txt sera copié dans le répertoire personnel de l'utilisateur lui de la machine percival, en s'identifiant comme ce dernier. Le caractère : permet de séparer le nom d'hôte du chemin vers le fichier distant.
scp supporte aussi beaucoup d'options. On notera les suivantes :

- -p préserve les attributs des fichiers copiés (date de modification, droits, etc.).
- -r copie récursivement les répertoires donnés.
- -C, comme son homologue de SSH, compresse les données durant le transfert.

A un plus haut niveau, existe également le protocole SFTP. SFTP n'est pas une version de FTP sur SSH, mais au contraire un nouveau protocole. Toutefois, les commandes supportées par le client sont très semblables à celle d'un client FTP :

```
$ sftp lui@percival
Connecting to percival...
Password:
sftp> ls
Desktop
Bin
sftp> put monfichier.txt
Uploading monfichier.txt to /home/lui/monfichier.txt
monfichier.txt                                100%   12KB  12.1KB/s   00:00
sftp> quit
```

~~scp~~ et ~~sftp~~ disposent de nombreux clients graphiques, comme ~~lftp~~ ou ~~krusader~~. Nous verrons également à la fin de l'article que ce protocole est bien intégré aux environnements de bureau modernes.
Ces quelques exemples d'utilisation constituent les bases de SSH. A première vue, SSH n'est qu'un équivalent sécurisé à ~~telnet~~ ou ~~rcp~~... En fait, il se comporte juste comme tel.

Utilisation avancée

Nous savons maintenant utiliser SSH comme on utiliserait ~~telnet~~ ou ~~rcp~~. C'est fort bien, mais ce serait se limiter grandement que d'en rester là : SSH peut grandement faciliter la vie de celui qui comprend comment il fonctionne.

Clés SSH et méthodes d'authentification

Bien que l'utilisation de SSH du point de vue de l'utilisateur semble aussi simple que celle de ~~telnet~~, une importante négociation se produit entre client et serveur quand vous vous connectez.
Celle-ci a deux buts : s'assurer de l'identité de la personne qui se connecte, et établir une connexion chiffrée entre client et serveur. Observons ce qui se passe en détail...
La cryptographie de SSH est dite « à clé publique ». S'il y a clé publique, il y a forcément... clé privée. Un couple clé publique/clé privée a la propriété suivante : tout ce qui est chiffré par la clé publique ne peut être déchiffré que par la clé privée, et vice-versa.
Il est également impossible de dériver l'une à partir de l'autre. Comme son nom l'indique, la clé publique est destinée à être diffusée, tandis que la clé privée ne doit être connue que de son propriétaire.
Ce genre de cryptographie a de nombreuses autres applications, comme la signature ou le chiffrage des emails. En fait, l'authentification de SSH ne fonctionne pas différemment de ces applications.
Pour pouvoir se connecter à un serveur SSH, il faut donc un tel couple de clés. Celles-ci sont personnelles à l'utilisateur. Il existe également une paire de clés pour l'hôte, qui se trouvent généralement dans ~~/etc/ssh~~.
L'intérêt de disposer de sa propre clé réside dans le fait qu'il est alors possible de s'affranchir de toute authentification manuelle, comme nous allons le voir.
Les clés d'un utilisateur se trouvent dans ~~~/.ssh/~~, et sont contenues dans les fichiers ~~id_algo[.pub]~~, où ~~algo~~ désigne l'algorithme de cryptage employé. La clé privée n'a pas d'extension, la clé publique a l'extension ~~.pub~~.
Pour générer ces fameuses clés, nous pouvons utiliser le programme ssh-keygen :

```
$ ssh-keygen -t dsa -b 4096
```

Cette commande suffira pour générer une paire de clés DSA d'une taille de 4096 bits. L'algorithme RSA peut également être utilisé, mais le DSA reste en général préféré.

La génération prend un certain temps, après lequel il est demandé dans quel fichier sauvegarder la clé privée (laissez les valeurs par défaut), ainsi qu'une phrase de passe (passphrase). C'est cette phrase qui permettra d'utiliser la clé.
Elle constitue une sécurité supplémentaire, car dans le cas où votre clé privée se fait

subtiliser, l'auteur du forfait ne pourra rien en tirer s'il ne connaît pas la phrase de passe. Choisissez-la bien. Vous pouvez également décider de vous en passer... à vos risques et périls.
Une fois l'exécution de ssh-keygen terminée, vous vous retrouvez donc avec deux fichiers ~~id_dsa~~ et ~~id_dsa.pub~~ dans le répertoire ~~~/.ssh~~. Comment en tirer parti ? Pour commencer, nous allons nous authentifier sur le serveur non plus avec notre mot de passe utilisateur, mais avec notre clé SSH.
Pour cela, nous allons nous connecter sur le serveur et créer ou éditer le fichier ~~~/.ssh/authorized_keys~~ pour y ajouter notre clé publique (par copier/coller de l'intégralité du fichier ou en utilisant ~~scp~~).
Ce fichier est censé contenir une clé publique par ligne, et indique que ces clés peuvent être utilisées pour authentifier l'utilisateur qui souhaite se connecter.
Attention, comme les clés sont longues, à ne pas insérer de retours à la ligne lors du copier/coller !
Maintenant, comment activer l'authentification par clé lors de la connexion ? Eh bien, il n'y a strictement rien à faire. SSH essaie successivement différentes méthodes d'authentification pour s'assurer de l'identité d'un utilisateur, et la méthode consistant à demander son mot de passe sur le système arrive en dernier.
L'authentification par clé est l'étape précédente. Il existe encore une autre méthode d'authentification par hôte, venant avant l'authentification par clé. Elle est souvent désactivée par défaut et nous ne la couvrirons donc pas ici.
Réessayons de nous connecter, en passant l'option ~~-v~~ à notre client pour voir ce qui se passe :

```
$ ssh -v percival
...
debug1: Authentications that can continue: publickey,keyboard-interactive
debug1: Next authentication method: publickey
...
debug1: Trying private key: /home/moi/.ssh/id_rsa
debug1: Offering public key: /home/moi/.ssh/id_dsa
debug1: Server accepts key: pkalg ssh-dss blen 818
debug1: PEM_read_PrivateKey failed
debug1: read PEM private key done: type <unknown>
Enter passphrase for key '/home/moi/.ssh/id_dsa':
```

Oh ! On nous demande cette fois le mot de passe de la clé privée. Une fois tapé, l'accès à ~~percival~~ est donné.
Comment fonctionne l'authentification par clé ? Le serveur SSH, par le biais du fichier ~~authorized_keys~~, dispose d'une liste de clés publiques qu'il peut utiliser pour authentifier l'utilisateur.
Pour ce faire, il va envoyer au client un « challenge », c'est-à-dire un court message chiffré avec la clé publique. Si le client est capable de le déchiffrer et de répondre au challenge, c'est qu'il dispose de la clé privée, et donc qu'il est autorisé à se connecter.
Tout cela est bien joli, mais il faut tout de même taper la phrase de passe de sa clé pour accéder à notre hôte distant... Pas forcément, car on peut utiliser ssh-agent pour autoriser l'utilisation de notre clé privée une fois pour toute.

Le gardien des clés

ssh-agent est un petit programme qui gardera nos clés privées ouvertes pour les utiliser à chaque fois que nécessaire, sans redemander leur mot de passe. Si vous le lancez en ligne de commande, vous ne manquerez pas d'être perplexe par son comportement :

```
$ ssh-agent
SSH_AUTH_SOCK=/tmp/ssh-sciaTo7933/agent.7933; export SSH_AUTH_SOCK;
SSH_AGENT_PID=7934; export SSH_AGENT_PID;
echo Agent pid 7934;
$
```

C'est déjà fini ? Oui, et en plus, ça n'a servi à rien ! :)
ssh-agent est en fait un programme qui est prévu pour être lancé en début de session, avec comme paramètre le programme initial de la session. Si vous travaillez en mode

avec comme paramètre le programme initial de la session. Si vous travaillez en mode console, ce pourra être bash, si vous êtes sous X, ce sera sans doute votre gestionnaire de fenêtres.
Comme sa sortie l'indique, ssh-agent définit quelques variables d'environnement, puis lance le programme passé en paramètre. Si ce programme est celui qui va gérer la session, tous les programmes lancés par la suite seront des fils de ssh-agent et auront donc accès à ces variables d'environnement qui leur permettront de communiquer avec lui.
De nombreuses distributions lancent le gestionnaire de fenêtre avec ssh-agent. Si vous êtes logué via l'interface graphique, il y a de grandes chances qu'il soit déjà lancé. Pour vérifier :

```
$ ps x |grep ssh-agent
 7313 ?        Ss     0:00 /usr/bin/ssh-agent /usr/bin/dbus-launch --exit-with-session x-session-ma
```

Ce résultat variera selon ce que vous utilisez, mais si vous voyez ssh-agent apparaître, c'est qu'il est déjà lancé. Si vous ne le voyez pas et que vous voulez l'essayer rapidement, vous pouvez lancer ssh-agent bash dans une console par exemple. Tous les programmes que vous lancerez à partir de cette console auront alors accès à votre agent.
Une fois ssh-agent lancé, il suffit de lui indiquer que nous voulons garder notre clé privée ouverte :

```
$ ssh-add
Enter passphrase for /home/moi/.ssh/id_dsa:
```

Entrez votre phrase de passe, et voilà ! Vous pouvez dorénavant vous connecter sur ~~percival~~ sans taper votre mot de passe utilisateur ou la phrase de passe de votre clé. Lors de la phase d'authentification par clé, le client SSH vérifie si ssh-agent est lancé et si oui, lui demande l'accès à la clé privée. Comme nous l'avons ouverte avec ~~ssh-add~~, elle est disponible sans plus de formalités. ~~ssh-add~~ peut prendre en paramètre le fichier de la clé privée à ajouter. S'il est absent, votre clé par défaut sera ouverte. Il reconnaît également les options suivantes :

- ~~-D~~ ferme toutes les clés ouvertes ;
- ~~-d~~ ferme la clé dont le fichier est donné en paramètre ;
- ~~-t~~ durée ne garde les clés ouvertes que pendant une durée limitée (en secondes).

Par ailleurs, ssh est également capable de rediriger les requêtes d'authentification par agent. Si vous créez une chaîne de connexions ssh pour atteindre, par exemple, un hôte se trouvant dans un réseau privé, vous pouvez demander à ce que les requêtes d'authentification par agent soient redirigées jusqu'à votre connexion d'origine. Cela vous permet d'éviter d'avoir à relancer l'agent sur chaque machine de la chaîne. Pour cela, rajoutez l'option -A à vos lignes de commande ssh.
Un bon contrôle de vos clés SSHvous garantit non seulement sécurité et confidentialité, mais aussi confort d'utilisation.

SSH et les tubes

Une grande partie de la puissance du shell réside dans l'utilisation des tubes, permettant de relier la sortie d'une commande à l'entrée d'une autre. Le fait de passer par SSH ne rompt pas cette chaîne, et il est ainsi possible de brancher la sortie de commandes locales sur des commandes distantes, et vice-versa. Tout d'abord, il est possible d'utiliser SSH pour exécuter une simple commande sur la machine distante et récupérer sa sortie. Il suffit de passer la commande à exécuter après le nom d'hôte (de préférence entre guillemets pour la protéger) :

```
moi@lancelot$ ssh lui@percival «ls»
Password:
Desktop bin monfichier.txt
moi@lancelot$
```

Notez que la connexion est coupée dès que la commande distante est exécutée.
Voyons maintenant comment nous pouvons tirer parti de cette forme d'appel à SSH.
Un problème récurrent pour bon nombre d'entre nous consiste à faire des sauvegardes de ses données les plus importantes. Par sécurité, il est pertinent de stocker ces données sauvegardées sur une autre machine. En combinant bash et SSH, nous allons rendre cette opération possible en une seule ligne de commande qui pourra facilement être définie comme tâche périodique à l'aide de cron.
Lorsque l'on demande à SSH d'exécuter une simple commande distante, cette commande peut être branchée avec une commande locale. SSH redirige en effet son entrée standard vers l'entrée standard de la commande distante, et la sortie standard de la commande distante vers la sienne.
Voici la ligne permettant de faire une sauvegarde du répertoire ~~Documents~~ sur ~~percival~~ :

```
$ tar czf - Documents |ssh percival «cat >sauvegarde.tar.gz»
```

Le paramètre - donné à ~~tar~~ lui demande de sortir le flux compressé sur la sortie standard. Ce flux est récupéré par SSH qui le transfère sur ~~percival~~ et l'utilise comme entrée standard de ~~cat.~~ Tel qu'il est appelé, ~~cat~~ ne fait que copier son entrée standard vers le fichier ~~sauvegarde.tar.gz~~. Ainsi, nous avons réalisé une sauvegarde distante et compressée en une seule ligne.
On peut également, si la bande passante n'est pas une ressource critique, réaliser la compression côté serveur :

```
$ tar cf - Documents |ssh percival «gzip -c >sauvegarde.tar.gz»
```

La restauration de cette sauvegarde pourra se faire de la manière suivante :

```
$ ssh percival «cat sauvegarde.tar.gz» |tar xzf
```

On peut donc faire passer les sorties standards de commandes à travers une connexion SSH. Ce serait dommage de s'arrêter là – on peut en effet y faire passer bien d'autres choses !

Tunnels SSH

Pour ce scénario, nous allons compléter notre schéma réseau : ~~lancelot~~ et ~~percival~~ ne sont pas dans le même réseau local, mais appartiennent à deux réseaux différents connectés par internet. ~~percival~~ est derrière un pare-feu qui filtre tous les ports à l'exception du port 22 (le port utilisé par SSH). Derrière ~~percival~~ se trouve ~~arthur~~, qui est un serveur POP3 (email). A partir de ~~lancelot~~, nous voudrions bien récupérer nos mails qui se trouvent sur ~~arthur~~. Il y a plusieurs problèmes ici. Premièrement, ~~arthur~~ n'est pas directement accessible depuis ~~lancelot~~. Seul ~~percival~~ a une adresse internet publique, arthur n'étant pas visible de l'extérieur du réseau. Deuxièmement, le pare-feu filtre tous les ports à l'exception du port 22. Or, le serveur POP3 écoute sur le port 110... SSH va nous permettre de nous affranchir de ces contraintes.
L'option -L de SSH permet de créer un tunnel. Sous ce nom se cache le fait d'employer un port de la machine locale pour transporter des données à travers la connexion SSH et les rediriger où l'on veut à partir de la machine distante. Le tunnel est également utilisable en sens inverse. Cette option est très puissante. Dans notre cas, nous voulons nous servir de ~~percival~~ comme intermédiaire pour nous connecter à ~~arthur~~, comme le montre la figure 2. Nous allons ouvrir notre tunnel avec la commande suivante:

```
$ ssh -L 2500:arthur:110 percival
```

Fig. 2 : Utilisation d'un tunnel SSH pour atteindre une machine inaccessible à partir du réseau visible.
Du point de vue d'arthur, ce sera percival qui se connecte, pas lancelot !

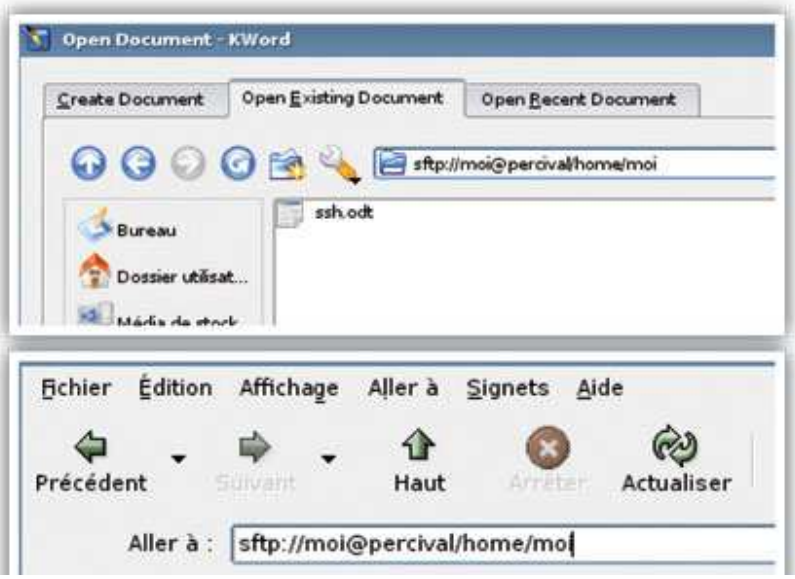
Nous demandons à SSH de nous connecter à ~~percival~~. Jusque-là tout est normal. Mais en plus, l'option ~~-L~~ demande à rediriger le port local ~~2500~~ vers le port ~~110~~ de l'hôte ~~arthur~~. Celui-ci est spécifié du point de vue de ~~percival~~ – rappelons ~~qu'arthur~~ n'est même pas visible pour lancelot. Le tunnel se fermera avec la session ~~ssh~~. Du côté du client mail, il suffira de d'indiquer ~~localhost~~ comme serveur de mail, avec le port ~~2500~~. Et voilà ! Tant que la connexion SSH restera ouverte, le fait de nous connecter sur le port ~~2500~~ de ~~lancelot~~ équivaudra à se connecter sur le port ~~110~~ de ~~arthur~~, et nous pourrons récupérer nos mails à partir de ce dernier, qui transiteront cryptés dans le tunnel SSH.

Intégration et transparence

Maintenant que nous connaissons les principes de SSH et que nous maîtrisons l'outil en ligne de commande, nous allons voir quelques exemples d'intégration dans des applications modernes qui rendent son utilisation plus confortable.

Intégration à KDE et Gnome

Les deux principaux environnements de bureau libres offrent la possibilité de passer par un serveur SSH pour accéder à des fichiers distants. KDE et Gnome proposent une couche d'accès à différents systèmes de fichiers (les ~~KIO~~ pour KDE, ~~gnome-vfs~~ pour Gnome) qui gèrent (entre autres) le protocole SFTP. Cette fonctionnalité très puissante mériterait un article à elle seule. Pour accéder à vos fichiers distants, il suffit de préciser un emplacement sous une forme similaire à celle utilisée pour ~~scp~~, préfixé de ~~sftp://~~. Cette syntaxe peut être utilisée dans n'importe quelle boîte d'ouverture/d'enregistrement de fichiers, ou tout simplement dans la barre d'URL de Konqueror ou Nautilus (figure 3). Par exemple : ~~sftp://moi@percival/home/moi/~~ vous placera dans le répertoire ~~/home/moi de percival~~, en vous authentifiant en tant que moi.



Si un mot de passe est nécessaire (celui du compte utilisateur distant ou de votre clé SSH), il vous sera demandé au travers d'une boîte de dialogue. KDE propose également un protocole supplémentaire : ~~fish://~~. Celui-ci n'utilise pas le protocole SFTP, mais passe par une connexion SSH classique assistée d'un script perl. Il peut se révéler utile pour vous connecter à un serveur qui a désactivé le protocole SFTP.

SSHFS

Les intégrations que nous venons de couvrir sont très pratiques et puissantes, mais ne constituent néanmoins pas une solution universelle : elles sont limitées aux applications KDE ou Gnome. Et si je veux ouvrir un fichier distant avec Emacs ? Ou regarder un film avec Mplayer sans avoir à le copier préalablement ? Un autre désavantage de ces couches d'accès est qu'elles traitent les fichiers distants comme des flux et ne supportent pas (du moins à l'heure actuelle) le déplacement. Impossible par exemple de se déplacer dans un fichier son en cours de lecture...
La solution à tous ces problèmes consiste à placer l'accès distant à un niveau plus bas que le middleware applicatif : au niveau du noyau. SSHFS est un système de fichiers utilisant FUSE, le module noyau permettant de développer des systèmes de fichiers au niveau utilisateur.
Beaucoup de systèmes de fichiers existent pour FUSE, comme le fameux ~~GmailFS~~, ou ce module permettant de monter une KIO... FUSE et SSHFS sont disponibles dans certaines distributions (Ubuntu Breezy notamment). Si la vôtre ne les fournit pas, vous pouvez vous reporter sur la page du projet pour la procédure d'installation.
Attention ! Il faut en général être dans le groupe ~~fuse~~ pour pouvoir monter un système de fichiers avec ce dernier. Une fois installé, vous pouvez invoquer SSHFS comme vous invoqueriez mount :

```
$ mkdir remote
$ sshfs moi@percival:/home/moi remote
```

Cette commande montera le répertoire ~~/home/moi~~ sur la machine ~~percival~~ dans le répertoire ~~remote~~. L'authentification SSH se fera pour l'utilisateur ~~moi~~. Pour démonter le système de fichiers :

```
$ fusermount -u percival
```

SSHFS vous offre ainsi un accès distant totalement transparent et sécurisé et un moyen simple d'accéder à vos données de n'importe où !

Vous pouvez par exemple laisser votre collection musicale, vos albums photo ou vos documents de travail sur votre serveur personnel et les avoir toujours à disposition, pourvu que vous disposiez d'une connexion internet.

Conclusion

Nous avons rapidement découvert l'utilisation de base de SSH, ses méthodes d'authentification ainsi que le tunneling. L'utilisation de ce protocole comme moyen d'accès à des fichiers distants, au travers de KDE, Gnome ou FUSE, reste cependant l'utilisation la plus pratique (et aussi la plus méconnue) de ce protocole.

Particulièrement adapté à l'utilisation contemporaine des réseaux, SSH permet

Particulièrement adapté à l'utilisation contemporaine des réseaux, SSH permet d'accéder à ses données personnelles de manière simple et sécurisée à partir de n'importe quelle connexion réseau.

Liens:

- Openssh : http://www.openssh.org/
- FUSE : http://fuse.sourceforge.net/

Port Forwarding iptables et authentification d'hôte par SSH - par Denis Bodor

A chaque première connexion SSH à un hôte distant, ssh, sur le poste client, demande à l'utilisateur de confirmer l'identité de la machine en fournissant l'empreinte de sa clef RSA.
Dans la plupart des cas, avouez-le, vous répondez simplement yes sans vérifier l'empreinte. C'est une très mauvaise chose puisque, si un attaquant est déjà en lice, vous pourriez littéralement accepter une attaque Man-In-the-Middle et toutes celles qui suivront. L'acceptation entraîne l'enregistrement de la clef publique dans un fichier ~/.ssh/known_hosts.
Celle-ci sera vérifiée à chaque connexion par la suite. Encore récemment, les entrées dans ce fichier débutaient par l'adresse IP ou le nom de la machine distante.
A présent, cette information est chiffrée. Un problème survient lorsque, pour une même adresse IP, nous avons plusieurs hôtes serveur SSH à contacter. Le cas typique est celui d'une passerelle/NAT faisant fonctionner un serveur SSH sur le port 22 mais faisant suivre son port 2222 sur le port 22 d'une machine du LAN.
Ainsi, en utilisant l'option -p de ssh on accèdera soit à la passerelle, soit à la machine du LAN. Cependant, dans ~/.ssh/known_hosts, une seule clef sera stockée et un message d'alerte sera affiché si la vérification échoue.
Il existe une solution permettant de régler le problème et de différencier les deux hôtes. Il suffit pour cela de créer des profiles dans votre ~/.ssh/config :

```
Host gateway
        Hostname truc.dom.fr
        CheckHostIP no
        Port 22
        HostKeyAlias gateway

Host mamachine
        Hostname truc.dom.fr
        CheckHostIP no
        Port 2222
        HostKeyAlias mamachine
```

On se connectera alors en utilisant ssh gateway ou ssh mamachine. C'est HostKeyAlias qui permet de créer des entrées différentes dans known_hosts. Notez également la directive CheckHostIP annulant la vérification IP pour une passerelle ADSL se voyant attribuer une adresse dynamiquement.