SSH – Authentification par clés

Génération des clés

Les clés seront générées et sauvegardées automatiquement dans le répertoire *$HOME/.ssh* de l’utilisateur courant.

Une passphrase (un mot de passe) sera demandée pendant la procédure de

génération : elle permet de protéger la clé privée au cas où un tiers s'en emparerait. La clé privée protégée ne pourra pas être utilisée sans la passphrase que vous avez renseignée.

Si vous ne souhaitez pas que la clé soit protégée, vous pouvez laisser le champ vide et valider.

Notez qu'on peut ajouter un commentaire, dans le fichier de la clé publique, avec la commande [redacted].

Vous obtenez donc 2 fichiers :

- « *id_rsa* » , la clé privée
- « *id_rsa.pub* » , la clé publique

Téléchargez la clé privée sur votre ordinateur local (le client), est sauvegardez-là en lieu sûr : elle ne doit en aucun cas être volée par un tiers, puisque c'est elle qui autorise les

connexions.

Une fois téléchargée, vous devez supprimer la clé privée du serveur :

N.B. : ici nous avons généré les clés sur un serveur distant, il aurait été plus convenable de les générer sur le poste client (on évite ainsi le transfert de la clé privée).

Conversion de la clé privée au format PuTTY

Pour pouvoir vous connecter avec PuTTY ou WinSCP, la clé privée doit être au format PuTTY, voici donc la procédure à suivre :

·téléchargez PuTTYgen (site

officiel),

- exécutez PuTTYgen et allez dans « Conversions » puis « Import key »,
- ensuite sélectionnez la clé privée « *id_rsa* » pour la charger dans le programme,
- cliquez sur le bouton « Save private key » et choisissez un fichier de destination pour sauvegarder la clé au format PuTTY « *.ppk*« .

Autoriser la clé publique

Nous devons autoriser les connexions par jeu de clés sur le serveur, pour cela il suffit d’ajouter la clé publique au fichier « *authorized_keys* » (ou « *authorized_keys2* » mais ce nom de

fichier est déprécié, même si les deux fonctionnent normalement) de l’utilisateur avec lequel on va se connecter.

Veuillez noter que le serveur peut gérer plusieurs clés publiques : il suffit d’ajouter une clé par ligne dans le fichier « *authorized_keys*« .

Activer l’authentification par clés

Vous devez ajouter/modifier les directives suivantes dans le fichier de configuration d’OpenSSH (*/etc/ssh/sshd_config*) :

Rechargez la configuration du serveur OpenSSH :

Testez la connexion en utilisant la clé privée avec PuTTY ou WinSCP.

Désactiver les mots de passe

Vous pouvez désactiver la connexion par mot de passe si le test a été concluant.

Vous devez ajouter/modifier les directives suivantes dans le fichier de configuration d’OpenSSH (*/etc/ssh/sshd_config*) :

Rechargez la configuration du serveur OpenSSH :

Limiter les connexions

Vous pouvez spécifier quels utilisateurs sont autorisés à se connecter via le serveur SSH, il suffit pour cela d’ajouter leur nom à la directive « *AllowUsers* » en les séparant par un espace, toujours dans le fichier de configuration d’OpenSSH (*/etc/ssh/sshd_config*) :

Rechargez la configuration du serveur OpenSSH :

Connexion

Pour se connecter avec le client SSH dans un terminal :

La spécification du port avec le paramètre [redacted] est facultative, par défaut le port 22 est utilisé.

On peut forcer l’utilisation d’une clé privée avec le paramètre [redacted], on peut l’omettre si les clés sont bien présentes dans le dossier [redacted] (le client tentera alors la connexion avec chaque clé privée trouvée), ou dans le cas où un fichier de configuration [redacted] existe et est correctement renseigné.

A des fins de debugging, pour savoir quelles clés sont proposées au serveur, on peut lancer une connexion en mode verbeux et chercher les lignes commençant par « Offering… »

Configuration de

connexion client

Le fichier [redacted] permet de s’affranchir du passage de plusieurs paramètres au client SSH. Voici le format utilisé :

On utilisera le client de la manière suivante :

Notes

Les utilisateurs créés avec la commande « *useradd* » doivent avoir un mot de passe valide, dans le cas contraire, les utilisateurs seront bloqués et ne pourront pas se connecter. On peut vérifier la présence d’un mot de passe dans le fichier « */etc/shadow* » : celui-ci contient un « *!* » après le nom d’un utilisateur

bloqué.

Voir aussi : https://wiki.debian.org/fr/SSH.

Mémo MySQL

Connexions persistantes

Les connexions persistantes permettent de ré-utiliser une connexion déjà ouverte par PHP. Elles ne se terminent pas à la fin de l’exécution d’un script. L’intérêt est

de limiter le temps de latence dû à l'ouverture de la connexion, ce qui peut-être utile lorsque le serveur de base de données et HTTP ne se situent pas sur le même serveur physique.

Dans le cas d'Apache, une connexion par processus fils est ouverte, il faut donc en tenir compte lorsqu'on fixe la valeur « max_connections » du serveur MySQL. En effet, deux requêtes consécutives peuvent être gérées par des processus Apache différents, ce qui engendre l'ouverture d'une connexion persistante pour chaque processus ! La charge de la machine, en terme de mémoire utilisée notamment, va augmenter de façon proportionnelle aux nombres de connexions simultanées.

Attention, une transaction qui n'est pas terminée, restera active entre les requêtes Apache ! De même, quand vous bloquez (lock) une table, normalement elle est débloquée lorsque la connexion est fermée, mais comme les connexions persistantes ne se ferment pas (sic), les tables que vous avez quittées en état

bloquées resteront bloquées, et la seule façon de les déloquer sera d’attendre que la connexion atteigne le timeout (« wait_timeout ») ou de tuer le processus MySQL. Pour finir, les options de connexions restent actives entre les appels, par exemple :

Dans MySQL, les tables temporaires sont visibles uniquement par la connexion courante, mais si vous avez une connexion persistante, la table temporaire sera visible par tous les scripts qui partagent la même connexion persistante.

Comet

Depuis quelques semaines, je cherche un moyen permettant à mes sites de proposer des intéractions en temps réel avec les utilisateurs. Il y a bien sûr toutes les techniques liées à AJAX, mais elles sont gourmandes en nombres de requêtes lorsqu'on veut, par exemple, gérer un chat. D'ailleurs en ce qui concerne les chats, on se trouve rapidement confronté au problème de latence du réseau, ce qui fait que dans la plupart des applications testées, les messages arrivent avec un certains délais. On pourrait utiliser Flash pour ce genre d'application, mais il faut un plugin et généralement les sites basés sur cette technologie sont lourds.

En cherchant sur le web, on trouve un autre ensemble de « méthodes » permettant de gérer des communications bi-directionnelles (serveur-client et client-serveur) : il s'agit de Comet.

J'ai donc rapidement étudié cette

technique (vous trouverez quelques infos sur Comet Daily), ce qui m’a amené à installer le serveur Meteor. Le choix de ce serveur n’est pas un hasard : cette page est un bref comparatif des solutions actuelles et j’en ai déduis que Meteor était le plus approprié pour mon utilisation.

Après quelques tests, je me suis rendu compte de ses limitations (ainsi que du support quasi-nul) et j’ai donc passé mon chemin. J’ai trouvé une autre solution, appelée APE, pour Ajax Push Engine. Il s’agit d’un serveur codé en C, récent et qui supporte des extensions en JavaScript du côté serveur. Un framework client est mis à disposition et permet de communiquer avec le serveur assez facilement.

Des performances flatteuses (il n’y a pas de benchmark « officiel », mais c’est ce qui ressort dans les différents articles que j’ai vu sur le web), peu de limitations et beaucoup d’avantages :

- modularité du serveur,

- la possibilité de créer des applications cross-domains,
- le choix du mode de transport (JSONP, XHRStreaming, XLHttpRequest etc…),
- un développement actif

Prochain billet : l’installation du serveur APE.