Haute École de Bruxelles-Brabant
École Supérieure d'Informatique
Bachelor en Informatique

2018 – 2019
DEV1

DEV1 – Laboratoires d'environnement

TD 08 – Secure Shell

Objectifs

À la fin de ce module, vous pourrez différencier les étapes et systèmes intervenants dans une connexion *ssh*, utiliser et configurer la commande *ssh* et différentes options dans différents contextes et décrire les algorithmes sous-jacents.

Secure SHell (*ssh*) permet d'obtenir un ***shell* sécurisé** à distance.

- ▷ un ***shell*** — comme *bash* — est un interprèteur de commandes. C'est lui l'interface entre l'utilisateur et le système d'exploitation. Il est en mode texte.
- ▷ il est **sécurisé** parce que les messages échangés sont chiffrés.

Lorsque la communication est chiffrée, un utilisateur malveillant ne pourra pas écouter cette communication. Plus précisément, il pourra peut-être l'écouter mais ne la comprendra pas.

ssh fonctionne en « client-serveur » :

- ▷ un programme sur l'une des machines est le **client** ;
  Ce *client* se connecte au serveur. Il ouvre un canal de communication vers une certaine adresse et envoie un message de demande de connexion. Il envoie également des identifiants de connexion.
- ▷ un programme sur l'autre machine est le **serveur** ;
  Ce *serveur* écoute continuellement les demandes de connexion. Dès qu'un message lui parvient, le *serveur* échange des données avec le *client*.
  Ce programme exécuté en tâche de fond, c'est un *dæmon*, écoute (généralement) les demandes de connexion. Dès que le client s'authentifie correctement, le programme fournit l'environnement — un *shell* — prévu.

ssh offre les fonctionnalités suivantes :

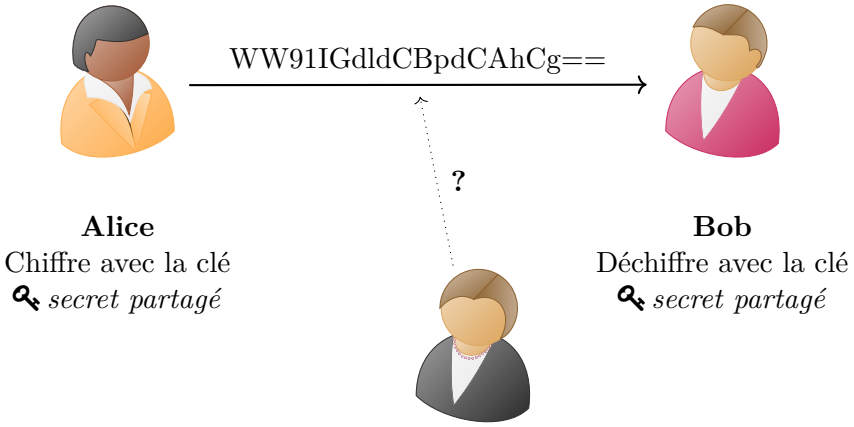
- ▷ l'authentification du serveur ;
- ▷ la confidentialité des données échangées (ou session chiffrée) ;
- ▷ l'intégrité des données échangées ;
- ▷ l'authentification du client.

Version du document
21 novembre 2018

1 Vocabulaire

1.1 Chiffrement symétrique

Le **chiffrement symétrique**, appelé aussi chiffrement par échange d'une clé ou par clé partagée ou encore par clé secrète, est un système qui consiste au chiffrement des messages échangés, avec une clé de chiffrement commune et partagée « à l'avance » par les intervenants.



Voici quelques noms d'algorithmes de chiffrements symétriques [1] : 3des-cbc blowfish-cbc cast128-cbc arcfour arcfour128 arcfour256 aes128-cbc aes192-cbc aes256-cbc rijndael-cbc@lysator.liu.se aes128-ctr aes192-ctr aes256-ctr aes128-gcm@openssh.com aes256-gcm@openssh.com chacha20-poly1305@openssh.com

Exercice 1 **Chiffrement, déchiffrement**

La commande 'openssl' permet d'encoder (chiffrer) et déchiffrer un texte. Voyons ça :

- ▷ créez et éditez un fichier nommé 'mysecrets' contenant quelques phrases ;
- ▷ chiffrez ce fichier (vous devrez entrer un mot de passe) ;

```
$ openssl enc -aes-256-cbc -out mysecrets.ciphered
    in mysecrets
```
terminal

- ▷ observez le contenu du fichier chiffré ;

1. Cette liste est accessible par la commande `ssh -Q cipher`

▷ déchiffrez-le et affichez le contenu déchiffré sur la sortie standard ;

```
$ openssl enc -aes-256-cbc -in mysecrets.ciphered -d
```

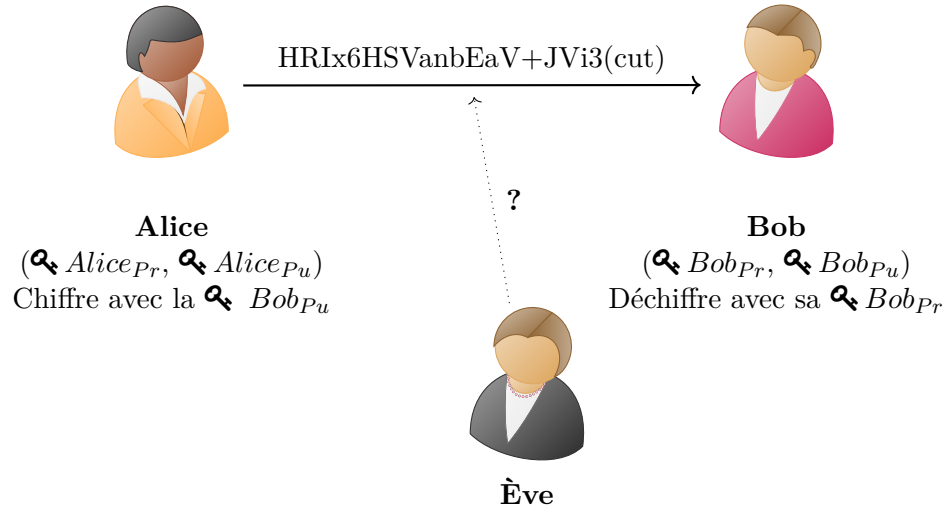
terminal

Fin Exercice 1

1.2 Chiffrement asymétrique

Le **chiffrement asymétrique** appelé aussi chiffrement à clé publique est un système qui consiste au chiffrement des messages échangés, à l'aide de plusieurs clés.

Supposons qu'Alice veuille envoyer un message à Bob de manière confidentielle et qu'ils ne détiennent pas de clé secrète partagée. Ils ne peuvent pas faire transiter la (future) clé secrète sur le canal non sûr par peur que cette clé ne soit interceptée. Le chiffrement asymétrique peut venir à leur secours.



Alice et Bob possèdent chacun deux clés — une clé publique (Pu) et une clé privée (Pr) — telles que :

- ▷ il est facile de chiffrer un message avec la clé publique ;
- ▷ il est impossible en un temps raisonnable de déchiffrer le message avec la clé publique ;
- ▷ il est facile de déchiffrer le message avec la clé privée.

Dans ces conditions, il suffit de publier sa clé publique et de conserver précieusement sa clé privée. Toute personne voulant m'envoyer un message le chiffrera avec ma clé publique… et je le déchiffrerai avec ma clé privée.

Voici quelques noms d'algorithmes de chiffrements asymétriques : RSA, DSA, ECDSA.

Remarque

Il est possible de générer une clé RSA|DSA avec `ssh-keygen` et `openssl`. Les deux programmes génèreront une clé privée de format et de taille voulue.

- `ssh-keygen` génèrera directement la clé publique au format *OpenSSH*
- `openssl` ne génèrera pas de clé publique automatiquement… mais il est possible de le lui demander. Dans ce cas, cette clé sera au format PEM.

Exercice 2 — Chiffrement, déchiffrement asymétrique

La commande `openssl` permet toujours de chiffrer et déchiffrer un message… cette fois de manière asymétrique.

Vous pouvez faire cet exercice avec votre voisin ou voisine.

Faites donc :

- générez une paire de clé *ssh* si vous ne l'avez pas déjà fait. Pour RSA 4096 bits la commande suivante créera deux fichiers ; `id_rsa` et `id_rsa.pub`.

```
$ ssh-keygen -t rsa -b 4096
```
terminal

  Regardez quelles permissions ces fichiers ont.
- le format OpenSSH de la clé publique ne conviendra pas pour la suite, créons un fichier contenant cette même clé au format PEM ;

```
$ openssl rsa -in id_rsa -pubout -out id_rsa.pub.pem
```
terminal

- vous pouvez « diffuser » votre clé publique (en la mettant dans un répertoire accessible par votre voisin·e, il ou elle pourra l'utiliser par exemple) ;
- chiffrez un message avec une clé publique (celle de votre voisin·e par exemple) et partagez le message chiffré ;

```
$ echo "Message super secret"
    | openssl pkeyutl -encrypt -pubin
        -inkey <la clé publique du voisin>
        -out encrypted-message.bin
```
terminal

- cherchez dans la page de manuel de `openssl` comment déchiffrer le message avec votre clé privée… et déchiffrez le message reçu de votre voisin·e.

Fin Exercice 2

1.3 Signature des messages

Pour vérifier l'intégrité d'un message lors de sa transmission, le message est signé. Cette signature atteste que le message (chiffré) transmis n'a pas été altéré pendant le transport.

Cette signature — généralement plus courte que le message lui-même — accompagne le message. Le récepteur du message signe le message reçu et compare la signature reçue avec la signature qu'il vient de calculer.

Cette signature MAC (*message authentication code*) peut s'accompagner d'une clé secrète, on parle alors de HMAC (*keyed-hash message authentication code*). Dans le cas de *ssh*, c'est bien HMAC qui est utilisé avec la clé de session (cfr section 2).

Voici quelques algorithmes `HMAC`[2] : hmac-sha1 hmac-sha1-96 hmac-sha2-256 hmac-sha2-512 hmac-md5 hmac-md5-96 hmac-ripemd160 hmac-ripemd160@openssh.com umac-64-@opessh.com umac-128@openssh.com hmac-sha1-etm@openssh.com hmac-sha1-96-etm-@openssh.com hmac-sha2-256-etm@openssh.com hmac-sha2-512-etm@openssh.com h-mac-md5-etm@openssh.com hmac-md5-96-etm@openssh.com hmac-ripemd160-etm@o-penssh.com umac-64-etm@openssh.com umac-128-etm@openssh.com

Exercice 3 **Altération d'un message**

Vérifions l'intégrité d'un message avec SHA-512.

- ▷ éditez un nouveau fichier en y écrivant quelques lignes de texte ;
- ▷ calculez le *hash* du message ;

```
$ sha512sum <fichier>
```
terminal

- ▷ modifiez un caractère de votre fichier, recalculez le *hash* et constatez les différences (ou plus exactement, cherchez une similitude).

Ceci montre que si le message est altéré — même très peu — son *hash* est complètement différent. En regardant le *hash*, il est donc très facile et rapide de vérifier l'intégrité du message.

Fin Exercice 3

Remarque

Cette technique est utilisée lors du téléchargement de certains fichiers par exemple. Ces fichiers sont accompagnés d'un *hash* qu'il est facile de vérifier.
Dans certains cas, le *hash* est disponible sur le site officiel et le fichier est répliqué à différents endroits. Vérifier le *hash* permet d'être sûr que le fichier n'a pas été altéré lorsqu'il a été partagé sur un autre site.

2. Accessible par la commande `ssh -Q mac`

Exercice 4 **Vérification d'une archive**

À l'adresse https://cdimage.debian.org/debian-cd/current/amd64/iso-cd, vous trouverez en téléchargement des *iso* debian à télécharger.

Ces fichiers sont accompanés de fichiers de somme de contrôle (*hash*). Ces *hashs* servent à vérifier l'intégrité de l'*iso* téléchargé.

1. Téléchargez l'iso correspondant au *pattern* (schéma) `debian*-netinst.iso` ;
2. calculez le *hash* du fichier à l'aide d'une des commandes `sha512sum`, `sha256sum` voire `md5sum` ;
3. comparez la valeur obtenue avec la valeur se trouvant dans le fichier correspondant au *hash* que vous avez choisi ;

   Si les valeurs correspondent, c'est que le fichier téléchargé est bien le fichier qui a été déposé par les responsables debian... sinon, ne l'utilisez pas.

4. effacez le fichier que vous venez de télécharger [3]

Pour être complet, il faudrait également vérifier que ces fichiers de somme de contrôle sont eux-même correct... mais ça sort du cadre de ce premier td. Le lecteur intéressé suivra le lien [4] renseigné sur le site de debian.

Fin Exercice 4

2 Comment se passe la connexion ?

Extrait d'une page de manuel de `sshd` :

> *The OpenSSH SSH daemon supports SSH protocol 2 only. Each host has a host-specific key, used to identify the host. Whenever a client connects, the daemon responds with its public host key. The client compares the host key against its own database to verify that it has not changed. Forward security is provided through a Diffie-Hellman key agreement. This key agreement results in a shared session key. The rest of the session is encrypted using a symmetric cipher, currently 128-bit AES, Blowfish, 3DES, CAST128, Arcfour, 192-bit AES, or 256-bit AES. The client selects the encryption algorithm to use from those offered by the server. Additionally, session integrity is provided through a cryptographic message authentication code (hmac-md5, hmac- sha1, umac-64, umac-128, hmac-ripemd160, hmac-sha2-256 or hmac-sha2-512).*
>
> *Finally, the server and the client enter an authentication dialog. The client tries to authenticate itself using host-based authentication, public key authentication, challenge-response authentication, or password authentication.*

Si l'on traduit et résume, nous obtenons :

1. le client fait une demande de connexion et le serveur répond en fournissant sa clé publique ;
2. le client compare cette clé avec celle qu'il connait (c'est le *host checking*) :
   - ▷ s'il l'a connait et qu'elle correspond à l'adresse IP ou au nom associé, le serveur est reconnu ;

3. Cette étape est importante pour éviter que ce fichier de près de 300Mib reste en beaucoup d'exemplaire sur linux1. Merci de la part des administrateurs de cette machine.
4. https://www.debian.org/CD/verify

▷ s'il ne l'a connait pas, il demande à l'utilisateur s'il est d'accord de l'ajouter à la liste des hôtes connus en affichant un message à l'allure suivante :

```
The authenticity of host 'host@example.org (192.168.1.112)' can't
be established.
ECDSA key fingerprint is SHA256:t1NxK(CUT)iPOets84eeIPO1z.
Are you sure you want to continue connecting (yes/no)?
```

3. en utilisant Diffie-Hellman, qui est un algorithme asymétrique, le client et le serveur se partagent une **clé de session** (*shared session key*) ;
4. le reste de la session utilisera un chiffrement symétrique en utilisant cette clé de session nouvellement partagée. Chaque message sera signé avec un algorithme HMAC. Les algorithmes choisis sont les premiers communs au serveur et au client ;
5. c'est à ce moment que le client va s'authentifier auprès du serveur, par exemple en communiquant son mot de passe ou par échange de clés.

Exercice 5 — Connexion ssh verbeuse

L'option `-v` de *ssh* rend le programme beaucoup plus verbeux. Il affiche dans la console toute une série de commentaires. Parmi ceux-ci se trouve l'information de quelle algorithme est utilisé pour faire l'échange de clés.

Entrez la commande suivante sous *Git Bash* et cherchez cette information :

```
$ ssh -v g12345@linux1
```
terminal

Si vous ne trouvez pas, entrez ces deux commandes, un peu plus précises :

```
$ ssh -v g12345@linux1 true 2>&1 |grep "server->client"
$ ssh -v g12345@linux1 true 2>&1 |grep "client->server"
```
terminal

Tiens ! Que font ces deux commandes ?

Fin Exercice 5

3 Connexion SSH

3.1 Connexion simple

Pour se connecter à un serveur ssh, il suffit de connaitre son nom ou son adresse IP et d'utiliser un **client ssh**. Sous linux, la commande s'appelle *ssh* et sous Windows, c'est souvent *PuTTY*[5] qui est utilisé mais *Git Bash* fait bien l'affaire aussi.

Exercice 6

Connexion à linux1

En utilisant PuTTY, se connecter à *linux1*.

En utilisant Git Bash, se connecter à *linux1*. Pour ce faire, lancer une console Git Bash et entrer la commande `ssh g12345@linux1`

Fin Exercice 6

Définition : port

Pour qu'une machine puisse faire tourner plusieurs programmes « serveurs » (qui sont des programmes continuellement à l'écoute d'une connexion réseau), il est nécessaire de donner une information supplémentaire à l'adresse de la machine. Nous dirons qu'un programme écoute sur un certain **port**.
Un *port* peut être comparé à une porte numérotée et une machine sera dotée de plusieurs portes portant un numéro. Le programme client, lors de sa connexion, se connectera au serveur avec son adresse et son numéro de port(e).
Certains programmes ont un numéro de port réservé. Par exemple [a] :

- ssh, 22 ;
- serveur web, 80 ;
- serveur mail, 25 ;
- ...

a. Pour en voir une liste plus complète : `cat /etc/services`

5. À l'heure de Windows 10 et de l'intégration de `bash`, les utilisateurs et utilisatrices de Windows, vont pouvoir se détacher de PuTTY. Voir par exemple http://namok.be/blog/?post/2018/10/05/debian-microsoft

3.2 Paramètres de connexion

Lors d'une connexion, il faut au minimum renseigner l'adresse du serveur. Il arrive parfois que l'on se connecte avec un autre nom d'utilisateur ou sur un autre *port*. La commande de connexion aura alors plutôt cette allure :

```
$ ssh <user>@<host>:<port>
$ ssh anakin@example.org:2222
```
terminal

Certains serveurs *ssh* ferment automatiquement la connexion dès lors que l'utilisateur ou l'utilisatrice est inactif·ve trop longtemps. *ssh* peut s'arranger pour garder la connexion active.

Il est possible de paramétrer *ssh*. Son fichier de configuration est *~/.ssh/config*.

Pour se connecter à *linux1*, un fichier de configuration pourrait avoir l'allure suivante :

```
Host *
    ServerAliveinterval 120

Host labo
    HostName linux1
    Port 22
    User g12345
```

La connexion à `linux1` en tant qu'utilisateur *g12345* se fera par la commande :

```
$ ssh labo
```
terminal

Exercice 7 **Connexion paramétrée**

Ajoutez un fichier de configuration sous Windows `Z:/.ssh/config` contenant les paramètres ci-dessus et testez la connexion avec *Git Bash*.

Si l'on voulait faire le même paramétrage avec *PuTTY*, cherchez où se trouvent ces mêmes paramètres dans l'interface de *PuTTY*.

Fin Exercice 7

3.3 Connexion par échange de clés

Plutôt que de se connecter en utilisant son mot de passe, il est possible de se connecter par échange de clés cryptographiques. Pour ce faire, il suffit de déposer sa clé publique sur le serveur. Lors de la connexion, le client enverra ses clés publiques et le serveur vérifiera s'il en possède une pour l'utilisateur ou l'utilisatrice donné·e. Si c'est le cas, l'authentification est faite.

Le serveur conserve les clés publiques autorisées dans les fichiers `/home/<user>/.ssh/authorized_keys`.

Exercice 8 **Connexion à linux1 à partir de Windows par échange de clés**

Sous *Git Bash*, si ce n'est pas déjà fait :

1. créez une paire de clés ssh ;

```
$ ssh-keygen -t rsa -b 4096 -C "12345@etu.he2b.be"
```
terminal

2. cherchez où se trouvent les deux fichiers créés ;
3. copiez la clé publique sur *linux1* grâce à votre mot de passe ;

```
$ ssh-copy-id -i <votre clé> <user>@linux1
```
terminal

4. connectez vous à *linux1* sans entrer de mot de passe ;
5. vérifiez sur *linux1* qu'un fichier `/.ssh/authorized_keys` vient d'être ajouté (ou modifié) en regardant sa date de dernière modification et constatez qu'il contient bien la clé publique ajoutée.

Maintenant que notre clé est déposée sur *linux1*, comment faire pour faire la même chose — une connexion par échange de clés — avec *PuTTY* cette fois ?

Fin Exercice 8

3.4 Codes d'échappement

Pendant une connexion *ssh*, il est possible de contrôler la connexion en envoyant des caractères d'échappement (*escape characters*[6]).

Les codes d'échappement commencent par le caractère `~` (sauf configuration spéciale) et doivent suivre un passage à la ligne. Il est donc recommandé d'entrer la touche `[Enter]` avant. Voici quelques codes :

- ▷ `~.` demande une déconnexion au serveur ;
- ▷ `~[Ctrl-Z]` passe la connexion en *background* ;
  Pour la ramener en avant plan, `fg`, pour voir les tâches, `jobs`. Ramener une tâche spécifique `fg %<i>`

Exercice 9 **Déconnexion rapide**

Après vous être connecté à *linux1*, déconnectez vous en utilisant le code d'échappement plutôt que `exit`

Fin Exercice 9

6. Le fait de connaitre le terme exact en français et en anglais permet d'être plus efficace lors d'une recherche dans une page `man` par exemple.

3.5 Ne pas demander de shell

Par défaut, une demande de connexion *ssh* procure un *shell*. Ce comportement n'est pas obligatoire. Il est possible de ne demander l'exécution que d'une seule commande avant déconnexion.

La syntaxe est la suivante :

```
ssh <user>@<host> <command>
```

Exercice 10 **ssh, une seule commande**

À partir de *Git Bash* connectez vous à *linux1* pour n'exécuter que la commande `df . -h`.

Tiens, que fait cette commande ?

Fin Exercice 10