Rapport de soutenance 1
OUSSH : Open Unsecure Shell

Nom de groupe : Les Mécréants du Temps

BERGER Théo - berger_e (a quitté EPITA)
CARON–LASNE Maxence - caron-_m
KHOUDLI Younes - khoudl_y

13 mars 2017

Sommaire

1 Introduction

Notre projet de quatrième semestre à l'EPITA, "OUSSH", consiste en un logiciel de communication entre un serveur et un client via le shell. Le but pour le client est de pouvoir entrer des commandes dans le shell du serveur en étant sûr d'une part de l'authenticité du serveur, et d'autre part de l'incapacité d'un utilisateur tiers à pouvoir lire les informations circulant entre le client et le serveur. Le serveur quant à lui doit pouvoir s'assurer que le client est un utilisateur de confiance.

Notre logiciel OUSSH aura donc un comportement assez similaire à des logiciels tels que openSSH par exemple. Nous offrons une méthode de connexion distantes sécurisée.

Ce début de projet a été marqué par le départ d'un des membres de notre groupe, Théo, qui a quitté EPITA. Ce départ a causé quelques réarrangements dans la répartition des tâches. Néanmoins, nous avons décidé de conserver le planning intact.

Dans ce rapport de soutenance, nous allons d'abord vous rappeler les objectifs de cette soutenance, puis nous allons vous présenter les fonctionnalités qui ont été implémentées. Enfin, après avoir précisé les problèmes survenus depuis le début du projet, nous finirons sur la conclusion de cette première soutenance.

2 Rappel des objectifs de la soutenance

Pour cette première soutenance, nous avions défini trois objectifs principaux.

Le premier objectif était de pouvoir, à partir d'un terminal virtuel, communiquer avec un autre terminal virtuel. Le but est de pouvoir, à partir d'un terminal 1, envoyer des commandes au terminal 2, tandis que celui-ci renvoie au terminal 1 les résultats des commandes reçues ("output" et erreurs).

Le deuxième objectif était d'implémenter la possibilité pour le serveur d'authentifier le client, c'est-à-dire déterminer si le client tentant de communiquer avec le serveur est un utilisateur de confiance. Pour cela, la solution choisie avait été l'authentification par mot de passe.

Enfin, le dernier objectif pour cette première soutenance était d'implémenter un prototype de communication serveur-client, en clair, par le biais de sockets.

Voyons maintenant si ces objectifs ont été tenus.

3 Fonctionnalités implémentées

3.1 Communication entre terminaux virtuels

La fonctionnalité de base de OUSSH est de pouvoir, à partir d'un terminal virtuel classique, comme ceux que l'on utilise tout les jours sous GNU+Linux, de pouvoir exécuter des commandes dans un autre terminal virtuel, situé éventuellement sur une machine distante.

La problématique est donc de savoir comment faire communiquer deux processus s'exécutant dans deux terminaux virtuels différents.

Tout d'abord, il faut observer que cette problématique est d'abord une question d'entrée-sortie. En effet, l'objectif est de faire en sorte que la sortie du processus client soit l'entrée du processus serveur, et que la sortie ainsi que le canal d'erreur du processus serveur soit l'entrée du processus client.

Cette première problématique est néanmoins assez vite surmontée grâce à l'architecture Unix-like de Linux, rendant cette opération assez basique.

La vraie difficulté ce trouve au niveau de la communication entre deux processus s'exécutant sur deux terminaux virtuels (ou pseudo-terminaux). Difficulté accentuée par le fait que les propriétés des terminaux sont importantes, notamment pour l'affichage et les droits du pseudo-terminal.

Pour résoudre ces problèmes, nous allons utiliser les propriétés des pseudo-terminaux.

Un pseudo-terminal est en fait une paire de PTY, des gestionnaires de caractères. L'un de ces PTY est "le maître", l'autre est "l'esclave". Ces deux PTY sont connectés entre eux de telle manière que tout ce qui arrive en entrée de l'esclave passe en sortie du maître, et tout ce qui arrive en entrée du maître passe en sortie de l'esclave (voir Figure 1).

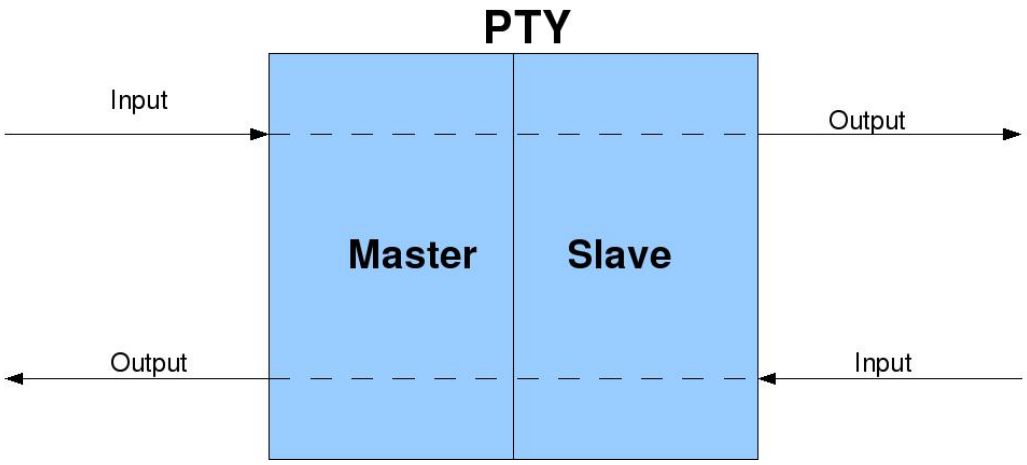

FIGURE 1 – Schéma du fonctionnement d'un PTY

C'est exactement ce dont on a besoin. Le principe est de relier le pseudo-terminal client à un pseudo-terminal du côté serveur, créé par un processus côté serveur, dans lequel sera exécuté un shell.

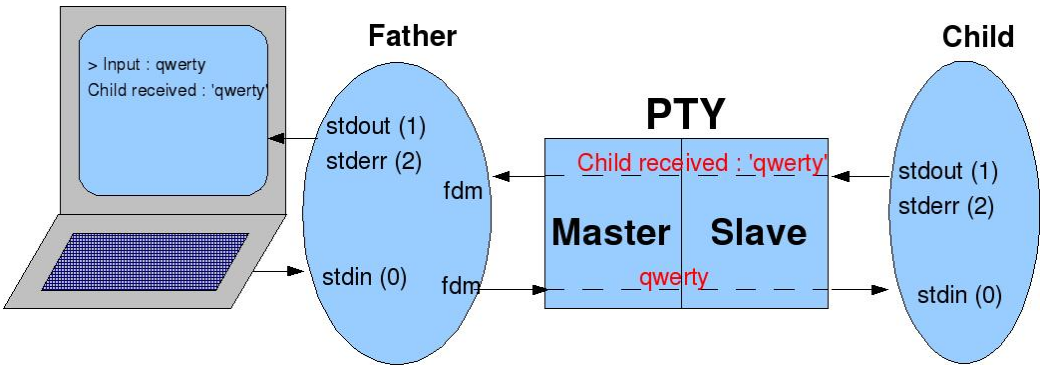

FIGURE 2 – Organisation des entrées et sorties entre les deux processus

3.1.1 Problèmes rencontrés

Certaines choses ne sont toutefois pas évidente dans leur utilisation et la documentation a leur propos est relativement éparse. En effet, les terminaux ne se comportent pas comme de simples pipes. Ils manipulent les données

entre leur entrée et leur sortie, gèrent l'écho des caractères entrés, la discipline de ligne, connaissent leur taille, etc. Il faut donc payer une attention particulière a la façon dont nous les réglons, puisque les données en traversent deux (le terminal du client, et celui entre le serveur et le shell).

3.2 Communication serveur-client

3.2.1 Généralités

Pour communiquer entre le client et le serveur, nous avons, dans un premier temps, décidé de simplement utiliser une socket Unix, ce qui permet d'éviter certaines complications induites par l'utilisation de TCP/IP.

Le client et le serveur communique donc en passant par cette socket, et échangent, en plus des informations entrées par l'utilisateur et celles affichées par le shell, des paquets de contrôle, contenant par exemple la taille du terminal ou l'annonce d'une déconnexion gracieuse.

3.3 Authentification du client auprès du serveur

Une des problématique d'un logiciel de communication sécurisé comme OUSSH est l'authentification du client auprès du serveur.

Il est important pour le serveur d'être sûr que la personne voulant se connecter à lui est un utilisateur de confiance. En effet, une personne tentant de se connecter pourrait très bien être un utilisateur mal-intentionné cherchant à prendre le contrôle du serveur.

Une manière simple et efficace pour s'assurer de l'identité d'un client est le mot de passe. Lors de la demande de connexion du client, le serveur demande à celui-ci son nom d'utilisateur et son mot de passe. Si le mot de passe donnée par l'utilisateur correspond bien au couple utilisateur/mot de passe stocké du côté serveur, le client peut se connecter. C'est très classique, mais c'est efficace.

Néanmoins, cette méthode soulève une problématique de sécurité très importante : comment stocker les mots de passe des utilisateurs sur le serveur ?

En effet, si les mots de passes sont stockés "en clair" sur le serveur, c'est-à-dire sans système de chiffrement, la moindre personne réussissant à s'introduire sur le serveur peut avoir accès aux mots de passe de tous les utilisateurs. De plus, même si on considère que le serveur est absolument imprenable, le fait que tous les utilisateurs de confiance du serveurs peuvent voir les mots de passes de tous les autres utilisateurs reste un problème.

Il faut donc que les mots de passe soient stockés de telle manière qu'ils ne puissent pas être reconnus.

Pour cela, il faudrait une méthode permettant de déterminer si une chaîne de caractère est bel et bien le mot de passe, sans pour autant pouvoir récupérer le mot de passe. En bref, il faut une fonction étant capable de convertir une chaîne de caractères en une unique autre suite de caractères (de telle sorte à ce qu'une chaîne de caractères produise toujours la même autre suite de caractères), mais que l'opération inverse soit impossible.

Pour cela, il nous faut une fonction de hachage cryptographique.

Une fonction de hachage cryptographique est une fonction qui à partir d'une chaîne de caractère de taille quelconque (le message) associe une chaîne de caractères de taille fixe (la valeur de hachage), de telle manière que la fonction inverse soit quasiment impossible à calculer (du moins pas dans des temps de calculs raisonnables).

D'autres propriétés essentielles de la fonction de hachage sont d'une part l'impossibilité d'aboutir à une même valeur de hachage à partir de deux messages différents, d'autre part l'impossibilité de modifier un message sans modifier la valeur de hachage.

Grâce à ce type de fonction, on peut stocker les valeurs de hachage en étant sur que les mots de passe associés ne puissent être récupérées dans un temps raisonnable.

La méthode est la suivante. Lorsqu'un utilisateur choisit son mot de passe, celui-ci est entrée dans la fonction de hachage. La valeur de hachage qui en résulte est envoyée au serveur qui le stocke dans un fichier, associée à son identifiant, qui est lui stocké en clair (sachant que l'identifiant peut aussi servir à la création de la valeur de hachage). Lorsque l'utilisateur veut ensuite se connecter au serveur, il envoie la valeur de hachage ainsi que son nom d'utilisateur. Si le hash correspond au hash associé au nom d'utilisateur (si les deux valeurs de hachage sont les mêmes), la connexion est acceptée.

Une fois la connexion acceptée, un utilisateur voulant savoir les mots de passe stockés sur le serveur sera bloqué : il ne pourra les trouver à partir des valeurs de hachage.

Nous avons choisi pour notre fonction de hachage la fonction SHA-512, de la famille SHA-2 (Secure Hash Algorithm). Cette fonction est disponible dans la glibc grâce à la fonction crypt(3).

4 Objectifs pour la prochaine soutenance

4.1 Trouver une solution à l'authentification du serveur auprès du client

Le serveur doit être authentifié par le client. En effet, le client doit être sur de ne pas être en train de communiquer avec un autre serveur que celui visé, puisqu'il pourrait lui transférer des données confidentielles (un mot de passe, par exemple).

Pour cela, nous utiliserons nécessairement de la cryptographie asymétrique, puisqu'on ne peut pas demander au serveur de taper un mot de passe.

4.2 Algorithme de chiffrement symétrique

Une des techniques de base pour faire du chiffrement est de se baser sur un système de clé symétrique. Le client et l'utilisateur ont la même clé, servant à chiffrer et à déchiffrer. C'est une méthode simple ayant pour avantage d'être rapide.

Il y a néanmoins un problème : la clé crée par le serveur doit être envoyée au client, car le serveur et le client ont la même clé. Si cet échange se fait sur le réseau, la clé envoyée ne pourra pas être chiffrée, car le client n'a pas encore de clé de déchiffrement. Si la clé est échangée sur le réseau en clair, un utilisateur pourra l'intercepter et l'utiliser pour déchiffrer tous les messages échangés entre le serveur et le client.

Nous utiliserons la méthode de Diffie-Hellman pour l'échange initial des clefs.

L'avantage en contrepartie de ces méthode est leur performance plus élevé pour le même niveau de sécurité.

4.3 Algorithme de chiffrement asymétrique

Une autre technique résolvant ce problème est le chiffrement asymétrique. Avec cette technique, le serveur crée deux clés : une clé publique, servant à chiffrer un message et une clé privée, servant à déchiffrer les messages chiffrés avec la clé publique associée. La clé publique, ne servant qu'à chiffrer des messages, peut être partagée publiquement sans risque. Par contre, la clé privée, servant à déchiffrer, ne devra jamais être partagée. Notons qu'avec cette méthode, une paire de clé ne peut servir que pour des transmissions d'un utilisateur vers le créateur des clés. Pour une communication bilatérale, on utilisera deux paires de clés.

Le principal problème du chiffrement asymétrique est que le chiffrement et le déchiffrement sont longs. Pour résoudre ce problème, il faut que le serveur crée une paire de clés asymétriques, qu'il envoie la clé publique à l'utilisateur pour que celui-ci chiffre sa clé de chiffrement symétrique et l'envoie au serveur. Le serveur et le client ont maintenant tous les deux une même clé de chiffrement symétrique, ils peuvent communiquer de manière sécurisée avec un temps de chiffrement/déchiffrement raisonnable.

5 Conclusion

Cette période de développement entre le rendu de cahier des charges et la première soutenance a été assez intense, surtout à cause du départ de Théo. Nous avons néanmoins tenu à ne pas changer le planning, et à prendre le plus d'avance possible pour que cette difficulté ne se répercute pas sur le logiciel final.

Cet objectif est un succès. En effet, nous sommes, à l'heure de cette première soutenance, en avance sur notre planning : en effet, les fonctionnalités réseaux sont opérationnelles.

Cette avance va nous permettre de nous concentrer sur les deux clés de ce projet : les deux algorithmes de chiffrements de données. Ces deux algorithmes sont cruciaux, il nécessiterons un soin particulier.

Cette avance nous permettra aussi de rendre OUSSH toujours moins "unsecure", c'est-à-dire d'ajouter des fonctionnalités sécurisant encore plus les communications.

6 Annexes

6.1 Table des matières

Table des matières

6.2 Tableaux récapitulatifs

6.2.1 Répartition des tâches

	caron-_m	khoudl_y
Creer un terminal virtuel	x	x
Communication avec un serveur distant		x
Authentification du serveur auprès du client	x	
Authentification du client auprès du serveur		x
Chiffrement symétrique	x	
Chiffrement asymétrique		x

6.2.2 Soutenance 1

	Avancement
Creer un terminal virtuel	100%
Communication avec un serveur distant	30%
Authentification du serveur auprès du client	0%
Authentification du client auprès du serveur	100%
Chiffrement symétrique	20%
Chiffrement asymétrique	20%

6.2.3 Soutenance 2

	Avancement
Creer un terminal virtuel	100%
Communication avec un serveur distant	70%
Authentification du serveur auprès du client	50%
Authentification du client auprès du serveur	100%
Chiffrement symétrique	60%
Chiffrement asymétrique	60%

6.2.4 Soutenance 3

	Avancement
Accès à un terminal virtuel	100%
Communication avec un serveur distant	100%
Authentification du serveur auprès du client	100%
Authentification du client auprès du serveur	100%
Chiffrement symétrique	100%
Chiffrement asymétrique	100%

6.3 Sources

Figure 1 et 2 :
`http://rachid.koucha.free.fr/tech_corner/pty_pdip.html`