- Projet Réseau -

Hermes

0.1 Introduction

1 Chatty

1.1 Principe

Chatty est une application de chat en ligne de commande permettant à deux utilisateurs de discuter en toute sécurité grâce au protocole RSA.
Un utilisateur se connecte en tant que serveur, l'autre en tant que client. Le serveur doit physiquement donner son adresse IP au client et le client doit la rentrer dans le logiciel pour permettre la connexion. Ici, on utilise des sockets pour établir une connexion. La connexion par socket n'étant pas chiffré, on utilise un système de chiffrement RSA pour, d'une part permettre l'échange des clés en toute sécurité, et d'autre part, une communication sécurisée.

2 Hermes

2.1 Principe

Ici, on attaque des choses un peu plus compliquées. Un serveur accepte des connexion socket de clients et s'occupe de router les messages reçus vers les destinataires désignés. On utilise des Threads pour permettre d'écouter et de gérer chaque client connecté séparément.

2.2 La structure des messages - Classe `Package`

Comme pour Chatty, on utilise une classe spéciale pour gérer le chiffrement/déchiffrement des messages à envoyer via le socket. Cela permet, d'une part de factoriser le code, côté server et client et d'autre part, de rajouter une couche d'abstraction. Les attributs de la classe `Package` sont majoritairement privés, permettant une meilleure sécurité.

2.2.1 Gestionnaire AES

On souhaite pouvoir échanger des messages via socket sans craindre de se faire observer. On va donc chiffrer tous les flux traversant le socket via un algorithme de chiffrement, ici AES. C'est un algorithme de chiffrement dit symétrique (même clé pour le chiffrement et déchiffrement). On utilise principalement 4 méthodes pour permettre le chiffrement/déchiffrement avec AES :

- `cipherStringAES()` : Permet simplement le chiffrement d'une chaîne de caractère avec AES. On l'utilise pour envoyer le nom d'utilisateur du client au serveur à l'initialisation de connexion (utile pour le routage des paquets).
- `decipherToStringAES()` : L'inverse de la méthode précédente, elle déchiffre des tableaux de bits (`byte[]`) avec AES. Elle sert lors de la réception du nom d'utilisateur lors de l'initialisation de la connexion. On s'en sert aussi lors du routage des paquets pour déchiffrer le nom d'utilisateur de l'envoyer et du destinataire.
- `cipherMessageAES()` : Côté client, elle permet de créer un paquet contenant date et heure d'envoi, utilisateur envoyeur du paquet, destinataire et message envoyé. Le tout méticuleusement chiffré avec une clé AES de 256 bits.

- `decipherMessageAES()` : coté client récepteur d'un paquet, elle extrait toutes les informations de celui-ci pour permettre leur traitement par le client après l'avoir déchiffré.

Le problème est que, malgré la force d'AES, sa rapidité d'exécution et sa fiabilité, cela reste un algorithme de chiffrement symétrique. Il faut donc que les deux parties s'échangent la clé à l'initialisation de la connexion pour permettre l'échange de messages chiffrés. Il est bien évidement inenvisageable d'envoyer la clé AES en clair sur le réseau, la rendant visible pour toute personne écoutant le flux de paquets. On utilise pour cela RSA...

2.2.2 Gestionnaire RSA

Tout comme AES, la classe `Package`, permet aux classes l'utilisant de chiffrer/déchiffrer avec l'algorithme RSA. RSA est un algorithme de chiffrement asymétrique paru en 1977. Une clé (dite publique) sert à chiffrer les messages et une autre (dite privée) à les déchiffrer. La force de cet algorithme est le fait que l'on ne peut retrouver la clé privée à partir de la clé publique en temps raisonnable.
On va donc se servir de RSA pour partager la clé AES entre le serveur et le client et ainsi permettre un échange de paquets en toute sécurité.
Ainsi, à l'initialisation de la connexion, le client, qui a généré des clés RSA à son instanciation envoie sa clé publique au serveur pour que celui-ci lui renvoie la clé AES chiffrée en RSA qu'il utilise pour communiquer avec chaque client. Le client reçoit alors un paquet constitué d'une séquence de bits apparement illisibles mais qu'il peut déchiffrer en une parfaite clé AES pour ensuite envoyer des messages chiffrés avec celle-ci.
La classe `Package` permet différentes deux utilisations de l'algorithme RSA au travers de deux méthodes :

- `getAESCiphered()` : Utilisée côté serveur, elle sert à chiffrer la clé AES générale avec la clé publique RSA du client pour la lui envoyer en toute sécurité.

- `setAESCiphered()` : Côté client, elle déchiffre le paquet reçu par le serveur chiffré en RSA pour récupérer la clé AES et la rajouter à son trousseau. La discussion peut ensuite commencer entre le serveur et le client sans que personne ne puisse savoir ce qu'il se disent.

On choisit volontairement d'utiliser deux méthodes de chiffrement pour une simple raison. RSA étant très puissant, il est malgré tout très lent comparé à AES du fait qu'il utilise un algorithme assez complexe. AES, quand à lui, est basé sur DES, algorithme effectuant seulement des permutations et xor de bits. Ainsi, pour chiffrer un gros volume de bits et permettre l'envoi de long messages rapidement, sans trop de délai, AES est une option de choix. On utilise donc RSA seulement pour envoyer la clé AES au début de la connexion.

2.2.3 Structure d'un paquet

Les paquets envoyés dans le Socket par les client sont seulement des matrices de bits dont chaque ligne correspond à une information :

- Les destinataire du paquet pour savoir, côté serveur, à qui faire parvenir le paquet.

- La personne ayant envoyé le paquet, pour éviter de lui renvoyer le paquet en cas de broadcast.

- Le message envoyé ainsi que la date et l'heure de l'envoi.

Le tout, bien entendu, chiffré avec AES.

2.3 Le serveur - Classe `ServerHermes`

La classe `ServerHermes` constitue le coeur même du serveur. Elle permet deux choses :

- **Initialiser le Socket :** à son lancement, elle ouvre un socket sur un port déterminé et génère les outils de sécurité nécessaires au bon fonctionnement d'*Hermes*.
- **Accepter les connexions :** Un Thread permet d'accepter en continu les connexions de clients (classes `ClientHermes`). Une fois un client accepté, la méthode `runServer()` créé un nouveau `ClientHandler` pour gérer séparément la connexion avec le nouveau client connecté.

2.4 Le gestionnaire de clients - Classe `ClientHandler`

2.5 Le client - Classe `ClientHermes`