Université du Québec
à Chicoutimi

TP Sécurité

Cryptage asymétrique et Hachage

Travail réalisé dans le cadre de 8SEC201 -
Cybersécurité défensive : vulnérabilité et incidents

Écrit par :

Jerôme Baldo - Bastien Lefumeux

Jeudi, 16 février 2023

Table des matières

I. Algorithme RSA

L'algorithme asymétrique de cryptographie RSA est fondé sur l'utilisation d'une paire de clés composée d'une clé publique pour chiffrer et d'une clé privée pour déchiffrer ou signer des données confidentielles. La clé publique correspond à une clé qui est accessible par n'importe quelle personne souhaitant chiffrer des informations. La clé privée permettant le chiffrement est quant à elle réservée à la personne ayant créé la paire de clés.
Lorsque deux personnes, ou plus, souhaitent échanger des données confidentielles, une personne prend en charge la création de la paire de clés, envoie sa clé publique aux autres personnes qui peuvent alors chiffrer les données confidentielles à l'aide de celle-ci puis envoyer les données chiffrées à la personne ayant créé la paire de clés. Cette dernière peut alors déchiffrer les données confidentielles à l'aide de sa clé privée

Source : Introduction du TP

II. Création d'un algorithme RSA

A. Création des clés

But :
L'objectif est de générer des clés pour les 2 acteurs, dans notre code on les a représentés par Alice et Bob. Pour pouvoir les générer, on aura besoin d'une longueur de clé (128 bits même si pour une protection optimale il faudrait monter à 2048 bits) et de la bibliothèque SecureRandom qui va nous permettre de fournir un générateur de nombres aléatoires cryptographiquement fort.
Solution :
Pour générer la clé publique, on va devoir créer un exposant de chiffrement.Pour ce faire on a besoin en entrée :

- L'indicateur d'Euler φ(n) (où n est le produit de deux nombres premiers)
- La longueur en bits du nombre premier utilisé pour générer le nombre premier probablement premier.

Donc on va générer génère un nombre premier probablement premier, et on vérifiera que celui-ci est compris entre 1 et φ(n), et que leur plus grand commun diviseur est égal à 1.
On récupère le nombre premier généré, on l'assigne comme exposant de chiffrement et notre clé publique est créée.
Et pour générer la clé privée, on va créer l'exposant de déchiffrement qui va prendre en entrée :

- La clé publique
- L'indicateur d'Euler φ(n)

Puis après on va calculer notre clé privé avec cette formule : d ≡ e^(-1) mod φ(n) où :

- d : L'exposant de déchiffrement
- e : L'exposant de chiffrement
- n : le module de chiffrement

Après ça on récupère notre exposant de déchiffrement qui nous permet de créer notre clé privé.

B. Numérisation et Dénumérisation du message

But :

La numérisation de notre message consistait à transformer chaque caractère de notre String en int. Et on doit réaliser l'opération inverse pour la dénumérisation.

Problème :

Comment faire pour reconnaître chaque caractère lorsqu'on voudra dénumériser notre message ?

Solution :

Pour y parvenir, on s'est dit qu'on allait tout simplement appliquer une suite de chiffres qui sera notre délimiteur.

On a juste 2 contraintes :

- Avoir une suite de chiffre peu probable à traiter
- Ne pas mettre un nombre trop grand car dans notre code à un moment on traite certaines chose avec des Integers (Dépasserait la limite qui nous est imposé par Integer)

C. Découpage et Assemblage du message

But :

L'objectif de cette partie est de découper notre message qui est numérisé afin de pouvoir le chiffrer ensuite en petit bout. Et l'assemblage réalise l'effet inverse en réunissant les bouts décryptés et les rassemble pour faire qu'un seul String

1er Problème :

Si notre String ne peut pas être découpé en part égale ou alors si notre String est trop petit pour être découpé, comment gérer notre découpage ?

Solution :

Dans le cas où la taille de la coupe est trop petite par rapport à la longueur de la chaîne, on créera juste un tableau avec une seule case qui contiendra une seule coupe qui contiendra l'ensemble du string. Et dans le cas où la division n'est pas exacte, le dernier élément du tableau contiendra le reste de la chaîne qui n'a pas pu être découpé en parties égales.

2ème Problème :

Si la taille n'est pas exacte, on se retrouve avec des cases de tableau avec des valeurs nulles. Comment faire pour éviter ce genre de cas ?

Solution :

En utilisant la méthode filter() qui va supprimer toutes les valeurs null du tableau.

D. Chiffrage et Déchiffrage

But :

Notre but ici serait de crypter et de chiffrer à l'aide de notre module de chiffrement, l'exposant de chiffrement et l'exposant de déchiffrement.

Pour pouvoir chiffrer nos données, on fera le modulo de notre message numérisé à l'aide de notre exposant de chiffrement (clé publique) ainsi que le module de chiffrement

Tandis que pour déchiffrer notre message chiffré, on fera aussi le modulo de notre message crypté seulement on utilisera notre exposant de déchiffrement (clé privé) ainsi que le module de chiffrement pour pouvoir avoir notre message en clair

III. Hashage

A. Problème de taille

En suivant les consignes nous avons remarquer que la taille nominal de la matrice par défaut est de 1.6 Ko car c'est 16 octets (bytes) multipliés par 100 (d'où 1600 octets).

B. Structure de l'application

L'application est structuré en 2 fichiers : le main et le GenerHash.

main:
Sachant que la matrice a une taille limite si l'on prend 16*100 (équivaut à 1.6 Ko), nous avons crée une méthode permettant d'agrandir le tableau selon la taille. La méthode CalibrerLignesColonnes() agrandit la future matrice en multipliant la taille par 10 jusqu'à ce que la matrice soit plus grande que le fichier.
GenerHash:
La classe prend en entrée le chemin du fichier, la taille de la matrice à créer(lignes + colonnes) et le taille du hash final.
Cette classe a pour méthodes:

- lireDansFichier => génére le hash à partir d'un fichier en utilisant une matrice de bytes:
- genererHash => lit les données d'un fichier en blocs de 16 bytes
- xor => effectue une opération XOR ("OU exclusif")

C. Description des méthodes générant le hash

lireDansFichier

Cette méthode lit les données d'un fichier en blocs de 16 bytes à l'aide d'un flux d'entrée BufferedInputStream. Il y a ensuite un stockage des données lues dans la matrice tridimensionnelle. La méthode utilise des boucles for et des indices de matrice pour stocker les données dans la matrice dans l'ordre correct. Elle retourne le nombre total de bytes lus à partir du fichier.

genererHash

Cette méthode a pour objectif de générer un hash à partir d'un fichier en utilisant une matrice de bytes. Pour cela elle va appeler la méthode lireDansFichier pour stocker les octets dans la matrice servant au hash.
Par la suite la méthode calcule ensuite XOR (en appelant la méthode xor) de chaque colonne de la matrice en initialisant le buffer à 0 et en calculant le XOR de chaque ligne de la colonne j. Le résultat est stocké dans le buffer.

Ensuite le hash courant est XOR avec le contenu du buffer.
Si le nombre d'octets lus est inférieur à la taille de la matrice, remplissage des cases restantes avec des 0(parcours matrice).

xor

La méthode effectue une opération XOR ("OU exclusif") entre chaque élément de deux tableaux de bytes (buffer et ligne) de même taille. La méthode utilise une boucle for pour parcourir les deux tableaux et effectuer l'opération XOR sur les éléments correspondants.