Rapport TP1 RS40

Professeur : Abdeljalil ABBAS-TURKI
Réalisé par : Gabriella Ndjamba Batomen

Printemps 2023

Table des matières

INTRODUCTION

Le TP considère un message envoyé de Bob vers Alice, où chacun dispose d'une paire de clés (publique, privée) pour le chiffrement RSA. Bob chiffre le message avec la clé publique d'Alice. Il procède aussi à la signature de l'empreinte numérique du message avec sa clé privée. Alice reçoit le message le déchiffre et vérifie la signature de Bob (Ce n'est pas sécurisé).
Le programme a besoin de quelques améliorations :

1. Ecrire les fonction home_mod_exponent and home_ext_euclide
2. Améliorer le processus de vérification de la signature de Bob (Utilisation d'une fonction de hachage approriée de la librairie hashlib : sha256).
3. Augmenter la taille des messages échangés.
4. Réduire le temps de calcul à l'aide du théorème du reste chinois.
5. Utiliser le bourrage suivant les instructions fourni dans le sujet.

Conception

1. Fonction d'euclide étendu pour la recherche de l'exposant secret

La fonction home_ext_euclide implémente l'algorithme d'Euclide étendu pour la recherche de l'exposant secret.

Le code est donc :

```
def home_ext_euclide(y,b):
    #Etape 1: Recherche du PGCD(y,b)
    #les variable w et h sont initialiser
    w = y
    h = b
    p = [] #la liste est cree et contiendra les coefficient de chaque calcul
    #Effectue les divisions successives jusqu'à ce que h devienne égal à 0.
    while h!=0 :
     p.append(w//h)
     w,h = h, w%h
    d = [0,1]
    p.pop()
    #Etape 2: Calcul de l'inverse
    for i in range(len(p)):
        t = (d[i] - p[i]*d[i+1])%y #on calcule dn
        d.append(t)
    return d[-1] #l'inverse est le derneir dn calculer
```

2. Fonction d'exponentiation modulaire

home_mod_expnoent(x,y,n) est la fonction qui permet de réaliser l'exponentiation modulaire x^y%n.

Le code est donc :

```
def home_mod_exponent(x,y,n): #exponentiation modulaire
    r=1
    while y>0:
        if(y%2==1) :
            r=(r*x)%n
        x=(x*x)%n
        y=y//2
    return r
```

3. Fonction de hachage SHA256

Il existe plusieurs fonctions de hachage parmi lesquelles MD5, SHA1, SHA2, SHA256/512. Chaque fonction utilise un algorithme particulier. Cependant ces fonctions ont vu le jour au fur et à mesure pour la plupart dans le but de pallier le défaut des précédentes. De nos jours l'on opte pour la fonction SHA256 car elle n'a pas encore été cassée contrairement à MD5, SHA1 et SHA2. Nous allons donc mettre en œuvre cette fonction.

Le code de la fonction est donc :

```
def sign(message):
    return hashlib.sha256(message.encode(encoding='UTF-8',errors='strict')).digest()
```

Commentaire :

- *Paramètres :*
  - *Message : la chaîne de caractères à hacher ;*
- *La valeur de retour la chaîne de caractères correspondant à la version hacher du message.*

4. Le théorème du reste chinois (CRT)

Rappelons avant d'aborder la suite qu'une opération de chiffrement ou signature pour RSA, c'est mathématiquement la même chose : Cette opération nécessite une exponentiation modulaire qui est particulièrement couteuse lorsque N est grand (ce qui est le cas dans la pratique). Il serait donc intéressant si nous trouvions une méthode qui soit moins gourmande et qui nous fournisse le même résultat. C'est là qu'intervient le théorème du reste chinois. Il permet une optimisation majeure dans les temps de calculs mais en contrepartie il doit utiliser des informations dérivées de la clé privée. Son utilisation est donc réservée aux opérations de signature ou déchiffrement.

On pose les notations suivantes :

d_p = d mod (p-1)

d_q = d mod (q-1)

qInv = q^{-1} mod p

L'opération M = c^d mod n est alors équivalente aux suivantes :

S_p = c^{d_p} mod p

S_q = c^{d_q} mod q

h = qInv.(S_p - S_q) mod p

S = S_q + h.q

Soit S = (((S_p - S_q) * q^{-1}) mod p) * q +S_q

M=(S_q+h*q)%n

Le code de la fonction est donc :

```
def CRT_home_mod_exp(u,n,dq,dp,q,p,q_inv):
    Mq = home_mod_exponent(u,dq,q)
    Mp = home_mod_exponent(u,dp,p)
    u = ((Mp-Mq)*q_inv)%p
    f = (Mq + u*q)%n
       return f
```

5. Taille du message

Parler d'augmenter la taille des messages pouvant être transmis revient à générer des clés plus grandes. Pour le faire nous pouvons aller sur bigprimes.org pour générer des clés de plus grande taille. Dans ce TP nous utilisons des clés de 60 bits ce qui nous permet de pour pouvoir écrire un secret d'une taille maximal de 25 caractères.

6. PKCS#1 v1.5

PKCS#1 v1.5 est un standard de formatage utilisé pour le chiffrement et le déchiffrement des messages à l'aide de l'algorithme de chiffrement RSA. Il définit la structure des données avant le chiffrement (padding) et après le déchiffrement (unpadding) pour garantir la sécurité et l'intégrité des données.

Le formatage PKCS#1 v1.5 permet d'assurer la sécurité en introduisant de l'aléa dans le chiffrement et de récupérer le message d'origine après le déchiffrement en éliminant les octets de formatage ajoutés.

a. Formater

La fonction **formater(msg, k)** découpe le message msg en blocs de taille maximale k/2 (si la taille du message est supérieure à k) ou en deux moitiés égales (si la taille du message est inférieure à k). Chaque bloc est ensuite formatté selon les règles spécifiées dans le Tp :

- Un en-tête '0002' est ajouté au début de chaque bloc.
- Des octets aléatoires sont générés et ajoutés au bloc pour atteindre une taille de k - len(block) - 3.
- Le caractère de fin '00' est ajouté après les octets aléatoires.
- Chaque caractère du bloc est traduit en hexadécimal et ajouté au bloc formatté.
- La fonction retourne une liste Msg_formater contenant tous les blocs formattés.

Le code de la fonction est donc :

```
    def formater(msg,k):
  Block_msg = [] # découpage du message sans le formattage
  length = len(msg)

  # Si le message fait moins de k caractères on le découpe en deux moitiés
  # pour s'assurer que chaque bloc contient au maximum 50% du message
  if length < k:
     Block_msg = [msg[:length//2],msg[length//2:]]
  # Sinon on le découpe en blocs de k/2 caractères
  else:
     i =0
     j = 0
     taille = k//2
     while (j < length):
        j = i + taille
        if j > length: j = length # Si on arrive au bout du message on fait un bloc plus petit
        Block_msg.append(msg[i:j]) # On ajoute le découpage du message allant de i à j
        i += taille

  Msg_formater =[] # message découpé et formatté en hexadecimal

  for block in  Block_msg:

     Block_formater = '0002'

     for i in range(k - len(block) - 3): # On ajoute autant d'octets aléatoires que nécessaire
        Block_formater += Aleabyte()

     Block_formater += '00'

     for char in block: # On traduit les caractères en hexadécimal et on les ajoute au bloc
        Block_formater += "{:02x}".format(ord(char))

     Msg_formater.append(Block_formater)

  return Msg_formater
```

Commentaire :

La fonction formater utilise une autre fonction Aleabyte() qui génère un octet aléatoire en hexadécimal (deux caractères hexadécimaux).

Voici le code de cette fonction :

```
hex_sym = ['0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f']

def Aleabyte():
  k = '00'
  while k == '00':
     k = random.choice(hex_sym) + random.choice(hex_sym)

  return k
```

b. Deformater

La fonction **deformater(Msg_formater)** prend en entrée la liste des blocs formattés Msg_formater et déformatte chaque bloc en une chaîne de caractères. Elle parcourt chaque bloc et extrait les données en sautant les octets aléatoires et en traduisant les caractères hexadécimaux en caractères de texte. La fonction retourne le message initial sous forme de chaîne de caractères.

Le code de la fonction est donc :

```
def deformater(Msg_formater):
    message = ''
    for block in Msg_formater :
        j = 4 # On va directement en index 4 pour sauter les '0002'
        # On passe les octets aléatoires jusqu'à tomber sur '00'
        while block[j] + block[j+1] != '00' :
            j += 2
        j += 2 # On se place après le '00' pour lire les données
        # On traduit l'hexadéciamal en string et on ajoute ça au message
        try:
            message += bytes.fromhex(block[j:len(block)]).decode('utf-8')
        except:
            print('Utilisez des caractères utf-8 svp')

    return message
```

7. Chiffrement et déchiffrement

a. Chiffrement

Pour le chiffrement la fonction formater on découpe le message en blocs et on ajoute une quantité d'octets aléatoires à chaque bloc. Ensuite, la fonction Aleabyte génère des octets aléatoires en hexadécimal qui sont ajoutés au message lors du bourrage. Puis, chaque bloc est chiffré en utilisant l'exponentiation modulaire avec la clé publique ea et le modulo na. Le résultat de chaque opération de chiffrement est stocké dans la liste Block_chiffrer.

Le code est donc :

```
Block_hex = formater(secret,k)

print("voici le message découpé en blocs et bourré : \n",Block_hex)

int_Block = []
for block in Block_hex:
    int_Block.append(int(block,16))

print("voici les blocs en décimal :\n",int_Block)

Block_chiffrer = []
for block in int_Block:
  Block_chiffrer.append(home_mod_exponent(block, ea, na))
```

b. Dechiffrement

Les blocs chiffrés sont déchiffrés en utilisant l'algorithme CRT (Chinese Remainder Theorem. Les blocs déchiffrés sont ensuite convertis en représentation hexadécimale et déformatés pour obtenir le message initial.

Le code est donc :

```
Block_dechiffrer = []
for block in Block_chiffrer:
Block_dechiffrer.append(CRT_home_mod_exp(block, na, dqa, dpa, qa, pa, qa_inv))

print(Block_dechiffrer)
print("Alice converti les blocs en hexadécimal :")

Block_hex = []
for block in Block_dechiffrer:
 Block_hex.append('000' + f'{block:x}')

print(Block_hex)
```

dechif = deformater(Block_hex)

Exécution

```
Vous êtes Bob, vous souhaitez envoyer un secret à Alice
voici votre clé publique que tout le monde a le droit de consulter
n = 22262005302964707027842641155694454609045524670178746722411620440748813205546124873785589375458674431351006389502760784 9
exposant : 23
voici votre précieux secret
d = 20971454270908781982750314132175935501274769616835051260242739022131821978579506396481256461058489012605642054224766073
**********************************************************************
Voici aussi la clé publique d'Alice que tout le monde peut conslter
n = 24937594645748001056153624715717385144510416201234763430897136331748719529555252091638029819519169442833082658337772412 3
exposent : 17
**********************************************************************
il est temps de lui envoyer votre precieux secret
**********************************************************************
appuyer sur entrer
donner un secret de 25 caractères au maximum : J'aime RSA!
**********************************************************************
```

```
voici le message découpé en blocs et bourré
 ['0002f0004a276169', '0002e5006d652052', '000209e800534121']
voici les blocs en décimal
 [826833988182377, 814739951525970, 573841995940129]
voici les blocs chiffré avec la clé publique d'Alice  [204265207495217990778264453841791881456366654928629366875656724398300181930050304974783741097362115
009661707461234747460989682576681972154738023108206023128932356745699366264636309742844845358322066507763266855456, 174770716024215424031382497076010779527541547145120974956965306757631334723113768940004568731
**********************************************************************
On utilise la fonction de hashage SHA256 pour obtenir le hash du message J'aime RSA!
voici le hash en nombre décimal
23781725608284159056811986338664941308421228924260814630329263743155346699937164652014755032007263022108465216
voici la signature avec la clé privée de Bob du hachis
10685675401297300082307752206033611679911767741166427231444827002035654837536623756169379705225728151648603375410605200 2
**********************************************************************
Bob envoie
        1-le message découpé en blocs chiffrés avec la clé public d'Alice
        2-et le hash signé
 10685675401297300082307752206033611679911767741166427231444827002035654837536623756169379705225728151648603375410605200 2
**********************************************************************
appuyer sur entrer
```

```
**********************************************************************
Alice déchiffre les blocs chiffré de manière sécurisée avec le CRT ce qui donne
[826833988182377, 814739951525970, 573841995940129]
Alice converti les blocs en hexadécimal
['0002f0004a276169', '0002e5006d652052', '000209e800534121']
J'aime RSA!
**********************************************************************
Alice déchiffre la signature de Bob
 10685675401297300082307752206033611679911767741166427231444827002035654837536623756169379705225728151648603375410605200 2
 ce qui donne  en décimal
23781725608284159056811986338664941308421228924260814630329263743155346699937164652014755032007263022108465216
Alice vérifie si elle obtient la même chose avec le hash de
 J'aime RSA!
23781725608284159056811986338664941308421228924260814630329263743155346699937164652014755032007263022108465216
La différence = 0
Felicitation!!

Alice : Bob m'a envoyé :  J'aime RSA!
Tout se passe trés bien
```

Conclusion

Ce projet ma permit de mettre en pratique les connaissances acquises en cours de l'UV RS40. Il était question pour nous de mettre en place une communication entre deux utilisateurs (Alice et Bob) en utilisant le chiffrement asymétrique RSA et PKCS#1 v1.5. Au sortir de ce Tp nous avons pu renforcer notre compréhension de ces 2 modes de chiffrement à travers les difficultés rencontrées. Lesdites difficultés concernaient majoritairement l'augmentation de la taille des messages échangés. Pour ce qui est de PKCS#1 v1.5, il était intéressant de nous confronter à la méthodologie employée dans ce mode.