RS40 TP1 /yann derré

Préparations

Le programme fourni possède de base un set de premiers p et q. Il faut les remplacer, les hash du message générés étant trop grands pour la clé. La fonction MD5 n'est également plus fiable, j'ai donc décidé de la remplacer par l'algorithme de hachage SHA-256.
Une fois les premiers remplacés, et MD5 remplacé, nous pourrons générer normalement en suivant l'algo RSA.

I - Implémentation RSA

a) home_mod_exponent

```
def home_mod_expnoent(x, y, n):
    result = 1
    while y > 0:
        if y % 2 == 1:
            result = (result * x) % n
        x = (x * x) % n
        y = y // 2
    return result
```

On calcule l'exponentiation modulaire, c'est à dire x^y [n]. Pour cela, on divise y par 2, et on met à jour x en l'élevant au carré (modulo n) à chaque itération.

b) Algorithme d'Euclide étendu

```
def home_ext_euclide(y,b): #algorithme d'euclide étendu pour la recherche de l'exposant secret
    if y==0: #si y=0 alors pgcd(b,0)=b et x=0 et y=1
        return b,0,1
    else: #sinon
        pgcd,x1,y1 = home_ext_euclide(b%y, y) #on applique l'algorithme d'euclide étendu à b%y et y
        x = y1 - (b//y) * x1 #on détermine x
        y = x1 #on détermine y
        return pgcd,x,y #on renvoie le pgcd, x et y
```

L'algorithme d'Euclide étendu est une implémentation standard de l'algo vu en cours. Elle nous permet de trouver l'exposant secret utilisé pour le chiffrement. On peut ainsi obtenir l'inverse du modulo.

c) Extra

Les premiers de Bob sont générés à l'exécution via un générateur de grand nombre premiers :

```python
def generate_large_prime():
    while True:
        p = random.getrandbits(1024)
        if is_prime(p):
            return p

def is_prime(n, k=5):
    if n <= 3:
        return n == 2 or n == 3
    for i in range(k):
        a = random.randint(2, n-2)
        x = pow(a, n-1, n)
        if x != 1:
            return False
    return True
```

```
(base) yannderre@MacBook-Pro-2 [22:57:27] [~/TP1 RS40 ] [main *]
-> % /opt/homebrew/bin/python3.10 "/Users/yannderre/TP1 RS40 /RSA CRT.py"
11812299266315789576886069703725651726593230374059744536446506446763578007090928995582409899029183520722161506361112443861679651336833366644772010526998058591423900913832186207840868728555504473234453653487511807074243934936872909632940872371115734423933433598749668925745807299495386374815649466023784921017608009113635700069452064937854657520174622307606203814967339382786788023374737587353635635911412941832362842134338854976343404818651210053931288395052870462037693387398804098048334268197018883193529631888875638528370691933640910449230808869524269043586031002245450773696743093528552312557338157674291095446112
```

P et Q de Bob, générés dynamiquement

La sécurité théorique de ma fonction repose sur la libraire *random* de Python et si sa génération est *truly random.* Des méthodes existent pour influencer la génération pseudo aléatoire, si la seed utilisée pour la génération peut être reproduite/récupérée (https://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator)

II - CRT

Chinese Remainder Theorem nous permet de réduire le temps de calcul de l'exponentiel modulaire au déchiffrage, lorsque p et q sont connus. On peut facilement obtenir le message original, comparé à la méthode standard.

```python
# Compute CRT Alice parameters
dp = da % (x2a - 1)
dq = da % (x1a - 1)
qinv = home_mod_inv(x1a, x2a)
print(qinv*x1a%x2a)
```

```python
# Decrypt using CRT
vp = home_mod_expnoent(chif, dp, x2a)
vq = home_mod_expnoent(chif, dq, x1a)
u = (qinv * (vp - vq)) % x1a
r = (vq + u * x1a) % nb
dechif = home_int_to_string(r)
print(dechif)
```

III - Splitting blocks + Padding

Pour utiliser RSA avec des longs messages, nous devons découper en blocs les messages, les padder pour remplir le block, chiffrer individuellement, déchiffrer puis dépadder avant de recoller le message original ensemble.

```
print("Mesage en bytes : ")
num_sec = num_sec.to_bytes((num_sec.bit_length() + 7) // 8, 'little')
print(num_sec)
print("**********************************************************************")
print("Mesage avec block :" + str(blocksize//2) + " bytes")
for i in range(0, len(num_sec), blocksize//2):
    blocklist.append(num_sec[i:i+blocksize//2])
print(blocklist)
```

Pour cela, il est nécessaire de convertir le message en octets. On utilise la méthode **to_bytes**. On le découpe en blocksize//2. J'ai arbitrairement choisi un blocksize de 10.

On s'occupe ensuite du bourrage/padding du message. Il permet d'envoyer des blocks de même longueur à chaque fois, mais une fois déchiffrés nous permettent de retrouver facilement le message initial tout en empêchant l'inférence du message via une attaque par taille d'extension (https://en.wikipedia.org/wiki/Padding_(cryptography) + https://en.wikipedia.org/wiki/Length_extension_attack)

```
# Ajout du padding aux blocs
paddinglist = []
for i in range(len(blocklist)):
    length = len(blocklist[i])
    padding = b'\x00\x02' + bytes([random.getrandbits(8) for _ in range(blocksize - length - 3)]) + b'\x00' + blocklist[i]
    paddinglist.append(padding)
print(paddinglist)


print("voici le message chiffré avec la publique d'Alice : ")
cryptlist = []
for i in range(0, len(paddinglist)):
    chif=home_mod_expnoent(int.from_bytes(paddinglist[i], byteorder='little'), ea, na)
    cryptlist.append(chif.to_bytes((chif.bit_length() + 7) // 8, 'little'))
print(chif)

print("**********************************************************************")
```

Le bourrage est constitué de plusieurs parties: ***b'\x00\x02'***, le flag de type de bourrage, une generation d'octets aléatoire entre le prefixe et nos données de longueur ***blocksize - length - 3***, notre flag de séparation ***b'\x00' (un byte nul), puis enfin*** *notre bloc* de données. Le bloc paddé obtenu est stocké dans paddinglist, puis chiffré.

```
dechiflist = []
message=""
for chif in cryptlist:
    vp = home_mod_expnoent(int.from_bytes(chif, byteorder='little'), dp, x2a)
    vq = home_mod_expnoent(int.from_bytes(chif, byteorder='little'), dq, x1a)
    u = (qinv * (vp - vq)) % x1a
    r = (vq + u * x1a) % nb
    dechif=r.to_bytes((r.bit_length() + 7) // 8, 'little')
    i=len(dechif)
    while dechif[i-1]!=0:
        i=i-1
    message=message + "".join(dechif[i:].decode())

print("Message déchiffré :")
print(message)
print("**********************************************************************")
```

Au déchiffrage, nous effectuons l'étape inverse, on utilise le CRT pour **déchiffrer chaque block** de **cryptlist** (l'array où nos blocks chiffrés précédemment sont stockés).

On reconvertit notre résultat déchiffré en octets pour retravailler dessus. Le padding se terminant avec un octet nul (de séparation), on peut itérer jusqu'à le trouver et atteindre le conteneur de données. On peut maintenant extraire notre block et reconsituter blocklist (le message initial).

Conclusion

Ce TP m'aura permis de gagner une maîtrise du fonctionnement de RSA. Les notions du cours pouvant être abstraites, il est pratique de les utiliser en pratique.

Cela permet également de me rendre compte des étapes de chiffrement et déchiffrement ainsi que les différentes optimisations qu'une messagerie chiffrée e2e peut utiliser (Signal par exemple, même si le protocole de chiffrage est différent).