TP6-Python

March 7, 2022

1 TP6 : Mise en accord par Diffie-Hellman, dérivation de clé et chiffrement à clé secrète

Installez tout d'abord le paquet `cryptography` qui est une surcouche `Python` à la librairie `OpenSSL`. Le but de ce TP est de réaliser une mise en accord par DH pour obtenir une clé maître, nommée `MasterKey`. La clé maître sera ensuite dérivée avec `pbkdf2`pour obtenir la clé de session `SessionKey` ainsi qu'une valeur initiale `IV`. La clé de session et la valeur initiale seront ensuite utilisées pour réaliser un chiffrement `AES` par blocs en mode `CTR`, plus simple à utiliser. Le TP est à réaliser préférablement en binôme et les messages seront échangés sur `Discord` entre les deux participants du binôme (Alice et Bob).

1.1 1. Mise en accord par Diffie Hellman

On s'inspire largement de la documentation en la complétant pour une approche opérationnelle. Prenez tout d'abord connaissance du code d'exemple ci-dessous:

[1]:
```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.backends import default_backend

# Generate some parameters. These can be reused.
parameters = dh.generate_parameters(generator=2, key_size=1024,
                                    backend=default_backend())

# Generate a private key for use in the exchange.
private_key = parameters.generate_private_key()

# In a real handshake the peer_public_key will be received from the
# other party. For this example we'll generate another private key and
# get a public key from that. Note that in a DH handshake both peers
# must agree on a common set of parameters.
peer_public_key = parameters.generate_private_key().public_key()
shared_key = private_key.exchange(peer_public_key)
```

Exercice 1 Dans le monde réel, les paramètres DH sont à récupérer et à transmettre à l'autre partie. On suppose pour la suite qu'Alice est l'initiatrice de l'échange. La méthode `dh.generate_parameters` construit un objet qui regroupe le grand entier premier p et g le générateur du groupe. Retrouvez ces deux entier au moyen de leurs accesseurs `p` et `g` et de la méthode

d'instance `parameter_numbers()`. Pour tester, rangez-les dans deux variables, `p` et `g` pour réaliser les tests.

[2]:
```
```

```
paramètre p 15285041406029521219526334911862515329401508222585285023158593367363
43143933961486960434975845741144135382651926617092691267003388009684802021709371
82961124862042655430002649026865959242387777271663768036035133454683874184542677
62428357384808504623028998504605692662569613414706338829104788164614880838216984
7
paramètre g 2
```

Les entiers p et g peuvent être transmis en clair sur le canal (ici dans un message `Discord`) mais doivent ensuite être regroupés dans un objet `parameters` comme décrit dans le code exemple de la documentation et recopié ci-dessous

[3]:
```
pn = dh.DHParameterNumbers(p, g)
parameters = pn.parameters()
```

Exercice 2 Reconstruisez les paramètres DH de Bob et affichez-les.

[ ]:
```
```

Une fois les paramètres p et g partagés, Alice doit construire sa clé privée et sa clé publique. Elle partage ensuite sa clé publique sur le canal en utilisant la méthode d'instance `public_numbers()` et l'accesseur `y`.

Exercice 3 Construisez la clé privée d'Alice, sa clé publique puis affichez la clé publique qui sera transmise sur le canal.

[ ]:
```
```

Exercice 4 Construisez la clé privée de Bob, sa clé publique puis affichez la clé publique qui sera transmise sur le canal.

[ ]:
```
```

Exercice 5 retrouvez la clé partagée par Alice et Bob en faisant les calculs du côté d'Alice et du côté de Bob. Attention, il faut rassembler les informations transmises dans un objet en s'inspirant du code suivant:

[ ]:
```
pn = dh.DHParameterNumbers(p, g)
parameters = pn.parameters()
peer_public_numbers = dh.DHPublicNumbers(y, pn)
peer_public_key = peer_public_numbers.public_key()
```

[ ]:
```
```

2 2. Dérivation de clé

Alice et Bob partagent maintenant un secret, `MasterKey`. Il vont devoir utiliser un algorithme de dérivation de clé pour construire: - la clé AES de 256 bit (ou 32 octets) - une valeur initiale de 128 bits (ou 16 octets)

On s'inspire de la documentation

[17]:
```
import os
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
# Salts should be randomly generated
salt = os.urandom(16)
# derive
kdf = PBKDF2HMAC(
    algorithm=hashes.SHA256(),
    length=32,
    salt=salt,
    iterations=390000,
)
key = kdf.derive(b"my great password")
print(key)
# verify
kdf = PBKDF2HMAC(
    algorithm=hashes.SHA256(),
    length=32,
    salt=salt,
    iterations=390000,
)
print(kdf.verify(b"my great password", key))
```

```
b"\xc3\x8c\xe0\x88T\xbe\xa9\xd3\x8a\x0f\xa5\xa6\x80P.\xc5}\x8dV|\xa1\xe1m'\x9b\x
d10X7\xc9m}"
None
```

Exercice 6 Dérivez `MasterKey` du côté d'Alice et du côté de Bob pour obtenir le clé de session `SessionKey` de 32 octets. **Attention** il y a un piège !

[ ]:
```

```

Exercice 7 Dérivez `MasterKey` du côté d'Alice et de Bob pour obtenir la valeur initiale `IV` de 16 octets

[ ]:
```

```

A présent, Alice et Bob disposent des même paramètres et peuvent enfin échanger des messages chiffrés par `AES-256` en mode `CTR` en utilisant `SessionKey`et `IV`.

3 3. Chiffrement des messages

En vous inspirant du code ci-dessous provenant de la page, écrivez une fonction `chiffre()` et une fonction `dechiffre()` qui travaillent en bytecode pour le clair et en `ASCII` pour les chiffrés au moyen de la fonction `b2a_hex` de `binascii`. La méthode `finalize` sert à gérer le dernier bloc à traiter. Attention, il faut d'abord réaliser le bourrage du clair au bon format.

[49]:
```
import os
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import padding
import sys

import binascii

def go_encrypt(msg,method,mode):
  cipher = Cipher(method, mode)
  encryptor = cipher.encryptor()
  ct = encryptor.update(msg) + encryptor.finalize()
  return (ct)


def go_decrypt(ct,method,mode):
  cipher = Cipher(method, mode)
  decryptor = cipher.decryptor()
  return (decryptor.update(ct) + decryptor.finalize())

def pad(data,size=128):
  padder = padding.PKCS7(size).padder()
  padded_data = padder.update(data)
  padded_data += padder.finalize()
  return(padded_data)

def unpad(data,size=128):
  padder = padding.PKCS7(size).unpadder()
  unpadded_data = padder.update(data)
  unpadded_data += padder.finalize()
  return(unpadded_data)


key = os.urandom(32)
iv = os.urandom(16)
msg=b"Hello"


print ("Message:\t",msg.decode())
print ("Key:\t",binascii.b2a_hex(key))
print ("IV:\t",binascii.b2a_hex(iv))
```

```
padded_data=pad(msg)


print ("=== AES CBC === ")
cipher=go_encrypt(padded_data,algorithms.AES(key), modes.CBC(iv))

plain=go_decrypt(cipher,algorithms.AES(key), modes.CBC(iv))
data=unpad(plain)

print ("Cipher: ",binascii.b2a_hex(cipher))
print (f"Decrypted: {data.decode()}")

cipher=go_encrypt(padded_data,algorithms.AES(key), modes.CBC(iv))

print ("=== AES CFB === ")
cipher=go_encrypt(padded_data,algorithms.AES(key), modes.CFB(iv))

plain=go_decrypt(cipher,algorithms.AES(key), modes.CFB(iv))
data=unpad(plain)

print ("Cipher: ",binascii.b2a_hex(cipher))
print (f"Decrypted: {data.decode()}")

print ("=== AES CTR === ")
cipher=go_encrypt(padded_data,algorithms.AES(key), modes.CTR(iv))

plain=go_decrypt(cipher,algorithms.AES(key), modes.CTR(iv))
data=unpad(plain)

print ("Cipher: ",binascii.b2a_hex(cipher))
print (f"Decrypted: {data.decode()}")
```

```
Message:         Hello
Key:     b'07dc24a4d54e8f434a51104d8736893f9c492c8cbf3c2854bda1104cd9d2cf92'
IV:      b'aff4b1b20b55bfd1259c4d7003d1deed'
=== AES CBC ===
Cipher:  b'e5560bbb44361d17b5c9e9997afb40d4'
Decrypted: Hello
=== AES CFB ===
Cipher:  b'5da6e593d3785cb1a0c6c8d028e4f2e1'
Decrypted: Hello
=== AES CTR ===
Cipher:  b'5da6e593d3785cb1a0c6c8d028e4f2e1'
Decrypted: Hello
```

Exercice 8 Ecrivez la fonction `chiffre()` qui prend en entrée un clair, la clé et l'IV et retourne un chiffré.

[59]:

[60]:
```
crypto=chiffre(b'Test petit message',SessionKey,IV)
print(binascii.b2a_hex(crypto))
```

```
b'e497607136e268a23c2baec9c9407bf07608f8ae8b86a76da4fb737415623366'
```

Exercice 9 Ecrivez la fonction `dechiffre()`qui prend en entrée un chiffré, la clé et l'IV et qui retourne le clair.

[61]:

[62]:
```
plain=dechiffre(crypto,SessionKey,IV)
print (f"Decrypted: {plain.decode()}")
```

```
Decrypted: Test petit message
```

A présent vous êtes en mesure d'échanger entre vous des messages chiffrés sur `Discord`. Vous pouvez vous aider de fonctions pour générer les messages qu'Alice envoie à Bob (et réciproquement) ainsi que sur les fonctions écrites précédemment.

[ ]: