Tema 3 - Cifrado de flujo con ChaCha20

En Python tenemos dos posibilidades para probar criptografía: el paquete `PyCryptodome` y el paquete `cryptography`. Ambos son opciones válidas. Las prácticas de este curso las haremos con PyCryptodome. Puedes encontrar la ayuda en: https://pycryptodome.readthedocs.io/en/latest/ (https://pycryptodome.readthedocs.io/en/latest/)

Si no lo tienes instalado: `python3 -m pip install pycryptodome`

Empezamos importando lo que vamos a necesitar:

In [2]:
```
from base64 import b64encode, b64decode
from Crypto.Cipher import ChaCha20
from Crypto.Random import get_random_bytes
```

Cifrado y envío de datos

Los módulos de criptografía suelen necesitar una etapa inicial de configuración. Cada módulo se configura a su manera. A continuación encontrarás la etapa de configuración de ChaCha20 para PyCryptodome.

Fíjate que la clave se crea al azar con algoritmos criptográficos `Crypto.Random.get_random_bytes()`: **es fundamental que las claves sean totalmente aleatorias y creadas también con algoritmos criptográficos**. Como habrás visto en el ejercicio "creando azar" de este mismo tema, no todas las funciones de creación de azar son válidas.

In [3]:
```
key = get_random_bytes(32)
cipher_emisor = ChaCha20.new(key=key, nonce=None)
print('Longitud de la clave: {} bits'.format(8 * len(key)))
```

```
Longitud de la clave: 256 bits
```

En PyCryptodome el *nonce* se puede pasar al algoritmo durante la configuración. Si, como en este caso, no se pasa *nonce* durante la creación, la librería crea un *nonc* al azar que podemos recuperar. Si decides crear tú el *nonce*, recuerda que también tiene que ser un número aleatorio creado con algoritmos criptográficos, igual que la clave.

In [4]:
```
nonce = b64encode(cipher_emisor.nonce)
print('Longitud del nonce: {} bits'.format(8 * len(cipher_emisor.nonce)))
print(nonce)
```

```
Longitud del nonce: 64 bits
b'G6ltoknWs54='
```

El emisor cifra el mensaje `Atacaremos al amanecer` y envía al receptor `result`, es decir, tanto como mensaje cifrado como el *nonce*. Fíjate: el nonce se puede enviar por un canal inseguro, así que se asume que el atacante lo conocerá.

Observa que el resultado lo codificamos en Base64 (https://es.wikipedia.org/wiki/Base64 (https://es.wikipedia.org/wiki/Base64)). Aunque no es necesario, sí que es común hacerlo así porque algunos protocolos (correo electrónico, JSON...) solo puede enviar caracteres imprimibles. No pierdes ni ganas seguridad si decides usar o no Base64, es más una exigencia de tu sistema de comunicaciones. Fíjate que he usado la expresión "codificamos en Base64", no ciframos. Base64 es un algoritmo de codificación de bytes, no tiene claves, cualquier lo puede codificar y decodificar y por tanto no es un cifrado.

In [5]:
```
plaintext = b'Atacaremos al amanecer'
ciphertext = cipher_emisor.encrypt(plaintext)
ct = b64encode(ciphertext)
result = {'nonce':nonce, 'ciphertext':ct}
print(result)
```

```
{'nonce': b'G6ltoknWs54=', 'ciphertext': b'qcObonyni22l9w4wtbj1vxnQhkSpbg=='}
```

Recepción y descifrado

El receptor toma el *nonce* y el *ciphertext*, primero decodifica el base64, configura el *cipher* con la clave que conoce (ya veremos cómo la conoce en el tema 4 y 5) y el *nonce* que ha recibido y descifra:

In [6]:
```
received_nonce = b64decode(result['nonce'])
```

```
received_ciphertext = b64decode(result['ciphertext'])
cipher_receptor = ChaCha20.new(key=key, nonce=received_nonce)
plaintext = cipher_receptor.decrypt(received_ciphertext)
print(plaintext)
```

```
b'Atacaremos al amanecer'
```

Siguientes mensajes: sincronización entre ciphers

Supongamos que el usuario vuelve a enviar el mismo mensaje, con el mismo cipher (fíjate que no volvemos a definir `cipher_emisor` : lo estamos reaprovechando)

In [7]:
```
plaintext = b'Atacaremos al amanecer'
ciphertext = cipher_emisor.encrypt(plaintext)
ct = b64encode(ciphertext)
result = {'nonce':nonce, 'ciphertext':ct}
print(result)
```

```
{'nonce': b'G6ltoknWs54=', 'ciphertext': b'HMwsXjNsRRejnGX/zLZyTEvm7tI7BQ=='}
```

Fíjate: estamos cifrando el mismo mensaje con el mismo nonce... pero el ciphertext es diferente. ¿Recuerdas que nunca se debe cifrar el mismo texto con la misma clave? ChaCha20 nos ayuda a que no lo hagamos, ni siquiera por equivocación, mediante el uso de un contador.

Supongamos que el receptor crea un nuevo cipher, con la misma configuración de key y nonce, e intenta descifrar:

In [8]:
```
received_nonce = b64decode(result['nonce'])
received_ciphertext = b64decode(result['ciphertext'])
cipher_receptor = ChaCha20.new(key=key, nonce=received_nonce)
plaintext = cipher_receptor.decrypt(received_ciphertext)
print(plaintext)
```

```
b'\xf4{\xd6\x9f.\xb9\xab\x17i\x18K\xae\x15.\xe6\x9e3X\r\xf5\xf7\x19'
```

¿Qué ha pasado? ¿Por qué no se descifra? Recuerda que ChaCha20 tiene un contador adicional interno. Es decir: **emisor y receptor tienen que estar sincronizados** Es decir: para descifrar el byte número 22 tenemos que decirle al receptor que han pasado 22 bytes antes, aunque no los haya visto.

(nota: 22 es el tamaño en bytes de la cadena "Atacaremos al amanecer", que fue el contenido del primer mensaje)

Si volvemos a intentar descifrar, ahora sí que podemos hacerlo:

In [9]:
```
cipher_receptor.seek(22)
plaintext = cipher_receptor.decrypt(received_ciphertext)
print(plaintext)
```

```
b'Atacaremos al amanecer'
```

PyCryptodome y todos los demás están sincronizados siempre que descifremos los mismos bytes que hemos cifrado desde que se han creado los dos ciphers, el de emisión y el de recepción.

Si alguno de los dos pierde la sincronización (por ejemplo, porque se reinicia), entonces es necesario volver a sincronizarlos con un "seek": "ya envié XX bytes aunque no los hayas visto, mueve el estado a esta posición"

Poder volver a sincronizar los dos streams es una enorme ventaja de ChaCha20 y eso es por el parámetro `pos` autoincremental que forma parte de la matriz de estado. No todos los algoritmos permiten sincronizar los flujos si se pierde la sincronización.

In [ ]:
```
```