3 - Modos de cifrado AES

April 2, 2023

1 Modos de cifrado AES

AES se puede utilizar en varios modos. Vamos a verlos en esta actividad.

Vamos a crear: - un mensaje de 128 bits, el tamaño de bloque de AES. - una clave k de 128 bits que usaremos durante todo el ejercicio.

[1]:
```
!python3 -m pip install pycryptodome
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from base64 import b64encode, b64decode

m = b'abcdefghabcdefgh'
k = get_random_bytes(16)
print(f'Mensaje: "{m}" Tamaño={len(m) * 8} bits')
print(f'Clave: {b64encode(k)} Tamaño={len(k) * 8} bits')
```

```
Requirement already satisfied: pycryptodome in
/home/gattes/.cache/pypoetry/virtualenvs/jupyter-notebooks-QRvsWska-
py3.10/lib/python3.10/site-packages (3.17)

[notice] A new release of pip is
available: 23.0 -> 23.0.1
[notice] To update, run:
pip install --upgrade pip
Mensaje: "b'abcdefghabcdefgh'" Tamaño=128 bits
Clave: b'qIfY1jvouUHqM+8CwJREug==' Tamaño=128 bits
```

1.1 Modo ECB

Ciframos dos veces el mismo mensaje.

Observa que no hay memoria, y que cifrar dos veces el mismo mensaje con la misma clave produce el mismo texto cifrado.

[8]:
```
cipher = AES.new(k, AES.MODE_ECB)
c1 = cipher.encrypt(m)
c2 = cipher.encrypt(m)
print(b64encode(c1))
print(b64encode(c2))
```

```
b'6DPHXhGKp2YBlCLQQO5MJg=='
b'6DPHXhGKp2YBlCLQQO5MJg=='
```

[9]:
```
decipher = AES.new(k, AES.MODE_ECB)
m1 = decipher.decrypt(c1)
m2 = decipher.decrypt(c2)
print(m1)
print(m2)
```

```
b'abcdefghabcdefgh'
b'abcdefghabcdefgh'
```

1.2 Modo CBC

Ciframos dos veces el mismo mensaje. Observa que: - tenemos que crear un IV (Vector de Inicialización), y que este IV se lo tenemos que enviar al receptor. El envío del IV puede ser en claro - Ahora los dos cifrados son diferentes, a pesar de que estamos cifrando el mismo mensaje. ¿Por qué sucede eso?

Respuesta del alumno: Porque en modo `CBC` cada bloque depende del anterior. El IV se inserta inicialmente junto con el mensaje en texto plano (en una XOR) y eso va a ser la entrada del primer bloque, luego se encripta usando AES con la key `k` predefinida. En los subsecuentes bloques se reutiliza la salida del bloque N0 junto con el mensaje en texto plano y se realiza una XOR, y asi en los subsecuentes bloques. Esto hace que aunque la entrada en texto plano sea siempre la misma, la salida del texto cifrado va a ser siempre diferente.

[11]:
```
iv = get_random_bytes(16)
cipher = AES.new(k, AES.MODE_CBC, iv=iv)
c1 = cipher.encrypt(m)
c2 = cipher.encrypt(m)
print(b64encode(c1))
print(b64encode(c2))
```

```
b'pRRa7Flar7VjZz7Og2qdvA=='
b'KqbtmK6t0t2fGS6xaUqYqg=='
```

Descifrado: necesita la clave y el IV. La clave es secreta y el receptor tiene que haberla recibido por un canal secreto (lo veremos) pero el IV puede recibirse sin protección al inicio de la comunicación.

[12]:
```
decipher = AES.new(k, AES.MODE_CBC, iv=iv)
m1 = decipher.decrypt(c1)
m2 = decipher.decrypt(c2)
print(m1)
print(m2)
```

```
b'abcdefghabcdefgh'
b'abcdefghabcdefgh'
```

2 Ejercicios (opcional)

- ¿Puedes programar el modo CBC a partir del modo ECB? ECB es la caja AES básica, así que es posible programar (¡como ejercicio solamente!) el modo CBC como composición de ECB
- ¿Puedes programar los demás modos?

Ejemplo de solución (solo parte de cifrado) de la primera pregunta. Observa que el resultado es el mismo de antes al cifrar `m` en modo CBC

[97]:
```
from Crypto.Util.strxor import strxor

class AES_CBC():
    def __init__(self, k, iv):
        self.iv = iv
        self.cipher = AES.new(k, AES.MODE_ECB)
    def encrypt(self, msg):
        # Primero hacemos XOR del mensaje con el IV que tenemos
        m = strxor(msg, self.iv)
        c = self.cipher.encrypt(m)
        # Para la siguiente ronda, el IV es el propio texto cifrado
        self.iv = c
        return c

k = get_random_bytes(16)
iv = get_random_bytes(16)
mycbc = AES_CBC(k, iv)

print("Mensaje en texto plano: {}".format(m.decode("utf-8")))

m1 = mycbc.encrypt(m)
m2 = mycbc.encrypt(m)

print(b64encode(m1))
print(b64encode(m2))
```

```
Mensaje en texto plano: abcdefghabcdefgh
b'mfPjR7PBvPX5UlcvNi9PVg=='
b'S4eOXsPXunn+rR1NXHDuDQ=='
```

3 Respuesta del alumno

3.1 Modo CBC

- El siguiente es el ejercicio completo para encriptar y desencriptar utilizando el modo CBC de AES, a partir del modo ECB

[98]:
```
from Crypto.Util.strxor import strxor
```

```
class AES_CBC():
    def __init__(self, k, iv):
        self.iv = iv
        self.cipher = AES.new(k, AES.MODE_ECB)
    def encrypt(self, msg):
        # Primero hacemos XOR del mensaje con el IV que tenemos
        m = strxor(msg, self.iv)
        c = self.cipher.encrypt(m)
        # Para la siguiente ronda, el IV es el propio texto cifrado
        self.iv = c
        return c
    def decrypt(self, encrypted_msg):
        out = self.cipher.decrypt(encrypted_msg)
        c = strxor(out, self.iv)
        self.iv = c
        return c


m = b'abcdefghabcdefgh'
print("Mensaje en texto plano: {}".format(m.decode("utf-8")))

k = get_random_bytes(16)
iv = get_random_bytes(16)
mycbc = AES_CBC(k, iv)

m1 = mycbc.encrypt(m)
print("Mensaje encriptado m1: {}".format(b64encode(m1)))

mycbcdecrypt = AES_CBC(k, iv)
print("Mensaje desencriptado m1: {}".format(mycbcdecrypt.decrypt(m1)))
```

```
Mensaje en texto plano: abcdefghabcdefgh
Mensaje encriptado m1: b'CJ+efD5Z3mnhJ6EQug6wHg=='
Mensaje desencriptado m1: b'abcdefghabcdefgh'
```

4 Modo OFB

- El siguiente es el ejercicio completo para encriptar y desencriptar utilizando el modo OFB de AES, a partir del modo ECB

[162]:
```
from Crypto.Util.strxor import strxor

class AES_OFB():
    def __init__(self, k, iv):
        self.iv = iv
        self.cipher = AES.new(k, AES.MODE_ECB)
    def encrypt(self, msg):
```

```
        # Primero encriptamos el IV con la key `k`
        c = self.cipher.encrypt(self.iv)
        # Luego hacemos XOR del mensaje en plano con el block cipher que tenemos
        m = strxor(msg, c)
        # Para la siguiente ronda, el IV es el output `m`
        self.iv = c
        # Output es el mensaje encriptado
        return m
    def decrypt(self, encrypted_msg):
        # Encriptamos el IV con la key `k`
        c = self.cipher.encrypt(self.iv)
        # Aplicamos una XOR al mensaje encriptado con el IV encriptado
        m = strxor(c, encrypted_msg)
        # Para la siguiente ronda, el IV es el output de `c`
        self.iv = c
        # El output es el texto en plano
        return m


m = b'abcdefghabcdefgh'
print("Mensaje en texto plano: {}".format(m.decode("utf-8")))

k = get_random_bytes(16)
iv = get_random_bytes(16)
myofb = AES_OFB(k, iv)

m1 = myofb.encrypt(m)
print("Mensaje encriptado m1: {}".format(b64encode(m1)))

myofbdecrypt = AES_OFB(k, iv)
print("Mensaje desencriptado m1: {}".format(myofbdecrypt.decrypt(m1)))
```

```
Mensaje en texto plano: abcdefghabcdefgh
Mensaje encriptado m1: b'7JltYNMu1v3FPEPhi6W1bQ=='
Mensaje desencriptado m1: b'abcdefghabcdefgh'
```

5 Modo CTR

- El modo que resta es el CTR, que es similar al modo OFB dado que hay que hacer un XOR con los IV (en este caso se llaman nonces) encriptados y el texto plano.

6 Padding

¿Qué pasa si tenemos que enviar mensajes más cortos que la longitud de bloque de AES? Entonces tenemos que usar algún algoritmo de padding. Es decir: marcar la longitud del mensaje original.

Con Cryptodome podemos usar las funciones pad y unpad

Observa: no ponemos IV, así que la librería lo escoge aleatorio. En modo CBC solo tenemos que enviar el IV la primera vez,

[106]:
```
from Crypto.Util.Padding import pad, unpad

# mensaje corto
m = b'1234'
cipher = AES.new(k, AES.MODE_CBC)
c = cipher.encrypt(pad(m, AES.block_size))
print({'iv':b64encode(cipher.iv), 'ciphertext':b64encode(c)})
```

```
{'iv': b'9nwK3rIe/trm8OgoxcWVhw==', 'ciphertext': b'vjqrOzPAMT1+jtECvQL/0Q=='}
```

Recepción:

[107]:
```
decipher = AES.new(k, AES.MODE_CBC, cipher.iv)
pt = unpad(decipher.decrypt(c), AES.block_size)
print("The message was: ", pt)
```

```
The message was:  b'1234'
```

¿Qué pasa si no usamos unpad? AES es un cifrado de bloque, así que los mensajes en AES tienen obligatoriamente un tamaño igual al bloque AES (128 bits), así que vemos el *padding* que sobra. Las función *unpad()* nos hubiese cortado esos bytes sobrantes.

(Observa: tenemos que volver a recrear el decipher, porque tiene memoria y queremos volver a descifrar el mismo mensaje)

[108]:
```
decipher = AES.new(k, AES.MODE_CBC, cipher.iv)
pt = decipher.decrypt(c)
print(f"The message was: {pt} (longitud {len(pt) * 8} bits)")
```

```
The message was: b'1234\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c'
(longitud 128 bits)
```