```
Acuerdo de claves Diffie-Hellman
        Recuerda, el protocolo es:
         1. Acuerdan g y p primos entre sí
         2. Escogen números en secreto a y b
         3. Se envian entre ellos:
            ullet \ Alice 
ightarrow Bob: A=g^a \mod p
            ullet \ Bob 
ightarrow Alice: B=g^b \mod p
         4. Calculan en secreto:
            • Alice: s = B^a \mod p = g^{ab} \mod p
            • Alice: s = A^b \mod p = g^{ab} \mod p
         5. Y usan s como clave de cifrado un algoritmo simétrico
        A continuación está el código de la librería https://github.com/amiralis/pyDH de Amirali Sanatinia, que es sencillo de leer y entender.
        Aunque no parece haber errores evidentes, es obligatorio utilizar librerías auditadas. Seguiremos esta por su valor educativo, no porque sea recomendable su uso.
In [56]: # Apache License
               Version 2.0, January 2004
            Copyright 2015 Amirali Sanatinia
        """ Pure Python Diffie Hellman implementation
       Source: https://github.com/amiralis/pyDH"""
        import os
       import binascii
        import hashlib
       # RFC 3526 - More Modular Exponential (MODP) Diffie-Hellman groups for
       # Internet Key Exchange (IKE) https://tools.ietf.org/html/rfc3526
       primes = {
              # 1536-bit
              "generator": 2
              # 2048-bit
              "generator": 2
              # 3072-bit
              "generator": 2
              # 4096-bit
              16: {
              "generator": 2
              # 6144-bit
               "generator": 2
              # 8192-bit
              "generator": 2
       class DiffieHellman:
              """ Class to represent the Diffie-Hellman key exchange protocol """
              # Current minimum recommendation is 2048 bit.
              def __init__(self, group=14):
                     if group in primes:
                            self.p = primes[group]["prime"]
                            self.g = primes[group]["generator"]
                     else:
                            raise Exception("Group not supported")
                     self.__a = int(binascii.hexlify(os.urandom(32)), base=16)
               def get_private_key(self):
                     """ Return the private key (a) """
                     return self.__a
               def gen_public_key(self):
                     """ Return A, A = g \wedge a \mod p """
                     # calculate G^a mod p
                     return pow(self.g, self._a, self.p)
              def check_other_public_key(self, other_contribution):
                     # check if the other public key is valid based on NIST SP800-56
                     # 2 <= g^b <= p-2 and Lagrange for safe primes (g^bq)=1, q=(p-1)/2
                     if 2 <= other_contribution and other_contribution <= self.p - 2:</pre>
                            if pow(other_contribution, (self.p - 1) // 2, self.p) == 1:
                                   return True
                     return False
              def gen_shared_key(self, other_contribution):
                     """ Return g ^ ab mod p """
                     # calculate the shared key G^ab mod p
                     if self.check_other_public_key(other_contribution):
                            self.shared_key = pow(other_contribution, self.__a, self.p)
                            return hashlib.sha256(str(self.shared_key).encode()).digest()
                     else:
                            raise Exception("Bad public key from other party")
       Alice
        Vamos a generar primero las claves de Alice
In [57]: alice = DiffieHellman()
       alice_pubkey = alice.gen_public_key()

    Claves de Alice:

            • Clave pública: \{g, p, g^a\}
            Clave privada: a
        Esta es la clave privada de Alice, que es lo que le envía a Bob.
        En realidad g y p suelen escogerse como valores conocidos, así que Alice y Bob ya los tienen y solo se envía g^a
In [ ]: print(f'g={alice.g}\n')
       print(f'p={alice.p}\n')
       print(f'g^a={alice_pubkey}\n')
        Esta es la clave privada de Alice, que nunca sale de su ordenador
In [4]: print(f'a={alice.get_private_key()}\n')
      a=29531639291412788471218505032353322400563739404441334427474931093959470601014
        Bob
        Cálculo de las claves públicas y privadas de bob
 In [ ]: bob = DiffieHellman()
       bob_pubkey = bob.gen_public_key()
       print(f'g={bob.g}\n')
       print(f'p={bob.p}\n')
       print(f'g^b={bob_pubkey}\n')
       Fíjate: la g de Alice y la de Bob es 2. Aunque podría ser cualquier, es común que g sea 2 siempre porque así se aceleran los cálculos. Esto no reduce la seguridad del algoritmo, según los matemáticos
       Alice y Bob: cálculo de la clave compartida
In [6]: alice_sharedkey = alice.gen_shared_key(bob_pubkey)
       print(alice_sharedkey)
      b'`x9c\xb10x\xbe\x85{\x19\x9a\xa8Y\x84\x92c\xc2\x03\xed1\xed\xf9}\xdf\xce\x92\x9d2v*\x95\x9e\xda'
 In [7]: bob_sharedkey = bob.gen_shared_key(alice_pubkey)
       print(bob_sharedkey)
       print(len(bob_sharedkey))
      b'`x9c\xb10x\xbe\x85{\x19\x9a\xa8Y\x84\x92c\xc2\x03\xed1\xed\xf9}\xdf\xce\x92\x9d2v*\x95\x9e\xda'
       Y podemos comprobar que los dos tienen la misma clave compartida
 In [8]: print(bob_sharedkey == alice_sharedkey)
      True
        Ejercicios
       Ya tenemos una "clave compartida", pero aún hay que adaptarla para poder usarla en AES-256 o ChaCha20. ¿Cómo lo harías?
        Demostracion para ChaCha20
In [15]: # Vemos que la sharedkey es de 32b length
       print(f"El largo de la shared key de Bob es: {len(bob_sharedkey)}")
       print(f"El largo de la shared key de Alice es: {len(alice_sharedkey)}")
      El largo de la shared key de Bob es: 32
      El largo de la shared key de Alice es: 32
       Nota de autor: Ya sabemos que ambas keys son iguales, el print es a modo demostrativo
         • Como Bob: Generaremos un mensaje con nuestra shared key bob_sharedkey y se lo enviaremos a Alice
In [26]: !python3 -m pip install pycryptodome
       from base64 import b64encode, b64decode
        from Crypto.Cipher import ChaCha20
        from Crypto.Random import get_random_bytes
       # Nuestra sharedkey es la que vamos a usar en ChaCha20
       cipher_emisor = ChaCha20.new(key=bob_sharedkey, nonce=None)
       print('Longitud de la clave compartida: {} bits'.format(8 * len(key)))
       nonce = b64encode(cipher_emisor.nonce)
       print('Longitud del nonce: {} bits'.format(8 * len(cipher_emisor.nonce)))
       plaintext = b'Hola Alice, soy Bob, te estoy enviando un mensaje usando ChaCha20!'
       ciphertext = cipher_emisor.encrypt(plaintext)
       ct = b64encode(ciphertext)
       result = {'nonce':nonce, 'ciphertext':ct}
       print(f'Nonce: {result["nonce"]}')
       print(f'Mensaje encriptado por Bob: {result["ciphertext"]}')
      Requirement already satisfied: pycryptodome in /home/gattes/.cache/pypoetry/virtualenvs/crypto-83WUx_V0-py3.10/lib/python3.10/site-packages (3.17)
      [notice] A new release of pip is available: 23.0 -> 23.1
      [notice] To update, run: pip install --upgrade pip
      Longitud de la clave compartida: 256 bits
      Longitud del nonce: 64 bits
      Nonce: b'Ew1SWv0YSHY='
      Mensaje encriptado por Bob: b'nwEPv7y28owepB+Y9jRVK08gSAvHH+y7nkvKxLtfb9V2Pb3jTASyAiTdIPDzAubcSq/VRm/+RKHDhs7zsQylHUpQ'
         • Como Alice: Recibimos el mensaje de Bob y usamos la key alice_sharedkey para desencriptarlo
In [35]: received_nonce = b64decode(result['nonce'])
       received_ciphertext = b64decode(result['ciphertext'])
       cipher_receptor = ChaCha20.new(key=alice_sharedkey, nonce=received_nonce)
       plaintext = cipher_receptor.decrypt(received_ciphertext)
       print(f"Mensaje desencriptado por Alice: {plaintext}")
      Mensaje desencriptado por Alice: b'Hola Alice, soy Bob, te estoy enviando un mensaje usando ChaCha20!'
       Demostracion para AES256 (Modo CBC)
         • Como Bob: Generaremos un mensaje con nuestra shared key bob_sharedkey y se lo enviaremos a Alice
In [36]: !python3 -m pip install pycryptodome
       from Crypto.Cipher import AES
       from Crypto.Random import get_random_bytes
       from base64 import b64encode, b64decode
       m = b'Hola Alice, soy Bob, te estoy enviando un mensaje usando AES256!'
       print(f'Mensaje: "{m}" Tamaño={len(m) * 8} bits')
       iv = get_random_bytes(16)
       cipher = AES.new(bob_sharedkey, AES.MODE_CBC, iv=iv)
       c1 = cipher.encrypt(m)
       print(b64encode(c1))
      Requirement already satisfied: pycryptodome in /home/gattes/.cache/pypoetry/virtualenvs/crypto-83WUx_V0-py3.10/lib/python3.10/site-packages (3.17)
      [notice] A new release of pip is available: 23.0 -> 23.1
      [notice] To update, run: pip install --upgrade pip
      Mensaje: "b'Hola Alice, soy Bob, te estoy enviando un mensaje usando AES256!'" Tamaño=512 bits
      b'edASA5XpD7xWqNrpZM/RORVItxjwym0a/fb5A7lC7c/vSFaFD4aJ6pFjNwtn6jJxe4v82zbMfabXdIKCc7W8hw=='
         • Como Alice: Recibimos el mensaje de Bob y usamos la key alice_sharedkey para desencriptarlo
In [37]: decipher = AES.new(alice_sharedkey, AES.MODE_CBC, iv=iv)
       m1 = decipher.decrypt(c1)
       print(m1)
      b'Hola Alice, soy Bob, te estoy enviando un mensaje usando AES256!'
        Nota del autor: El IV (Vector de inicializacion) lo estamos pasando en texto plano en un primer caso pero podriamos encriptarlo usando ChaCha20, enviarlo y una vez compartido utilizar AES256. Por motivos practicos se demuestra con el IV en texto plano.
        ¿Puedes probar a implementar tú D-H sin mirar el código de la librería?
In [55]: import hashlib
        import secrets
       from typing import Tuple
       # Cambiar RANDOM_BITS por otro numero que se desee
        RANDOM_BITS = 16
        def get_prime() -> int:
           Funcion de armado muy basico para obtener
           numeros primos de tamaño RANDOM_BITS
           is_prime = False
           while is_prime == False:
              is_prime = True
              num = secrets.randbits(RANDOM_BITS)
              for n in range(2, num):
                 if num % n == 0:
                     isPrime = False
           return num
        def get_prime_numbers() -> Tuple[int, int]:
           Funcion para obtener 2 numeros primos distintos
           a = 0
           b = 0
           while (a == b):
             a = get_prime()
             b = get_prime()
           return a, b
        def get_shared_key(num: int) -> bytes:
           return hashlib.sha256(str(num).encode()).digest()
       # Obtenemos g y p primos acordados entre Alice y Bob
       # Ambos seran compartidos por canales no seguros
       g, p = get_prime_numbers()
       # Alice calcula su numero `x` en secreto y calcula su `A`
       x = secrets.randbits(RANDOM_BITS)
       A = pow(g, x) \% p
       # Bob calcula su numero `y` en secreto y calcula su `B`
       y = secrets.randbits(RANDOM_BITS)
       B = pow(g, y) \% p
       # Ambos comparten sus A y B respectivamente y calculan la shared key K
       Alice_K = pow(B, x) \% p
       Bob_K = pow(A, y) \% p
       # Finalmente, ambos llegan al mismo numero o shared key `K`
       print(f"Shared Key de Alice (SHA256): {get_shared_key(Alice_K)}")
       print(f"Shared Key de Alice (SHA256): {get_shared_key(Bob_K)}")
       print(f"Shared Key de Alice en plaintext: {Alice_K}")
       print(f"Shared Key de Bob en plaintext: {Bob_K}")
      Shared Key de Alice (SHA256): b's\x85\xca\xe6\x11\t\x9e\x13\xd3v@\xd1\xc0\xee\x8d\x0b~W~\xbb\xd7\xa6+\x04z\x87\x91Y\x03\x8e/\xdf'
      Shared Key de Alice en plaintext: 5329
      Shared Key de Bob en plaintext: 5329
```

Nota 1: Se utilizaron numeros bajos solo a modo demostrativo, tranquilamente se podrian haber utilizado numeros exponencialmente mas altos siguiendo las convenciones de RFC7919 (ver siguiente apartado)

Los parámetros p y g de la librería son muy antiguos (RFC3526). ¿Puedes buscar otros más modernos?

Hoy en dia se utilizan parametros mas modernos como los de las tablas publicas RFC7919 de 2016

Alli se ve como definen g = 2 y estandares de calculo para p, por ejemplo:

Nota 2: La libreria secrets de python utiliza los modulos de seguridad del SO y es confiable. Si bien la randomizacion puede fallar y todos los random numbers fueran a ser iguales, se pueden subir el valor de RANDOM_BITS a mas grandes. Pero por conveniencia y solo para demostrar el caso se utiliza un numero bajo de bits