```
# OTR
# By Amanda Flote & Olivia Mattsson

import hashlib, socket, random, math, binascii

sharedSecret = "eitn41 <3"
p = int("FFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC74020BBEA63B139B22514A08798E3404DDEF9519B3CD3A431B302B0A6DF25F14374FE
g = 2



def recv():
    return soc.recv(4096).decode('utf8').strip()

def recv_int():
    return int(soc.recv(4096).decode('utf8').strip(), 16)

def send(x):
    soc.send(format(x, 'x').encode('utf8'))

def main():
    # Diffie Hellman exchange:
    g_x1 = recv_int()
    x2 = random.randrange(2,p)
    g_x2 = pow(g, x2, p)
    send(g_x2)
    print ('\nsent g_x2:', soc.recv(4096).decode('utf8').strip())
    #Generating shared key
    sharedKey = pow(g_x1, x2, p)
    sharedKey = I20SP(sharedKey, (sharedKey.bit_length() + 7) // 8)

    # SMP OTR protocol:
    g1_a2 = recv_int()
    b2 = random.randrange(2,p)
    g1_b2 = pow(g, b2, p)
    send(g1_b2)
    print ('\nsent g1_b2:', soc.recv(4096).decode('utf8').strip())

    g1_a3 = int(soc.recv(4096).decode('utf8').strip(), 16)
    b3 = random.randrange(2,p)
    g1_b3 = pow(g, b3, p)
    send(g1_b3)
    print ('\nsent g1_b3:', soc.recv(4096).decode('utf8').strip())

    Pa  = recv_int()
    g2 = pow(g1_a2, b2, p)
    g3 = pow(g1_a3, b3, p)
    P,Q = calculatePQ(g, g2, g3, sharedKey, p)

    send(P)
    print ('\nsent P:', soc.recv(4096).decode('utf8').strip())

    Q_a  = recv_int()
    send(Q)
    print ('\nsent Q:', soc.recv(4096).decode('utf8').strip())

    Ra = recv_int()
    Q_b = inverse(Q)
    Q_c = Q_a * Q_b
    Rb = pow(Q_c, b3, p)
    send(Rb)
    print ('\nsent Rb:', soc.recv(4096).decode('utf8').strip())
    print ('\nauthentication:', soc.recv(4096).decode('utf8').strip())

    #Send message:
    message = "e21def8129b64b0899ea941d4597ca9f27d98228"
    padding = bytes(len(sharedKey) - len(message) // 2)
    message = padding + binascii.unhexlify(message)
    encryptedMessage = xor(message, sharedKey)
    send(int.from_bytes(encryptedMessage, 'big'))
    receivedMessage = recv()
    print ('\nreceived back:', receivedMessage)


def xor(message, sharedKey):
    # Found from this page: https://nitratine.net/blog/post/xor-python-byte-strings/
    return bytes(a ^ b for a, b in zip(message, sharedKey))

def calculatePQ(gen1, gen2, gen3, sharedKey, p):
    b = random.randrange(2,p)
    y = int(sha_hash(sharedKey + sharedSecret.encode('utf8')), 16)
    P= pow(gen3, b, p)
    Q= pow(gen1, b, p) * pow(gen2, y, p)
    return P,Q

# I20SP function
def I20SP (x, xLen):
    xLen = math.ceil(xLen)
    # If x >= 256^xLen, output "integer too large" and stop.
    if x >= 256**xLen:
        print("Integer too large")
        return
    # Make a representation of X in base 256:
    X = bytearray(xLen)
    for i in range(0,xLen-1):
        # For each value on i, we want to find out what we need to multiply 256^(xLen-i) with to get the part
        X[i] = x // (256**(xLen-1-i))
        x = x % (256**(xLen-1-i))
    X[xLen - 1] = x
    # Return the base 256 X value:
    return bytes(X)


def inverse(val):
    # We need to find x so that x*val % p is equivalent to 1, using the
    # extended Euclidean algorithm:
    y, x = euc_alg(val, p)
    if y == 1:
        return x % p
```

```
def euc_alg(val, p):
    # Extended Euclidean algorithm, inspiration from wikipedia and other source:
    # https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm#Modular_integers
    # http://anh.cs.luc.edu/331/notes/xgcd.pdf , page 2
    t, newt = 1, 0
    r, newr = 0, 1
    while p != 0:
        quotient = val // p
        val, p = p, val % p
        t, newt = newt, t - (quotient * newt)
        r, newr = newr, r - (quotient * newr)
    if t < 0:
        t = t + p
    return  val, t

# Byte array to hash
def sha_hash(inputVal):
    h = hashlib.sha1(inputVal).hexdigest()
    return h

if __name__ == '__main__':
    soc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    soc.connect(("eitn41.eit.lth.se", 1337))
    main()
```