```
# OAEP for RSA
# By Amanda Flote & Olivia Mattsson
import math, hashlib

# LÃ¤nk till RFC: https://tools.ietf.org/html/rfc8017

# The length of SHA1 digest is 20 bytes:
hLen = 20
k = 128

def main():
    mgfseed = '4ca586d5705a0b82ce5c5ca03dcd90'
    maskLen = 25
    m = "0d4413b8823db607b594f3d7e86c4db168a4a17eb4fffd97bb71"
    seed = "e1683401d63da920ccced24b47c53cca7479f0ec"
    encMessage = "0043759f100e1b0ffbaed6b5e234f085cfd20cb94962f786195f85f8d337481f2abb06da0f3f9b1a5e413d31e347a179461d13c47b4f6893c022209324
    # RSAES-OAEP Scheme
    res = MGF1(mgfseed, maskLen)
    print("MGF1: {0}".format(res))
    print("-------------------------")

    # EME-OAEP encoding: Step 2 in 7.1.1
    c = OAEPencode(seed, m)
    print("Encrypted message: {0}".format(c))
    print("-------------------------")
    # EME-OAEP decoding: Step 3 in 7.1.2
    decrypted = OAEPdecode(encMessage)
    print("Decrypted message: {0}".format(decrypted))


# EME-OAEP encoding
def OAEPencode(seed, message):
    # Computes hash of empty string
    lHash = sha_hash("")
    mLen = int(len(message)/2)
    PS = (k - mLen - 2*hLen) - 2
    padding = "".zfill((PS*2))
    DB = lHash + padding + "01" + message
    dbMask = MGF1(seed, (k - hLen - 1))
    maskedDB = hex(int(DB,16) ^ int(dbMask, 16))[2:]
    seedMask = MGF1(maskedDB, hLen).lstrip("0x")
    maskedSeed = hex(int(seed,16) ^ int(seedMask, 16))[2:]
    encMessage = "00" + maskedSeed + maskedDB
    if ((len(encMessage)/2) < 128):
        encMessage = "00" + encMessage
    return encMessage


# EME-OAEP decoding
def OAEPdecode(message):
    # Computes hash of empty string
    lHash = sha_hash("")
    Y = message[:2]
    # Retrieves masked seed & db from encrypted message
    maskedSeed = message[2:hLen*2 + 2]
    maskedDB = message[2*hLen+2:]
    seedMask = MGF1(maskedDB, hLen)
    seed = hex(int(maskedSeed,16) ^(int(seedMask,16)))[2:]
    dbMask = MGF1(seed, (k-hLen-1))
    db =  hex(int(maskedDB,16) ^(int(dbMask,16)))[2:]
    lHash2 = db[:2*hLen]
    message = db[2*hLen:]
    indexMess = message.find("01")
    if lHash != lHash2 or indexMess == -1:
        return "Decryption error"
    message = message[indexMess+2:]
    return message


# Mask Genereation Function
def MGF1(mgfSeed, maskLen):
    # If maskLen > 2^32 hLen, output "mask too long" and stop.
    if maskLen > (2**32) * hLen:
        print("Mask too long")
        return
    #  Let T be the empty octet string (byte string)
    T = ""
    # For counter from 0 to \ceil (maskLen / hLen) - 1:
    for i in range(0,math.ceil(maskLen/hLen)):
        # Convert counter to an octet string C of length 4 octets:
        C = I2OSP (i, 4)

        # Concatenate the hash of the seed mgfSeed and C to the octet string T:
        T += sha_hash(mgfSeed + C)

    # Output the leading maskLen octets of T as the octet string mask.

    return T[:2*maskLen]

# I2OSP function
def I2OSP(x, 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 = ""
    for i in range(1,xLen+1):
        if i == xLen:
            res = x
        elif (x / (256**(xLen-i))) > 1:
            res = int(x - math.floor(x / (256**(xLen-i))))
            x = x - res
        else:
            res = "0"
        X += str(res)
    # Return the base 256 X value:
    return (X.zfill(2*xLen))
```

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

if __name__ == '__main__':
    main()
```