# Author: Trevor Perrin

# See the LICENSE file for legal information regarding use of this file.

"""Pure-Python RSA implementation."""

from .cryptomath import \*

from .asn1parser import ASN1Parser

from .rsakey import \*

from .pem import \*

class Python\_RSAKey(RSAKey):

def \_\_init\_\_(self, n=0, e=0, d=0, p=0, q=0, dP=0, dQ=0, qInv=0):

if (n and not e) or (e and not n):

raise AssertionError()

self.n = n

self.e = e

self.d = d

self.p = p

self.q = q

self.dP = dP

self.dQ = dQ

self.qInv = qInv

self.blinder = 0

self.unblinder = 0

def hasPrivateKey(self):

return self.d != 0

def \_rawPrivateKeyOp(self, m):

#Create blinding values, on the first pass:

if not self.blinder:

self.unblinder = getRandomNumber(2, self.n)

self.blinder = powMod(invMod(self.unblinder, self.n), self.e,

self.n)

#Blind the input

m = (m \* self.blinder) % self.n

#Perform the RSA operation

c = self.\_rawPrivateKeyOpHelper(m)

#Unblind the output

c = (c \* self.unblinder) % self.n

#Update blinding values

self.blinder = (self.blinder \* self.blinder) % self.n

self.unblinder = (self.unblinder \* self.unblinder) % self.n

#Return the output

return c

def \_rawPrivateKeyOpHelper(self, m):

#Non-CRT version

#c = powMod(m, self.d, self.n)

#CRT version (~3x faster)

s1 = powMod(m, self.dP, self.p)

s2 = powMod(m, self.dQ, self.q)

h = ((s1 - s2) \* self.qInv) % self.p

c = s2 + self.q \* h

return c

def \_rawPublicKeyOp(self, c):

m = powMod(c, self.e, self.n)

return m

def acceptsPassword(self): return False

def generate(bits):

key = Python\_RSAKey()

p = getRandomPrime(bits//2, False)

q = getRandomPrime(bits//2, False)

t = lcm(p-1, q-1)

key.n = p \* q

key.e = 65537

key.d = invMod(key.e, t)

key.p = p

key.q = q

key.dP = key.d % (p-1)

key.dQ = key.d % (q-1)

key.qInv = invMod(q, p)

return key

generate = staticmethod(generate)

def parsePEM(s, passwordCallback=None):

"""Parse a string containing a PEM-encoded <privateKey>."""

if pemSniff(s, "PRIVATE KEY"):

bytes = dePem(s, "PRIVATE KEY")

return Python\_RSAKey.\_parsePKCS8(bytes)

elif pemSniff(s, "RSA PRIVATE KEY"):

bytes = dePem(s, "RSA PRIVATE KEY")

return Python\_RSAKey.\_parseSSLeay(bytes)

else:

raise SyntaxError("Not a PEM private key file")

parsePEM = staticmethod(parsePEM)

def \_parsePKCS8(bytes):

p = ASN1Parser(bytes)

version = p.getChild(0).value[0]

if version != 0:

raise SyntaxError("Unrecognized PKCS8 version")

rsaOID = p.getChild(1).value

if list(rsaOID) != [6, 9, 42, 134, 72, 134, 247, 13, 1, 1, 1, 5, 0]:

raise SyntaxError("Unrecognized AlgorithmIdentifier")

#Get the privateKey

privateKeyP = p.getChild(2)

#Adjust for OCTET STRING encapsulation

privateKeyP = ASN1Parser(privateKeyP.value)

return Python\_RSAKey.\_parseASN1PrivateKey(privateKeyP)

\_parsePKCS8 = staticmethod(\_parsePKCS8)

def \_parseSSLeay(bytes):

privateKeyP = ASN1Parser(bytes)

return Python\_RSAKey.\_parseASN1PrivateKey(privateKeyP)

\_parseSSLeay = staticmethod(\_parseSSLeay)

def \_parseASN1PrivateKey(privateKeyP):

version = privateKeyP.getChild(0).value[0]

if version != 0:

raise SyntaxError("Unrecognized RSAPrivateKey version")

n = bytesToNumber(privateKeyP.getChild(1).value)

e = bytesToNumber(privateKeyP.getChild(2).value)

d = bytesToNumber(privateKeyP.getChild(3).value)

p = bytesToNumber(privateKeyP.getChild(4).value)

q = bytesToNumber(privateKeyP.getChild(5).value)

dP = bytesToNumber(privateKeyP.getChild(6).value)

dQ = bytesToNumber(privateKeyP.getChild(7).value)

qInv = bytesToNumber(privateKeyP.getChild(8).value)

return Python\_RSAKey(n, e, d, p, q, dP, dQ, qInv)

\_parseASN1PrivateKey = staticmethod(\_parseASN1PrivateKey)