# Author: Trevor Perrin

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

"""Abstract class for RSA."""

from .cryptomath import \*

class RSAKey(object):

"""This is an abstract base class for RSA keys.

Particular implementations of RSA keys, such as

L{openssl\_rsakey.OpenSSL\_RSAKey},

L{python\_rsakey.Python\_RSAKey}, and

L{pycrypto\_rsakey.PyCrypto\_RSAKey},

inherit from this.

To create or parse an RSA key, don't use one of these classes

directly. Instead, use the factory functions in

L{tlslite.utils.keyfactory}.

"""

def \_\_init\_\_(self, n=0, e=0):

"""Create a new RSA key.

If n and e are passed in, the new key will be initialized.

@type n: int

@param n: RSA modulus.

@type e: int

@param e: RSA public exponent.

"""

raise NotImplementedError()

def \_\_len\_\_(self):

"""Return the length of this key in bits.

@rtype: int

"""

return numBits(self.n)

def hasPrivateKey(self):

"""Return whether or not this key has a private component.

@rtype: bool

"""

raise NotImplementedError()

def hashAndSign(self, bytes):

"""Hash and sign the passed-in bytes.

This requires the key to have a private component. It performs

a PKCS1-SHA1 signature on the passed-in data.

@type bytes: str or L{bytearray} of unsigned bytes

@param bytes: The value which will be hashed and signed.

@rtype: L{bytearray} of unsigned bytes.

@return: A PKCS1-SHA1 signature on the passed-in data.

"""

hashBytes = SHA1(bytearray(bytes))

prefixedHashBytes = self.\_addPKCS1SHA1Prefix(hashBytes)

sigBytes = self.sign(prefixedHashBytes)

return sigBytes

def hashAndVerify(self, sigBytes, bytes):

"""Hash and verify the passed-in bytes with the signature.

This verifies a PKCS1-SHA1 signature on the passed-in data.

@type sigBytes: L{bytearray} of unsigned bytes

@param sigBytes: A PKCS1-SHA1 signature.

@type bytes: str or L{bytearray} of unsigned bytes

@param bytes: The value which will be hashed and verified.

@rtype: bool

@return: Whether the signature matches the passed-in data.

"""

hashBytes = SHA1(bytearray(bytes))

# Try it with/without the embedded NULL

prefixedHashBytes1 = self.\_addPKCS1SHA1Prefix(hashBytes, False)

prefixedHashBytes2 = self.\_addPKCS1SHA1Prefix(hashBytes, True)

result1 = self.verify(sigBytes, prefixedHashBytes1)

result2 = self.verify(sigBytes, prefixedHashBytes2)

return (result1 or result2)

def sign(self, bytes):

"""Sign the passed-in bytes.

This requires the key to have a private component. It performs

a PKCS1 signature on the passed-in data.

@type bytes: L{bytearray} of unsigned bytes

@param bytes: The value which will be signed.

@rtype: L{bytearray} of unsigned bytes.

@return: A PKCS1 signature on the passed-in data.

"""

if not self.hasPrivateKey():

raise AssertionError()

paddedBytes = self.\_addPKCS1Padding(bytes, 1)

m = bytesToNumber(paddedBytes)

if m >= self.n:

raise ValueError()

c = self.\_rawPrivateKeyOp(m)

sigBytes = numberToByteArray(c, numBytes(self.n))

return sigBytes

def verify(self, sigBytes, bytes):

"""Verify the passed-in bytes with the signature.

This verifies a PKCS1 signature on the passed-in data.

@type sigBytes: L{bytearray} of unsigned bytes

@param sigBytes: A PKCS1 signature.

@type bytes: L{bytearray} of unsigned bytes

@param bytes: The value which will be verified.

@rtype: bool

@return: Whether the signature matches the passed-in data.

"""

if len(sigBytes) != numBytes(self.n):

return False

paddedBytes = self.\_addPKCS1Padding(bytes, 1)

c = bytesToNumber(sigBytes)

if c >= self.n:

return False

m = self.\_rawPublicKeyOp(c)

checkBytes = numberToByteArray(m, numBytes(self.n))

return checkBytes == paddedBytes

def encrypt(self, bytes):

"""Encrypt the passed-in bytes.

This performs PKCS1 encryption of the passed-in data.

@type bytes: L{bytearray} of unsigned bytes

@param bytes: The value which will be encrypted.

@rtype: L{bytearray} of unsigned bytes.

@return: A PKCS1 encryption of the passed-in data.

"""

paddedBytes = self.\_addPKCS1Padding(bytes, 2)

m = bytesToNumber(paddedBytes)

if m >= self.n:

raise ValueError()

c = self.\_rawPublicKeyOp(m)

encBytes = numberToByteArray(c, numBytes(self.n))

return encBytes

def decrypt(self, encBytes):

"""Decrypt the passed-in bytes.

This requires the key to have a private component. It performs

PKCS1 decryption of the passed-in data.

@type encBytes: L{bytearray} of unsigned bytes

@param encBytes: The value which will be decrypted.

@rtype: L{bytearray} of unsigned bytes or None.

@return: A PKCS1 decryption of the passed-in data or None if

the data is not properly formatted.

"""

if not self.hasPrivateKey():

raise AssertionError()

if len(encBytes) != numBytes(self.n):

return None

c = bytesToNumber(encBytes)

if c >= self.n:

return None

m = self.\_rawPrivateKeyOp(c)

decBytes = numberToByteArray(m, numBytes(self.n))

#Check first two bytes

if decBytes[0] != 0 or decBytes[1] != 2:

return None

#Scan through for zero separator

for x in range(1, len(decBytes)-1):

if decBytes[x]== 0:

break

else:

return None

return decBytes[x+1:] #Return everything after the separator

def \_rawPrivateKeyOp(self, m):

raise NotImplementedError()

def \_rawPublicKeyOp(self, c):

raise NotImplementedError()

def acceptsPassword(self):

"""Return True if the write() method accepts a password for use

in encrypting the private key.

@rtype: bool

"""

raise NotImplementedError()

def write(self, password=None):

"""Return a string containing the key.

@rtype: str

@return: A string describing the key, in whichever format (PEM)

is native to the implementation.

"""

raise NotImplementedError()

def generate(bits):

"""Generate a new key with the specified bit length.

@rtype: L{tlslite.utils.RSAKey.RSAKey}

"""

raise NotImplementedError()

generate = staticmethod(generate)

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Helper Functions for RSA Keys

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

def \_addPKCS1SHA1Prefix(self, bytes, withNULL=True):

# There is a long history of confusion over whether the SHA1

# algorithmIdentifier should be encoded with a NULL parameter or

# with the parameter omitted. While the original intention was

# apparently to omit it, many toolkits went the other way. TLS 1.2

# specifies the NULL should be included, and this behavior is also

# mandated in recent versions of PKCS #1, and is what tlslite has

# always implemented. Anyways, verification code should probably

# accept both. However, nothing uses this code yet, so this is

# all fairly moot.

if not withNULL:

prefixBytes = bytearray(\

[0x30,0x1f,0x30,0x07,0x06,0x05,0x2b,0x0e,0x03,0x02,0x1a,0x04,0x14])

else:

prefixBytes = bytearray(\

[0x30,0x21,0x30,0x09,0x06,0x05,0x2b,0x0e,0x03,0x02,0x1a,0x05,0x00,0x04,0x14])

prefixedBytes = prefixBytes + bytes

return prefixedBytes

def \_addPKCS1Padding(self, bytes, blockType):

padLength = (numBytes(self.n) - (len(bytes)+3))

if blockType == 1: #Signature padding

pad = [0xFF] \* padLength

elif blockType == 2: #Encryption padding

pad = bytearray(0)

while len(pad) < padLength:

padBytes = getRandomBytes(padLength \* 2)

pad = [b for b in padBytes if b != 0]

pad = pad[:padLength]

else:

raise AssertionError()

padding = bytearray([0,blockType] + pad + [0])

paddedBytes = padding + bytes

return paddedBytes