# Author: Trevor Perrin

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

"""Factory functions for asymmetric cryptography.

@sort: generateRSAKey, parsePEMKey, parseAsPublicKey

"""

from .compat import \*

from .rsakey import RSAKey

from .python\_rsakey import Python\_RSAKey

from tlslite.utils import cryptomath

if cryptomath.m2cryptoLoaded:

from .openssl\_rsakey import OpenSSL\_RSAKey

if cryptomath.pycryptoLoaded:

from .pycrypto\_rsakey import PyCrypto\_RSAKey

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

# Factory Functions for RSA Keys

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

def generateRSAKey(bits, implementations=["openssl", "python"]):

"""Generate an RSA key with the specified bit length.

@type bits: int

@param bits: Desired bit length of the new key's modulus.

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

@return: A new RSA private key.

"""

for implementation in implementations:

if implementation == "openssl" and cryptomath.m2cryptoLoaded:

return OpenSSL\_RSAKey.generate(bits)

elif implementation == "python":

return Python\_RSAKey.generate(bits)

raise ValueError("No acceptable implementations")

#Parse as an OpenSSL or Python key

def parsePEMKey(s, private=False, public=False, passwordCallback=None,

implementations=["openssl", "python"]):

"""Parse a PEM-format key.

The PEM format is used by OpenSSL and other tools. The

format is typically used to store both the public and private

components of a key. For example::

-----BEGIN RSA PRIVATE KEY-----

MIICXQIBAAKBgQDYscuoMzsGmW0pAYsmyHltxB2TdwHS0dImfjCMfaSDkfLdZY5+

dOWORVns9etWnr194mSGA1F0Pls/VJW8+cX9+3vtJV8zSdANPYUoQf0TP7VlJxkH

dSRkUbEoz5bAAs/+970uos7n7iXQIni+3erUTdYEk2iWnMBjTljfgbK/dQIDAQAB

AoGAJHoJZk75aKr7DSQNYIHuruOMdv5ZeDuJvKERWxTrVJqE32/xBKh42/IgqRrc

esBN9ZregRCd7YtxoL+EVUNWaJNVx2mNmezEznrc9zhcYUrgeaVdFO2yBF1889zO

gCOVwrO8uDgeyj6IKa25H6c1N13ih/o7ZzEgWbGG+ylU1yECQQDv4ZSJ4EjSh/Fl

aHdz3wbBa/HKGTjC8iRy476Cyg2Fm8MZUe9Yy3udOrb5ZnS2MTpIXt5AF3h2TfYV

VoFXIorjAkEA50FcJmzT8sNMrPaV8vn+9W2Lu4U7C+K/O2g1iXMaZms5PC5zV5aV

CKXZWUX1fq2RaOzlbQrpgiolhXpeh8FjxwJBAOFHzSQfSsTNfttp3KUpU0LbiVvv

i+spVSnA0O4rq79KpVNmK44Mq67hsW1P11QzrzTAQ6GVaUBRv0YS061td1kCQHnP

wtN2tboFR6lABkJDjxoGRvlSt4SOPr7zKGgrWjeiuTZLHXSAnCY+/hr5L9Q3ZwXG

6x6iBdgLjVIe4BZQNtcCQQDXGv/gWinCNTN3MPWfTW/RGzuMYVmyBFais0/VrgdH

h1dLpztmpQqfyH/zrBXQ9qL/zR4ojS6XYneO/U18WpEe

-----END RSA PRIVATE KEY-----

To generate a key like this with OpenSSL, run::

openssl genrsa 2048 > key.pem

This format also supports password-encrypted private keys. TLS

Lite can only handle password-encrypted private keys when OpenSSL

and M2Crypto are installed. In this case, passwordCallback will be

invoked to query the user for the password.

@type s: str

@param s: A string containing a PEM-encoded public or private key.

@type private: bool

@param private: If True, a L{SyntaxError} will be raised if the

private key component is not present.

@type public: bool

@param public: If True, the private key component (if present) will

be discarded, so this function will always return a public key.

@type passwordCallback: callable

@param passwordCallback: This function will be called, with no

arguments, if the PEM-encoded private key is password-encrypted.

The callback should return the password string. If the password is

incorrect, SyntaxError will be raised. If no callback is passed

and the key is password-encrypted, a prompt will be displayed at

the console.

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

@return: An RSA key.

@raise SyntaxError: If the key is not properly formatted.

"""

for implementation in implementations:

if implementation == "openssl" and cryptomath.m2cryptoLoaded:

key = OpenSSL\_RSAKey.parse(s, passwordCallback)

break

elif implementation == "python":

key = Python\_RSAKey.parsePEM(s)

break

else:

raise ValueError("No acceptable implementations")

return \_parseKeyHelper(key, private, public)

def \_parseKeyHelper(key, private, public):

if private:

if not key.hasPrivateKey():

raise SyntaxError("Not a private key!")

if public:

return \_createPublicKey(key)

if private:

if hasattr(key, "d"):

return \_createPrivateKey(key)

else:

return key

return key

def parseAsPublicKey(s):

"""Parse a PEM-formatted public key.

@type s: str

@param s: A string containing a PEM-encoded public or private key.

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

@return: An RSA public key.

@raise SyntaxError: If the key is not properly formatted.

"""

return parsePEMKey(s, public=True)

def parsePrivateKey(s):

"""Parse a PEM-formatted private key.

@type s: str

@param s: A string containing a PEM-encoded private key.

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

@return: An RSA private key.

@raise SyntaxError: If the key is not properly formatted.

"""

return parsePEMKey(s, private=True)

def \_createPublicKey(key):

"""

Create a new public key. Discard any private component,

and return the most efficient key possible.

"""

if not isinstance(key, RSAKey):

raise AssertionError()

return \_createPublicRSAKey(key.n, key.e)

def \_createPrivateKey(key):

"""

Create a new private key. Return the most efficient key possible.

"""

if not isinstance(key, RSAKey):

raise AssertionError()

if not key.hasPrivateKey():

raise AssertionError()

return \_createPrivateRSAKey(key.n, key.e, key.d, key.p, key.q, key.dP,

key.dQ, key.qInv)

def \_createPublicRSAKey(n, e, implementations = ["openssl", "pycrypto",

"python"]):

for implementation in implementations:

if implementation == "openssl" and cryptomath.m2cryptoLoaded:

return OpenSSL\_RSAKey(n, e)

elif implementation == "pycrypto" and cryptomath.pycryptoLoaded:

return PyCrypto\_RSAKey(n, e)

elif implementation == "python":

return Python\_RSAKey(n, e)

raise ValueError("No acceptable implementations")

def \_createPrivateRSAKey(n, e, d, p, q, dP, dQ, qInv,

implementations = ["pycrypto", "python"]):

for implementation in implementations:

if implementation == "pycrypto" and cryptomath.pycryptoLoaded:

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

elif implementation == "python":

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

raise ValueError("No acceptable implementations")