RSA

RSA is the most widespread and used public key algorithm. Its security is based on the difficulty of factoring large integers. The algorithm has withstood attacks for more than 30 years, and it is therefore considered reasonably secure for new designs.

The algorithm can be used for both confidentiality (encryption) and authentication (digital signature). It is worth noting that signing and decryption are significantly slower than verification and encryption.

The cryptographic strength is primarily linked to the length of the RSA modulus *n*. In 2017, a sufficient length is deemed to be 2048 bits. For more information, see the most recent ECRYPT report.

Both RSA ciphertexts and RSA signatures are as large as the RSA modulus *n* (256 bytes if *n* is 2048 bit long).

The module `Crypto.PublicKey.RSA` provides facilities for generating new RSA keys, reconstructing them from known components, exporting them, and importing them.

As an example, this is how you generate a new RSA key pair, save it in a file called `mykey.pem`, and then read it back:

```
>>> from Crypto.PublicKey import RSA
>>>
>>> key = RSA.generate(2048)
>>> f = open('mykey.pem','wb')
>>> f.write(key.export_key('PEM'))
>>> f.close()
...
>>> f = open('mykey.pem','r')
>>> key = RSA.import_key(f.read())
```

Crypto.PublicKey.RSA.generate(*bits*, *randfunc=None*, *e=65537*)

Create a new RSA key pair.

The algorithm closely follows NIST FIPS 186-4 in its sections B.3.1 and B.3.3. The modulus is the product of two non-strong probable primes. Each prime passes a suitable number of Miller-Rabin tests with random bases and a single Lucas test.

Parameters:

- `bits` (*integer*) – Key length, or size (in bits) of the RSA modulus. It must be at least 1024, but `2048 is recommended`. The FIPS standard only defines 1024, 2048 and 3072.
- `randfunc` (*callable*) – Function that returns random bytes. The default is `Crypto.Random.get_random_bytes()`.

- e (*integer*) – Public RSA exponent. It must be an odd positive integer. It is typically a small number with very few ones in its binary representation. The FIPS standard requires the public exponent to be at least 65537 (the default).

Returns: an RSA key object (`RsaKey`, with private key).

Crypto.PublicKey.RSA.construct(*rsa_components*, *consistency_check=True*)

Construct an RSA key from a tuple of valid RSA components.

The modulus n must be the product of two primes. The public exponent e must be odd and larger than 1.

In case of a private key, the following equations must apply:

$$p * q = n$$
$$e * d \equiv 1 (\mathrm{mod}\ \mathrm{lcm}[(p-1)(q-1)])\ p*q=ne*d\equiv1(\mathrm{mod}\ \mathrm{lcm}[(p-1)(q-1)])p*u\equiv1(\mathrm{mod}\ q)$$
$$p * u \equiv 1 (\mathrm{mod}\ q)$$

Parameters:

- rsa_components (*tuple*) –
  A tuple of integers, with at least 2 and no more than 6 items. The items come in the following order:
  1. RSA modulus *n*.
  2. Public exponent *e*.
  3. Private exponent *d*. Only required if the key is private.
  4. First factor of *n* (*p*). Optional, but the other factor *q* must also be present.
  5. Second factor of *n* (*q*). Optional.
  6. CRT coefficient *q*, that is $p^{-1} \mathrm{mod}\ q$p−1mod q. Optional.
- consistency_check (*boolean*) – If `True`, the library will verify that the provided components fulfil the main RSA properties.

Raises: `ValueError` – when the key being imported fails the most basic RSA validity checks.

Returns: An RSA key object (`RsaKey`).

Crypto.PublicKey.RSA.import_key(*extern_key*, *passphrase=None*)

Import an RSA key (public or private).

Parameters:

- extern_key (*string or byte string*) –
  The RSA key to import.

The following formats are supported for an RSA public key:

- X.509 certificate (binary or PEM format)
- X.509 `subjectPublicKeyInfo` DER SEQUENCE (binary or PEM encoding)
- PKCS#1 `RSAPublicKey` DER SEQUENCE (binary or PEM encoding)
- An OpenSSH line (e.g. the content of `~/.ssh/id_ecdsa`, ASCII)

The following formats are supported for an RSA private key:

- PKCS#1 `RSAPrivateKey` DER SEQUENCE (binary or PEM encoding)
- PKCS#8 `PrivateKeyInfo` or `EncryptedPrivateKeyInfo` DER SEQUENCE (binary or PEM encoding)
- OpenSSH (text format, introduced in OpenSSH 6.5)

For details about the PEM encoding, see RFC1421/RFC1423.

- passphrase (*string or byte string*) – For private keys only, the pass phrase that encrypts the key.

Returns: An RSA key object (`RsaKey`).

Raises: `ValueError/IndexError/TypeError` – When the given key cannot be parsed (possibly because the pass phrase is wrong).

class **Crypto.PublicKey.RSA.RsaKey**(*****kwargs***)

Class defining an actual RSA key. Do not instantiate directly.
Use `generate()`, `construct()` or `import_key()` instead.

Variables:

- n (*integer*) – RSA modulus
- e (*integer*) – RSA public exponent
- d (*integer*) – RSA private exponent
- p (*integer*) – First factor of the RSA modulus
- q (*integer*) – Second factor of the RSA modulus
- u – Chinese remainder component ($p^{-1} \bmod q$ $p^{-1} \bmod q$)

exportKey(***format='PEM'***, ***passphrase=None***, ***pkcs=1***, ***protection=None***, ***randfunc=None***)

Export this RSA key.

Parameters:

- format (*string*) –
  The format to use for wrapping the key:
  - *'PEM'*. (*Default*) Text encoding, done according to RFC1421/RFC1423.
  - *'DER'*. Binary encoding.
  - *'OpenSSH'*. Textual encoding, done according to OpenSSH specification. Only

suitable for public keys (not private keys).

- passphrase (*string*) – (*For private keys only*) The pass phrase used for protecting the output.
- pkcs (*integer*) –
  (*For private keys only*) The ASN.1 structure to use for serializing the key. Note that even in case of PEM encoding, there is an inner ASN.1 DER structure.

  With `pkcs=1` (*default*), the private key is encoded in a simple PKCS#1 structure (`RSAPrivateKey`).

  With `pkcs=8`, the private key is encoded in a PKCS#8 structure (`PrivateKeyInfo`).

  Note

  This parameter is ignored for a public key. For DER and PEM, an ASN.1 DER `SubjectPublicKeyInfo` structure is always used.

- protection (*string*) –
  (*For private keys only*) The encryption scheme to use for protecting the private key.

  If `None` (default), the behavior depends on `format`:

  - For *'DER'*, the *PBKDF2WithHMAC-SHA1AndDES-EDE3-CBC* scheme is used. The following operations are performed:
    1. A 16 byte Triple DES key is derived from the passphrase using `Crypto.Protocol.KDF.PBKDF2()` with 8 bytes salt, and 1 000 iterations of `Crypto.Hash.HMAC`.
    2. The private key is encrypted using CBC.
    3. The encrypted key is encoded according to PKCS#8.
  - For *'PEM'*, the obsolete PEM encryption scheme is used. It is based on MD5 for key derivation, and Triple DES for encryption.

  Specifying a value for `protection` is only meaningful for PKCS#8 (that is, `pkcs=8`) and only if a pass phrase is present too.

  The supported schemes for PKCS#8 are listed in the `Crypto.IO.PKCS8` module (see `wrap_algo` parameter).
- randfunc (*callable*) – A function that provides random bytes. Only used for PEM encoding. The default is `Crypto.Random.get_random_bytes()`.

Returns: the encoded key

Return type: byte string

Raises: `ValueError` – when the format is unknown or when you try to encrypt a private key with *DER* format and PKCS#1.

Warning

If you don't provide a pass phrase, the private key will be exported in the clear!

export_key(*format='PEM'*, *passphrase=None*, *pkcs=1*, *protection=None*, *randfunc=None*)

Export this RSA key.

Parameters:

- format (*string*) –
  The format to use for wrapping the key:
  - *'PEM'*. (*Default*) Text encoding, done according to RFC1421/RFC1423.
  - *'DER'*. Binary encoding.
  - *'OpenSSH'*. Textual encoding, done according to OpenSSH specification. Only suitable for public keys (not private keys).
- passphrase (*string*) – (*For private keys only*) The pass phrase used for protecting the output.
- pkcs (*integer*) –
  (*For private keys only*) The ASN.1 structure to use for serializing the key. Note that even in case of PEM encoding, there is an inner ASN.1 DER structure.

  With `pkcs=1` (*default*), the private key is encoded in a simple PKCS#1 structure (`RSAPrivateKey`).

  With `pkcs=8`, the private key is encoded in a PKCS#8 structure (`PrivateKeyInfo`).

  Note

  This parameter is ignored for a public key. For DER and PEM, an ASN.1 DER `SubjectPublicKeyInfo` structure is always used.

- protection (*string*) –
  (*For private keys only*) The encryption scheme to use for protecting the private key.

  If `None` (default), the behavior depends on `format`:
  - For *'DER'*, the *PBKDF2WithHMAC-SHA1AndDES-EDE3-CBC* scheme is used. The following operations are performed:
    1. A 16 byte Triple DES key is derived from the passphrase using `Crypto.Protocol.KDF.PBKDF2()` with 8 bytes salt, and 1 000 iterations of `Crypto.Hash.HMAC`.

2. The private key is encrypted using CBC.
3. The encrypted key is encoded according to PKCS#8.

- For *'PEM'*, the obsolete PEM encryption scheme is used. It is based on MD5 for key derivation, and Triple DES for encryption.

Specifying a value for `protection` is only meaningful for PKCS#8 (that is, `pkcs=8`) and only if a pass phrase is present too.

The supported schemes for PKCS#8 are listed in the `Crypto.IO.PKCS8` module (see `wrap_algo` parameter).

- `randfunc` (*callable*) – A function that provides random bytes. Only used for PEM encoding. The default is `Crypto.Random.get_random_bytes()`.

Returns: the encoded key

Return type: byte string

Raises: `ValueError` – when the format is unknown or when you try to encrypt a private key with *DER* format and PKCS#1.

Warning

If you don't provide a pass phrase, the private key will be exported in the clear!

has_private()

Whether this is an RSA private key

publickey()

A matching RSA public key.

Returns: a new `RsaKey` object

size_in_bits()

Size of the RSA modulus in bits

size_in_bytes()

The minimal amount of bytes that can hold the RSA modulus

Crypto.PublicKey.RSA.oid= *'1.2.840.113549.1.1.1'*

Object ID for the RSA encryption algorithm. This OID often indicates a generic RSA key, even when such key will be actually used for digital signatures.