Danger

This is a "Hazardous Materials" module. You should **ONLY** use it if you're 100% absolutely sure that you know what you're doing because this module is full of land mines, dragons, and dinosaurs with laser guns.

RSA

RSA is a public-key algorithm for encrypting and signing messages.

Generation

Unlike symmetric cryptography, where the key is typically just a random series of bytes, RSA keys have a complex internal structure with specific mathematical properties.

cryptography.hazmat.primitives.asymmetric.rsa.generate_private_key(*public_exponent, key_size, backend=None*)

New in version 0.5.

Changed in version 3.0: Tightened restrictions on `public_exponent`.

Generates a new RSA private key using the provided `backend`. `key_size` describes how many bits long the key should be. Larger keys provide more security; currently `1024` and below are considered breakable while `2048` or `4096` are reasonable default key sizes for new keys. The `public_exponent` indicates what one mathematical property of the key generation will be. Unless you have a specific reason to do otherwise, you should always use 65537.

```
>>> from cryptography.hazmat.primitives.asymmetric import rsa
>>> private_key = rsa.generate_private_key(
...     public_exponent=65537,
...     key_size=2048,
... )
```

Parameters:

- **public_exponent** (*int*) – The public exponent of the new key. Either 65537 or 3 (for legacy purposes). Almost everyone should use 65537.
- **key_size** (*int*) – The length of the modulus in bits. For keys generated in 2015 it is strongly recommended to be at least 2048 (See page 41). It must not be less than 512. Some backends may have additional limitations.
- **backend** – An optional backend which implements `RSABackend`.

Returns: An instance of `RSAPrivateKey`.

Raises: cryptography.exceptions.UnsupportedAlgorithm – This is raised if the provided `backend` does not implement `RSABackend`

Key loading

If you already have an on-disk key in the PEM format (which are recognizable by the distinctive `-----BEGIN {format}-----` and `-----END {format}-----` markers), you can load it:

```
>>> from cryptography.hazmat.primitives import serialization

>>> with open("path/to/key.pem", "rb") as key_file:
...     private_key = serialization.load_pem_private_key(
...         key_file.read(),
...         password=None,
...     )
```

Serialized keys may optionally be encrypted on disk using a password. In this example we loaded an unencrypted key, and therefore we did not provide a password. If the key is encrypted we can pass a `bytes` object as the `password` argument.

There is also support for `loading public keys in the SSH format`.

Key serialization

If you have a private key that you've loaded you can use `private_bytes()` to serialize the key.

```
>>> from cryptography.hazmat.primitives import serialization
>>> pem = private_key.private_bytes(
...    encoding=serialization.Encoding.PEM,
...    format=serialization.PrivateFormat.PKCS8,
...    encryption_algorithm=serialization.BestAvailableEncryption(b'mypassword')
... )
>>> pem.splitlines()[0]
b'-----BEGIN ENCRYPTED PRIVATE KEY-----'
```

It is also possible to serialize without encryption using `NoEncryption`.

```
>>> pem = private_key.private_bytes(
...    encoding=serialization.Encoding.PEM,
...    format=serialization.PrivateFormat.TraditionalOpenSSL,
...    encryption_algorithm=serialization.NoEncryption()
... )
>>> pem.splitlines()[0]
b'-----BEGIN RSA PRIVATE KEY-----'
```

For public keys you can use `public_bytes()` to serialize the key.

```
>>> from cryptography.hazmat.primitives import serialization
>>> public_key = private_key.public_key()
>>> pem = public_key.public_bytes(
...    encoding=serialization.Encoding.PEM,
...    format=serialization.PublicFormat.SubjectPublicKeyInfo
... )
>>> pem.splitlines()[0]
b'-----BEGIN PUBLIC KEY-----'
```

Signing

A private key can be used to sign a message. This allows anyone with the public key to verify that the message was created by someone who possesses the corresponding private key. RSA signatures require a specific hash function, and padding to be used. Here is an example of signing `message` using RSA, with a secure hash function and padding:

```
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric import padding
>>> message = b"A message I want to sign"
>>> signature = private_key.sign(
...     message,
...     padding.PSS(
...         mgf=padding.MGF1(hashes.SHA256()),
...         salt_length=padding.PSS.MAX_LENGTH
...     ),
...     hashes.SHA256()
... )
```

Valid paddings for signatures are `PSS` and `PKCS1v15`. `PSS` is the recommended choice for any new protocols or applications, `PKCS1v15` should only be used to support legacy protocols.

If your data is too large to be passed in a single call, you can hash it separately and pass that value using `Prehashed`.

```
>>> from cryptography.hazmat.primitives.asymmetric import utils
>>> chosen_hash = hashes.SHA256()
>>> hasher = hashes.Hash(chosen_hash)
>>> hasher.update(b"data & ")
>>> hasher.update(b"more data")
>>> digest = hasher.finalize()
>>> sig = private_key.sign(
...     digest,
...     padding.PSS(
...         mgf=padding.MGF1(hashes.SHA256()),
...         salt_length=padding.PSS.MAX_LENGTH
...     ),
...     utils.Prehashed(chosen_hash)
... )
```

Verification

The previous section describes what to do if you have a private key and want to sign something. If you have a public key, a message, a signature, and the signing algorithm that was used you can check that the private key associated with a given public key was used to sign that specific message. You can obtain a public key to use in verification using `load_pem_public_key()`, `load_der_public_key()`, `public_key()`, or `public_key()`.

```
>>> public_key = private_key.public_key()
>>> public_key.verify(
...     signature,
...     message,
...     padding.PSS(
...         mgf=padding.MGF1(hashes.SHA256()),
...         salt_length=padding.PSS.MAX_LENGTH
...     ),
...     hashes.SHA256()
... )
```

If the signature does not match, `verify()` will raise an `InvalidSignature` exception.

If your data is too large to be passed in a single call, you can hash it separately and pass that value using `Prehashed`.

```
>>> chosen_hash = hashes.SHA256()
>>> hasher = hashes.Hash(chosen_hash)
>>> hasher.update(b"data & ")
>>> hasher.update(b"more data")
>>> digest = hasher.finalize()
>>> public_key.verify(
...     sig,
...     digest,
...     padding.PSS(
...         mgf=padding.MGF1(hashes.SHA256()),
...         salt_length=padding.PSS.MAX_LENGTH
...     ),
...     utils.Prehashed(chosen_hash)
... )
```

Encryption

RSA encryption is interesting because encryption is performed using the **public** key, meaning anyone can encrypt data. The data is then decrypted using the **private** key.

Like signatures, RSA supports encryption with several different padding options. Here's an example using a secure padding and hash function:

```
>>> message = b"encrypted data"
>>> ciphertext = public_key.encrypt(
...     message,
...     padding.OAEP(
...         mgf=padding.MGF1(algorithm=hashes.SHA256()),
...         algorithm=hashes.SHA256(),
...         label=None
...     )
... )
```

Valid paddings for encryption are `OAEP` and `PKCS1v15`. `OAEP` is the recommended choice for any new protocols or applications, `PKCS1v15` should only be used to support legacy protocols.

Decryption

Once you have an encrypted message, it can be decrypted using the private key:

```
>>> plaintext = private_key.decrypt(
...     ciphertext,
...     padding.OAEP(
...         mgf=padding.MGF1(algorithm=hashes.SHA256()),
...         algorithm=hashes.SHA256(),
...         label=None
...     )
... )
>>> plaintext == message
True
```

Padding

class **cryptography.hazmat.primitives.asymmetric.padding.AsymmetricPadding**

New in version 0.2.

name

class **cryptography.hazmat.primitives.asymmetric.padding.PSS**(*mgf*, *salt_length*)

New in version 0.3.

Changed in version 0.4: Added `salt_length` parameter.

PSS (Probabilistic Signature Scheme) is a signature scheme defined in **RFC 3447**. It is more complex than PKCS1 but possesses a security proof. This is the recommended padding algorithm for RSA signatures. It cannot be used with RSA encryption.

Parameters:

- **mgf** – A mask generation function object. At this time the only supported MGF is `MGF1`.
- **salt_length** (*int*) – The length of the salt. It is recommended that this be set to `PSS.MAX_LENGTH`.

MAX_LENGTH

Pass this attribute to `salt_length` to get the maximum salt length available.

class **cryptography.hazmat.primitives.asymmetric.padding.OAEP**(*mgf*, *algorithm*, *label*)

New in version 0.4.

OAEP (Optimal Asymmetric Encryption Padding) is a padding scheme defined in **RFC 3447**. It provides probabilistic encryption and is proven secure against several attack types. This is the recommended padding algorithm for RSA encryption. It cannot be used with RSA signing.

Parameters:

- **mgf** – A mask generation function object. At this time the only supported MGF is `MGF1`.
- **algorithm** – An instance of `HashAlgorithm`.
- **label** (*bytes*) – A label to apply. This is a rarely used field and should typically be set to `None` or `b""`, which are equivalent.

class **cryptography.hazmat.primitives.asymmetric.padding.PKCS1v15**

New in version 0.3.

PKCS1 v1.5 (also known as simply PKCS1) is a simple padding scheme developed for use with RSA keys. It is defined in RFC 3447. This padding can be used for signing and encryption.

It is not recommended that `PKCS1v15` be used for new applications, `OAEP` should be preferred for encryption and `PSS` should be preferred for signatures.

> **Warning**
>
> Our implementation of PKCS1 v1.5 decryption is not constant time. See Known security limitations for details.

`cryptography.hazmat.primitives.asymmetric.padding.calculate_max_pss_salt_length`(*key*, *hash_algorithm*)

New in version 1.5.

Parameters:
- **key** – An RSA public or private key.
- **hash_algorithm** – A `cryptography.hazmat.primitives.hashes.HashAlgorithm`.

Returns int: The computed salt length.

Computes the length of the salt that `PSS` will use if `PSS.MAX_LENGTH` is used.

Mask generation functions

---

class **`cryptography.hazmat.primitives.asymmetric.padding.MGF1`**(*algorithm*)

New in version 0.3.

Changed in version 0.6: Removed the deprecated `salt_length` parameter.

MGF1 (Mask Generation Function 1) is used as the mask generation function in `PSS` and `OAEP` padding. It takes a hash algorithm.

Parameters: **algorithm** – An instance of `HashAlgorithm`.

Numbers

These classes hold the constituent components of an RSA key. They are useful only when more traditional Key Serialization is unavailable.

---

class **`cryptography.hazmat.primitives.asymmetric.rsa.RSAPublicNumbers`**(*e*, *n*)

New in version 0.5.

The collection of integers that make up an RSA public key.

n

Type: int

The public modulus.

e

Type: int

The public exponent.

public_key(*backend=None*)

Parameters: **backend** – An optional instance of `RSABackend`.

Returns: A new instance of `RSAPublicKey`.

class **cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateNumbers**(*p, q, d, dmp1, dmq1, iqmp, public_numbers*)

New in version 0.5.

The collection of integers that make up an RSA private key.

Warning

With the exception of the integers contained in the `RSAPublicNumbers` all attributes of this class must be kept secret. Revealing them will compromise the security of any cryptographic operations performed with a key loaded from them.

public_numbers

Type: `RSAPublicNumbers`

The `RSAPublicNumbers` which makes up the RSA public key associated with this RSA private key.

p

Type: int

`p`, one of the two primes composing `n`.

q

Type: int

`q`, one of the two primes composing `n`.

d

Type: int

The private exponent.

dmp1

Type: int

A Chinese remainder theorem coefficient used to speed up RSA operations. Calculated as: d mod (p-1)

dmq1

Type: int

A Chinese remainder theorem coefficient used to speed up RSA operations. Calculated as: d mod (q-1)

iqmp

Type: int

A Chinese remainder theorem coefficient used to speed up RSA operations. Calculated as: q^{-1} mod p

private_key(*backend=None*)

Parameters: **backend** – An optional instance of `RSABackend`.

Returns: An instance of `RSAPrivateKey`.

Handling partial RSA private keys

If you are trying to load RSA private keys yourself you may find that not all parameters required by `RSAPrivateNumbers` are available. In particular the Chinese Remainder Theorem (CRT) values `dmp1`, `dmq1`, `iqmp` may be missing or present in a different form. For example, OpenPGP does not include the `iqmp`, `dmp1` or `dmq1` parameters.

The following functions are provided for users who want to work with keys like this without having to do the math themselves.

cryptography.hazmat.primitives.asymmetric.rsa.rsa_crt_iqmp(*p*, *q*)

New in version 0.4.

Computes the `iqmp` (also known as `qInv`) parameter from the RSA primes `p` and `q`.

`cryptography.hazmat.primitives.asymmetric.rsa.rsa_crt_dmp1`(*private_exponent*, *p*)

New in version 0.4.

Computes the `dmp1` parameter from the RSA private exponent (`d`) and prime `p`.

`cryptography.hazmat.primitives.asymmetric.rsa.rsa_crt_dmq1`(*private_exponent*, *q*)

New in version 0.4.

Computes the `dmq1` parameter from the RSA private exponent (`d`) and prime `q`.

`cryptography.hazmat.primitives.asymmetric.rsa.rsa_recover_prime_factors`(*n*, *e*, *d*)

New in version 0.8.

Computes the prime factors `(p, q)` given the modulus, public exponent, and private exponent.

> **Note**
>
> When recovering prime factors this algorithm will always return `p` and `q` such that `p > q`. Note: before 1.5, this function always returned `p` and `q` such that `p < q`. It was changed because libraries commonly require `p > q`.

Returns: A tuple `(p, q)`

Key interfaces

class `cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateKey`

New in version 0.2.

An RSA private key.

`decrypt`(*ciphertext*, *padding*)

New in version 0.4.

Decrypt data that was encrypted with the public key.

Parameters:

- **ciphertext** (*bytes*) – The ciphertext to decrypt.
- **padding** – An instance of `AsymmetricPadding`.

Return bytes: Decrypted data.

`public_key()`

Returns: `RSAPublicKey`

An RSA public key object corresponding to the values of the private key.

`key_size`

Type: int

The bit length of the modulus.

`sign(`*data*, *padding*, *algorithm*`)`

New in version 1.4.

Changed in version 1.6: `Prehashed` can now be used as an `algorithm`.

Sign one block of data which can be verified later by others using the public key.

Parameters:
- **data** (*bytes*) – The message string to sign.
- **padding** – An instance of `AsymmetricPadding`.
- **algorithm** – An instance of `HashAlgorithm` or `Prehashed` if the `data` you want to sign has already been hashed.

Return bytes: Signature.

`private_numbers()`

Create a `RSAPrivateNumbers` object.

Returns: An `RSAPrivateNumbers` instance.

`private_bytes(`*encoding*, *format*, *encryption_algorithm*`)`

Allows serialization of the key to bytes. Encoding (`PEM` or `DER`), format (`TraditionalOpenSSL` , `OpenSSH` or `PKCS8`) and encryption algorithm (such as `BestAvailableEncryption` or `NoEncryption`) are chosen to define the exact serialization.

Parameters:
- **encoding** – A value from the `Encoding` enum.
- **format** – A value from the `PrivateFormat` enum.
- **encryption_algorithm** – An instance of an object conforming to the `KeySerializationEncryption` interface.

Return bytes: Serialized key.

class **cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateKeyWithSerialization**

New in version 0.8.

Alias for `RSAPrivateKey`.

class **cryptography.hazmat.primitives.asymmetric.rsa.RSAPublicKey**

New in version 0.2.

An RSA public key.

encrypt(*plaintext*, *padding*)

New in version 0.4.

Encrypt data with the public key.

Parameters:
- **plaintext** (*bytes*) – The plaintext to encrypt.
- **padding** – An instance of `AsymmetricPadding`.

Return bytes: Encrypted data.

Raises: ValueError – The data could not be encrypted. One possible cause is if `data` is too large; RSA keys can only encrypt data that is smaller than the key size.

key_size

Type: int

The bit length of the modulus.

public_numbers()

Create a `RSAPublicNumbers` object.

Returns: An `RSAPublicNumbers` instance.

public_bytes(*encoding*, *format*)

Allows serialization of the key to bytes. Encoding (`PEM` or `DER`) and format (`SubjectPublicKeyInfo` or `PKCS1`) are chosen to define the exact serialization.

Parameters:
- **encoding** – A value from the `Encoding` enum.
- **format** – A value from the `PublicFormat` enum.

Return bytes: Serialized key.

`verify`(*signature*, *data*, *padding*, *algorithm*)

New in version 1.4.

Changed in version 1.6: `Prehashed` can now be used as an `algorithm`.

Verify one block of data was signed by the private key associated with this public key.

Parameters:
- **signature** (*bytes*) – The signature to verify.
- **data** (*bytes*) – The message string that was signed.
- **padding** – An instance of `AsymmetricPadding`.
- **algorithm** – An instance of `HashAlgorithm` or `Prehashed` if the `data` you want to verify has already been hashed.

Raises: cryptography.exceptions.InvalidSignature – If the signature does not validate.

`recover_data_from_signature`(*signature*, *padding*, *algorithm*)

New in version 3.3.

Recovers the signed data from the signature. The data typically contains the digest of the original message string. The `padding` and `algorithm` parameters must match the ones used when the signature was created for the recovery to succeed.

The `algorithm` parameter can also be set to `None` to recover all the data present in the signature, without regard to its format or the hash algorithm used for its creation.

For `PKCS1v15` padding, this method returns the data after removing the padding layer. For standard signatures the data contains the full `DigestInfo` structure. For non-standard signatures, any data can be returned, including zero-length data.

Normally you should use the `verify()` function to validate the signature. But for some non-standard signature formats you may need to explicitly recover and validate the signed data. The following are some examples:

- Some old Thawte and Verisign timestamp certificates without `DigestInfo`.
- Signed MD5/SHA1 hashes in TLS 1.1 or earlier (RFC 4346, section 4.7).
- IKE version 1 signatures without `DigestInfo` (RFC 2409, section 5.1).

Parameters:

- **signature** (*bytes*) – The signature.
- **padding** – An instance of `AsymmetricPadding`. Recovery is only supported with some of the padding types. (Currently only with `PKCS1v15`).
- **algorithm** – An instance of `HashAlgorithm`. Can be `None` to return the all the data present in the signature.

Return bytes: The signed data.

Raises:

- cryptography.exceptions.InvalidSignature – If the signature is invalid.
- cryptography.exceptions.UnsupportedAlgorithm – If signature data recovery is not supported with the provided `padding` type.

class **`cryptography.hazmat.primitives.asymmetric.rsa.RSAPublicKeyWithSerialization`**

New in version 0.8.

Alias for `RSAPublicKey`.