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.

Key Serialization

There are several common schemes for serializing asymmetric private and public keys to bytes. They generally support encryption of private keys and additional key metadata.

Many serialization formats support multiple different types of asymmetric keys and will return an instance of the appropriate type. You should check that the returned key matches the type your application expects when using these methods.

```
>>> from cryptography.hazmat.primitives.asymmetric import dsa, rsa
>>> from cryptography.hazmat.primitives.serialization import load_pem_private_key
>>> key = load_pem_private_key(pem_data, password=None)
>>> if isinstance(key, rsa.RSAPrivateKey):
...     signature = sign_with_rsa_key(key, message)
... elif isinstance(key, dsa.DSAPrivateKey):
...     signature = sign_with_dsa_key(key, message)
... else:
...     raise TypeError
```

Key dumping

The `serialization` module contains functions for loading keys from `bytes`. To dump a `key` object to `bytes`, you must call the appropriate method on the key object. Documentation for these methods in found in the `rsa`, `dsa`, and `ec` module documentation.

PEM

PEM is an encapsulation format, meaning keys in it can actually be any of several different key types. However these are all self-identifying, so you don’t need to worry about this detail. PEM keys are recognizable because they all begin with `-----BEGIN {format}-----` and end with `-----END {format}-----`.

Note

A PEM block which starts with `-----BEGIN CERTIFICATE-----` is not a public or private key, it’s an X.509 Certificate. You can load it using `load_pem_x509_certificate()` and extract the public key with `certificate.public_key`.

`cryptography.hazmat.primitives.serialization.load_pem_private_key`_(data, password, backend=None)_

New in version 0.6.

Deserialize a private key from PEM encoded data to one of the supported asymmetric private key types.

Parameters:
- **data** (bytes-like) – The PEM encoded key data.
- **password** – The password to use to decrypt the data. Should be `None` if the private key is not encrypted.
- **backend** – An optional instance of `PEMSerializationBackend`.

Returns: One of `Ed25519PrivateKey`, `X25519PrivateKey`, `Ed448PrivateKey`, `X448PrivateKey`, `RSAPrivateKey`, `DSAPrivateKey`, `DHPrivateKey`, or `EllipticCurvePrivateKey` depending on the contents of `data`.

Raises:
- ValueError – If the PEM data could not be decrypted or if its structure could not be decoded successfully.
- TypeError – If a `password` was given and the private key was not encrypted. Or if the key was encrypted but no password was supplied.
- cryptography.exceptions.UnsupportedAlgorithm – If the serialized key is of a type that is not supported by the backend.

`cryptography.hazmat.primitives.serialization.load_pem_public_key`_(data, backend=None)_

New in version 0.6.

Deserialize a public key from PEM encoded data to one of the supported asymmetric public key types. The PEM encoded data is typically a `subjectPublicKeyInfo` payload as specified in RFC 5280.

```
>>> from cryptography.hazmat.primitives.serialization import load_pem_public_key
>>> key = load_pem_public_key(public_pem_data)
>>> isinstance(key, rsa.RSAPublicKey)
True
```

Parameters:
- **data** (*bytes*) – The PEM encoded key data.
- **backend** – An optional instance of `PEMSerializationBackend`.

Returns: One of `Ed25519PublicKey`, `X25519PublicKey`, `Ed448PublicKey`, `X448PublicKey`, `RSAPublicKey`, `DSAPublicKey`, `DHPublicKey`, or `EllipticCurvePublicKey` depending on the contents of `data`.

Raises:

- ValueError – If the PEM data's structure could not be decoded successfully.
- cryptography.exceptions.UnsupportedAlgorithm – If the serialized key is of a type that is not supported by the backend.

cryptography.hazmat.primitives.serialization.load_pem_parameters(*data, backend=None*)

New in version 2.0.

Deserialize parameters from PEM encoded data to one of the supported asymmetric parameters types.

```
>>> from cryptography.hazmat.primitives.serialization import load_pem_parameters
>>> from cryptography.hazmat.primitives.asymmetric import dh
>>> parameters = load_pem_parameters(parameters_pem_data)
>>> isinstance(parameters, dh.DHParameters)
True
```

Parameters:

- **data** (*bytes*) – The PEM encoded parameters data.
- **backend** – An optional instance of `PEMSerializationBackend`.

Returns: Currently only `DHParameters` supported.

Raises:

- ValueError – If the PEM data's structure could not be decoded successfully.
- cryptography.exceptions.UnsupportedAlgorithm – If the serialized parameters is of a type that is not supported by the backend.

DER

DER is an ASN.1 encoding type. There are no encapsulation boundaries and the data is binary. DER keys may be in a variety of formats, but as long as you know whether it is a public or private key the loading functions will handle the rest.

cryptography.hazmat.primitives.serialization.load_der_private_key(*data, password, backend=None*)

New in version 0.8.

Deserialize a private key from DER encoded data to one of the supported asymmetric private key types.

Parameters:

- **data** (bytes-like) – The DER encoded key data.
- **password** (bytes-like) – The password to use to decrypt the data. Should be `None` if the private key is not encrypted.
- **backend** – An optional instance of `DERSerializationBackend`.

Returns: One of `Ed25519PrivateKey`, `X25519PrivateKey`, `Ed448PrivateKey`, `X448PrivateKey`, `RSAPrivateKey`, `DSAPrivateKey`, `DHPrivateKey`, or `EllipticCurvePrivateKey` depending on the contents of `data`.

Raises:

- ValueError – If the DER data could not be decrypted or if its structure could not be decoded successfully.
- TypeError – If a `password` was given and the private key was not encrypted. Or if the key was encrypted but no password was supplied.
- cryptography.exceptions.UnsupportedAlgorithm – If the serialized key is of a type that is not supported by the backend.

```
>>> from cryptography.hazmat.primitives.asymmetric import rsa
>>> from cryptography.hazmat.primitives.serialization import load_der_private_key
>>> key = load_der_private_key(der_data, password=None)
>>> isinstance(key, rsa.RSAPrivateKey)
True
```

`cryptography.hazmat.primitives.serialization.load_der_public_key`*(data, backend=None)*

New in version 0.8.

Deserialize a public key from DER encoded data to one of the supported asymmetric public key types. The DER encoded data is typically a `subjectPublicKeyInfo` payload as specified in RFC 5280.

Parameters:

- **data** (*bytes*) – The DER encoded key data.
- **backend** – An optional instance of `DERSerializationBackend`.

Returns: One of `Ed25519PublicKey`, `X25519PublicKey`, `Ed448PublicKey`, `X448PublicKey`, `RSAPublicKey`, `DSAPublicKey`, `DHPublicKey`, or `EllipticCurvePublicKey` depending on the contents of `data`.

Raises:

- ValueError – If the DER data’s structure could not be decoded successfully.
- cryptography.exceptions.UnsupportedAlgorithm – If the serialized key is of a type that is not supported by the backend.

```
>>> from cryptography.hazmat.primitives.asymmetric import rsa
>>> from cryptography.hazmat.primitives.serialization import load_der_public_key
>>> key = load_der_public_key(public_der_data)
>>> isinstance(key, rsa.RSAPublicKey)
True
```

`cryptography.hazmat.primitives.serialization.load_der_parameters`*(data, backend=None)*

New in version 2.0.

Deserialize parameters from DER encoded data to one of the supported asymmetric parameters types.

Parameters:
- **data** (*bytes*) – The DER encoded parameters data.
- **backend** – An optional instance of `DERSerializationBackend`.

Returns: Currently only `DHParameters` supported.

Raises:
- ValueError – If the DER data's structure could not be decoded successfully.
- cryptography.exceptions.UnsupportedAlgorithm – If the serialized key is of a type that is not supported by the backend.

```
>>> from cryptography.hazmat.primitives.asymmetric import dh
>>> from cryptography.hazmat.primitives.serialization import load_der_parameters
>>> parameters = load_der_parameters(parameters_der_data)
>>> isinstance(parameters, dh.DHParameters)
True
```

OpenSSH Public Key

The format used by OpenSSH to store public keys, as specified in **RFC 4253**.

An example RSA key in OpenSSH format (line breaks added for formatting purposes):

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDDu/XRP1kyK6Cgt36gts9XAk
FiiuJLW6RU0j3KKVZSs1I7Z3UmU9/9aVh/rZV43WQG8jaR6kkcP4stOR0DEtll
PDA7ZRBnrfiHpSQYQ874AZaAoIjgkv7DBfsE6gcDQLub0PFjWyrYQUJhtOLQEK
vY/G0vt2iRL3juawWmCFdTK3W3XvwAdgGk71i6lHt+deOPNEPN2H58E4odrZ2f
sxn/adpDqfb2sM0kPwQs0aWvrrKGvUaustkivQE4XWiSFnB0oJB/lKK/CKVKuy
///ImSCGHQRvhwariN2tvZ6CBNSLh3iQgeB0AkyJlng7MXB2qYq/Ci2FUOryCX
2MzHvnbv testkey@localhost
```

DSA keys look almost identical but begin with `ssh-dss` rather than `ssh-rsa`. ECDSA keys have a slightly different format, they begin with `ecdsa-sha2-{curve}`.

`cryptography.hazmat.primitives.serialization.load_ssh_public_key`(*data, backend=None*)

New in version 0.7.

Deserialize a public key from OpenSSH (**RFC 4253** and PROTOCOL.certkeys) encoded data to an instance of the public key type for the specified backend.

Parameters:
- **data** (bytes-like) – The OpenSSH encoded key data.
- **backend** – An optional backend which implements `RSABackend`, `DSABackend`, or `EllipticCurveBackend` depending on the key's type.

Returns: One of `RSAPublicKey`, `DSAPublicKey`, `EllipticCurvePublicKey`, or `Ed25519PublicKey`, depending on the contents of `data`.

Raises:
- ValueError – If the OpenSSH data could not be properly decoded or if the key is not in the proper format.
- cryptography.exceptions.UnsupportedAlgorithm – If the serialized key is of a type that is not supported.

OpenSSH Private Key

The format used by OpenSSH to store private keys, as approximately specified in PROTOCOL.key.

An example ECDSA key in OpenSSH format:

```
-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAAAaAAAABNlY2RzYS
1zaGEyLW5pc3RwMjU2AAAACG5pc3RwMjU2AAAAQQRI0fWnI1CxX7qYqp0ih6bxjhGmUrZK
/Axf8vhM8Db3oH7CFR+JdL715lUdu4XCWvQZKVf60/h3kBFhuxQC23XjAAAAqKPzVaOj81
WjAAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBEjR9acjULFfupiq
nSKHpvGOEaZStkr8DF/y+EzwNvegfsIVH4l0vvXmVR27hcJa9BkpV/rT+HeQEWG7FALbde
MAAAAga/VGV2asRlL3kXXao0aochQ59nXHA2xEGeAoQd952r0AAAAJbWFya29AdmZmAQID
BAUGBw==
-----END OPENSSH PRIVATE KEY-----
```

cryptography.hazmat.primitives.serialization.load_ssh_private_key(*data, password, backend=None*)

New in version 3.0.

Deserialize a private key from OpenSSH encoded data to an instance of the private key type for the specified backend.

Parameters:
- **data** (bytes-like) – The PEM encoded OpenSSH private key data.
- **password** (*bytes*) – Password bytes to use to decrypt password-protected key. Or `None` if not needed.
- **backend** – An optional backend which implements `RSABackend`, `DSABackend`, or `EllipticCurveBackend` depending on the key's type.

Returns: One of `RSAPrivateKey`, `DSAPrivateKey`, `EllipticCurvePrivateKey` or `Ed25519PrivateKey`, depending on the contents of `data`.

Raises:
- ValueError – If the OpenSSH data could not be properly decoded, if the key is not in the proper format or the incorrect password was provided.
- cryptography.exceptions.UnsupportedAlgorithm – If the serialized key is of a type that is not supported.

PKCS12

PKCS12 is a binary format described in RFC 7292. It can contain certificates, keys, and more. PKCS12 files commonly have a `pfx` or `p12` file suffix.

Note

`cryptography` only supports a single private key and associated certificates when parsing PKCS12 files at this time.

`cryptography.hazmat.primitives.serialization.pkcs12.load_key_and_certificates`*(data, password, backend=None)*

New in version 2.5.

Deserialize a PKCS12 blob.

Parameters:
- **data** (bytes-like) – The binary data.
- **password** (bytes-like) – The password to use to decrypt the data. `None` if the PKCS12 is not encrypted.
- **backend** – An optional backend instance.

Returns: A tuple of `(private_key, certificate, additional_certificates)`. `private_key` is a private key type or `None`, `certificate` is either the `Certificate` whose public key matches the private key in the PKCS 12 object or `None`, and `additional_certificates` is a list of all other `Certificate` instances in the PKCS12 object.

`cryptography.hazmat.primitives.serialization.pkcs12.serialize_key_and_certificates`*(name, key, cert, cas, encryption_algorithm)*

New in version 3.0.

Warning

PKCS12 encryption is not secure and should not be used as a security mechanism. Wrap a PKCS12 blob in a more secure envelope if you need to store or send it safely. Encryption is provided for compatibility reasons only.

Serialize a PKCS12 blob.

Note

Due to a bug in Firefox it's not possible to load unencrypted PKCS12 blobs in Firefox.

Parameters:

- **name** (*bytes*) – The friendly name to use for the supplied certificate and key.
- **key** (An `RSAPrivateKeyWithSerialization` , `EllipticCurvePrivateKeyWithSerialization` , or `DSAPrivateKeyWithSerialization` object.) – The private key to include in the structure.
- **cert** (`Certificate` or `None`) – The certificate associated with the private key.
- **cas** (list of `Certificate` or `None`) – An optional set of certificates to also include in the structure.
- **encryption_algorithm** – The encryption algorithm that should be used for the key and certificate. An instance of an object conforming to the `KeySerializationEncryption` interface. PKCS12 encryption is **very weak** and should not be used as a security boundary.

Return bytes: Serialized PKCS12.

PKCS7

PKCS7 is a format described in **RFC 2315**, among other specifications. It can contain certificates, CRLs, and much more. PKCS7 files commonly have a `p7b` , `p7m` , or `p7s` file suffix but other suffixes are also seen in the wild.

Note

`cryptography` only supports parsing certificates from PKCS7 files at this time.

`cryptography.hazmat.primitives.serialization.pkcs7.load_pem_pkcs7_certificates`(*data*)

New in version 3.1.

Deserialize a PEM encoded PKCS7 blob to a list of certificates. PKCS7 can contain many other types of data, including CRLs, but this function will ignore everything except certificates.

Parameters: **data** (*bytes*) – The data.

Returns: A list of `Certificate` .

Raises:

- ValueError – If the PKCS7 data could not be loaded.
- cryptography.exceptions.UnsupportedAlgorithm – If the PKCS7 data is of a type that is not supported.

`cryptography.hazmat.primitives.serialization.pkcs7.load_der_pkcs7_certificates`(*data*)

New in version 3.1.

Deserialize a DER encoded PKCS7 blob to a list of certificates. PKCS7 can contain many other types of data, including CRLs, but this function will ignore everything except certificates.

Parameters: **data** (*bytes*) – The data.

Returns: A list of `Certificate`.

Raises:

- ValueError – If the PKCS7 data could not be loaded.
- cryptography.exceptions.UnsupportedAlgorithm – If the PKCS7 data is of a type that is not supported.

class `cryptography.hazmat.primitives.serialization.pkcs7.PKCS7SignatureBuilder`

The PKCS7 signature builder can create both basic PKCS7 signed messages as well as S/MIME messages, which are commonly used in email. S/MIME has multiple versions, but this implements a subset of RFC 2632, also known as S/MIME Version 3.

New in version 3.2.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.primitives import hashes, serialization
>>> from cryptography.hazmat.primitives.serialization import pkcs7
>>> cert = x509.load_pem_x509_certificate(ca_cert)
>>> key = serialization.load_pem_private_key(ca_key, None)
>>> options = [pkcs7.PKCS7Options.DetachedSignature]
>>> pkcs7.PKCS7SignatureBuilder().set_data(
...     b"data to sign"
... ).add_signer(
...     cert, key, hashes.SHA256()
... ).sign(
...     serialization.Encoding.SMIME, options
... )
b'...'
```

`set_data`(*data*)

Parameters: **data** (bytes-like) – The data to be hashed and signed.

`add_signer`(*certificate*, *private_key*, *hash_algorithm*)

Parameters:

- **certificate** – The `Certificate`.
- **private_key** – The `RSAPrivateKey` or `EllipticCurvePrivateKey` associated with the certificate provided.
- **hash_algorithm** – The `HashAlgorithm` that will be used to generate the signature. This must be an instance of `SHA1`, `SHA224`, `SHA256`, `SHA384`, or `SHA512`.

`add_certificate`(*certificate*)

Add an additional certificate (typically used to help build a verification chain) to the PKCS7 structure. This method may be called multiple times to add as many certificates as desired.

Parameters: **certificate** – The `Certificate` to add.

`sign`(*encoding*, *options*, *backend=None*)

Parameters:

- **encoding** – `PEM`, `DER`, or `SMIME`.
- **options** – A list of `PKCS7Options`.
- **backend** – An optional backend.

Return bytes: The signed PKCS7 message.

class **`cryptography.hazmat.primitives.serialization.pkcs7.PKCS7Options`**

New in version 3.2.

An enumeration of options for PKCS7 signature creation.

`Text`

The text option adds `text/plain` headers to an S/MIME message when serializing to `SMIME`. This option is disallowed with `DER` serialization.

`Binary`

Signing normally converts line endings (LF to CRLF). When passing this option the data will not be converted.

`DetachedSignature`

Don't embed the signed data within the ASN.1. When signing with `SMIME` this also results in the data being added as clear text before the PEM encoded structure.

`NoCapabilities`

PKCS7 structures contain a `MIMECapabilities` section inside the `authenticatedAttributes`. Passing this as an option removes `MIMECapabilities`.

`NoAttributes`

PKCS7 structures contain an `authenticatedAttributes` section. Passing this as an option removes that section. Note that if you pass `NoAttributes` you can't pass `NoCapabilities` since `NoAttributes` removes `MIMECapabilities` and more.

`NoCerts`

Don't include the signer's certificate in the PKCS7 structure. This can reduce the size of the signature but requires that the recipient can obtain the signer's certificate by other means (for example from a previously signed message).

Serialization Formats

class **`cryptography.hazmat.primitives.serialization.PrivateFormat`**

New in version 0.8.

An enumeration for private key formats. Used with the `private_bytes` method available on `RSAPrivateKeyWithSerialization` , `EllipticCurvePrivateKeyWithSerialization` , `DHPrivateKeyWithSerialization` and `DSAPrivateKeyWithSerialization` .

`TraditionalOpenSSL`

Frequently known as PKCS#1 format. Still a widely used format, but generally considered legacy.

A PEM encoded RSA key will look like:

```
-----BEGIN RSA PRIVATE KEY-----
...
-----END RSA PRIVATE KEY-----
```

`PKCS8`

A more modern format for serializing keys which allows for better encryption. Choose this unless you have explicit legacy compatibility requirements.

A PEM encoded key will look like:

```
-----BEGIN PRIVATE KEY-----
...
-----END PRIVATE KEY-----
```

`Raw`

New in version 2.5.

A raw format used by X448 key exchange. It is a binary format and is invalid for other key types.

OpenSSH

New in version 3.0.

Custom private key format for OpenSSH, internals are based on SSH protocol and not ASN1. Requires `PEM` encoding.

A PEM encoded OpenSSH key will look like:

```
-----BEGIN OPENSSH PRIVATE KEY-----
...
-----END OPENSSH PRIVATE KEY-----
```

class **`cryptography.hazmat.primitives.serialization.PublicFormat`**

New in version 0.8.

An enumeration for public key formats. Used with the `public_bytes` method available on `RSAPublicKeyWithSerialization` , `EllipticCurvePublicKeyWithSerialization` , `DHPublicKeyWithSerialization` , and `DSAPublicKeyWithSerialization` .

`SubjectPublicKeyInfo`

This is the typical public key format. It consists of an algorithm identifier and the public key as a bit string. Choose this unless you have specific needs.

A PEM encoded key will look like:

```
-----BEGIN PUBLIC KEY-----
...
-----END PUBLIC KEY-----
```

`PKCS1`

Just the public key elements (without the algorithm identifier). This format is RSA only, but is used by some older systems.

A PEM encoded key will look like:

```
-----BEGIN RSA PUBLIC KEY-----
...
-----END RSA PUBLIC KEY-----
```

`OpenSSH`

New in version 1.4.

The public key format used by OpenSSH (e.g. as found in `~/.ssh/id_rsa.pub` or `~/.ssh/authorized_keys`).

`Raw`

New in version 2.5.

A raw format used by X448 key exchange. It is a binary format and is invalid for other key types.

`CompressedPoint`

New in version 2.5.

A compressed elliptic curve public key as defined in ANSI X9.62 section 4.3.6 (as well as SEC 1 v2.0).

`UncompressedPoint`

New in version 2.5.

An uncompressed elliptic curve public key as defined in ANSI X9.62 section 4.3.6 (as well as SEC 1 v2.0).

class **`cryptography.hazmat.primitives.serialization.ParameterFormat`**

New in version 2.0.

An enumeration for parameters formats. Used with the `parameter_bytes` method available on `DHParametersWithSerialization`.

`PKCS3`

ASN1 DH parameters sequence as defined in PKCS3.

Serialization Encodings

class **`cryptography.hazmat.primitives.serialization.Encoding`**

An enumeration for encoding types. Used with the `private_bytes` method available on `RSAPrivateKeyWithSerialization` , `EllipticCurvePrivateKeyWithSerialization` , `DHPrivateKeyWithSerialization`, `DSAPrivateKeyWithSerialization`, and `X448PrivateKey` as well as `public_bytes` on `RSAPublicKey`, `DHPublicKey`, `EllipticCurvePublicKey`, and `X448PublicKey`.

`PEM`

New in version 0.8.

For PEM format. This is a base64 format with delimiters.

`DER`

New in version 0.9.

For DER format. This is a binary format.

`OpenSSH`

New in version 1.4.

The format used by OpenSSH public keys. This is a text format.

`Raw`

New in version 2.5.

A raw format used by X448 key exchange. It is a binary format and is invalid for other key types.

`X962`

New in version 2.5.

The format used by elliptic curve point encodings. This is a binary format.

`SMIME`

New in version 3.2.

An output format used for PKCS7. This is a text format.

Serialization Encryption Types

class **`cryptography.hazmat.primitives.serialization.KeySerializationEncryption`**

Objects with this interface are usable as encryption types with methods like `private_bytes` available on `RSAPrivateKey` , `EllipticCurvePrivateKey` , `DHPrivateKey` and `DSAPrivateKey` . All other classes in this section represent the available choices for encryption and have this interface.

class **`cryptography.hazmat.primitives.serialization.BestAvailableEncryption`**(*password*)

Encrypt using the best available encryption for a given key's backend. This is a curated encryption choice and the algorithm may change over time.

Parameters: **password** (*bytes*) – The password to use for encryption.

class **`cryptography.hazmat.primitives.serialization.NoEncryption`**

Do not encrypt.