

public_key

Copyright © 2008-2026 Ericsson AB, All Rights Reserved

public_key 1.15.1.7
November 2, 2026

Copyright © 2008-2026 Ericsson AB, All Rights Reserved
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. The Initial Developer of the Original Code is Ericsson AB. Ericsson AB, All Rights Reserved.

November 2, 2026

1 public_key User's Guide

This application provides an API to public-key infrastructure from **RFC 5280** (X.509 certificates) and public-key formats defined by the **PKCS** standard.

1.1 Introduction

1.1.1 Purpose

The Public Key application deals with public-key related file formats, digital signatures, and **X-509 certificates**. It is a library application that provides encode/decode, sign/verify, encrypt/decrypt, and similar functionality. It does not read or write files, it expects or returns file contents or partial file contents as binaries.

1.1.2 Prerequisites

It is assumed that the reader is familiar with the Erlang programming language and has a basic understanding of the concepts of using public-keys and digital certificates.

1.1.3 Performance Tips

The Public Key decode- and encode-functions try to use the NIFs in the ASN.1 compilers runtime modules, if they can be found. Thus, to have the ASN1 application in the path of your system gives the best performance.

1.2 Public-Key Records

This chapter briefly describes Erlang records derived from ASN.1 specifications used to handle public key infrastructure. The scope is to describe the data types of each component, not the semantics. For information on the semantics, refer to the relevant standards and RFCs linked in the sections below.

Use the following include directive to get access to the records and constant macros described in the following sections:

```
-include_lib("public_key/include/public_key.hrl").
```

1.2.1 Data Types

Common non-standard Erlang data types used to describe the record fields in the following sections and which are not defined in the Public Key Reference Manual follows here:

```
time() =
    utc_time() | general_time()
utc_time() =
    {utcTime, "YYMMDDHHMMSSZ"}
general_time() =
    {generalTime, "YYYYMMDDHHMMSSZ"}
general_name() =
    {rfc822Name, string()}
```

```
    | {dNSName, string()}
    | {x400Address, string()}
    | {directoryName, {rdnSequence, [#AttributeTypeAndValue'{}]}}
    | {ediPartyName, special_string()}
    | {ediPartyName, special_string(), special_string()}
    | {uniformResourceIdentifier, string()}
    | {iPAddress, string()}
    | {registeredId, oid()}
    | {otherName, term()}
special_string() =
    {teletexString, string()}
    | {printableString, string()}
    | {universalString, string()}
    | {utf8String, binary()}
    | {bmpString, string()}
dist_reason() =
    unused
    | keyCompromise
    | cACompromise
    | affiliationChanged
    | superseded
    | cessationOfOperation
    | certificateHold
    | privilegeWithdrawn
    | aACompromise
OID_macro() =
    ?OID_name()
OID_name() =
    atom()
```

1.2.2 RSA

Erlang representation of **Rivest-Shamir-Adleman cryptosystem (RSA)** keys follows:

```
#'RSAPublicKey'{
   modulus,       % integer()
   publicExponent % integer()
   }.

#'RSAPrivateKey'{
          version,         % two-prime | multi
   modulus,         % integer()
   publicExponent,  % integer()
   privateExponent, % integer()
   prime1,          % integer()
   prime2,          % integer()
   exponent1,       % integer()
   exponent2,       % integer()
   coefficient,     % integer()
   otherPrimeInfos  % [#OtherPrimeInfo{}] | asn1_NOVALUE
  }.

#'OtherPrimeInfo'{
 prime,           % integer()
 exponent,        % integer()
 coefficient      % integer()
  }.

#'RSASSA-PSS-params'{hashAlgorithm,     % #'HashAlgorithm'{}},
              maskGenAlgorithm,  % #'MaskGenAlgorithm'{}},
       saltLength,        % integer(),
       trailerField,      % integer()
       }.

#'HashAlgorithm'{algorithm,  % oid()
                 parameters  % defaults to asn1_NOVALUE
                 }.

#'MaskGenAlgorithm'{algorithm,  % oid()
                    parameters, % defaults to asn1_NOVALUE
                   }.
```

1.2.3 DSA

Erlang representation of **Digital Signature Algorithm (DSA)** keys

```
#'DSAPrivateKey',{
   version,      % integer()
   p,            % integer()
   q,            % integer()
   g,            % integer()
   y,            % integer()
   x             % integer()
   }.

#'Dss-Parms',{
         p,         % integer()
  q,         % integer()
  g          % integer()
  }.
```

1.2.4 ECDSA and EDDSA

Erlang representation of **Elliptic Curve Digital Signature Algorithm (ECDSA)** and **Edwards-Curve Digital Signature Algorithm (EDDSA)** where parameters in the private key will be `{namedCurve, ?'id-Ed25519' | ?'id-Ed448'}`.

```
#'ECPrivateKey'{
          version,       % integer()
   privateKey,    % binary()
          parameters,    % {ecParameters, #'ECParameters'{}} |
                         % {namedCurve, Oid::tuple()} |
                         % {implicitlyCA, 'NULL'}
   publicKey      % bitstring()
   }.

#'ECParameters'{
      version,    % integer()
      fieldID,    % #'FieldID'{}
      curve,      % #'Curve'{}
      base,       % binary()
      order,      % integer()
      cofactor    % integer()
      }.

#'Curve'{
 a,        % binary()
 b,        % binary()
 seed      % bitstring() - optional

 }.

#'FieldID'{
 fieldType,    % oid()
 parameters    % Depending on fieldType
 }.

#'ECPoint'{
      point %  binary() - the public key
      }.
```

1.2.5 PKIX Certificates

Erlang representation of PKIX certificates derived from ASN.1 specifications see also **X509 certificates (RFC 5280)**, also referred to as `plain` type, are as follows:

```
#'Certificate'{
  tbsCertificate,        % #'TBSCertificate'{}
  signatureAlgorithm,    % #'AlgorithmIdentifier'{}
  signature              % bitstring()
        }.

#'TBSCertificate'{
   version,              % v1 | v2 | v3
   serialNumber,         % integer()
   signature,            % #'AlgorithmIdentifier'{}
   issuer,               % {rdnSequence, [#AttributeTypeAndValue'{}]}
   validity,             % #'Validity'{}
   subject,              % {rdnSequence, [#AttributeTypeAndValue'{}]}
   subjectPublicKeyInfo, % #'SubjectPublicKeyInfo'{}
   issuerUniqueID,       % binary() | asn1_novalue
   subjectUniqueID,      % binary() | asn1_novalue
   extensions            % [#'Extension'{}]
  }.

#'AlgorithmIdentifier'{
   algorithm,  % oid()
   parameters  % der_encoded()
  }.
```

Erlang alternate representation of PKIX certificate, also referred to as `otp` type

```
#'OTPCertificate'{
  tbsCertificate,        % #'OTPTBSCertificate'{}
  signatureAlgorithm,    % #'SignatureAlgorithm'
  signature              % bitstring()
        }.

#'OTPTBSCertificate'{
   version,              % v1 | v2 | v3
   serialNumber,         % integer()
   signature,            % #'SignatureAlgorithm'
   issuer,               % {rdnSequence, [#AttributeTypeAndValue'{}]}
   validity,             % #'Validity'{}
   subject,              % {rdnSequence, [#AttributeTypeAndValue'{}]}
   subjectPublicKeyInfo, % #'OTPSubjectPublicKeyInfo'{}
   issuerUniqueID,       % binary() | asn1_novalue
   subjectUniqueID,      % binary() | asn1_novalue
   extensions            % [#'Extension'{}]
  }.

#'SignatureAlgorithm'{
   algorithm,  % id_signature_algorithm()
   parameters  % asn1_novalue | #'Dss-Parms'{}
  }.
```

`id_signature_algorithm() = OID_macro()`

The available OID names are as follows:

OID Name
id-dsa-with-sha1
id-dsaWithSHA1 (ISO or OID to above)
md2WithRSAEncryption
md5WithRSAEncryption
sha1WithRSAEncryption
sha-1WithRSAEncryption (ISO or OID to above)
sha224WithRSAEncryption
sha256WithRSAEncryption
sha512WithRSAEncryption
ecdsa-with-SHA1

Table 2.1: Signature Algorithm OIDs

The data type `'AttributeTypeAndValue'`, is represented as the following erlang record:

```
#'AttributeTypeAndValue'{
   type,   % id_attributes()
   value   % term()
  }.
```

The attribute OID name atoms and their corresponding value types are as follows:

OID Name	Value Type
id-at-name	special_string()
id-at-surname	special_string()
id-at-givenName	special_string()
id-at-initials	special_string()
id-at-generationQualifier	special_string()
id-at-commonName	special_string()
id-at-localityName	special_string()
id-at-stateOrProvinceName	special_string()
id-at-organizationName	special_string()
id-at-title	special_string()
id-at-dnQualifier	{printableString, string()}
id-at-countryName	{printableString, string()}
id-at-serialNumber	{printableString, string()}
id-at-pseudonym	special_string()

Table 2.2: Attribute OIDs

The data types `'Validity'`, `'SubjectPublicKeyInfo'`, and `'SubjectPublicKeyInfoAlgorithm'` are represented as the following Erlang records:

```
#'Validity'{
   notBefore, % time()
   notAfter   % time()
  }.

#'SubjectPublicKeyInfo'{
   algorithm,       % #AlgorithmIdentifier{}
   subjectPublicKey % binary()
  }.

#'SubjectPublicKeyInfoAlgorithm'{
   algorithm,  % id_public_key_algorithm()
   parameters  % public_key_params()
  }.
```

The public-key algorithm OID name atoms are as follows:

OID Name
rsaEncryption
id-dsa
dhpublicnumber
id-keyExchangeAlgorithm
id-ecPublicKey

Table 2.3: Public-Key Algorithm OIDs

```
#'Extension'{
   extnID,    % id_extensions() | oid()
   critical,  % boolean()
   extnValue  % der_encoded()
  }.
```

`id_extensions()` Standard Certificate Extensions, Private Internet Extensions, CRL Extensions and CRL Entry Extensions.

1.2.6 Standard Certificate Extensions

The standard certificate extensions OID name atoms and their corresponding value types are as follows:

OID Name	Value Type
id-ce-authorityKeyIdentifier	#'AuthorityKeyIdentifier'{}
id-ce-subjectKeyIdentifier	oid()
id-ce-keyUsage	[key_usage()]
id-ce-privateKeyUsagePeriod	#'PrivateKeyUsagePeriod'{}
id-ce-certificatePolicies	#'PolicyInformation'{}

id-ce-policyMappings	#'PolicyMappings_SEQOF'{}
id-ce-subjectAltName	general_name()
id-ce-issuerAltName	general_name()
id-ce-subjectDirectoryAttributes	[#'Attribute'{}]
id-ce-basicConstraints	#'BasicConstraints'{}
id-ce-nameConstraints	#'NameConstraints'{}
id-ce-policyConstraints	#'PolicyConstraints'{}
id-ce-extKeyUsage	[id_key_purpose()]
id-ce-cRLDistributionPoints	[#'DistributionPoint'{}]
id-ce-inhibitAnyPolicy	integer()
id-ce-freshestCRL	[#'DistributionPoint'{}]

Table 2.4: Standard Certificate Extensions

Here:

```
key_usage()
    =
    digitalSignature
    | nonRepudiation
    | keyEncipherment
    | dataEncipherment
    | keyAgreement
    | keyCertSign
    | cRLSign
    | encipherOnly
    | decipherOnly
```

And for `id_key_purpose()`:

OID Name
id-kp-serverAuth
id-kp-clientAuth
id-kp-codeSigning
id-kp-emailProtection

id-kp-timeStamping
id-kp-OCSPSigning

Table 2.5: Key Purpose OIDs

```
#'AuthorityKeyIdentifier'{
   keyIdentifier,     % oid()
   authorityCertIssuer,      % general_name()
   authorityCertSerialNumber % integer()
  }.

#'PrivateKeyUsagePeriod'{
   notBefore,   % general_time()
   notAfter     % general_time()
  }.

#'PolicyInformation'{
   policyIdentifier,  % oid()
   policyQualifiers   % [#PolicyQualifierInfo{}]
  }.

#'PolicyQualifierInfo'{
   policyQualifierId,   % oid()
   qualifier            % string() | #'UserNotice'{}
  }.

#'UserNotice'{
         noticeRef,   % #'NoticeReference'{}
  explicitText % string()
  }.

#'NoticeReference'{
         organization,    % string()
  noticeNumbers    % [integer()]
  }.

#'PolicyMappings_SEQOF'{
   issuerDomainPolicy,  % oid()
   subjectDomainPolicy  % oid()
  }.

#'Attribute'{
          type,  % oid()
   values % [der_encoded()]
   }).

#'BasicConstraints'{
   cA,      % boolean()
   pathLenConstraint % integer()
  }).

#'NameConstraints'{
   permittedSubtrees, % [#'GeneralSubtree'{}]
   excludedSubtrees   % [#'GeneralSubtree'{}]
  }).

#'GeneralSubtree'{
   base,    % general_name()
   minimum, % integer()
   maximum  % integer()
  }).

#'PolicyConstraints'{
   requireExplicitPolicy, % integer()
   inhibitPolicyMapping   % integer()
  }).

#'DistributionPoint'{
   distributionPoint, % {fullName, [general_name()]} | {nameRelativeToCRLIssuer,
   [#AttributeTypeAndValue{}]}
```

```
   reasons,           % [dist_reason()]
   cRLIssuer          % [general_name()]
  }).
```

1.2.7 Private Internet Extensions

The private internet extensions OID name atoms and their corresponding value types are as follows:

OID Name	Value Type
id-pe-authorityInfoAccess	[#'AccessDescription'{}]
id-pe-subjectInfoAccess	[#'AccessDescription'{}]

Table 2.6: Private Internet Extensions

```
#'AccessDescription'{
          accessMethod,    % oid()
    accessLocation   % general_name()
  }).
```

1.2.8 CRL and CRL Extensions Profile

Erlang representation of CRL and CRL extensions profile derived from ASN.1 specifications and RFC 5280 are as follows:

```
#'CertificateList'{
          tbsCertList,        % #'TBSCertList{}
          signatureAlgorithm, % #'AlgorithmIdentifier'{}
          signature           % bitstring()
   }).

#'TBSCertList'{
      version,             % v2 (if defined)
      signature,           % #AlgorithmIdentifier{}
      issuer,              % {rdnSequence, [#AttributeTypeAndValue'{}]}
      thisUpdate,          % time()
      nextUpdate,          % time()
      revokedCertificates, % [#'TBSCertList_revokedCertificates_SEQOF'{}]
      crlExtensions        % [#'Extension'{}]
      }).

#'TBSCertList_revokedCertificates_SEQOF'{
         userCertificate,      % integer()
   revocationDate,       % timer()
  crlEntryExtensions    % [#'Extension'{}]
  }).
```

CRL Extensions

The CRL extensions OID name atoms and their corresponding value types are as follows:

OID Name	Value Type
id-ce-authorityKeyIdentifier	#'AuthorityKeyIdentifier{}

id-ce-issuerAltName	{rdnSequence, [#AttributeTypeAndValue'{}]}
id-ce-cRLNumber	integer()
id-ce-deltaCRLIndicator	integer()
id-ce-issuingDistributionPoint	#'IssuingDistributionPoint'{}
id-ce-freshestCRL	[#'Distributionpoint'{}]

Table 2.7: CRL Extensions

Here, the data type `'IssuingDistributionPoint'` is represented as the following Erlang record:

```
#'IssuingDistributionPoint'{
          distributionPoint,         % {fullName, [general_name()]} | {nameRelativeToCRLIssuer,
   [#AttributeTypeAndValue'{}]}
   onlyContainsUserCerts,     % boolean()
   onlyContainsCACerts,       % boolean()
   onlySomeReasons,           % [dist_reason()]
   indirectCRL,               % boolean()
   onlyContainsAttributeCerts % boolean()
   }).
```

CRL Entry Extensions

The CRL entry extensions OID name atoms and their corresponding value types are as follows:

OID Name	**Value Type**
id-ce-cRLReason	crl_reason()
id-ce-holdInstructionCode	oid()
id-ce-invalidityDate	general_time()
id-ce-certificateIssuer	general_name()

Table 2.8: CRL Entry Extensions

Here:

```
crl_reason()
    =
    unspecified
    | keyCompromise
    | cACompromise
    | affiliationChanged
    | superseded
    | cessationOfOperation
```

```
   | certificateHold
   | removeFromCRL
   | privilegeWithdrawn
   | aACompromise
```

PKCS#10 Certification Request

Erlang representation of a PKCS#10 certification request derived from ASN.1 specifications and RFC 5280 are as follows:

```
#'CertificationRequest'{
          certificationRequestInfo #'CertificationRequestInfo'{},
   signatureAlgorithm    #'CertificationRequest_signatureAlgorithm'{}}.
   signature               bitstring()
   }

#'CertificationRequestInfo'{
   version       atom(),
   subject       {rdnSequence, [#AttributeTypeAndValue'{}]} ,
   subjectPKInfo #'CertificationRequestInfo_subjectPKInfo'{},
   attributes    [#'AttributePKCS-10' {}]
   }

#'CertificationRequestInfo_subjectPKInfo'{
          algorithm  #'CertificationRequestInfo_subjectPKInfo_algorithm'{}
   subjectPublicKey    bitstring()
   }

#'CertificationRequestInfo_subjectPKInfo_algorithm'{
     algorithm = oid(),
     parameters = der_encoded()
}

#'CertificationRequest_signatureAlgorithm'{
     algorithm = oid(),
     parameters = der_encoded()
     }

#'AttributePKCS-10'{
    type = oid(),
    values = [der_encoded()]
}
```

1.3 Getting Started

This section describes examples of how to use the Public Key API. Keys and certificates used in the following sections are generated only for testing the Public Key application.

Some shell printouts in the following examples are abbreviated for increased readability.

1.3.1 PEM Files

Public-key data (keys, certificates, and so on) can be stored in Privacy Enhanced Mail (PEM) format. The PEM files have the following structure:

```
    <text>
    -----BEGIN <SOMETHING>-----
    <Attribute> : <Value>
    <Base64 encoded DER data>
    -----END <SOMETHING>-----
    <text>
```

A file can contain several `BEGIN/END` blocks. Text lines between blocks are ignored. Attributes, if present, are ignored except for `Proc-Type` and `DEK-Info`, which are used when `DER` data is encrypted.

DSA Private Key

A DSA private key can look as follows:

Note:

File handling is not done by the Public Key application.

```
1> {ok, PemBin} = file:read_file("dsa.pem").
{ok,<<"-----BEGIN DSA PRIVATE KEY-----\nMIIBuw"...>>}
```

The following PEM file has only one entry, a private DSA key:

```
2> [DSAEntry] =  public_key:pem_decode(PemBin).
[{'DSAPrivateKey',<<48,130,1,187,2,1,0,2,129,129,0,183,
                    179,230,217,37,99,144,157,21,228,204,
      162,207,61,246,...>>,
      not_encrypted}]
```

```
3> Key = public_key:pem_entry_decode(DSAEntry).
#'DSAPrivateKey'{version = 0,
                 p = 12900045185019966618...6593,
                 q = 1216700114794736143432235288305776850295620488937,
                 g = 10442040227452349332...47213,
                 y = 872568079800305090 74...403143,
                 x = 5109685298560121463513173638073665750756458396 54}
```

RSA Private Key with Password

An RSA private key encrypted with a password can look as follows:

```
1> {ok, PemBin} = file:read_file("rsa.pem").
{ok,<<"Bag Attribute"...>>}
```

The following PEM file has only one entry, a private RSA key:

```
2>[RSAEntry] = public_key:pem_decode(PemBin).
[{'RSAPrivateKey',<<224,108,117,203,152,40,15,77,128,126,
                    221,195,154,249,85,208,202,251,109,
                    119,120,57,29,89,19,9,...>>,
                  {"DES-EDE3-CBC",<<"kÙeø¼pµL">>}}]
```

In this following example, the password is `"abcd1234"`:

```
3> Key = public_key:pem_entry_decode(RSAEntry, "abcd1234").
    #'RSAPrivateKey'{version = 'two-prime',
                     modulus = 1112355156729921663373...2737107,
                     publicExponent = 65537,
                     privateExponent = 58064406231183...2239766033,
                     prime1 = 110347666146565984840 98...7326883017,
                     prime2 = 10080459293561036618240...77738643771,
                     exponent1 = 77928819327425934607...22152984217,
                     exponent2 = 36287623121853605733...20588523793,
                     coefficient = 924840412626098444...41820968343,
                     otherPrimeInfos = asn1_NOVALUE}
```

X509 Certificates

The following is an example of X509 certificates:

```
1> {ok, PemBin} = file:read_file("cacerts.pem").
{ok,<<"-----BEGIN CERTIFICATE-----\nMIIC7jCCAl"...>>}
```

The following file includes two certificates:

```
2> [CertEntry1, CertEntry2] = public_key:pem_decode(PemBin).
[{'Certificate',<<48,130,2,238,48,130,2,87,160,3,2,1,2,2,
                  9,0,230,145,97,214,191,2,120,150,48,13,
                  ...>>,
                not_encrypted},
 {'Certificate',<<48,130,3,200,48,130,3,49,160,3,2,1,2,2,1,
                  1,48,13,6,9,42,134,72,134,247,...>>,
                not_encrypted}]
```

Certificates can be decoded as usual:

```
2> Cert = public_key:pem_entry_decode(CertEntry1).
#'Certificate'{
    tbsCertificate =
        #'TBSCertificate'{
            version = v3,serialNumber = 16614168075301976214,
            signature =
                #'AlgorithmIdentifier'{
                    algorithm = {1,2,840,113549,1,1,5},
                    parameters = <<5,0>>},
            issuer =
                {rdnSequence,
                    [[#'AttributeTypeAndValue'{
                          type = {2,5,4,3},
                          value = <<19,8,101,114,108,97,110,103,67,65>>}],
                     [#'AttributeTypeAndValue'{
                          type = {2,5,4,11},
                          value = <<19,10,69,114,108,97,110,103,32,79,84,80>>}],
                     [#'AttributeTypeAndValue'{
                          type = {2,5,4,10},
                          value = <<19,11,69,114,105,99,115,115,111,110,32,65,66>>}],
                     [#'AttributeTypeAndValue'{
                          type = {2,5,4,7},
                          value = <<19,9,83,116,111,99,107,104,111,108,109>>}],
                     [#'AttributeTypeAndValue'{
                          type = {2,5,4,6},
                          value = <<19,2,83,69>>}],
                     [#'AttributeTypeAndValue'{
                          type = {1,2,840,113549,1,9,1},
                          value = <<22,22,112,101,116,101,114,64,101,114,...>>}]]},
            validity =
                #'Validity'{
                    notBefore = {utcTime,"0801090829292"},
                    notAfter = {utcTime,"0802080829292"}},
            subject =
                {rdnSequence,
                    [[#'AttributeTypeAndValue'{
                          type = {2,5,4,3},
                          value = <<19,8,101,114,108,97,110,103,67,65>>}],
                     [#'AttributeTypeAndValue'{
                          type = {2,5,4,11},
                          value = <<19,10,69,114,108,97,110,103,32,79,84,80>>}],
                     [#'AttributeTypeAndValue'{
                          type = {2,5,4,10},
                          value = <<19,11,69,114,105,99,115,115,111,110,32,...>>}],
                     [#'AttributeTypeAndValue'{
                          type = {2,5,4,7},
                          value = <<19,9,83,116,111,99,107,104,111,108,...>>}],
                     [#'AttributeTypeAndValue'{
                          type = {2,5,4,6},
                          value = <<19,2,83,69>>}],
                     [#'AttributeTypeAndValue'{
                          type = {1,2,840,113549,1,9,1},
                          value = <<22,22,112,101,116,101,114,64,...>>}]]},
            subjectPublicKeyInfo =
                #'SubjectPublicKeyInfo'{
                    algorithm =
                        #'AlgorithmIdentifier'{
                            algorithm = {1,2,840,113549,1,1,1},
                            parameters = <<5,0>>},
                    subjectPublicKey =
                        {0,<<48,129,137,2,129,129,0,203,209,187,77,73,231,90,...>>}},
            issuerUniqueID = asn1_NOVALUE,
            subjectUniqueID = asn1_NOVALUE,
            extensions =
                [#'Extension'{
```

```
                    extnID = {2,5,29,19},
                    critical = true,
                    extnValue = [48,3,1,1,255]},
                #'Extension'{
                    extnID = {2,5,29,15},
                    critical = false,
                    extnValue = [3,2,1,6]},
                #'Extension'{
                    extnID = {2,5,29,14},
                    critical = false,
                    extnValue = [4,20,27,217,65,152,6,30,142|...]},
                #'Extension'{
                    extnID = {2,5,29,17},
                    critical = false,
                    extnValue = [48,24,129,22,112,101,116,101|...]}]},
    signatureAlgorithm =
        #'AlgorithmIdentifier'{
            algorithm = {1,2,840,113549,1,1,5},
            parameters = <<5,0>>},
    signature =
    <<163,186,7,163,216,152,63,47,154,234,139,73,154,96,120,
    165,2,52,196,195,109,167,192,...>>}
```

Parts of certificates can be decoded with `public_key:der_decode/2`, using the ASN.1 type of that part. However, an application-specific certificate extension requires application-specific ASN.1 decode/encode-functions. In the recent example, the first value of `rdnSequence` is of ASN.1 type `'X520CommonName'. ({2,5,4,3} = ?id-at-commonName)`:

```
public_key:der_decode('X520CommonName', <<19,8,101,114,108,97,110,103,67,65>>).
{printableString,"erlangCA"}
```

However, certificates can also be decoded using `pkix_decode_cert/2`, which can customize and recursively decode standard parts of a certificate:

```
3>{_, DerCert, _} = CertEntry1.
```

```
4> public_key:pkix_decode_cert(DerCert, otp).
#'OTPCertificate'{
    tbsCertificate =
        #'OTPTBSCertificate'{
            version = v3,serialNumber = 16614168075301976214,
            signature =
                #'SignatureAlgorithm'{
                    algorithm = {1,2,840,113549,1,1,5},
                    parameters = 'NULL'},
            issuer =
                {rdnSequence,
                    [[#'AttributeTypeAndValue'{
                          type = {2,5,4,3},
                          value = {printableString,"erlangCA"}}],
                     [#'AttributeTypeAndValue'{
                          type = {2,5,4,11},
                          value = {printableString,"Erlang OTP"}}],
                     [#'AttributeTypeAndValue'{
                          type = {2,5,4,10},
                          value = {printableString,"Ericsson AB"}}],
                     [#'AttributeTypeAndValue'{
                          type = {2,5,4,7},
                          value = {printableString,"Stockholm"}}],
                     [#'AttributeTypeAndValue'{type = {2,5,4,6},value = "SE"}],
                     [#'AttributeTypeAndValue'{
                          type = {1,2,840,113549,1,9,1},
                          value = "peter@erix.ericsson.se"}]]},
            validity =
                #'Validity'{
                    notBefore = {utcTime,"080109082929Z"},
                    notAfter = {utcTime,"080208082929Z"}},
            subject =
                {rdnSequence,
                    [[#'AttributeTypeAndValue'{
                          type = {2,5,4,3},
                          value = {printableString,"erlangCA"}}],
                     [#'AttributeTypeAndValue'{
                          type = {2,5,4,11},
                          value = {printableString,"Erlang OTP"}}],
                     [#'AttributeTypeAndValue'{
                          type = {2,5,4,10},
                          value = {printableString,"Ericsson AB"}}],
                     [#'AttributeTypeAndValue'{
                          type = {2,5,4,7},
                          value = {printableString,"Stockholm"}}],
                     [#'AttributeTypeAndValue'{type = {2,5,4,6},value = "SE"}],
                     [#'AttributeTypeAndValue'{
                          type = {1,2,840,113549,1,9,1},
                          value = "peter@erix.ericsson.se"}]]},
            subjectPublicKeyInfo =
                #'OTPSubjectPublicKeyInfo'{
                    algorithm =
                        #'PublicKeyAlgorithm'{
                            algorithm = {1,2,840,113549,1,1,1},
                            parameters = 'NULL'},
                    subjectPublicKey =
                        #'RSAPublicKey'{
                            modulus =
                                1431267547247997...37419,
                            publicExponent = 65537}},
            issuerUniqueID = asn1_NOVALUE,
            subjectUniqueID = asn1_NOVALUE,
            extensions =
                [#'Extension'{
                     extnID = {2,5,29,19},
```

```
                    critical = true,
                    extnValue =
                        #'BasicConstraints'{
                            cA = true,pathLenConstraint = asn1_NOVALUE}},
                #'Extension'{
                    extnID = {2,5,29,15},
                    critical = false,
                    extnValue = [keyCertSign,cRLSign]},
                #'Extension'{
                    extnID = {2,5,29,14},
                    critical = false,
                    extnValue = [27,217,65,152,6,30,142,132,245|...]},
                #'Extension'{
                    extnID = {2,5,29,17},
                    critical = false,
                    extnValue = [{rfc822Name,"peter@erix.ericsson.se"}]}]},
    signatureAlgorithm =
        #'SignatureAlgorithm'{
            algorithm = {1,2,840,113549,1,1,5},
            parameters = 'NULL'},
    signature =
         <<163,186,7,163,216,152,63,47,154,234,139,73,154,96,120,
           165,2,52,196,195,109,167,192,...>>}
```

This call is equivalent to `public_key:pem_entry_decode(CertEntry1)`:

```
5> public_key:pkix_decode_cert(DerCert, plain).
#'Certificate'{ ...}
```

Encoding Public-Key Data to PEM Format

If you have public-key data and want to create a PEM file this can be done by calling functions `public_key:pem_entry_encode/2` and `pem_encode/1` and saving the result to a file. For example, assume that you have `PubKey = 'RSAPublicKey'{}`. Then you can create a PEM-"RSA PUBLIC KEY" file (ASN.1 type `'RSAPublicKey'`) or a PEM-"PUBLIC KEY" file (`'SubjectPublicKeyInfo'` ASN.1 type).

The second element of the PEM-entry is the ASN.1 `DER` encoded key data:

```
1> PemEntry = public_key:pem_entry_encode('RSAPublicKey', RSAPubKey).
{'RSAPublicKey', <<48,72,...>>, not_encrypted}

2> PemBin = public_key:pem_encode([PemEntry]).
<<"-----BEGIN RSA PUBLIC KEY-----\nMEgC...>>

3> file:write_file("rsa_pub_key.pem", PemBin).
ok
```

or:

```
1> PemEntry = public_key:pem_entry_encode('SubjectPublicKeyInfo', RSAPubKey).
{'SubjectPublicKeyInfo', <<48,92...>>, not_encrypted}

2> PemBin = public_key:pem_encode([PemEntry]).
<<"-----BEGIN PUBLIC KEY-----\nMFw...>>

3> file:write_file("pub_key.pem", PemBin).
ok
```

1.3.2 RSA Public-Key Cryptography

Suppose you have the following private key and a corresponding public key:

- `PrivateKey = #'RSAPrivateKey{}'` and the plaintext `Msg = binary()`

- `PublicKey = #'RSAPublicKey'{}`

Then you can proceed as follows:

Encrypt with the private key:

```
RsaEncrypted = public_key:encrypt_private(Msg, PrivateKey),
Msg = public_key:decrypt_public(RsaEncrypted, PublicKey),
```

Encrypt with the public key:

```
RsaEncrypted = public_key:encrypt_public(Msg, PublicKey),
Msg = public_key:decrypt_private(RsaEncrypted, PrivateKey),
```

Note:

You normally do only one of the encrypt or decrypt operations, and the peer does the other. This normally used in legacy applications as a primitive digital signature.

1.3.3 Digital Signatures

Suppose you have the following private key and a corresponding public key:

- `PrivateKey = #'RSAPrivateKey{}'` or `#'DSAPrivateKey'{}` and the plaintext `Msg = binary()`
- `PublicKey = #'RSAPublicKey'{}` or `{integer(), #'DssParams'{}}`

Then you can proceed as follows:

```
Signature = public_key:sign(Msg, sha, PrivateKey),
true = public_key:verify(Msg, sha, Signature, PublicKey),
```

Note:

You normally do only one of the sign or verify operations, and the peer does the other.

It can be appropriate to calculate the message digest before calling `sign` or `verify`, and then use `none` as second argument:

```
Digest = crypto:sha(Msg),
Signature = public_key:sign(Digest, none, PrivateKey),
true = public_key:verify(Digest, none, Signature, PublicKey),
```

1.3.4 Verifying a certificate hostname

Background

When a client checks a server certificate there are a number of checks available like checks that the certificate is not revoked, not forged or not out-of-date.

There are however attacks that are not detected by those checks. Suppose a bad guy has succeeded with a DNS infection. Then the client could believe it is connecting to one host but ends up at another but evil one. Though it is evil, it could have a perfectly legal certificate! The certificate has a valid signature, it is not revoked, the certificate chain is not faked and has a trusted root and so on.

To detect that the server is not the intended one, the client must additionally perform a *hostname verification*. This procedure is described in **RFC 6125**. The idea is that the certificate lists the hostnames it could be fetched from. This

is checked by the certificate issuer when the certificate is signed. So if the certificate is issued by a trusted root the client could trust the host names signed in it.

There is a default hostname matching procedure defined in **RFC 6125, section 6** as well as protocol dependent variations defined in **RFC 6125 appendix B**. The default procedure is implemented in public_key:pkix_verify_hostname/2,3. It is possible for a client to hook in modified rules using the options list.

Some terminology is needed: the certificate presents hostname(s) on which it is valid. Those are called *Presented IDs*. The hostname(s) the client believes it connects to are called *Reference IDs*. The matching rules aims to verify that there is at least one of the Reference IDs that matches one of the Presented IDs. If not, the verification fails.

The IDs contains normal fully qualified domain names like e.g `foo.example.com`, but IP addresses are not recommended. The rfc describes why this is not recommended as well as security considerations about how to acquire the Reference IDs.

Internationalized domain names are not supported.

The verification process

Traditionally the Presented IDs were found in the `Subject` certificate field as `CN` names. This is still quite common. When printing a certificate they show up as:

```
$ openssl x509 -text < cert.pem
...
Subject: C=SE, CN=example.com, CN=*.example.com, O=erlang.org
...
```

The example `Subject` field has one C, two CN and one O part. It is only the CN (Common Name) that is used by hostname verification. The two other (C and O) is not used here even when they contain a domain name like the O part. The C and O parts are defined elsewhere and meaningful only for other functions.

In the example the Presented IDs are `example.com` as well as hostnames matching `*.example.com`. For example `foo.example.com` and `bar.example.com` both matches but not `foo.bar.example.com`. The name `erlang.org` matches neither since it is not a CN.

In case where the Presented IDs are fetched from the `Subject` certificate field, the names may contain wildcard characters. The function handles this as defined in **chapter 6.4.3 in RFC 6125**.

There may only be one wildcard character and that is in the first label, for example: `*.example.com`. This matches `foo.example.com` but neither `example.com` nor `foo.bar.example.com`.

There may be label characters before or/and after the wildcard. For example: `a*d.example.com` matches `abcd.example.com` and `ad.example.com`, but not `ab.cd.example.com`.

In the previous example there is no indication of which protocols are expected. So a client has no indication of whether it is a web server, an ldap server or maybe a sip server it is connected to. There are fields in the certificate that can indicate this. To be more exact, the rfc introduces the usage of the `X509v3 Subject Alternative Name` in the `X509v3 extensions` field:

```
$ openssl x509 -text < cert.pem
...
X509v3 extensions:
    X509v3 Subject Alternative Name:
        DNS:kb.example.org, URI:https://www.example.org
...
```

Here `kb.example.org` serves any protocol while `www.example.org` presents a secure web server.

The next example has both `Subject` and `Subject Alternate Name` present:

```
$ openssl x509 -text < cert.pem
...
Subject: C=SE, CN=example.com, CN=*.example.com, O=erlang.org
...
X509v3 extensions:
    X509v3 Subject Alternative Name:
        DNS:kb.example.org, URI:https://www.example.org
...
```

The RFC states that if a certificate defines Reference IDs in a `Subject Alternate Name` field, the `Subject` field MUST NOT be used for host name checking, even if it contains valid CN names. Therefore only `kb.example.org` and `https://www.example.org` matches. The match fails both for `example.com` and `foo.example.com` because they are in the `Subject` field which is not checked because the `Subject Alternate Name` field is present.

Function call examples

Note:

Other applications like ssl/tls or https might have options that are passed down to the `public_key:pkix_verify_hostname`. You will probably not have to call it directly

Suppose our client expects to connect to the web server https://www.example.net. This URI is therefore the Reference IDs of the client. The call will be:

```
public_key:pkix_verify_hostname(CertFromHost,
                                [{uri_id, "https://www.example.net"}
                                ]).
```

The call will return `true` or `false` depending on the check. The caller do not need to handle the matching rules in the rfc. The matching will proceed as:

- If there is a `Subject Alternate Name` field, the `{uri_id,string()}` in the function call will be compared to any `{uniformResourceIdentifier,string()}` in the Certificate field. If the two `strings()` are equal (case insensitive), there is a match. The same applies for any `{dns_id,string()}` in the call which is compared with all `{dNSName,string()}` in the Certificate field.
- If there is NO `Subject Alternate Name` field, the `Subject` field will be checked. All `CN` names will be compared to all hostnames *extracted* from `{uri_id,string()}` and from `{dns_id,string()}`.

Extending the search mechanism

The caller can use own extraction and matching rules. This is done with the two options `fqdn_fun` and `match_fun`.

Hostname extraction

The `fqdn_fun` extracts hostnames (Fully Qualified Domain Names) from uri_id or other ReferenceIDs that are not pre-defined in the public_key function. Suppose you have some URI with a very special protocol-part: `myspecial://example.com"`. Since this a non-standard URI there will be no hostname extracted for matching CN-names in the `Subject`.

To "teach" the function how to extract, you can give a fun which replaces the default extraction function. The `fqdn_fun` takes one argument and returns either a `string()` to be matched to each CN-name or the atom `default` which will invoke the default fqdn extraction function. The return value `undefined` removes the current URI from the fqdn extraction.

```
...
Extract = fun({uri_id, "myspecial://"++HostName}) -> HostName;
             (_Else) -> default
          end,
...
public_key:pkix_verify_hostname(CertFromHost, RefIDs,
                                [{fqdn_fun, Extract}])
...
```

Re-defining the match operation

The default matching handles dns_id and uri_id. In an uri_id the value is tested for equality with a value from the `Subject Alternate Name`. If some other kind of matching is needed, use the `match_fun` option.

The `match_fun` takes two arguments and returns either `true`, `false` or `default`. The value `default` will invoke the default match function.

```
...
Match = fun({uri_id,"myspecial://"++A},
            {uniformResourceIdentifier,"myspecial://"++B}) ->
                                                  my_match(A,B);
           (_RefID, _PresentedID) ->
                               default
        end,
...
public_key:pkix_verify_hostname(CertFromHost, RefIDs,
                                [{match_fun, Match}]),
...
```

In case of a match operation between a ReferenceID and a CN value from the `Subject` field, the first argument to the fun is the extracted hostname from the ReferenceID, and the second argument is the tuple `{cn, string()}` taken from the `Subject` field. That makes it possible to have separate matching rules for Presented IDs from the `Subject` field and from the `Subject Alternate Name` field.

The default matching transformes the ascii values in strings to lowercase before comparing. The `match_fun` is however called without any transformation applied to the strings. The reason is to enable the user to do unforeseen handling of the strings where the original format is needed.

"Pinning" a Certificate

The **RFC 6125** defines *pinning* as:

"The act of establishing a cached name association between the application service's certificate and one of the client's reference identifiers, despite the fact that none of the presented identifiers matches the given reference identifier. ..."

The purpose is to have a mechanism for a human to accept an otherwise faulty Certificate. In for example a web browser, you could get a question like

Warning: you wanted to visit the site www.example.com, but the certificate is for shop.example.com. Accept anyway (yes/no)?"

This could be accomplished with the option `fail_callback` which will be called if the hostname verification fails:

```
-include_lib("public_key/include/public_key.hrl"). % Record def
...
Fail = fun(#'OTPCertificate'{}=C) ->
             case in_my_cache(C) orelse my_accept(C) of
                 true ->
                      enter_my_cache(C),
                      true;
                 false ->
                      false
        end,
...
public_key:pkix_verify_hostname(CertFromHost, RefIDs,
                                [{fail_callback, Fail}]),
...
```

2 Reference Manual

The `public_key` application provides functions to handle public-key infrastructure from RFC 3280 (X.509 certificates) and parts of the PKCS standard.

public_key

Application

Provides encode/decode of different file formats (PEM, OpenSSH), digital signature and verification functions, validation of certificate paths and certificate revocation lists (CRLs) and other functions for handling of certificates, keys and CRLs.

- Supports **RFC 5280** - Internet X.509 Public-Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Certificate policies supported since OTP-26.2
- Supports **PKCS-1** - RSA Cryptography Standard
- Supports **DSS** - Digital Signature Standard (DSA - Digital Signature Algorithm)
- Supports **PKCS-3** - Diffie-Hellman Key Agreement Standard
- Supports **PKCS-5** - Password-Based Cryptography Standard
- Supports **AES** - Use of the Advanced Encryption Standard (AES) Algorithm in Cryptographic Message Syntax (CMS)
- Supports **PKCS-8** - Private-Key Information Syntax Standard
- Supports **PKCS-10** - Certification Request Syntax Standard

DEPENDENCIES

The `public_key` application uses the Crypto application to perform cryptographic operations and the ASN-1 application to handle PKIX-ASN-1 specifications, hence these applications must be loaded for the `public_key` application to work. In an embedded environment this means they must be started with `application:start/[1,2]` before the `public_key` application is started.

ERROR LOGGER AND EVENT HANDLERS

The `public_key` application is a library application and does not use the error logger. The functions will either succeed or fail with a runtime error.

SEE ALSO

application(3)

public_key

Erlang module

Provides functions to handle public-key infrastructure, for details see public_key(6).

Common Records and ASN.1 Types

Note:

All records used in this Reference Manual are generated from ASN.1 specifications and are documented in the User's Guide. See Public-key Records.

Use the following include directive to get access to the records and constant macros described here and in the User's Guide:

```
-include_lib("public_key/include/public_key.hrl").
```

Data Types

```
oid() = tuple()
```

Object identifier, a tuple of integers as generated by the `ASN.1` compiler.

```
key_oid_name() =
    rsaEncryption | 'id-RSASSA-PSS' | 'id-ecPublicKey' |
    'id-Ed25519' | 'id-Ed448' | 'id-dsa'
```

Macro names for key object identifiers used by prefixing with ?

```
der_encoded() = binary()
pki_asn1_type() =
    'Certificate' | 'RSAPrivateKey' | 'RSAPublicKey' |
    'SubjectPublicKeyInfo' | 'DSAPrivateKey' | 'DHParameter' |
    'PrivateKeyInfo' | 'CertificationRequest' | 'ContentInfo' |
    'CertificateList' | 'ECPrivateKey' | 'OneAsymmetricKey' |
    'EcpkParameters'
asn1_type() = atom()
```

ASN.1 type present in the Public Key applications ASN.1 specifications.

```
pem_entry() =
    {pki_asn1_type(),
     der_or_encrypted_der(),
     not_encrypted | cipher_info()}
der_or_encrypted_der() = binary()
cipher_info() = {cipher(), cipher_info_params()}
cipher() = string()
salt() = binary()
cipher_info_params() =
    salt() |
    {#'PBEParameter'{}, digest_type()} |
```

```
    #'PBES2-params'{}
```

```
Cipher = "RC2-CBC" | "DES-CBC" | "DES-EDE3-CBC"
```

`Salt` could be generated with `crypto:strong_rand_bytes(8)`.

```
public_key() =
    rsa_public_key() |
    rsa_pss_public_key() |
    dsa_public_key() |
    ec_public_key() |
    ed_public_key()
rsa_public_key() = #'RSAPublicKey'{}
dss_public_key() = integer()
rsa_pss_public_key() =
    {rsa_pss_public_key(), #'RSASSA-PSS-params'{}}
dsa_public_key() = {dss_public_key(), #'Dss-Parms'{}}
ec_public_key() = {#'ECPoint'{}, ecpk_parameters_api()}
public_key_params() =
    'NULL' |
    #'RSASSA-PSS-params'{} |
    {namedCurve, oid()} |
    #'ECParameters'{} |
    #'Dss-Parms'{}
ecpk_parameters() =
    {ecParameters, #'ECParameters'{}} |
    {namedCurve, Oid :: tuple()}
ecpk_parameters_api() =
    ecpk_parameters() |
    #'ECParameters'{} |
    {namedCurve, Name :: crypto:ec_named_curve()}
public_key_info() =
    {key_oid_name(),
     rsa_public_key() | #'ECPoint'{} | dss_public_key(),
     public_key_params()}
ed_public_key() = {#'ECPoint'{}, ed_params()}
ed_params() = {namedCurve, ed_oid_name()}
private_key() =
    rsa_private_key() |
    rsa_pss_private_key() |
    dsa_private_key() |
    ec_private_key() |
    ed_private_key()
rsa_private_key() = #'RSAPrivateKey'{}
rsa_pss_private_key() =
```

```
    {#'RSAPrivateKey'{}, #'RSASSA-PSS-params'{}}
dsa_private_key() = #'DSAPrivateKey'{}
ec_private_key() = #'ECPrivateKey'{}
ed_private_key() = #'ECPrivateKey'{parameters = ed_params()}
ed_oid_name() = 'id-Ed25519' | 'id-Ed448'
```

Macro names for object identifiers for EDDSA curves used by prefixing with ?

```
key_params() =
    #'DHParameter'{} |
    {namedCurve, oid()} |
    #'ECParameters'{} |
    {rsa, Size :: integer(), PubExp :: integer()}
digest_type() =
    none | sha1 |
    crypto:rsa_digest_type() |
    crypto:dss_digest_type() |
    crypto:ecdsa_digest_type()
issuer_name() = {rdnSequence, [[#'AttributeTypeAndValue'{}]]}
referenceIDs() = [referenceID()]
referenceID() =
    {uri_id | dns_id | ip | srv_id | atom() | oid(), string()} |
    {ip, inet:ip_address() | string()}
cert_id() = {SerialNr :: integer(), issuer_name()}
cert() = der_cert() | otp_cert()
otp_cert() = #'OTPCertificate'{}
der_cert() = der_encoded()
combined_cert() =
    #cert{der = public_key:der_encoded(),
          otp = #'OTPCertificate'{}}
policy_node() =
    #{valid_policy := public_key:oid(),
      qualifier_set := [#'UserNotice'{} | {uri, string()}],
      expected_policy_set := [public_key:oid()]}
bad_cert_reason() =
    cert_expired | invalid_issuer | invalid_signature |
    name_not_permitted | missing_basic_constraint |
    invalid_key_usage |
    {key_usage_mismatch, term()} |
    duplicate_cert_in_path |
    {policy_requirement_not_met, term()} |
    {invalid_policy_mapping, term()} |
    {revoked, crl_reason()} |
    invalid_validity_dates | policy_tree_exceeded |
    atom()
crl_reason() =
    unspecified | keyCompromise | cACompromise |
    affiliationChanged | superseded | cessationOfOperation |
    certificateHold | privilegeWithdrawn | aACompromise
chain_opts() =
```

```
    #{chain_end() := [cert_opt()],
      intermediates => [[cert_opt()]]}
chain_end() = root | peer
cert_opt() =
    {digest, public_key:digest_type()} |
    {key, public_key:key_params() | public_key:private_key()} |
    {validity,
     {From :: erlang:timestamp(), To :: erlang:timestamp()}} |
    {extensions, [#'Extension'{}]}
test_root_cert() =
    #{cert := der_encoded(), key := public_key:private_key()}
test_config() =
    #{server_config := [conf_opt()],
      client_config := [conf_opt()]}
conf_opt() =
    {cert, public_key:der_encoded()} |
    {key, public_key:private_key()} |
    {cacerts, [public_key:der_encoded()]}
```

Exports

`cacerts_clear() -> boolean()`

Clears any loaded CA certificates, returns true if any was loaded.

`cacerts_get() -> [combined_cert()]`

Returns the trusted CA certificates if any are loaded, otherwise uses cacerts_load/0 to load them. The function fails if no `cacerts` could be loaded.

`cacerts_load() -> ok | {error, Reason :: term()}`

Loads the OS supplied trusted CA certificates.

```
cacerts_load(File :: file:filename_all()) ->
                ok | {error, Reason :: term()}
```

Loads the trusted CA certificates from a file.

`compute_key(OthersECDHkey, MyECDHkey) -> SharedSecret`

Types:

```
OthersECDHkey = #'ECPoint'{}
MyECDHkey = #'ECPrivateKey'{}
SharedSecret = binary()
```

Computes shared secret.

`compute_key(OthersDHkey, MyDHkey, DHparms) -> SharedSecret`

Types:

```
   OthersDHkey = crypto:dh_public()
   MyDHkey = crypto:dh_private()
   DHparms = #'DHParameter'{}
   SharedSecret = binary()
```

Computes shared secret.

```
decrypt_private(CipherText, Key) -> PlainText
decrypt_private(CipherText, Key, Options) -> PlainText
```

Types:

```
   CipherText = binary()
   Key = rsa_private_key()
   Options = crypto:pk_encrypt_decrypt_opts()
   PlainText = binary()
```

Public-key decryption using the private key. See also crypto:private_decrypt/4

```
decrypt_public(CipherText, Key) -> PlainText
decrypt_public(CipherText, Key, Options) -> PlainText
```

Types:

```
   CipherText = binary()
   Key = rsa_public_key()
   Options = crypto:pk_encrypt_decrypt_opts()
   PlainText = binary()
```

Public-key decryption using the public key. See also crypto:public_decrypt/4

```
der_decode(Asn1Type, Der) -> Entity
```

Types:

```
   Asn1Type = asn1_type()
   Der = der_encoded()
   Entity = term()
```

Decodes a public-key ASN.1 DER encoded entity.

```
der_encode(Asn1Type, Entity) -> Der
```

Types:

```
   Asn1Type = asn1_type()
   Entity = term()
   Der = binary()
```

Encodes a public-key entity with ASN.1 DER encoding.

```
dh_gex_group(MinSize, SuggestedSize, MaxSize, Groups) ->
                {ok, {Size, Group}} | {error, term()}
```

Types:

```
MinSize = SuggestedSize = MaxSize = integer() >= 1
Groups = undefined | [{Size, [Group]}]
Size = integer() >= 1
Group = {G, P}
G = P = integer() >= 1
```

Selects a group for Diffie-Hellman key exchange with the key size in the range `MinSize...MaxSize` and as close to `SuggestedSize` as possible. If `Groups == undefined` a default set will be used, otherwise the group is selected from `Groups`.

First a size, as close as possible to SuggestedSize, is selected. Then one group with that key size is randomly selected from the specified set of groups. If no size within the limits of `MinSize` and `MaxSize` is available, `{error,no_group_found}` is returned.

The default set of groups is listed in `lib/public_key/priv/moduli`. This file may be regenerated like this:

```
$> cd $ERL_TOP/lib/public_key/priv/
$> generate
       ---- wait until all background jobs has finished. It may take several days !
$> cat moduli-* > moduli
$> cd ..; make
```

encrypt_private(PlainText, Key) -> CipherText
encrypt_private(PlainText, Key, Options) -> CipherText

Types:

```
PlainText = binary()
Key = rsa_private_key()
Options = crypto:pk_encrypt_decrypt_opts()
CipherText = binary()
```

Public-key encryption using the private key. See also crypto:private_encrypt/4.

encrypt_public(PlainText, Key) -> CipherText
encrypt_public(PlainText, Key, Options) -> CipherText

Types:

```
PlainText = binary()
Key = rsa_public_key()
Options = crypto:pk_encrypt_decrypt_opts()
CipherText = binary()
```

Public-key encryption using the public key. See also crypto:public_encrypt/4.

generate_key(Params :: DHparams | ECparams | RSAparams) ->
DHkeys | ECkey | RSAkey

Types:

```
DHparams = #'DHParameter'{}
DHkeys = {PublicDH :: binary(), PrivateDH :: binary()}
ECparams = ecpk_parameters_api()
ECkey = #'ECPrivateKey'{}
RSAparams = {rsa, Size, PubExp}
Size = PubExp = integer() >= 1
RSAkey = #'RSAPrivateKey'{}
```

Generates a new key pair. Note that except for Diffie-Hellman the public key is included in the private key structure. See also crypto:generate_key/2

pem_decode(PemBin :: binary()) -> [pem_entry()]

Decodes PEM binary data and returns entries as ASN.1 DER encoded entities.

Example `{ok, PemBin} = file:read_file("cert.pem"). PemEntries = public_key:pem_decode(PemBin).`

pem_encode(PemEntries :: [pem_entry()]) -> binary()

Creates a PEM binary.

pem_entry_decode(PemEntry) -> term()
pem_entry_decode(PemEntry, Password) -> term()

Types:

```
PemEntry = pem_entry()
Password = iodata() | fun(() -> iodata())
```

Decodes a PEM entry. `pem_decode/1` returns a list of PEM entries. Notice that if the PEM entry is of type 'SubjectPublickeyInfo', it is further decoded to an `rsa_public_key()` or `dsa_public_key()`.

Password can be either an octet string or function which returns same type.

pem_entry_encode(Asn1Type, Entity) -> pem_entry()
pem_entry_encode(Asn1Type, Entity, InfoPwd) -> pem_entry()

Types:

```
Asn1Type = pki_asn1_type()
Entity = term()
InfoPwd = {CipherInfo, Password}
CipherInfo = cipher_info()
Password = iodata()
```

Creates a PEM entry that can be feed to `pem_encode/1`.

If `Asn1Type` is `'SubjectPublicKeyInfo'`, `Entity` must be either an `rsa_public_key()`, `dsa_public_key()` or an `ec_public_key()` and this function creates the appropriate `'SubjectPublicKeyInfo'` entry.

pkix_decode_cert(Cert, Type) -> #'Certificate'{} | otp_cert()

Types:

```
Cert = der_cert()
Type = plain | otp
```

Decodes an ASN.1 DER-encoded PKIX certificate. Option `otp` uses the customized ASN.1 specification OTP-PKIX.asn1 for decoding and also recursively decode most of the standard parts.

`pkix_encode(Asn1Type, Entity, Type) -> Der`

Types:

```
Asn1Type = asn1_type()
Entity = term()
Type = otp | plain
Der = der_encoded()
```

DER encodes a PKIX x509 certificate or part of such a certificate. This function must be used for encoding certificates or parts of certificates that are decoded/created in the `otp` format, whereas for the plain format this function directly calls `der_encode/2`.

> **Note:**
>
> Subtle ASN-1 encoding errors in certificates may be worked around when decoding, this may have the affect that the encoding a certificate back to DER may generate different bytes then the supplied original.

`pkix_is_issuer(CertorCRL, IssuerCert) -> boolean()`

Types:

```
CertorCRL = cert() | #'CertificateList'{}
IssuerCert = cert()
```

Checks if `IssuerCert` issued `Cert`.

`pkix_is_fixed_dh_cert(Cert) -> boolean()`

Types:

```
Cert = cert()
```

Checks if a certificate is a fixed Diffie-Hellman certificate.

`pkix_is_self_signed(Cert) -> boolean()`

Types:

```
Cert = cert()
```

Checks if a certificate is self-signed.

`pkix_issuer_id(Cert, IssuedBy) ->`
`{ok, ID :: cert_id()} | {error, Reason}`

Types:

```
Cert = cert()
IssuedBy = self | other
Reason = term()
```

Returns the x509 certificate issuer id, if it can be determined.

pkix_normalize_name(Issuer) -> Normalized

Types:

Issuer = issuer_name() | der_encoded()

Normalized = issuer_name()

Normalizes an issuer name so that it can be easily compared to another issuer name.

```
pkix_path_validation(Cert, CertChain, Options) ->
                        {ok,
                         {PublicKeyInfo, ConstrainedPolicyNodes}} |
                        {error,
                         {bad_cert, Reason :: bad_cert_reason()}}
```

Types:

```
Cert = cert() | atom()
CertChain = [cert() | combined_cert()]
Options =
    [{max_path_length, integer()} |
     {verify_fun, {function(), term()}}]
PublicKeyInfo = public_key_info()
ConstrainedPolicyNodes = [policy_node()]
```

Performs a basic path validation according to **RFC 5280.** However, CRL validation is done separately by pkix_crls_validate/3 and is to be called from the supplied `verify_fun`. The policy tree check was added in OTP-26.2 and if the certificates include policies the constrained policy set with potential qualifiers will be returned, these values are derived from the policy tree created as part of the path validation algorithm. The constrained set can be constrained only by the Certificate Authorities or also by the user when the option `policy_set` is provided to this function. The qualifiers convey information about the valid policy and is intended as information to end users.

Available options:

{verify_fun, {fun(), InitialUserState::term()}

The fun must be defined as:

```
fun(OtpCert :: #'OTPCertificate'{},
    Event :: {bad_cert, Reason :: atom() | {revoked, atom()}} |
             {extension, #'Extension'{}},
    InitialUserState :: term()) ->
 {valid, UserState :: term()} |
 {valid_peer, UserState :: term()} |
 {fail, Reason :: term()} |
 {unknown, UserState :: term()}.
```

If the verify callback fun returns `{fail, Reason}`, the verification process is immediately stopped. If the verify callback fun returns `{valid, UserState}`, the verification process is continued. This can be used to accept specific path validation errors, such as `selfsigned_peer`, as well as verifying application-specific extensions. If called with an extension unknown to the user application, the return value `{unknown, UserState}` is to be used.

Warning:

Note that user defined custom `verify_fun` may alter original path validation error (e.g `selfsigned_peer`). Use with caution.

{max_path_length, integer()}
: The `max_path_length` is the maximum number of non-self-issued intermediate certificates that can follow the peer certificate in a valid certification path. So, if `max_path_length` is 0, the PEER must be signed by the trusted ROOT-CA directly, if it is 1, the path can be PEER, CA, ROOT-CA, if it is 2, the path can be PEER, CA, CA, ROOT-CA, and so on.

{policy_set, [oid()]}
: The set of policies that will be accepted, defaults to the special value `[?anyPolicy]` that will accept all policies.

{explicit_policy, boolean()}
: Explicitly require that each certificate in the path must include at least one of the certificate policies in the `policy_set`.

{inhibit_policy_mapping, boolean()}
: Prevent policies to be mapped to other policies.

{inhibit_any_policy, boolean()}
: Prevent the special policy `?anyPolicy` from being accepted.

Explanations of reasons for a bad certificate:

cert_expired
: Certificate is no longer valid as its expiration date has passed.

invalid_issuer
: Certificate issuer name does not match the name of the issuer certificate in the chain.

invalid_signature
: Certificate was not signed by its issuer certificate in the chain.

distinguished_name_not_permitted
: Subject Name does not adhere to name constraints which is mandatory in RFC 5280

name_not_permitted
: Subject Alternative Name extension does not adhere to name constraints which is optional in RFC 5280

missing_basic_constraint
: Certificate, required to have the basic constraints extension, does not have a basic constraints extension.

invalid_key_usage
: Certificate key is used in an invalid way according to the key-usage extension.

{revoked, crl_reason()}
: Certificate has been revoked.

invalid_validity_dates
: The validity section of the X.509 certificate(s) contains invalid date formats not matching the RFC.

policy_tree_exceeded
: The certificate chain's policy tree exceeded the maximum allowed node count. This indicates a malformed or malicious chain with exponentially expanding policy mappings. This error cannot be overridden by the `verify_fun`.

atom()
: Application-specific error reason that is to be checked by the `verify_fun`.

pkix_crl_issuer(CRL) -> Issuer

Types:

```
CRL = der_encoded() | #'CertificateList'{}
Issuer = issuer_name()
```

Returns the issuer of the CRL.

pkix_crls_validate(OTPcertificate, DPandCRLs, Options) -> CRLstatus

Types:

```
OTPcertificate = #'OTPCertificate'{}
DPandCRLs = [DPandCRL]
DPandCRL = {DP, {DerCRL, CRL}}
DP = #'DistributionPoint'{}
DerCRL = der_encoded()
CRL = #'CertificateList'{}
Options = [{atom(), term()}]
CRLstatus = valid | {bad_cert, BadCertReason}
BadCertReason =
    revocation_status_undetermined |
    {revocation_status_undetermined, Reason :: term()} |
    {revoked, crl_reason()}
```

Performs CRL validation. It is intended to be called from the verify fun of pkix_path_validation/3 .

Available options:

{update_crl, fun()}

The fun has the following type specification:

```
fun(#'DistributionPoint'{}, #'CertificateList'{}) ->
       #'CertificateList'{}
```

The fun uses the information in the distribution point to access the latest possible version of the CRL. If this fun is not specified, Public Key uses the default implementation:

```
fun(_DP, CRL) -> CRL end
```

{issuer_fun, fun()}

The fun has the following type specification:

```
fun(#'DistributionPoint'{}, #'CertificateList'{},
    {rdnSequence,[#'AttributeTypeAndValue'{}]}, term()) ->
 {ok, #'OTPCertificate'{}, [der_encoded]}
```

The fun returns the root certificate and certificate chain that has signed the CRL.

```
fun(DP, CRL, Issuer, UserState) -> {ok, RootCert, CertChain}
```

{undetermined_details, boolean()}

Defaults to false. When revocation status cannot be determined, and this option is set to true, details of why no CRLs where accepted are included in the return value.

pkix_crl_verify(CRL, Cert) -> boolean()

Types:

CRL = der_encoded() | #'CertificateList'{}

Cert = cert()

Verify that `Cert` is the `CRL` signer.

pkix_dist_point(Cert) -> DistPoint

Types:

Cert = cert()

DistPoint = #'DistributionPoint'{}

Creates a distribution point for CRLs issued by the same issuer as `Cert`. Can be used as input to pkix_crls_validate/3

pkix_dist_points(Cert) -> DistPoints

Types:

Cert = cert()

DistPoints = [#'DistributionPoint'{}]

Extracts distribution points from the certificates extensions.

```
pkix_hash_type(HashOid :: oid()) ->
                  DigestType ::
                      md5 | crypto:sha1() | crypto:sha2()
```

Translates OID to Erlang digest type

pkix_match_dist_point(CRL, DistPoint) -> boolean()

Types:

CRL = der_encoded() | #'CertificateList'{}

DistPoint = #'DistributionPoint'{}

Checks whether the given distribution point matches the Issuing Distribution Point of the CRL, as described in RFC 5280. If the CRL doesn't have an Issuing Distribution Point extension, the distribution point always matches.

pkix_sign(Cert, Key) -> Der

Types:

Cert = #'OTPTBSCertificate'{}

Key = private_key()

Der = der_encoded()

Signs an 'OTPTBSCertificate'. Returns the corresponding DER-encoded certificate.

pkix_sign_types(AlgorithmId) -> {DigestType, SignatureType}

Types:

AlgorithmId = oid()

DigestType = crypto:rsa_digest_type() | none

SignatureType = rsa | dsa | ecdsa | eddsa

Translates signature algorithm OID to Erlang digest and signature types.

The `AlgorithmId` is the signature OID from a certificate or a certificate revocation list.

pkix_test_data(ChainConf) -> TestConf

Types:

```
ChainConf =
    #{server_chain := chain_opts(), client_chain := chain_opts()} |
    chain_opts()
TestConf = test_config() | [conf_opt()]
```

Creates certificate configuration(s) consisting of certificate and its private key plus CA certificate bundle, for a client and a server, intended to facilitate automated testing of applications using X509-certificates, often through SSL/TLS. The test data can be used when you have control over both the client and the server in a test scenario.

When this function is called with a map containing client and server chain specifications; it generates both a client and a server certificate chain where the `cacerts` returned for the server contains the root cert the server should trust and the intermediate certificates the server should present to connecting clients. The root cert the server should trust is the one used as root of the client certificate chain. Vice versa applies to the `cacerts` returned for the client. The root cert(s) can either be pre-generated with pkix_test_root_cert/2 , or if options are specified; it is (they are) generated.

When this function is called with a list of certificate options; it generates a configuration with just one node certificate where `cacerts` contains the root cert and the intermediate certs that should be presented to a peer. In this case the same root cert must be used for all peers. This is useful in for example an Erlang distributed cluster where any node, towards another node, acts either as a server or as a client depending on who connects to whom. The generated certificate contains a subject altname, which is not needed in a client certificate, but makes the certificate useful for both roles.

Explanation of the options used to customize certificates in the generated chains:

{digest, digest_type()}

Hash algorithm to be used for signing the certificate together with the key option. Defaults to sha that is sha1.

{key, key_params() | private_key()}

Parameters to be used to call public_key:generate_key/1, to generate a key, or an existing key. Defaults to generating an ECDSA key. Note this could fail if Erlang/OTP is compiled with a very old cryptolib.

{validity, {From::erlang:timestamp(), To::erlang:timestamp()}}

The validity period of the certificate.

{extensions, [#'Extension'{}]}

Extensions to include in the certificate.

Default extensions included in CA certificates if not otherwise specified are:

```
[#'Extension'{extnID = ?'id-ce-keyUsage',
              extnValue = [keyCertSign, cRLSign],
              critical = false},
#'Extension'{extnID = ?'id-ce-basicConstraints',
             extnValue = #'BasicConstraints'{cA = true},
             critical = true}]
```

Default extensions included in the server peer cert if not otherwise specified are:

```
[#'Extension'{extnID = ?'id-ce-keyUsage',
              extnValue = [digitalSignature, keyAgreement],
              critical = false},
#'Extension'{extnID = ?'id-ce-subjectAltName',
             extnValue = [{dNSName, Hostname}],
             critical = false}]
```

Hostname is the result of calling net_adm:localhost() in the Erlang node where this function is called.

Note:

Note that the generated certificates and keys does not provide a formally correct PKIX-trust-chain and they cannot be used to achieve real security. This function is provided for testing purposes only.

pkix_test_root_cert(Name, Options) -> RootCert

Types:

Name = string()
Options = [cert_opt()]
RootCert = test_root_cert()

Generates a root certificate that can be used in multiple calls to pkix_test_data/1 when you want the same root certificate for several generated certificates.

pkix_subject_id(Cert) -> ID

Types:

Cert = cert()
ID = cert_id()

Returns the X509 certificate subject id.

pkix_verify(Cert, Key) -> boolean()

Types:

Cert = der_cert()
Key = public_key()

Verifies PKIX x.509 certificate signature.

pkix_verify_hostname(Cert, ReferenceIDs) -> boolean()
pkix_verify_hostname(Cert, ReferenceIDs, Options) -> boolean()

Types:

Cert = cert()
ReferenceIDs = referenceIDs()
Options = [{match_fun | fail_callback | fqdn_fun, function()}]

This function checks that the *Presented Identifier* (e.g hostname) in a peer certificate is in agreement with at least one of the *Reference Identifier* that the client expects to be connected to. The function is intended to be added as an extra client check of the peer certificate when performing public_key:pkix_path_validation/3

See **RFC 6125** for detailed information about hostname verification. The User's Guide and code examples describes this function more detailed.

The option funs are described here:

`match_fun`

```
fun(ReferenceId::ReferenceId() | FQDN::string(),
    PresentedId::{dNSName,string()} | {uniformResourceIdentifier,string() |
                 {iPAddress,list(byte())} | {OtherId::atom()|oid(),term()}})
```

This function replaces the default host name matching rules. The fun should return a boolean to tell if the Reference ID and Presented ID matches or not. The match fun can also return a third value, value, the atom `default`, if the default matching rules shall apply. This makes it possible to augment the tests with a special case:

```
fun(....) -> true;   % My special case
   (_, _) -> default % all others falls back to the inherit tests
end
```

See pkix_verify_hostname_match_fun/1 for a function that takes a protocol name as argument and returns a `fun/2` suitable for this option and Re-defining the match operation in the User's Guide for an example.

> **Note:**
>
> Reference Id values given as binaries will be converted to strings, and ip references may be given in string format that is "10.0.1.1" or "1234::5678:9012" as well as on the format inet:ip_address()

`fail_callback`

If a matching fails, there could be circumstances when the certificate should be accepted anyway. Think for example of a web browser where you choose to accept an outdated certificate. This option enables implementation of such an exception but for hostnames. This `fun/1` is called when no `ReferenceID` matches. The return value of the fun (a `boolean()`) decides the outcome. If `true` the the certificate is accepted otherwise it is rejected. See "Pinning" a Certificate in the User's Guide.

`fqdn_fun`

This option augments the host name extraction from URIs and other Reference IDs. It could for example be a very special URI that is not standardised. The fun takes a Reference ID as argument and returns one of:

- the hostname
- the atom `default`: the default host name extract function will be used
- the atom `undefined`: a host name could not be extracted. The pkix_verify_hostname/3 will return `false`.

For an example, see Hostname extraction in the User's Guide.

`pkix_verify_hostname_match_fun(Protocol) -> Result`

Types:

`Protocol = https`

`Result = function()`

The return value of calling this function is intended to be used in the `match_fun` option in pkix_verify_hostname/3.

The returned fun augments the verify hostname matching according to the specific rules for the protocol in the argument.

> **Note:**
>
> Currently supported https fun will allow wildcard certificate matching as specified by the HTTP standard. Note that for instance LDAP have a different set of wildcard matching rules. If you do not want to allow wildcard certificates (recommended from a security perspective) or otherwise customize the hostname match the default match function used by ssl application will be sufficient.

```
sign(Msg, DigestType, Key) -> Signature
sign(Msg, DigestType, Key, Options) -> Signature
```

Types:

```
Msg = binary() | {digest, binary()}
DigestType = digest_type()
Key = private_key()
Options = crypto:pk_sign_verify_opts()
Signature = binary()
```

Creates a digital signature.

The `Msg` is either the binary "plain text" data to be signed or it is the hashed value of "plain text", that is, the digest.

```
verify(Msg, DigestType, Signature, Key) -> boolean()
verify(Msg, DigestType, Signature, Key, Options) -> boolean()
```

Types:

```
Msg = binary() | {digest, binary()}
DigestType = digest_type()
Signature = binary()
Key = public_key()
Options = crypto:pk_sign_verify_opts()
```

Verifies a digital signature.

The `Msg` is either the binary "plain text" data or it is the hashed value of "plain text", that is, the digest.

```
short_name_hash(Name) -> string()
```

Types:

```
Name = issuer_name()
```

Generates a short hash of an issuer name. The hash is returned as a string containing eight hexadecimal digits.

The return value of this function is the same as the result of the commands `openssl crl -hash` and `openssl x509 -issuer_hash`, when passed the issuer name of a CRL or a certificate, respectively. This hash is used by the `c_rehash` tool to maintain a directory of symlinks to CRL files, in order to facilitate looking up a CRL by its issuer name.