



public\_key

Copyright © 2008-2019 Ericsson AB, All Rights Reserved  
public\_key 1.7  
November 21, 2019

---

**Copyright © 2008-2019 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 21, 2019**

# 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()}
```

## 1.2 Public-Key Records

---

```
| {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 % [#0therPrimeInfo{}} | asn1_NOVALUE
}.

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

```

### 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

Erlang representation of **Elliptic Curve Digital Signature Algorithm (ECDSA)** keys follows:

## 1.2 Public-Key Records

---

```
#'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()
}.
```



## 1.2 Public-Key Records

---

```
#'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'{} }

## 1.2 Public-Key Records

---

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

## 1.2 Public-Key Records

---

```
#'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{ }

## 1.2 Public-Key Records

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
```





```
3> Key = public_key.pem_entry_decode(RSAEntry, "abcd1234").
    #'RSAPrivateKey'{version = 'two-prime',
      modulus = 1112355156729921663373...2737107,
      publicExponent = 65537,
      privateExponent = 58064406231183...2239766033,
      prime1 = 11034766614656598484098...7326883017,
      prime2 = 10080459293561036618240...77738643771,
      exponent1 = 77928819327425934607...22152984217,
      exponent2 = 36287623121853605733...20588523793,
      coefficient = 924840412626098444...41820968343,
      otherPrimeInfos = asnl_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:



```

        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.

```

## 1.3 Getting Started

---

```
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},
```















# 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 are currently not supported.
- 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)*































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.