



public_key

Copyright © 2008-2017 Ericsson AB, All Rights Reserved
public_key 0.5
21 2017

Copyright © 2008-2017 Ericsson AB, All Rights Reserved

The contents of this file are subject to the Erlang Public License, Version 1.1, (the "License"); you may not use this file except in compliance with the License. You should have received a copy of the Erlang Public License along with this software. If not, it can be retrieved online at <http://www.erlang.org/>. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. The Initial Developer of the Original Code is Ericsson AB. Ericsson AB, All Rights Reserved.

21 2017

1 User's Guide

This application provides an API to public key infrastructure from RFC 3280 (X.509 certificates) and some public key formats defined by the PKCS-standard.

1.1 Introduction

1.1.1 Purpose

This application provides an API to public key infrastructure from RFC 3280 (X.509 certificates) and public key formats defined by the PKCS-standard.

1.1.2 Prerequisites

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

1.2 Public key records

This chapter briefly describes Erlang records derived from asn1 specifications used to handle public and private keys. The intent is to describe the data types and not to specify the meaning of each component for this we refer you to the relevant standards and RFCs.

Use the following include directive to get access to the records and constant macros used in the following sections.

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

1.2.1 RSA as defined by the PKCS-1 standard and RFC 3447.

```
#'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()
}
```

1.3 Certificate records

```
        coefficient      % integer()  
    }.
```

1.2.2 DSA as defined by Digital Signature Standard (NIST FIPS PUB 186-2)

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

1.3 Certificate records

This chapter briefly describes erlang records derived from asn1 specifications used to handle X509 certificates. The intent is to describe the data types and not to specify the meaning of each component for this we refer you to RFC 3280.

Use the following include directive to get access to the records and constant macros (OIDs) described in the following sections.

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

The used specification is available in OTP-PKIX.asn1, which is an amelioration of the PKIX1Explicit88.asn1, PKIX1Implicit88.asn1 and PKIX1Algorithms88.asn1 modules. You find all these modules in the asn1 subdirectory of the application public_key.

1.3.1 Common Data Types

Common non standard erlang data types used to described the record fields in the below sections are defined in *public key reference manual* or follows here.

```
time() = uct_time() | general_time()
```

```
uct_time() = {utcTime, "YYMMDDHHMMSSZ"}
```

```
general_time() = {generalTime, "YYYYMMDDHHMMSSZ"}
```

```
general_name() = {rfc822Name, string()} | {dNSName, string()} | {x400Address,  
string()} | {directoryName, {rdnSequence, [#AttributeTypeAndValue'{}]} }  
| | {eidPartyName, special_string()} | {eidPartyName, special_string(),  
special_string()} | {uniformResourceIdentifier, string()} | {ipAddress,  
string()} | {registeredId, oid()} | {otherName, term() }
```

```
special_string() = {teletexString, string()} | {printableString, string()} |  
{universalString, string()} | {utf8String, string()} | {bmpString, string() }
```

```
dist_reason() = unused | keyCompromise | cACompromise | affiliationChanged
| superseded | cessationOfOperation | certificateHold | privilegeWithdrawn |
aACompromise
```

1.3.2 PKIX Certificates

```
#'Certificate'{
  tbsCertificate,      % #'TBSCertificate'{}
  signatureAlgorithm,  % #'AlgorithmIdentifier'{}
  signature             % {0, binary()} - asn1 compact 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 % asn1_der_encoded()
}.

#'SignatureAlgorithm'{
  algorithm, % id_signature_algorithm()
  parameters % public_key_params()
}.
```

id_signature_algorithm() = ?oid_name_as_erlang_atom for available oid names see table below.
Ex: ?'id-dsa-with-sha1'

OID name
id-dsa-with-sha1
md2WithRSAEncryption
md5WithRSAEncryption
sha1WithRSAEncryption
ecdsa-with-SHA1

Table 3.1: Signature algorithm oids

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

1.3 Certificate records

```
    }.
```

id_attributes()

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 3.2: Attribute oids

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

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

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

id_public_key_algorithm()

OID name
rsaEncryption
id-dsa
dhpublicnumber
ecdsa-with-SHA1
id-keyExchangeAlgorithm

Table 3.3: Public key algorithm oids

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

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

1.3.3 Standard certificate extensions

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' { }

1.3 Certificate records

id-ce-extKeyUsage	[id_key_purpose()]
id-ce-cRLDistributionPoints	#'DistributionPoint'{}
id-ce-inhibitAnyPolicy	integer()
id-ce-freshestCRL	[#'DistributionPoint'{}]

Table 3.4: Standard Certificate Extensions

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

OID name
id-kp-serverAuth
id-kp-clientAuth
id-kp-codeSigning
id-kp-emailProtection
id-kp-timeStamping
id-kp-OCSPSigning

Table 3.5: 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 % [asn1_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, % general_name() | [#AttributeTypeAndValue{}]
    reasons, % [dist_reason()]
    cRLIssuer % general_name()
}).

```

1.3.4 Private Internet Extensions

OID name	Value type
id-pe-authorityInfoAccess	[#'AccessDescription'{}]
id-pe-subjectInfoAccess	[#'AccessDescription'{}]

Table 3.6: Private Internet Extensions

```

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

```

1.3 Certificate records

```
}).
```

1.3.5 CRL and CRL Extensions Profile

```
#'CertificateList'{
  tbsCertList,      % #'TBSCertList{}
  signatureAlgorithm, % #'AlgorithmIdentifier'{}
  signature          % {0, binary()} - asnl compact 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

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 3.7: CRL Extensions

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

CRL Entry Extensions

OID name	Value type
id-ce-cRLReason	crl_reason()
id-ce-holdInstructionCode	oid()
id-ce-invalidityDate	general_time()
id-ce-certificateIssuer	general_name()

Table 3.8: CRL Entry Extensions

```
crl_reason() = unspecified | keyCompromise | cACompromise | affiliationChanged  
| superseded | cessationOfOperation | certificateHold | removeFromCRL |  
privilegeWithdrawn | aACompromise
```

2 Reference Manual

Provides functions to handle public key infrastructure from RFC 3280 (X.509 certificates) and some parts of the PKCS-standard.

public_key

Erlang module

This module provides functions to handle public key infrastructure from RFC 3280 - X.509 certificates (will later be upgraded to RFC 5280) and some parts of the PKCS-standard. Currently this application is mainly used by the new ssl implementation. The API is yet under construction and only a few of the functions are currently documented and thereby supported.

COMMON DATA TYPES

Note:

All records used in this manual are generated from asn1 specifications and are documented in the User's Guide. See *Public key records* and *X.509 Certificate 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

```
boolean() = true | false
string = [bytes()]
asn1_der_encoded() = binary() | [bytes()]
der_bin() = binary()
oid() - a tuple of integers as generated by the asn1 compiler.
public_key() = rsa_public_key() | dsa_public_key()
rsa_public_key() = #'RSAPublicKey'{}
rsa_private_key() = #'RSAPrivateKey'{}
dsa_public_key() = integer()
public_key_params() = dsa_key_params()
dsa_key_params() = #'Dss-Parms'{}
private_key() = rsa_private_key() | dsa_private_key()
rsa_private_key() = #'RSAPrivateKey'{}
dsa_private_key() = #'DSAPrivateKey'{}
x509_certificate() = "#Certificate{}"
x509_tbs_certificate() = #'TBSCertificate'{}"
```

Exports

decode_private_key(KeyInfo) ->
decode_private_key(KeyInfo, Password) -> {ok, PrivateKey} | {error, Reason}

Types:

KeyInfo = {KeyType, der_bin(), CipherInfo}

As returned from pem_to_der/1 for private keys

KeyType = rsa_private_key | dsa_private_key

CipherInfo = opaque() | no_encryption

CipherInfo may contain encryption parameters if the private key is password protected, these are opaque to the user just pass the value returned by pem_to_der/1 to this function.

Password = string()

Must be specified if CipherInfo /= no_encryption

PrivateKey = private_key()

Reason = term()

Decodes an asn1 der encoded private key.

pem_to_der(File) -> {ok, [Entry]}

Types:

File = path()

Password = string()

Entry = {entry_type(), der_bin(), CipherInfo}

CipherInfo = opaque() | no_encryption

CipherInfo may contain encryption parameters if the private key is password protected, these will be handled by the function decode_private_key/2.

entry_type() = cert | cert_req | rsa_private_key | dsa_private_key | dh_params

Reads a PEM file and translates it into its asn1 der encoded parts.

pkix_decode_cert(Cert, Type) -> {ok, DecodedCert} | {error, Reason}

Types:

Cert = asn1_der_encoded()

Type = plain | otp

DecodedCert = x509_certificate()

When type is specified as otp the asn1 spec OTP-PKIX.asn1 is used to decode known extensions and enhance the signature field in #Certificate'{} and #TBSCertificate'{}'. This is currently used by the new ssl implementation but not documented and supported for the public_key application.

Reason = term()

Decodes an asn1 encoded pkix certificate.