public_key

public_key 0.20
September 19, 2013

September 19, 2013

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

public_key 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 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 will try to use the NIFs which are in the ASN1 compilers runtime modules if they can be found. So for the best performance you want to have the ASN1 application in the path of your system.

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()
 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` and `CertificationRequest`. The intent is to describe the data types and not to specify the meaning of each component for this we refer you to **RFC 5280** and **PKCS-10**.

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 ASN1 specifications are available `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  % der_encoded()
  }.
```

```
#'OTPCertificate'{
  tbsCertificate,        % #'OTPTBSCertificate'{}
  signatureAlgorithm,    % #'SignatureAlgorithm'
  signature              % {0, binary()} - ASN1 compact 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_name_as_erlang_atom` for available oid names see table below. Ex: ?'id-dsa-with-sha1'

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

Table 3.1: Signature algorithm oids

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

`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
id-keyExchangeAlgorithm
id-ecPublicKey

Table 3.3: 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*.

1.3.3 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'{}
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 % [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()]
  }).
```

1.3.4 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()
  }).
```

1.3.5 CRL and CRL Extensions Profile

```
#'CertificateList'{
          tbsCertList,        % #'TBSCertList{}
          signatureAlgorithm, % #'AlgorithmIdentifier'{}
          signature           % {0, binary()} - ASN1 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,         % {fullName, [general_name()]} | {nameRelativeToCRLIssuer,
   [#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
```

PKCS#10 Certification Request

```
#'CertificationRequest'{
          certificationRequestInfo #'CertificationRequestInfo'{},
   signatureAlgorithm    #'CertificationRequest_signatureAlgorithm'{}}.
   signature                {0, binary()} - ASN1 compact bitstring
   }

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

#'CertificationRequestInfo_subjectPKInfo'{
          algorithm  #'CertificationRequestInfo_subjectPKInfo_algorithm'{}
   subjectPublicKey   {0, binary()} - ASN1 compact bitstring
   }

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

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

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

1.4 Getting Started

1.4.1 General information

This chapter is dedicated to showing some examples of how to use the public_key API. Keys and certificates used in the following sections are generated only for the purpose of testing the public key application.

Note that some shell printouts, in the following examples, have been abbreviated for increased readability.

1.4.2 PEM files

Public key data (keys, certificates etc) may be stored in PEM format. PEM files comes from the Private Enhanced Mail Internet standard and has a structure that looks like this:

```
<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 currently ignored except for `Proc-Type` and `DEK-Info` that are used when the DER data is encrypted.

DSA private key

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"...>>}
```

This PEM file only has 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 = 87256807980030509074...403143,
                 x = 510968529856012146351317363807366575075645839654}
```

RSA private key encrypted with a password.

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

This PEM file only has 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 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 = 11034766614656598484098...7326883017,
                 prime2 = 10080459293561036618240...77738643771,
                 exponent1 = 77928819327425934607...22152984217,
                 exponent2 = 36287623121853605733...20588523793,
                 coefficient = 924840412626098444...41820968343,
                 otherPrimeInfos = asn1_NOVALUE}
```

X509 Certificates

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

This 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 may of course 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,"080109082929Z"},
                    notAfter = {utcTime,"080208082929Z"}},
            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 =
        {0,
         <<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 that parts ASN.1 type. Although application specific certificate extension requires application specific ASN.1 decode/encode-functions. Example, the first value of the rdnSequence above 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"}
```

... but certificates can also be decode using the pkix_decode_cert/2 that 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 =
        {0,
         <<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 and want to create a PEM file you can do that by calling the functions public_key:pem_entry_encode/2 and pem_encode/1 and then saving the result to a file. For example assume 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 will be 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> PemBin = 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
```

1.4.3 RSA public key cryptography

Suppose you have PrivateKey = #'RSAPrivateKey{}' and the plaintext Msg = binary() and the corresponding public key PublicKey = #'RSAPublicKey'{} then you can do the following. Note that you normally will only do one of the encrypt or decrypt operations and the peer will do the other.

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),
```

1.4.4 Digital signatures

Suppose you have PrivateKey = #'RSAPrivateKey{}'or #'DSAPrivateKey'{} and the plaintext Msg = binary() and the corresponding public key PublicKey = #'RSAPublicKey'{} or {integer(), #'DssParams'{}} then you can do the following. Note that you normally will only do one of the sign or verify operations and the peer will do the other.

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

It might be appropriate to calculate the message digest before calling sign or verify and then you can use the none as second argument.

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

1.4.5 SSH files

SSH typically uses PEM files for private keys but has its own file format for storing public keys. The erlang public_key application can be used to parse the content of SSH public key files.

RFC 4716 SSH public key files

RFC 4716 SSH files looks confusingly like PEM files, but there are some differences.

```
1> {ok, SshBin} = file:read_file("ssh2_rsa_pub").
{ok, <<"---- BEGIN SSH2 PUBLIC KEY ----\nAAAA"...>>}
```

This is equivalent to calling public_key:ssh_decode(SshBin, rfc4716_public_key).

```
2> public_key:ssh_decode(SshBin, public_key).
[{#'RSAPublicKey'{modulus = 794430685...91663,
                  publicExponent = 35}, []}]
```

Openssh public key format

```
1> {ok, SshBin} = file:read_file("openssh_dsa_pub").
{ok,<<"ssh-dss AAAAB3Nza"...>>}
```

This is equivalent to calling public_key:ssh_decode(SshBin, openssh_public_key).

```
2>  public_key:ssh_decode(SshBin, public_key).
[{{15642692...694280725,
   #'Dss-Parms'{p = 17291273936...696123221,
                q = 1255626590179665817295475654204371833735706001853,
                g = 10454211196...480338645}},
  [{comment,"dhopson@VMUbuntu-DSH"}]}]
```

Known hosts - openssh format

```
1> {ok, SshBin} = file:read_file("known_hosts").
{ok,<<"hostname.domain.com,192.168.0.1 ssh-rsa AAAAB...>>}
```

Returns a list of public keys and their related attributes each pair of key and attributes corresponds to one entry in the known hosts file.

```
2>  public_key:ssh_decode(SshBin, known_hosts).
[{#'RSAPublicKey'{modulus = 1498979460408...72721699,
                  publicExponent = 35},
  [{hostnames,["hostname.domain.com","192.168.0.1"]}]},
 {#'RSAPublicKey'{modulus = 14989794604088...2721699,
                  publicExponent = 35},
  [{comment,"foo@bar.com"},
   {hostnames,["|1|BWO5qDxk/cFH0wa05JLdHn+j6xQ=|rXQvIxh5cDD3C43k5DPDamawVNA="]}]}]
```

Authorized keys - openssh format

```
1> {ok, SshBin} = file:read_file("auth_keys").
{ok, <<"command=\"dump /home\",no-pty,no-port-forwarding ssh-rsa AAA...>>}
```

Returns a list of public keys and their related attributes each pair of key and attributes corresponds to one entry in the authorized key file.

```
2> public_key:ssh_decode(SshBin, auth_keys).
[{#'RSAPublicKey'{modulus = 794430685...691663,
                  publicExponent = 35},
  [{comment,"dhopson@VMUbuntu-DSH"},
   {options,["command=\"dump/home\"","no-pty",
             "no-port-forwarding"]}]},
 {{1564269258491...607694280725,
   #'Dss-Parms'{p = 17291273936185...763696123221,
                q = 1255626590179665817295475654204371833735706001853,
                g = 10454211195705...60511039590076780999046480338645}},
  [{comment,"dhopson@VMUbuntu-DSH"}]}]
```

Creating an SSH file from public key data

If you got a public key `PubKey` and a related list of attributes `Attributes` as returned by ssh_decode/2 you can create a new ssh file for example

```
N> SshBin = public_key:ssh_encode([{PubKey, Attributes}], openssh_public_key),
<<"ssh-rsa "...>>
N+1> file:write_file("id_rsa.pub", SshBin).
ok
```

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. It can encode/decode different file formats (PEM, openssh), sign and verify digital signatures and validate certificate paths and certificate revocation lists.

public_key

- public_key requires the crypto and asn1 applications, the latter since R16 (hopefully the runtime dependency on asn1 will be removed again in the future).
- Supports **RFC 5280** - Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile
- 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 **PKCS-8** - Private-Key Information Syntax Standard
- Supports **PKCS-10** - Certification Request Syntax Standard

COMMON DATA TYPES

Note:

All records used in this manual are generated from ASN.1 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

```
oid() - a tuple of integers as generated by the ASN1 compiler.
```

```
boolean() = true | false
```

```
string() = [bytes()]
```

```
der_encoded() = binary()
```

```
pki_asn1_type() = 'Certificate' | 'RSAPrivateKey'| 'RSAPublicKey' |
    'DSAPrivateKey' | 'DSAPublicKey' | 'DHParameter' | 'SubjectPublicKeyInfo' |
    'PrivateKeyInfo' | 'CertificationRequest' | 'ECPrivateKey'|
```

```
    'EcpkParameters'
```

```
pem_entry () = {pki_asn1_type(), binary(), %% DER or encrypted DER
          not_encrypted | cipher_info()}
```

```
cipher_info()  =  {"RC2-CBC | "DES-CBC" | "DES-EDE3-CBC", crypto:rand_bytes(8)} |
    'PBES2-params'}
```

```
rsa_public_key()  = #'RSAPublicKey'{}
```

```
rsa_private_key() = #'RSAPrivateKey'{}
```

```
dsa_public_key() = {integer(),  #'Dss-Parms'{}}
```

```
dsa_private_key() = #'DSAPrivateKey'{}
```

```
ec_public_key()  = {#'ECPoint'{}, #'EcpkParameters'{} | {namedCurve, oid()}}
```

```
ec_private_key()  = #'ECPrivateKey'{}
```

```
 public_crypt_options() = [{rsa_pad, rsa_padding()}].
```

```
 rsa_padding() =  'rsa_pkcs1_padding' | 'rsa_pkcs1_oaep_padding'
    | 'rsa_no_padding'
```

```
 rsa_digest_type()  = 'md5' | 'sha' | 'sha224' | 'sha256' | 'sha384' | 'sha512'
```

```
 dss_digest_type()  = 'sha'
```

```
 ecdsa_digest_type()  = 'sha'| 'sha224' | 'sha256' | 'sha384' | 'sha512'
```

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

```
 ssh_file()  = openssh_public_key | rfc4716_public_key |
    known_hosts | auth_keys
```

Exports

`compute_key(OthersKey, MyKey)->`
`compute_key(OthersKey, MyKey, Params)->`

Types:

`OthersKey = #'ECPoint'{} | binary(), MyKey = #'ECPrivateKey'{} | binary()`
`Params = #'DHParameter'{}`

Compute shared secret

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

Types:

`CipherText = binary()`
`Key = rsa_private_key()`
`Options = public_crypt_options()`

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

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

Types:

`CipherText = binary()`
`Key = rsa_public_key()`
`Options = public_crypt_options()`

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

`der_decode(Asn1type, Der) -> term()`

Types:

`Asn1Type = atom()`
ASN.1 type present in the public_key applications asn1 specifications.
`Der = der_encoded()`

Decodes a public key ASN.1 DER encoded entity.

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

Types:

`Asn1Type = atom()`
Asn1 type present in the public_key applications ASN.1 specifications.
`Entity = term()`
The erlang representation of `Asn1Type`

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

`generate_key(Params) -> {Public::binary(), Private::binary()} | #'ECPrivateKey'{}`

Types:

Params = #'DHParameter'{} | {namedCurve, oid()} | #'ECParameters'{}

Generates a new keypair

`pem_decode(PemBin) -> [pem_entry()]`

Types:

PemBin = binary()

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

Decode PEM binary data and return entries as ASN.1 DER encoded entities.

`pem_encode(PemEntries) -> binary()`

Types:

PemEntries = [pem_entry()]

Creates a PEM binary

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

Types:

PemEntry = pem_entry()
Password = string()

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

`pem_entry_encode(Asn1Type, Entity) -> pem_entry()`
`pem_entry_encode(Asn1Type, Entity, {CipherInfo, Password}) -> pem_entry()`

Types:

Asn1Type = pki_asn1_type()
Entity = term()

The Erlang representation of `Asn1Type`. If `Asn1Type` is 'SubjectPublicKeyInfo' then `Entity` must be either an rsa_public_key() or a dsa_public_key() and this function will create the appropriate 'SubjectPublicKeyInfo' entry.

CipherInfo = cipher_info()
Password = string()

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

`encrypt_private(PlainText, Key) -> binary()`

Types:

PlainText = binary()
Key = rsa_private_key()

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

`encrypt_public(PlainText, Key) -> binary()`

Types:

PlainText = binary()

Key = rsa_public_key()

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

pkix_decode_cert(Cert, otp|plain) -> #'Certificate'{} | #'OTPCertificate'{}

Types:

Cert = der_encoded()

Decodes an ASN.1 DER encoded PKIX certificate. The otp option will use the customized ASN.1 specification OTP-PKIX.asn1 for decoding and also recursively decode most of the standard parts.

pkix_encode(Asn1Type, Entity, otp | plain) -> der_encoded()

Types:

Asn1Type = atom()

The ASN.1 type can be 'Certificate', 'OTPCertificate' or a subtype of either .

Entity = #'Certificate'{} | #'OTPCertificate'{} | a valid subtype

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 will directly call der_encode/2.

pkix_is_issuer(Cert, IssuerCert) -> boolean()

Types:

Cert = der_encode() | #'OTPCertificate'{}

IssuerCert = der_encode() | #'OTPCertificate'{}

Checks if `IssuerCert` issued `Cert`

pkix_is_fixed_dh_cert(Cert) -> boolean()

Types:

Cert = der_encode() | #'OTPCertificate'{}

Checks if a Certificate is a fixed Diffie-Hellman Cert.

pkix_is_self_signed(Cert) -> boolean()

Types:

Cert = der_encode() | #'OTPCertificate'{}

Checks if a Certificate is self signed.

pkix_issuer_id(Cert, IssuedBy) -> {ok, IssuerID} | {error, Reason}

Types:

Cert = der_encode() | #'OTPCertificate'{}

IssuedBy = self | other

IssuerID = {integer(), {rdnSequence, [#'AttributeTypeAndValue'{}]}}

The issuer id consists of the serial number and the issuers name.

Reason = term()

Returns the issuer id.

pkix_normalize_name(Issuer) -> Normalized

Types:

Issuer = {rdnSequence,[#'AttributeTypeAndValue'{}]}

Normalized = {rdnSequence, [#'AttributeTypeAndValue'{}]}

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

pkix_path_validation(TrustedCert, CertChain, Options) -> {ok, {PublicKeyInfo, PolicyTree}} | {error, {bad_cert, Reason}}

Types:

TrustedCert = #'OTPCertificate'{} | der_encode() | unknown_ca | selfsigned_peer

Normally a trusted certificate but it can also be one of the path validation errors `unknown_ca` or `selfsigned_peer` that can be discovered while constructing the input to this function and that should be run through the `verify_fun`.

CertChain = [der_encode()]

A list of DER encoded certificates in trust order ending with the peer certificate.

Options = proplists:proplists()

PublicKeyInfo = {?'rsaEncryption' | ?'id-dsa', rsa_public_key() | integer(), 'NULL' | 'Dss-Parms'{}}

PolicyTree = term()

At the moment this will always be an empty list as Policies are not currently supported

Reason = cert_expired | invalid_issuer | invalid_signature | unknown_ca | selfsigned_peer | name_not_permitted | missing_basic_constraint | invalid_key_usage | crl_reason()

Performs a basic path validation according to **RFC 5280.** However CRL validation is done separately by *pkix_crls_validate/3* and should be called from the supplied `verify_fun`

Available options are:

{verify_fun, fun()}

The fun should be defined as:

```
fun(OtpCert :: #'OTPCertificate'{}, Event :: {bad_cert, Reason :: 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} should be used.

{max_path_length, integer()}

The `max_path_length` is the maximum number of non-self-issued intermediate certificates that may 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 1 the path can be PEER, CA, ROOT-CA, if it is 2 PEER, CA, CA, ROOT-CA and so on.

pkix_crls_validate(OTPCertificate, DPAndCRLs, Options) -> CRLStatus()

Types:

OTPCertificate = #'OTPCertificate'{}

DPAndCRLs = [{DP::#'DistributionPoint'{} ,CRL::#'CertificateList'{}}]

Options = proplists:proplists()

CRLStatus() = valid | {bad_cert, revocation_status_undetermined} | {bad_cert, {revoked, crl_reason()}}

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

Available options are:

{update_crl, fun()}

The fun has the following type spec:

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

The fun should use the information in the distribution point to acesses the lates possible version of the CRL. If this fun is not specified public_key will use the default implementation:

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

pkix_sign(#'OTPTBSCertificate'{}, Key) -> der_encode()

Types:

Key = rsa_public_key() | dsa_public_key()

Signs a 'OTPTBSCertificate'. Returns the corresponding der encoded certificate.

pkix_sign_types(AlgorithmId) -> {DigestType, SignatureType}

Types:

AlgorithmId = oid()

Signature oid from a certificate or a certificate revocation list

DigestType = rsa_digest_type() | dss_digest_type()

SignatureType = rsa | dsa

Translates signature algorithm oid to erlang digest and signature types.

pkix_verify(Cert, Key) -> boolean()

Types:

Cert = der_encode()

Key = rsa_public_key() | dsa_public_key()

Verify PKIX x.509 certificate signature.

sign(Msg, DigestType, Key) -> binary()

Types:

Msg = binary() | {digest,binary()}

The msg is either the binary "plain text" data to be signed or it is the hashed value of "plain text" i.e. the digest.

DigestType = rsa_digest_type() | dss_digest_type() | ecdsa_digest_type()

Key = rsa_private_key() | dsa_private_key() | ec_private_key()

Creates a digital signature.

`ssh_decode(SshBin, Type) -> [{public_key(), Attributes::list()}]`

Types:

SshBin = binary()

Example {ok, SshBin} = file:read_file("known_hosts").

Type = public_key | ssh_file()

If `Type` is `public_key` the binary may be either a rfc4716 public key or a openssh public key.

Decodes a ssh file-binary. In the case of know_hosts or auth_keys the binary may include one or more lines of the file. Returns a list of public keys and their attributes, possible attribute values depends on the file type represented by the binary.

rfc4716 attributes - see RFC 4716
 {headers, [{string(), utf8_string()}]}
auth_key attributes - see man sshd
 {comment, string()}
 {options, [string()]}
 {bits, integer()} - In ssh version 1 files
known_host attributes - see man sshd
 {hostnames, [string()]}
 {comment, string()}
 {bits, integer()} - In ssh version 1 files

`ssh_encode([{Key, Attributes}], Type) -> binary()`

Types:

Key = public_key()

Attributes = list()

Type = ssh_file()

Encodes a list of ssh file entries (public keys and attributes) to a binary. Possible attributes depends on the file type, see *ssh_decode/2*

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

Types:

Msg = binary() | {digest,binary()}

The msg is either the binary "plain text" data or it is the hashed value of "plain text" i.e. the digest.

DigestType = rsa_digest_type() | dss_digest_type() | ecdsa_digest_type()

Signature = binary()

Key = rsa_public_key() | dsa_public_key() | ec_public_key()

Verifies a digital signature