public_key

public_key 0.10
December 7 2010

December 7 2010

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()
```

```
         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 5280.

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.

`oid() - a tuple of integers as generated by the asn1 compiler.`

`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()
  }.
```

```
#'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, % #'SubjectPublicKeyInfo'{}
   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
md2WithRSAEncryption
md5WithRSAEncryption
sha1WithRSAEncryption
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
ecdsa-with-SHA1
id-keyExchangeAlgorithm

Table 3.3:   Public key algorithm oids

```
#'Extension'{
   extnID,    % id_extensions() | oid()
   critical,  % boolean()
   extnValue  % asn1_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 % [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()
  }).
```

1.3.5 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,         % 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 5280 - X.509 certificates and some parts of the PKCS-standard.

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()]`

`der_encoded() = binary()`

`decrypt_der() = binary()`

`pki_asn1_type() = 'Certificate' | 'RSAPrivateKey'| 'DSAPrivateKey' | 'DHParameter'`

`pem_entry () = {pki_asn1_type(), der_encoded() | decrypt_der(), not_encrypted | {"DES-CBC" | "DES-EDE3-CBC", crypto:rand_bytes(8)}}.`

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

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

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

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

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

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

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

`rsa_digest_type() = 'md5' | 'sha'`

`dss_digest_type() = 'none' | 'sha'`

Exports

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

Types:

CipherText = binary()

Key = rsa_private_key()

Options = public_crypt_options()

Public key decryption using the private key.

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

Types:

CipherText = binary()

Key = rsa_public_key()

Options = public_crypt_options()

Public key decryption using the public key.

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

Types:

Asn1Type = atom() -

Asn1 type present in the public_key applications asn1 specifications.

Der = der_encoded()

Decodes a public key asn1 der encoded entity.

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

Types:

Asn1Type = atom()

Asn1 type present in the public_key applications asn1 specifications.

Entity = term() - The erlang representation of `Asn1Type`

Encodes a public key entity with asn1 DER encoding.

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

Types:

PemBin = binary()

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

Decode PEM binary data and return entries as asn1 der encoded entities.

`pem_encode(PemEntries) -> binary()`

Types:

PemEntries = [pem_entry()]

Creates a PEM binary

`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.

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

Types:

Asn1Type = atom()

Entity = term()

CipherInfo = {"DES-CBC" | "DES-EDE3-CBC", crypto:rand_bytes(8)}

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.

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

Types:

PlainText = binary()

Key = rsa_public_key()

Public key encryption using the public key.

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

Types:

Cert = der_encoded()

Decodes an asn1 der encoded pkix certificate. The otp option will use the customized asn1 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 asn1 type can be 'Certificate', 'OTPCertificate' or a subtype of either .

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 on 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_sign(#'OTPTBSCertificate'{}, Key) -> der_encode()`

Types:

Key = rsa_public_key() | dsa_public_key()

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

`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()

The msg is either the binary "plain text" data to be signed or in the case that digest type is `none` it is the hashed value of "plain text" i.e. the digest.

DigestType = rsa_digest_type() | dsa_digest_type()

Key = rsa_public_key() | dsa_public_key()

Creates a digital signature.

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

Types:

Msg = binary()

The msg is either the binary "plain text" data or in the case that digest type is `none` it is the hashed value of "plain text" i.e. the digest.

DigestType = rsa_digest_type() | dsa_digest_type()

Signature = binary()

Key = rsa_public_key() | dsa_public_key()

Verifies a digital signature