Package ‘PKI’

November 28, 2022

Version 0.1-12
Title Public Key Infrastructure for R Based on the X.509 Standard
Author Simon Urbanek <Simon.Urbanek@r-project.org>
Maintainer Simon Urbanek <Simon.Urbanek@r-project.org>
Depends R (>= 2.9.0), base64enc
Enhances gmp
Description Public Key Infrastructure functions such as verifying certificates, RSA encryption and signing which can be used to build PKI infrastructure and perform cryptographic tasks.
License GPL-2 | GPL-3 | file LICENSE
URL http://www.rforge.net/PKI
SystemRequirements OpenSSL library and headers (openssl-dev or similar)
NeedsCompilation yes
Repository CRAN
Date/Publication 2022-11-28 07:32:03 UTC

R topics documented:

ASN1 ................................................................. 2
BIGNUMint .......................................................... 3
oid ................................................................. 4
PKI.crypt ............................................................ 6
PKI.digest .......................................................... 7
PKI.info ............................................................ 8
PKI.random .......................................................... 9
PKI.sign ............................................................ 9
PKI.sign.tar ....................................................... 11
raw2hex ............................................................. 12
RSA ................................................................. 13
X509 ................................................................. 15

Index 18
ASN1

Functions for handling ASN.1 format (typically DER)

Description

ASN1.decode decodes ASN.1 binary format into raw format chunks tagged with class types.
ASN1.encode converts structured objects into ASN.1 binary format.
ASN1.item creates an item - basic object in structures that can be encoded using ASN1.encode.
ASN1.type extracts the class type from an ASN.1 item

Usage

ASN1.decode(what)
ASN1.encode(what)
ASN1.item(what, type)
ASN1.type(what)

Arguments

what object to decode/encode/query

type class type of the item (integer value)

Details

This is a suite of low-level tools to deal with ASN.1 (Abstract Syntax Notation One) binary formats DER, BER and CER. The tools were written specifically to handle the various DER-encoded key structures so it provides only a subset of the ASN.1 specification. They are used internally by the PKI package.

ASN1.decode decodes the binary representation (as raw vector) into individual items. Sequences are converted into lists, all other objects are retained in their binary form and tagged with the integer class type - which can be obtained using ASN1.type function.

ASN1.encode expects item (or a list of items) either created using ASN1.decode or ASN1.item and converts them into DER binary format.

The result of ASN1.encode(ASN1.decode(x)) will be x if x was in DER format.

Value

ASN1.decode returns either one item or a list.
ASN1.encode returns a raw vector in DER format.
ASN1.type returns an integer class type
ASN1.item returns an ASN.1 item object
Note

ASN1.encode uses a fixed buffer for encoding which currently limits the total size of the resulting structure to 1MB.

Only definite length forms are supported. The validity of individual items is not checked.

Author(s)

Simon Urbanek

Examples

# generate a small key
key <- PKI.genRSAkey(bits = 512L)

# extract private and public parts in DER format
prv <- PKI.save.key(key, format="DER")
pub <- PKI.save.key(key, private=FALSE, format="DER")

# parse the public key
x <- ASN1.decode(pub)
x

# the second element is the actual key
# as a bit string that's itself in DER
# two integers - modulus and exponent
# Note that this is in fact the pure PKCS#1 key format
ASN1.decode(x[[2]])

# encoding it back should yield the same representation since it is DER
stopifnot(identical(ASN1.encode(x), as.raw(pub)))

BIGNUMint

Functions for BIGNUM representation of arbitrarily precise integers

Description

as.BIGNUMint encodes integer in BIGNUM format as raw vector as used by ASN.1 format.

Usage

as.BIGNUMint(what, scalar = TRUE)

Arguments

what representation of an integer or a vector thereof. Currently supported formats include "bigz" objects from the "gmp" package, integers and reals.

scalar if TRUE then the input is expected to be scalar and only the first element will be used (zero-length vectors raise an error). Otherwise the result will be a list of all converted elements.
Details

The BIGNUM representation as used in ASN.1 is a big-endian encoding of variable length stored in a raw vector. Negative numbers are stored in two-complement's encoding, but are currently unsupported by as.BIGNUMint.

Value

Raw vector in BIGNUM integer representation.

Note

Unless the input is of class "bigz" then 32-bit platforms only support integers up to 32-bit, 64-bit platforms up to 53-bit (when real vectors are used).

Author(s)

Simon Urbanek

Examples

as.BIGNUMint(65537)

oid OBJECT IDENTIFIER Functions

Description

Object Identifiers (OIDs) are entities defined by international standards (ITU-T, ISO, IEC) used to identify objects. In the PKI context they are used for example to identify encryption algorithms. Each root (first integer - see below) denotes the standards body governing the allocations.

OIDs consist of a hierarchy of integers with each component having a meaning in the hierarchy. For example, the OID of the DER encoding is defined in the ITU-T X.680 standard as joint-iso-itu-t(2) asnl1(1) ber-derived(2) distinguished-encoding(1) where the text before each integer describes its meaning in that context and the integer is the encoding of that meaning. So the OID itself would be in character form "2.1.2.1" (also called the dot notation introduced by IETF) and in R integer form c(2, 1, 2, 1). Internally, OIDs are represented in their ASN.1 encoding as raw vectors which is the way they are typically used in files or communication payload.

The following functions are used to operate on OIDs.

oid creates an OID.

Coercion methods as.integer and as.character convert the OID into its numeric and textural form respectively. as.oid is a generic for converting objects into OIDs and is implemented for at least the above cases.

is.oid returns TRUE if the object is an OID.
**Usage**

oid(x)

as.oid(x, ...)

## Default S3 method:

as.oid(x, ...)

is.oid(x)

## S3 method for class 'oid'

Ops(e1, e2)

## S3 method for class 'oid'

print(x, ...)

## S3 method for class 'oid'

as.character(x, ...)

## S3 method for class 'oid'

as.integer(x, ...)

**Arguments**

x  
object to covert/create/check

e1  
left-hand side argument for binary operators

e2  
right-hand side argument for binary operators

...  
further arguments (currently unused)

**Details**

The only allowed operators on OIDs are == and != which return TRUE or FALSE.

The oid(x) constructor (and also the as.oid default method) support following types: scalar string (expected to be in dot-notation), integer vector, numeric vector (it is coerced to integer vector implicitly), raw vector (must be ASN.1 encoding of the OID).

The S3 class of OID objects is "oid". It consists of a raw vector representing the ASN.1 encoded OID (without the type specifier). An additional attribute "type" is set to 6L for compatibility with ASN1::encode.

**Author(s)**

Simon Urbanek

**See Also**

ASN1::encode

**Examples**

## RSA algorithm OID:

## iso(1) member-body(2) us(840) rsadsi(113549)

## pkcs(1) pkcs-1(1) rsaEncryption(1)
PKI.crypt

PKI.crypt

PKI encryption/decryption functions

Description

PKI.encrypt encrypts a raw vector
PKI.decrypt decrypts a raw vector

Usage

PKI.encrypt(what, key, cipher = NULL, iv = NULL)
PKI.decrypt(what, key, cipher = NULL, iv = NULL)

Arguments

what raw vector to encrypt/decrypt. It must not exceed the key size minus padding
key key to use for encryption/decryption
cipher cipher to use for encryption/decryption
iv initialization vector for ciphers that use it (e.g., CBC). NULL corresponds to all-zeros IV, otherwise must be either a string or a raw vector with sufficiently many bytes to match the IV length for the cipher.

Value

Raw vector (encrypted/decrypted)

Note

The cipher is optional for key objects that already contain the cipher information such as RSA keys (in fact it is ignored in that case).

Supported symmetric ciphers are AES-128, AES-256 and BF (blowfish). Each cipher can be used in CBC (default), ECB or OFB modes which are specified as suffix, so "aes256ofb" would specify AES-256 in OFB mode. Case and non-alphanumeric characters are ignored, so the same could be specified as "AES-256-OFB". PKCS padding is used to fill up to the block size. Analogously, PKCS padding is expected when decoding.

Note that the payload for RSA encryption should be very small since it must fit into the key size including padding. For example, 1024-bit key can only encrypt 87 bytes, while 2048-bit key can encrypt 215 bytes. Therefore a typical use is to use RSA to transfer a symmetric key to the peer and subsequently use symmetric ciphers like AES for encryption of larger amounts of data.
**PKI.digest**

*Author(s)*

Simon Urbanek

*See Also*

PKI.genRSAkey, PKI.pubkey

*Examples*

```r
e <- PKI.encrypt(x, key)
y <- PKI.decrypt(e, key)
stopifnot(identical(x, y))
print(rawToChar(y))
```

```r
## AES symmetric - use SHA256 to support arbitrarily long key strings
key <- PKI.digest(charToRaw("hello"), "SHA256")
```

```r
ae <- PKI.encrypt(x, key, "aes256")
```

```r
ad <- PKI.decrypt(ae, key, "aes256")
```

*PKI.digest*  
*Compute digest sum based on SHA1, SHA256 or MD5 hash functions*

**Description**

PKI.digest computes digest sum based on the hash function specified.

**Usage**

```r
PKI.digest(what, hash = c("SHA1", "SHA256", "MD5"))
```

**Arguments**

- `what`: raw vector of bytes to digest
- `hash`: type of the hash function. Note that "MD5" should not be used for cryptographic purposes as it is not secure

**Value**

Raw vector containing the hash

**Author(s)**

Simon Urbanek
See Also

PKI.sign

Examples

PKI.digest(as.raw(1:10))

PKI.info

Retrieve PKI back-end information

Description

PKI.info returns information about the engine which is providing the PKI functionality.

Usage

PKI.info()

Value

Named list:

- engine: string, name of the engine, currently either "openssl" or "libressl"
- version: numeric, version of the engine as a real number in the form major.minor
- description: string, description of the engine, its version and any further information that the engine may provide

Note

This function should be treated as informational only. The return value is subject to change, mainly we may extend it to possibly supply information on available ciphers etc.

Older versions of OpenSSL did not provide functional API to retrieve version information, so versions < 1.1 may not reflect the true version, but rather the values from the headers at compile time which may not be the same as the loaded library at run-time.

Author(s)

Simon Urbanek

Examples

str(PKI.info())
PKI.random

Generate cryptographically strong pseudo-random bytes.

Description

PKI.random generates n cryptographically strong pseudo-random bytes.

Usage

PKI.random(n)

Arguments

n  non-negative integer, number of bytes to generate

Details

PKI.random is the preferred way to generate cryptographically strong random content that can be used as keys, seeds etc. Not to be confused with random number generators in R, it is entirely separate for cryptographics purposes.

Value

Raw vector of n cryptographically strong pseudo-random bytes.

Author(s)

Simon Urbanek

Examples

PKI.random(10)

PKI.sign

PKI: sign content or verify a signature

Description

PKI.sign signs content using RSA with the specified hash function
PKI.verify verifies a signature of RSA-signed content

Usage

PKI.sign(what, key, hash = c("SHA1", "SHA256", "MD5"), digest)
PKI.verify(what, signature, key, hash = c("SHA1", "SHA256", "MD5"), digest)
Arguments

what    raw vector: content to sign
key     RSA private key to use for signing; RSA public key or certificate to use for verification.
hash    hash function to use. "MD5" should not be used unless absolutely needed for compatibility as it is less secure.
digest raw vector: it is possible to supply the digest of the content directly instead of specifying what.
signature raw vector: signature

Details

Objects are signed by computing a hash function digest (typically using SHA1 hash function) and then signing the digest with a RSA key. Verification is done by computing the digest and then comparing the signature to the digest. Private key is needed for signing whereas public key is needed for verification.

Both functions call PKI.digest on what if digest is not specified.

Value

PKI.sign signature (raw vector)
PKI.verify logical: TRUE if the digest and signature match, FALSE otherwise

Author(s)

Simon Urbanek

See Also

PKI.pubkey, PKI.genRSAkey, PKI.digest

Examples

d <- PKI.genRSAkey(2048)
x <- charToRaw("My message to sign")
sig <- PKI.sign(x, d)
stopifnot(PKI.verify(x, sig, d))
Description

PKI.sign.tar appends a signature to a tar file
PKI.verify.tar verifies the signature in a tar file

Usage

PKI.sign.tar(tarfile, key, certificate, output = tarfile)
PKI.verify.tar(tarfile, key, silent = FALSE, enforce.cert = FALSE)

Arguments

tarfile  
string, file name of the file to sign

key  
PKI.sign.tar: private key to use for signing; PKI.verify.tar: optional, public key to use for verification

certificate  
optional, certificate to embed in the signature with the public key matching key. If not present the signature will only contain the public key.

output  
file name, connection or raw vector determining how to store the signed tar file

silent  
if TRUE then no warning are generated, otherwise a warning is issues for failed verification describing the reason for failure

enforce.cert  
if TRUE then a certificate is required in the signature. It can be also set to a valid certificate in which case the public key of the certificate in the signature must also match the public key in the supplied certificate.

Details

PKI.tar.sign adds extra entry .signature with the signature based on the contents of the tarfile. Note that any existing signatures are retained. key is a mandatory private key used to sign the content. certificate is optional but if present, it will be embedded in the signature.

The tarfile can be in compressed form (gzip, bzip2 or xz) in which case it is decompressed internally before the signature is applied. If output is a file name then the same compression is applied to the output, otherwise the output is uncompressed.

PKI.verify.tar retrieves the last .signature entry from the tar file (if tarfile is a file name then the same compression auto-detection is applied as above) and verifies the signature against either the supplied (public) key or against the key or certificate stored in the signature. The result is TRUE or FALSE except when enforce.cert is set. In that case the result is the certificate contained in the signature if the validation succeeded (and thus it can be further verified against a chain of trust), otherwise FALSE.
Note

The signature format is ASN.1 DER encoded as follows:
SEQ(signature BITSTRING, subjectPublicKeyInfo, Certificate[opt])

The subjectPublicKeyInfo can be NULL in which case the certificate must be present (in X.509 DER format).

The signature is appended as tar entry named .signature. However, terminating blocks are not removed from the file, so the signature is placed after the EOF blocks and thus doesn’t affect extraction.

Author(s)

Simon Urbanek

---

raw2hex

Convert raw vector to string hex representation

Description

raw2hex converts a raw vector into hexadecimal representation

Usage

raw2hex(what, sep, upper = FALSE)

Arguments

what          raw vector
sep           optional separator string
upper         logical, if TRUE then upper case letters are used, otherwise any letters will be lower case.

Details

If sep is omitted or NULL then the resulting character vector will have as many elements as the raw vector. Otherwise the elements are concatenated using the specified separator into one character string. This is much more efficient than using paste(raw2hex(x), collapse=sep), but has the same effect.

Value

Character vector with the hexadecimal representation of the raw vector.

Author(s)

Simon Urbanek
Examples

raw2hex(PKI.digest(raw(), "SHA1"), "")
raw2hex(PKI.digest(raw(), "MD5"), ":")

## this is just a performance comparison and a test that
## raw2hex can handle long strings
x <- as.raw(runif(1e5) * 255.9)
system.time(h1 <- raw2hex(x, ":")
system.time(h2 <- paste(raw2hex(x), collapse=" "))
stopifnot(identical(h1, h2))

PKI functions handling RSA keys

Description

PKI.load.key loads an RSA key in PKCS#1/8 PEM or DER format.
PKI.save.key creates a PEM or DER representation of a RSA key.
PKI.genRSAkey generates RSA public/private key pair.
PKI.mkRSApubkey creates a RSA public key with the supplied modulus and exponent.
PKI.load.OpenSSH.pubkey loads public key in OpenSSH format (as used in .ssh/authorized_keys file)

Usage

PKI.load.key(what, format = c("PEM", "DER"), private, file, password="")
PKI.save.key(key, format = c("PEM", "DER"), private, target)
PKI.genRSAkey(bits = 2048L)
PKI.mkRSApubkey(modulus, exponent=65537L, format = c("DER", "PEM", "key"))
PKI.load.OpenSSH.pubkey(what, first=TRUE, format = c("DER", "PEM", "key"))

Arguments

what  string, raw vector or connection to load the key from
key   RSA key object
format format - PEM is ASCII (essentially base64-encoded DER with header/footer),
         DER is binary and key means an actual key object
private logical, whether to use the private key (TRUE), public key (FALSE) or whichever
         is available (NA or missing).
file   filename to load the key from - what and file are mutually exclusive
password string, used only if what is an encrypted private key as the password to decrypt
         the key
target optional connection or a file name to store the result in. If missing, the result
         is just returned form the function as either a character vector (PEM) or a raw
         vector (DER).
bits size of the generated key in bits. Must be \(2^n\) with integer \(n > 8\).
modulus modulus either as a raw vector (see \texttt{as.BIGNUMint}) or bigz object (from \texttt{gmp} package) or an integer.
exponent exponent either as a raw vector (see \texttt{as.BIGNUMint}) or bigz object (from \texttt{gmp} package) or an integer.
first logical, if \texttt{TRUE} only the first key will be used, otherwise the result is a list of keys.

Value

\texttt{PKI.load.key}: private or public key object
\texttt{PKI.save.key}: raw vector (DER format) or character vector (PEM format).
\texttt{PKI.genRSAkey}: private + public key object
\texttt{PKI.mkRSApubkey, PKI.load.OpenSSH.pubkey}: raw vector (DER format) or character vector (PEM format) or a "public.key" object.

Note

The output format for private keys in PEM is PKCS#1, but for public keys it is X.509 SubjectPublicKeyInfo (certificate public key). This is consistent with OpenSSL RSA command line tool which uses the same convention.

\texttt{PKI.load.key} can auto-detect the contained format based on the header if ‘PEM’ format is used. In that case it supports PKCS#1 (naked RSA key), PKCS#8 (wrapped key with identifier - for public keys X.509 SubjectPublicKeyInfo) and encrypted private key in PKCS#8 (password must be passed to decrypt). ‘DER’ format provides no way to define the type so ‘private’ cannot be ‘NA’ and only the default format (PKCS#1 for private keys and X.509 SubjectPublicKeyInfo for public keys) is supported.

The OpenSSH format is one line beginning with "\texttt{ssh-rsa}". SSH2 PEM public keys (rfc4716) are supported in \texttt{PKI.load.key} and the binary payload is the same as the OpenSSH, only with different wrapping.

Author(s)

Simon Urbanek

See Also

\texttt{PKI.encrypt, PKI.decrypt, PKI.pubkey}

Examples

# generate 2048-bit RSA key
key <- PKI.genRSAkey(bits = 2048L)

# extract private and public parts as PEM
priv.pem <- PKI.save.key(key)
pub.pem <- PKI.save.key(key, private=FALSE)
# load back the public key separately
pub.k <- PKI.load.key(pub.pem)

# encrypt with the public key
x <- PKI.encrypt(charToRaw("Hello, world!"), pub.k)

# decrypt with private key
rawToChar(PKI.decrypt(x, key))

# compute SHA1 hash (fingerprint) of the public key
PKI.digest(PKI.save.key(key, "DER", private=FALSE))

# convert OpenSSH public key to PEM format
# (the example is split into multiple lines just
# so it is readable in the documentation, in reality you can
# simply use the full line from is_rsa.pub without gsub)
PKI.load.OpenSSH.pubkey(gsub("\n", "",
  "ssh-rsa AAAAB3NzaC1yc2EAAAABIAwAAAIEAuvOXqfZ3pJeWeqyQOIXZwmg
M1RBqPUnVx3XgntpaAYt0ZjKfuosJspg3LhBuI/wXx8LQZXNFibvX4qXZqoYsb
Hvkz2uoa3F7HRHCJRBJURR5n135znVqALzo328v86HDsVYWrZJ3yY1X8GI2R
2iKUMGXF0hVrphdwL8735CU= foo@mycomputer"), format="PEM")
default logical, if TRUE then root CAs known to OpenSSL will be added to the trust store. In that case ca can also be NULL if the certificate is directly signed by the root CA (very uncommon).

partial logical, if TRUE then the CAs listed in ca are trusted even if they are neither root nor self-signed CAs.

Details

PKI.verifyCA is used to verify the validity of a certificate by following a chain of trust. In the most simple case the certificate was issued by a certificate authority (CA) directly, which has a self-signed certificate. This is typically the case when you (or your organization) have created your own CA for internal use. In that case you only need to supply that CA’s certificate to ca and that’s it. It is also possible that your self-signed CA issued an intermediate certificate - if that is the case then pass a list of both certificates (order doesn’t matter) to ca.

Another use case is that you have a certificate which has been issued by publicly trusted CA - this is commonly the case with SSL certificates used by web servers. In that case, the chain doesn’t end with an internal self-signed certificate, but instead it will end with a publicly known root CA. OpenSSL manages a list of such trusted CAs and you can check against them with default=TRUE. However, in most cases your certificate won’t be issued directly by a root CA, but by an intermediate authority so you have to pass the intermediate certificate(s) in the ca argument.

Finally, it is sometimes possible that the default list of trusted certificates does not include the root CA that you need. If that is the case, and you still want to trust that chain, you can set partial=TRUE and then PKI.verifyCA will trust the certificates provided in ca unconditionally, even if they don’t lead to a trusted root or are not self-signed. Note, however, that this is the least secure option and you should only use it if the certificates are supplied by you and not the user. If you want to support user-supplied intermediate certificates then you can use PKI.verifyCA first to verify the integrity of the user-supplied chain with partial=TRUE and then verify just the intermediate certificate against your trusted certificate. That way you won’t trust the intermediate certificate inadvertently.

Value

PKI.load.cert: a certificate object
PKI.verifyCA: TRUE is the certificate can be trusted, FALSE otherwise
PKI.pubkey: public key object
PKI.get.subject: string containing the subject information in one-line RFC2253 format but in UTF8 encoding instead of MBS escapes. NOTE: this is experimental, we may choose to parse the contents and return it in native R form as a named vector instead.

Author(s)

Simon Urbanek

Examples

```r
(ca <- PKI.load.cert(file=system.file("certs", "RForge-ca.crt", package="PKI")))
(my.cert <- PKI.load.cert(readLines(system.file("certs", "demo.crt", package="PKI"))))
PKI.verifyCA(my.cert, ca)
PKI.pubkey(my.cert)
```
PKI.get.subject(my.cert)
PKI.get.cert.info(my.cert)
Index

* interface
  PKI.info, 8
* manip
  ASN1, 2
  BIGNUMint, 3
  oid, 4
  PKI.crypt, 6
  PKI.digest, 7
  PKI.random, 9
  PKI.sign, 9
  PKI.sign.tar, 11
  raw2hex, 12
  RSA, 13
  X509, 15

  as.BIGNUMint, 14
  as.BIGNUMint(BIGNUMint), 3
  as.character.oid(oid), 4
  as.integer.oid(oid), 4
  as.oid(oid), 4
  ASN1, 2
  ASN1.encode, 5

  BIGNUMint, 3
  is.oid(oid), 4

  oid, 4
  Ops.oid(oid), 4

  PKI.crypt, 6
  PKI.decrypt, 14
  PKI.decrypt(PKI.crypt), 6
  PKI.digest, 7, 10
  PKI.encrypt, 14
  PKI.encrypt(PKI.crypt), 6
  PKI.genRSAkey, 7, 10
  PKI.genRSAkey(RSA), 13
  PKI.get.cert.info(X509), 15
  PKI.get.subject(X509), 15
  PKI.info, 8
  PKI.load.cert(X509), 15
  PKI.load.key(RSA), 13
  PKI.load.OpenSSH.pubkey(RSA), 13
  PKI.mkRSApubkey(RSA), 13
  PKI.pubkey, 7, 10, 14
  PKI.pubkey(X509), 15
  PKI.random, 9
  PKI.save.key(RSA), 13
  PKI.sign, 8, 9
  PKI.sign.tar, 11
  PKI.verify(PKI.sign), 9
  PKI.verify.tar(PKI.sign.tar), 11
  PKI.verifyCA(X509), 15
  print.oid(oid), 4

  raw2hex, 12
  RSA, 13

  X509, 15