Package ‘openssl’

November 15, 2018

Type Package

Title Toolkit for Encryption, Signatures and Certificates Based on OpenSSL

Version 1.1

Description Bindings to OpenSSL libssl and libcrypto, plus custom SSH pubkey parsers. Supports RSA, DSA and EC curves P-256, P-384 and P-521. Cryptographic signatures can either be created and verified manually or via x509 certificates. AES can be used in cbc, ctr or gcm mode for symmetric encryption; RSA for asymmetric (public key) encryption or EC for Diffie Hellman. High-level envelope functions combine RSA and AES for encrypting arbitrary sized data. Other utilities include key generators, hash functions (md5, sha1, sha256, etc), base64 encoder, a secure random number generator, and 'bignum' math methods for manually performing crypto calculations on large multibyte integers.

License MIT + file LICENSE

URL https://github.com/jeroen/openssl#readme

BugReports https://github.com/jeroen/openssl/issues

SystemRequirements OpenSSL >= 1.0.1

VignetteBuilder knitr

Suggests testthat, digest, knitr, rmarkdown, jsonlite, jose

RoxygenNote 6.1.1

NeedsCompilation yes

Author Jeroen Ooms [aut, cre] (<https://orcid.org/0000-0002-4035-0289>), Oliver Keyes [ctb]

Maintainer Jeroen Ooms <jeroen@berkeley.edu>

Repository CRAN

Date/Publication 2018-11-15 14:20:03 UTC
Description

Low-level symmetric encryption/decryption using the AES block cipher in CBC mode. The key is a raw vector, for example a hash of some secret. When no shared secret is available, a random key can be used which is exchanged via an asymmetric protocol such as RSA. See rsa_encrypt for a worked example or encrypt_envelope for a high-level wrapper combining AES and RSA.

Usage

```r
aes_ctr_encrypt(data, key, iv = rand_bytes(16))

aes_ctr_decrypt(data, key, iv = attr(data, "iv"))

aes_cbc_encrypt(data, key, iv = rand_bytes(16))

aes_cbc_decrypt(data, key, iv = attr(data, "iv"))

aes_gcm_encrypt(data, key, iv = rand_bytes(12))

aes_gcm_decrypt(data, key, iv = attr(data, "iv"))

aes_keygen(length = 16)
```
askpass

Arguments

- **data**: raw vector or path to file with data to encrypt or decrypt
- **key**: raw vector of length 16, 24 or 32, e.g. the hash of a shared secret
- **iv**: raw vector of length 16 (aes block size) or NULL. The initialization vector is not secret but should be random
- **length**: how many bytes to generate. Usually 16 (128-bit) or 12 (92-bit) for aes_gcm

Examples

```r
# aes-256 requires 32 byte key
passphrase <- charToRaw("This is super secret")
key <- sha256(passphrase)

# symmetric encryption uses same key for decryption
x <- serialize(iris, NULL)
y <- aes_cbc_encrypt(x, key = key)
x2 <- aes_cbc_decrypt(y, key = key)
stopifnot(identical(x, x2))
```

Description

Function to prompt the user for a password to read a protected private key. Frontends can provide a custom password entry widget by setting the askpass option. If no such option is specified we default to **readline**.

Usage

```r
askpass(prompt = "Please enter your password: ")
```

Arguments

- **prompt**: the string printed when prompting the user for input.
base64_encode  
*Encode and decode base64*

**Description**

Encode and decode binary data into a base64 string. Character vectors are automatically collapsed into a single string.

**Usage**

```r
base64_encode(bin, linebreaks = FALSE)
```

```r
base64_decode(text)
```

**Arguments**

- `bin`  
  raw or character vector with data to encode into base64

- `linebreaks`  
  insert linebreaks in the base64 message to make it more readable

- `text`  
  string with base64 data to decode

**Examples**

```r
input <- charToRaw("foo = bar + 5")
message <- base64_encode(input)
output <- base64_decode(message)
identical(output, input)
```

bignum  
*Big number arithmetic*

**Description**

Basic operations for working with large integers. The `bignum` function converts a positive integer, string or raw vector into a bignum type. All basic Arithmetic and Comparison operators such as `+`, `-`, `*`, `/`, `%`, `%%`, `==`, `!=`, `<`, `<=`, `>`, `>=` are implemented for bignum objects. The Modular exponent (`a^b %% m`) can be calculated using `bignum_mod_exp` when `b` is too large for calculating `a^b` directly.

**Usage**

```r
bignum(x, hex = FALSE)
```

```r
bignum_mod_exp(a, b, m)
```

```r
bignum_mod_inv(a, m)
```
cert_verify

Arguments

X509 certificates

Arguments

cert
certificate (or certificate-chain) to be verified. Must be cert or list or path.
root	rusted pubkey or certificate(s) e.g. CA bundle.
host.string: hostname of the server to connect to
port.string or integer: port or protocol to use, e.g: 443 or "https"
ipv4_only.do not use IPv6 connections

Examples

# create a bignum
x <- bignum(123L)
y <- bignum("123456789123456789")
z <- bignum("D41D8CD98F00B204E980098ECF8427E", hex = TRUE)

# Basic arithmetic
div <- z %/% y
mod <- z %% y
z2 <- div * y + mod
stopifnot(z2 == z)
stopifnot(div < z)
Details

If https verification fails and you can't figure out why, have a look at https://ssldecoder.org.

See Also

read_cert

Examples

# Verify the r-project HTTPS cert
chain <- download_ssl_cert("www.r-project.org", 443)
print(chain)
print(as.list(chain[[1]])$pubkey)
cert_verify(chain, ca_bundle())

# Another example
chain <- download_ssl_cert("public.opencpu.org")
ocpu <- chain[[1]]
as.list(ocpu)$subject

# Write PEM format
write_pem(ocpu)

---

**ec_dh**

**Diffie-Hellman Key Agreement**

Description

Key agreement is one-step method of creating a shared secret between two peers. Both peers can independently derive the joined secret by combining his or her private key with the public key from the peer.

Usage

```
ec_dh(key = my_key(), peerkey, password = askpass)
```

Arguments

- **key**: your own private key
- **peerkey**: the public key from your peer
- **password**: passed to `read_key` for reading protected private keys

Details

Currently only Elliptic Curve Diffie Hellman (ECDH) is implemented.
**encrypt_envelope**

**References**


**Examples**

```r
## Not run:
# Need two EC keypairs from the same curve
alice_key <- ec_keygen("P-521")
bob_key <- ec_keygen("P-521")

# Derive public keys
alice_pub <- as.list(alice_key)$pubkey
bob_pub <- as.list(bob_key)$pubkey

# Both peers can derive the (same) shared secret via each other's pubkey
ec_dh(alice_key, bob_pub)
ec_dh(bob_key, alice_pub)

## End(Not run)
```

---

**encrypt_envelope**  
*Envelope encryption*

**Description**

An envelope contains ciphertext along with an encrypted session key and optionally and initialization vector. The `encrypt_envelope` generates a random IV and session-key which is used to encrypt the data with AES stream cipher. The session key itself is encrypted using the given RSA key (see `rsa_encrypt`) and stored or sent along with the encrypted data. Each of these outputs is required to decrypt the data with the corresponding private key.

**Usage**

```r
crypt_encrypt(data, pubkey = my_pubkey())

crypt_decrypt(data, iv, session, key = my_key(), password)
```

**Arguments**

- `data` raw data vector or file path for message to be signed. If hash == NULL then data must be a hash string or raw vector.
- `pubkey` public key or file path. See `read_pubkey`.
- `iv` 16 byte raw vector returned by `encrypt_envelope`.
- `session` raw vector with encrypted session key as returned by `encrypt_envelope`.
- `key` private key or file path. See `read_key`.
- `password` string or a function to read protected keys. See `read_key`. 
fingerprint

References

https://wiki.openssl.org/index.php/EVP_Asymmetric_Encryption_and_Decryption_of_an_Envelope

Examples

# Requires RSA key
key <- rsa_keygen()
pubkey <- key$pubkey
msg <- serialize(iris, NULL)

# Encrypt
out <- encrypt_envelope(msg, pubkey)
str(out)

# Decrypt
orig <- decrypt_envelope(out$data, out$iv, out$session, key)
stopifnot(identical(msg, orig))

---

fingerprint (OpenSSH fingerprint)

Description

Calculates the OpenSSH fingerprint of a public key. This value should match what you get to see when connecting with SSH to a server. Note that some other systems might use a different algorithm to derive a (different) fingerprint for the same keypair.

Usage

fingerprint(key, hashfun = md5)

Arguments

key a public or private key
hashfun which hash function to use to calculate the fingerprint

Examples

mykey <- rsa_keygen()
pubkey <- as.list(mykey)$pubkey
fingerprint(mykey)
fingerprint(pubkey)

# Some systems use other hash functions
fingerprint(pubkey, sha1)
fingerprint(pubkey, sha256)

# Other key types
fingerprint(dsa_keygen())
hashing

Vectorized hash/hmac functions

Description

All hash functions either calculate a hash-digest for key == NULL or HMAC (hashed message authentication code) when key is not NULL. Supported inputs are binary (raw vector), strings (character vector) or a connection object.

Usage

- sha1HxL(key = NULL)
- sha224HxL(key = NULL)
- sha256HxL(key = NULL)
- sha384HxL(key = NULL)
- sha512HxL(key = NULL)
- sha2(x, size = 256, key = NULL)
- md4HxL(key = NULL)
- md5HxL(key = NULL)
- blake2bHxL(key = NULL)
- blake2sHxL(key = NULL)
- ripemd160HxL(key = NULL)

Arguments

- x: character vector, raw vector or connection object.
- key: string or raw vector used as the key for HMAC hashing
- size: must be equal to 224 256 384 or 512

Details

Functions are vectorized for the case of character vectors: a vector with n strings returns n hashes. When passing a connection object, the contents will be stream-hashed which minimizes the amount of required memory. This is recommended for hashing files from disk or network.

The sha2 family of algorithms (sha224, sha256, sha384 and sha512) is generally recommended for sensitive information. While sha1 and md5 are usually sufficient for collision-resistant identifiers, they are no longer considered secure for cryptographic purposes.
In applications where hashes should be irreversible (such as names or passwords) it is often recommended to use a random \textit{key} for HMAC hashing. This prevents attacks where we can lookup hashes of common and/or short strings. See examples. A common special case is adding a random salt to a large number of records to test for uniqueness within the dataset, while simultaneously rendering the results incomparable to other datasets.

The \texttt{blake2b} and \texttt{blake2s} algorithms are only available if your system has libssl 1.1 or newer.

\textbf{References}

Digest types: \url{https://www.openssl.org/docs/man1.1/dgst.html}

\textbf{Examples}

\begin{verbatim}
# Support both strings and binary
md5(c("foo", "bar"))
md5("foo", key = "secret")

hash <- md5(charToRaw("foo"))
as.character(hash, sep = ":")

# Compare to digest
digest::digest("foo", "md5", serialize = FALSE)

# Other way around
digest::digest(cars, skip = 0)
md5(serialize(cars, NULL))

# Stream-verify from connections (including files)
myfile <- system.file("CITATION")
md5(file(myfile))
md5(file(myfile), key = "secret")

## Not run: check md5 from: http://cran.r-project.org/bin/windows/base/old/3.1.1/md5sum.txt
md5(url("http://cran.r-project.org/bin/windows/base/old/3.1.1/R-3.1.1-win.exe"))
## End(Not run)

# Use a salt to prevent dictionary attacks
sha1("admin") # googleable
sha1("admin", key = "random_salt_value") # not googleable

# Use a random salt to identify duplicates while anonymizing values
sha256("john") # googleable
sha256(c("john", "mary", "john"), key = "random_salt_value")
\end{verbatim}
my_key

Description

The keygen functions generate a random private key. Use `as.list(key)$pubkey` to derive the corresponding public key. Use `write_pem` to save a private key to a file, optionally with a password.

Usage

```r
rsa_keygen(bits = 2048)

dsa_keygen(bits = 1024)

ec_keygen(curve = c("P-256", "P-384", "P-521"))
```

Arguments

- `bits`: bitsize of the generated RSA/DSA key
- `curve`: which NIST curve to use

Examples

```r
# Generate keypair
key <- rsa_keygen()
pubkey <- as.list(key)$pubkey

# Write/read the key with a passphrase
write_pem(key, "id_rsa", password = "supersecret")
read_key("id_rsa", password = "supersecret")
```

my_key  

Default key

Description

The default user key can be set in the USER_KEY variable and otherwise is `~/.ssh/id_rsa`. Note that on Windows we treat `~` as the windows user home (and not the documents folder).

Usage

```r
my_key()

my_pubkey()
```

Details

The `my_pubkey()` function looks for the public key by appending `.pub` to the above key path. If this file does not exist, it reads the private key file and automatically derives the corresponding pubkey. In the latter case the user may be prompted for a passphrase if the private key is protected.
Examples

```r
# Set random RSA key as default
call("rsa_keygen()");
write_pem(key, temp <- tempfile(), password = "")
rm(key)
Sys.setenv("USER_KEY" = tmp)

# Check the new keys
print(my_key())
print(my_pubkey())
```

Description

Bindings to OpenSSL libssl and libcrypto, plus custom SSH pubkey parsers. Supports RSA, DSA and NIST curves P-256, P-384 and P-521. Cryptographic signatures can either be created and verified manually or via x509 certificates. The AES block cipher is used in CBC mode for symmetric encryption; RSA for asymmetric (public key) encryption. High-level envelope methods combine RSA and AES for encrypting arbitrary sized data. Other utilities include key generators, hash functions (md5, sha1, sha256, etc), base64 encoder, a secure random number generator, and bignum math methods for manually performing crypto calculations on large multibyte integers.

Author(s)

Jeroen Ooms, Oliver Keyes

Description

Shows libssl version and configuration information.

Usage

```r
openssl_config()
```
**Description**

this set of functions generates random bytes or numbers from OpenSSL. This provides a cryptographically secure alternative to R's default random number generator. `rand_bytes` generates \( n \) random cryptographically secure bytes

**Usage**

\[
\text{rand_bytes}(n = 1) \\
\text{rand_num}(n = 1)
\]

**Arguments**

\( n \)  
number of random bytes or numbers to generate

**References**


**Examples**

\[
\text{rnd} \leftarrow \text{rand_bytes}(10) \\
\text{as.numeric(rnd)} \\
\text{as.character(rnd)} \\
\text{as.logical(rawToBits(rnd))}
\]

# bytes range from 0 to 255
\[
\text{rnd} \leftarrow \text{rand_bytes}(100000) \\
\text{hist(\text{as.numeric(rnd)}, \text{breaks}=-1:255)}
\]

# Generate random doubles between 0 and 1
\[
\text{rand_num}(5)
\]

# Use CDF to map \([0,1]\) into random draws from a distribution
\[
\text{x} \leftarrow \text{qnorm(\text{rand_num}(1000), \text{mean}=100, \text{sd}=15)} \\
\text{hist(x)}
\]

\[
\text{y} \leftarrow \text{qbinom(\text{rand_num}(1000), \text{size}=10, \text{prob}=0.3)} \\
\text{hist(y)}
\]
read_key

Parsing keys and certificates

Description
The `read_key` function (private keys) and `read_pubkey` (public keys) support both SSH pubkey format and OpenSSL PEM format (base64 data with a `--BEGIN` and `--END` header), and automatically convert where necessary. The functions assume a single key per file except for `read_cert_bundle` which supports PEM files with multiple certificates.

Usage
```
read_key(file, password = askpass, der = is.raw(file))
read_pubkey(file, der = is.raw(file))
read_cert(file, der = is.raw(file))
read_cert_bundle(file)
read_pem(file)
```

Arguments
- **file**: Either a path to a file, a connection, or literal data (a string for pem/ssh format, or a raw vector in der format)
- **password**: A string or callback function to read protected keys
- **der**: set to TRUE if file is in binary DER format

Details
Most versions of OpenSSL support at least RSA, DSA and ECDSA keys. Certificates must conform to the X509 standard.

The `password` argument is needed when reading keys that are protected with a passphrase. It can either be a string containing the passphrase, or a custom callback function that will be called by OpenSSL to read the passphrase. The function should take one argument (a string with a message) and return a string. The default is to use `readline` which will prompt the user in an interactive R session.

Value
An object of class `cert`, `key` or `pubkey` which holds the data in binary DER format and can be decomposed using `as.list`.

See Also
- `download_ssl_cert`
rsa_encrypt

Examples

## Not run: # Read private key
key <- read_key("~/.ssh/id_rsa")
str(key)

# Read public key
pubkey <- read_pubkey("~/.ssh/id_rsa.pub")
str(pubkey)

# Read certificates
txt <- readLines("https://curl.haxx.se/ca/cacert.pem")
bundle <- read_cert_bundle(txt)
print(bundle)

## End(Not run)

---

rsa_encrypt | Low-level RSA encryption

Description

Asymmetric encryption and decryption with RSA. Because RSA can only encrypt messages smaller than the size of the key, it is typically used only for exchanging a random session-key. This session key is used to encipher arbitrary sized data via a stream cipher such as aes_cbc. See encrypt_envelope for a high-level wrappers combining RSA and AES in this way.

Usage

```r
rsa_encrypt(data, pubkey = my_pubkey())

rsa_decrypt(data, key = my_key(), password = askpass)
```

Arguments

data | raw vector of max 245 bytes (for 2048 bit keys) with data to encrypt/decrypt
pubkey | public key or file path. See read_pubkey.
key | private key or file path. See read_key.
password | string or a function to read protected keys. See read_key.

Examples

```r
# Generate test keys
key <- rsa_keygen()
pubkey <- key$pubkey

# Encrypt data with AES
tempkey <- rand_bytes(32)
iv <- rand_bytes(16)
```
signature_create

Description

Sign and verify a message digest. RSA supports both MD5 and SHA signatures whereas DSA and EC keys only support SHA.

Usage

signature_create(data, hash = sha1, key = my_key(),
password = askpass)

signature_verify(data, sig, hash = sha1, pubkey = my_pubkey())

Arguments

data raw data vector or file path for message to be signed. If hash == NULL then data must be a hash string or raw vector.
hash the digest function to use. Must be one of md5, sha1, sha256, sha512 or NULL.
key private key or file path. See read_key.
password string or a function to read protected keys. See read_key.
sig raw vector or file path for the signature data.
pubkey public key or file path. See read_pubkey.

Examples

# Generate a keypair
key <- rsa_keygen()
pubkey <- key$pubkey

# Sign a file
data <- system.file("DESCRIPTION")
sig <- signature_create(data, key = key)
stopifnot(signature_verify(data, sig, pubkey = pubkey))

# Sign raw data

blob <- aes_cbc_encrypt(system.file("CITATION"), tempkey, iv = iv)

# Encrypt tempkey using receivers public RSA key
ciphertext <- rsa_encrypt(tempkey, pubkey)

# Receiver decrypts tempkey from private RSA key
tempkey <- rsa_decrypt(ciphertext, key)
message <- aes_cbc_decrypt(blob, tempkey, iv)
out <- rawToChar(message)
write_p12

data <- serialize(iris, NULL)
sig <- signature_create(data, sha256, key = key)
stopifnot(signature_verify(data, sig, sha256, pubkey = pubkey))

# Sign a hash
md <- md5(data)
sig <- signature_create(md, hash = NULL, key = key)
stopifnot(signature_verify(md, sig, hash = NULL, pubkey = pubkey))

---

write_p12

**PKCS7 / PKCS12 bundles**

**Description**

PKCS7 and PKCS12 are container formats for storing multiple certificates and/or keys.

**Usage**

write_p12(key = NULL, cert = NULL, ca = NULL, name = NULL,
password = NULL, path = NULL)

write_p7b(ca, path = NULL)

read_p12(file, password = askpass)

read_p7b(file, der = is.raw(file))

**Arguments**

- **key**
  - a private key
- **cert**
  - certificate that matches ‘key’
- **ca**
  - a list of certificates (the CA chain)
- **name**
  - a friendly title for the bundle
- **password**
  - string or function to set/get the password.
- **path**
  - a file where to write the output to. If ‘NULL’ the output is returned as a raw vector.
- **file**
  - path or raw vector with binary PKCS12 data to parse
- **der**
  - set to TRUE for binary files and FALSE for PEM files

**Details**

The PKCS#7 or P7B format is a container for one or more certificates. It can either be stored in binary form or in a PEM file. P7B files are typically used to import and export public certificates.

The PKCS#12 or PFX format is a binary-only format for storing the server certificate, any intermediate certificates, and the private key into a single encryptable file. PFX files are usually found with
The extensions .pfx and .p12. PFX files are typically used to import and export certificates with their private keys. The PKCS formats also allow for including signatures and CRLs but this is quite rare and these are currently ignored.

---

**write_pem**

Export key or certificate

---

**Description**

The `write_pem` function exports a key or certificate to the standard base64 PEM format. For private keys it is possible to set a password.

**Usage**

```
write_pem(x, path = NULL, password = NULL)
write_der(x, path = NULL)
write_pkcs1(x, path = NULL, password = NULL)
write_ssh(pubkey, path = NULL)
```

**Arguments**

- `x`: a public/private key or certificate object
- `path`: file to write to. If `NULL` it returns the output as a string.
- `password`: string or callback function to set password (only applicable for private keys).
- `pubkey`: a public key

**Details**

The pkcs1 format is a legacy format which only supports RSA keys and should not be used anymore. It is only provided for compatibility with some old SSH clients. Simply use `write_pem` to export keys and certs to the recommended format.

**Examples**

```
# Generate RSA keypair
key <- rsa_keygen()
pubkey <- key$pubkey

# Write to output formats
write_ssh(pubkey)
write_pem(pubkey)
write_pem(key, password = "super secret")
```
AES, 7
AES block cipher, 12
aes_cbc, 2, 15
aes_cbc_decrypt (aes_cbc), 2
aes_cbc_encrypt (aes_cbc), 2
aes_ctr_decrypt (aes_cbc), 2
aes_ctr_encrypt (aes_cbc), 2
aes_gcm_decrypt (aes_cbc), 2
aes_gcm_encrypt (aes_cbc), 2
aes_keygen (aes_cbc), 2
Arithmetic, 4
askpass, 3
asymmetric (public key), 12
base64, 12
base64_decode (base64_encode), 4
base64_encode, 4
bignum, 4, 12
bignum_mod_exp, 4
bignum_mod_exp (bignum), 4
bignum_mod_inv (bignum), 4
blake2b (hashing), 9
blake2s (hashing), 9
certificates, 12
Comparison, 4
decrypt_envelope (encrypt_envelope), 7
download_ssl_cert, 14
download_ssl_cert (cert_verify), 5
dsa_keygen (keygen), 10
ec_dh, 6
ec_keygen (keygen), 10
encrypt (rsa_encrypt), 15
encrypt_envelope, 2, 7, 15
envelope, 12
envelope (encrypt_envelope), 7
fingerprint, 8
hashing, 9
hmac (hashing), 9
key generators, 12
keygen, 10
mac (hashing), 9
md4 (hashing), 9
md5, 12, 16
md5 (hashing), 9
my_key, 11
my_pubkey (my_key), 11
openssl, 12
openssl_package (openssl), 12
openssl_config, 12
pfx (write_p12), 17
pkcs12 (write_p12), 17
pubkey, 12
rand_bytes, 13
rand_num (rand_bytes), 13
random number generator, 12
read_cert, 6
read_cert (read_key), 14
read_cert_bundle (read_key), 14
read_key, 6, 7, 14, 15, 16
read_p12 (write_p12), 17
read_p7b (write_p12), 17
read_pem (read_key), 14
read_pubkey, 7, 15, 16
read_pubkey (read_key), 14
readline, 3
ripemd160 (hashing), 9
rsa (rsa_encrypt), 15
rsa_decrypt (rsa_encrypt), 15
rsa_encrypt, 2, 7, 15
rsa_keygen (keygen), 10
sha1, 12, 16
sha1 (hashing), 9
sha2 (hashing), 9
sha224 (hashing), 9
sha256, 12, 16
sha256 (hashing), 9
sha384 (hashing), 9
sha512, 16
sha512 (hashing), 9
signature_create, 16
signature_verify (signature_create), 16
signatures, 12

write_der (write_pem), 18
write_p12, 17
write_p7b (write_p12), 17
write_pem, 11, 18
write_pkcs1 (write_pem), 18
write_ssh (write_pem), 18