Package ‘openssl’

May 26, 2025

Type Package

Title Toolkit for Encryption, Signatures and Certificates Based on OpenSSL

Version 2.3.3

Description Bindings to OpenSSL libssl and libcrypto, plus custom SSH key parsers. Supports RSA, DSA and EC curves P-256, P-384, P-521, and curve25519. 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://jeroen.r-universe.dev/openssl

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

SystemRequirements OpenSSL >= 1.0.2

VignetteBuilder knitr

Imports askpass

Suggests curl, testthat (>= 2.1.0), digest, knitr, rmarkdown, jsonlite, jose, sodium

RoxygenNote 7.3.2

Encoding UTF-8

NeedsCompilation yes

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

Maintainer Jeroen Ooms <jeroenooms@gmail.com>

Repository CRAN

Date/Publication 2025-05-26 13:00:02 UTC

Contents

`aes_cbc` *Symmetric AES encryption*

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

```
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)
```

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

```
# 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))
```

`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

```
base64_encode(bin, linebreaks = FALSE)

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

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

---

bcrypt_pbkdf *Bcrypt PWKDF*

Description

Password based key derivation function with bcrypt. This is not part of openssl. It is needed to parse private key files which are encoded in the new openssh format.

Usage

```
bcrypt_pbkdf(password, salt, rounds = 16L, size = 32L)
```

Arguments

password	string or raw vector with password
salt	raw vector with (usually 16) bytes
rounds	number of hashing rounds
size	desired length of the output key

---

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 `+`, `-`, `*`, `^`, `%%`, `%/%`, `==`, `!=`, `<`, `<=`, `>` and `>=` 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

```
bignum(x, hex = FALSE)

bignum_mod_exp(a, b, m)

bignum_mod_inv(a, m)
```

Arguments

`x`	an integer, string (hex or dec) or raw vector
`hex`	set to TRUE to parse strings as hex rather than decimal notation
`a`	bignum value for `(a^b %% m)`
`b`	bignum value for `(a^b %% m)`
`m`	bignum value for `(a^b %% m)`

Examples

```
# create a bignum
x <- bignum(123L)
y <- bignum("123456789123456789")
z <- bignum("D41D8CD98F00B204E9800998ECF8427E", hex = TRUE)

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

`cert_verify` *X509 certificates*

Description

Read, download, analyze and verify X.509 certificates.

Usage

```
cert_verify(cert, root = ca_bundle())

download_ssl_cert(host = "localhost", port = 443, ipv4_only = FALSE)

ca_bundle()
```

Arguments

`cert`	certificate (or certificate-chain) to be verified. Must be cert or list or path.
`root`	trusted 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

See Also

read_cert

Examples

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

# Write cert in PEM format
cat(write_pem(chain[[1]]))
```

---

curve25519 *Curve25519*

Description

Curve25519 is a recently added low-level algorithm that can be used both for diffie-hellman (called X25519) and for signatures (called ED25519). Note that these functions are only available when building against version 1.1.1 or newer of the openssl library. The same functions are also available in the sodium R package.

Usage

```
read_ed25519_key(x)

read_ed25519_pubkey(x)

read_x25519_key(x)

read_x25519_pubkey(x)

ed25519_sign(data, key)

ed25519_verify(data, sig, pubkey)

x25519_diffie_hellman(key, pubkey)
```

Arguments

x	a 32 byte raw vector with (pub)key data
data	raw vector with data to sign or verify

key	private key as returned by `read_ed25519_key` or `ed25519_keygen`
sig	raw vector of length 64 with signature as returned by `ed25519_sign`
pubkey	public key as returned by `read_ed25519_pubkey` or `key$pubkey`

Examples

```
# Generate a keypair
if(openssl_config()$x25519){
key <- ed25519_keygen()
pubkey <- as.list(key)$pubkey

# Sign message
msg <- serialize(iris, NULL)
sig <- ed25519_sign(msg, key)

# Verify the signature
ed25519_verify(msg, sig, pubkey)

# Diffie Hellman example:
key1 <- x25519_keygen()
key2 <- x25519_keygen()

# Both parties can derive the same secret
x25519_diffie_hellman(key1, key2$pubkey)
x25519_diffie_hellman(key2, key1$pubkey)

# Import/export sodium keys
rawkey <- sodium::sig_keygen()
rawpubkey <- sodium::sig_pubkey(rawkey)
key <- read_ed25519_key(rawkey)
pubkey <- read_ed25519_pubkey(rawpubkey)

# To get the raw key data back for use in sodium
as.list(key)$data
as.list(pubkey)$data
}
```

---

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.

References

https://wiki.openssl.org/index.php/EVP_Key_Agreement, https://wiki.openssl.org/index.php/Elliptic_Curve_Diffie_Hellman

Examples

```
## 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

```
encrypt_envelope(data, pubkey = my_pubkey())

decrypt_envelope(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().

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 = sha256)
```

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

```
sha1(x, key = NULL)

sha224(x, key = NULL)

sha256(x, key = NULL)

sha384(x, key = NULL)

sha512(x, key = NULL)

keccak(x, size = 256, key = NULL)

sha2(x, size = 256, key = NULL)

sha3(x, size = 256, key = NULL)

md4(x, key = NULL)

md5(x, key = NULL)

blake2b(x, key = NULL)

blake2s(x, key = NULL)
```

```
ripemd160(x, key = NULL)

multihash(x, algos = c("md5", "sha1", "sha256", "sha384", "sha512"))
```

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
`algos`	string vector with names of hashing algorithms

Details

The most efficient way to calculate hashes is by using input connections, such as a file() or url() object. In this case the hash is calculated streamingly, using almost no memory or disk space, regardless of the data size. When using a connection input in the multihash function, the data is only read only once while streaming to multiple hash functions simultaneously. Therefore several hashes are calculated simultanously, without the need to store any data or download it multiple times.

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 *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 `blake2b` and `blake2s` algorithms are only available if your system has libssl 1.1 or newer.

NB R base `file()` function has a poor default `raw = FALSE` which causes files to get altereted (e.g. decompressed) when reading. Use `file(path, raw = TRUE)` to get the hash of the file as it exists on your disk.

References

Digest types: https://docs.openssl.org/1.1.1/man1/dgst/

Examples

```
# 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, raw = TRUE))
md5(file(myfile, raw = TRUE), 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")
```

---

keygen *Generate Key pair*

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

```
rsa_keygen(bits = 2048)

dsa_keygen(bits = 1024)

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

x25519_keygen()

ed25519_keygen()
```

Arguments

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

Examples

```
# 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")
unlink("id_rsa")
```

`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

```
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

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

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

openssl — *Toolkit for Encryption, Signatures and Certificates based on OpenSSL*

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

See Also

Useful links:

- https://jeroen.r-universe.dev/openssl
- Report bugs at https://github.com/jeroen/openssl/issues

openssl_config — *OpenSSL Configuration Info*

Description

Shows libssl version and configuration information.

Usage

```
openssl_config()

fips_mode()
```

Details

Note that the "fips" flag in `openssl_config` means that FIPS is supported, but it does not mean that it is currently enforced. If supported, it can be enabled in several ways, such as a kernel option, or setting an environment variable `OPENSSL_FORCE_FIPS_MODE=1`. The `fips_mode()` function shows if FIPS is currently enforced.

pkcs7_encrypt *Encrypt/decrypt pkcs7 messages*

Description

Encrypt or decrypt messages using PKCS7 smime format. Note PKCS7 only supports RSA keys.

Usage

```
pkcs7_encrypt(message, cert, pem = TRUE)

pkcs7_decrypt(input, key, der = is.raw(input))
```

Arguments

message	text or raw vector with data to encrypt
cert	the certificate with public key to use for encryption
pem	convert output pkcs7 data to PEM format
input	file path or string with PEM or raw vector with p7b data
key	private key to decrypt data
der	assume input is in DER format (rather than PEM)

See Also

encrypt_envelope

rand_bytes *Generate random bytes and numbers with OpenSSL*

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

```
rand_bytes(n = 1)

rand_num(n = 1)
```

Arguments

n	number of random bytes or numbers to generate

References

OpenSSL manual: https://docs.openssl.org/1.1.1/man3/RAND_bytes/

Examples

```
rnd <- rand_bytes(10)
as.numeric(rnd)
as.character(rnd)
as.logical(rawToBits(rnd))

# bytes range from 0 to 255
rnd <- rand_bytes(100000)
hist(as.numeric(rnd), breaks=-1:255)

# Generate random doubles between 0 and 1
rand_num(5)

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

y <- qbinom(rand_num(1000), size=10, prob=0.3)
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

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 or pkcs7_encrypt for a high-level wrappers combining RSA and AES in this way.

Usage

```
rsa_encrypt(data, pubkey = my_pubkey(), oaep = FALSE)

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

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()`.
`oaep`	if TRUE, changes padding to EME-OAEP as defined in PKCS #1 v2.0
`key`	private key or file path. See `read_key()`.
`password`	string or a function to read protected keys. See `read_key()`.

Examples

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

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

`signature_create` *Signatures*

Description

Sign and verify a message digest. RSA supports both MD5 and SHA signatures whereas DSA and EC keys only support SHA. ED25591 can sign any payload so you can set `hash` to `NULL` to sign the raw input data.

Usage

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

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

ecdsa_parse(sig)

ecdsa_write(r, s)
```

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()`.
`r`	bignum value for r parameter
`s`	bignum value for s parameter

Details

The `ecdsa_parse` and `ecdsa_write` functions convert (EC)DSA signatures between the conventional DER format and the raw `(r,s)` bignum pair. Most users won't need this, it is mostly here to support the JWT format (which does not use DER).

Examples

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

# Sign a file
data <- system.file("DESCRIPTION")
```

```
sig <- signature_create(data, sha256, key = key)
stopifnot(signature_verify(data, sig, sha256, pubkey = pubkey))

# Sign raw data
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 = sha256, key = key)
stopifnot(signature_verify(md, sig, hash = sha256, pubkey = pubkey))
#
# ECDSA example
data <- serialize(iris, NULL)
key <- ec_keygen()
pubkey <- key$pubkey
sig <- signature_create(data, sha256, key = key)
stopifnot(signature_verify(data, sig, sha256, pubkey = pubkey))

# Convert signature to (r, s) parameters and then back
params <- ecdsa_parse(sig)
out <- ecdsa_write(params$r, params$s)
identical(sig, out)
```

`ssl_ctx` *Hooks to manipulate the SSL context for curl requests*

Description

These functions allow for manipulating the SSL context from inside the CURLOPT_SSL_CTX_FUNCTION callback using the curl R package. Note that this is not fully portable and will only work on installations that use matching versions of libssl (see details). It is recommended to only use this locally and if what you need cannot be accomplished using standard libcurl TLS options, e.g. those listed in `curl::curl_options('ssl')` or `curl::curl_options('tls')`.

Usage

```
ssl_ctx_add_cert_to_store(ssl_ctx, cert)

ssl_ctx_set_verify_callback(ssl_ctx, cb)

ssl_ctx_curl_version_match()
```

Arguments

`ssl_ctx`	pointer object to the SSL context provided in the ssl_ctx_function callback.
`cert`	certificate object, e.g from read_cert or download_ssl_cert.
`cb`	callback function with 1 parameter (the server certificate) and which returns TRUE (for proceed) or FALSE (for abort).

Details

Curl allows for setting an option called `ssl_ctx_function`: this is a callback function that is triggered during the TLS initiation, before any https connection has been made. This serves as a hook to let you manipulate the TLS configuration (called `SSL_CTX` for historical reasons), in order to control how to curl will validate the authenticity of server certificates for upcoming TLS connections.

Currently we provide 2 such functions: ssl_ctx_add_cert_to_store injects a custom certificate into the trust-store of the current TLS connection. But most flexibility is provided via ssl_ctx_set_verify_callback which allows you to override the function that is used by validate if a server certificate should be trusted. The callback will receive one argument `cert` and has to return `TRUE` or `FALSE` to decide if the cert should be trusted.

By default libcurl re-uses connections, hence the cert validation is only performed in the first request to a given host. Subsequent requests use the already established TLS connection. For testing, it can be useful to set `forbid_reuse` in order to make a new connection for each request, as done in the examples below.

System compatibility

Passing the SSL_CTX between the curl and openssl R packages only works if they are linked to the same version of libssl. Use ssl_ctx_curl_version_match to test if this is the case. On Debian / Ubuntu you need to build the R curl package against `libcurl4-openssl-dev`, which is usually the case. On Windows you would need to set `CURL_SSL_BACKEND=openssl` in your `~/.Renviron` file. On MacOS things are complicated because it uses LibreSSL instead of OpenSSL by default. You can make it work by compiling the curl R package from source against the homebrew version of curl and then then set `CURL_SSL_BACKEND=openssl` in your `~/.Renviron` file. If your curl and openssl R packages use different versions of libssl, the examples may segfault due to ABI incompatibility of the SSL_CTX structure.

Examples

```
## Not run:
# Example 1: accept your local snakeoil https cert
mycert <- openssl::download_ssl_cert('localhost')[[1]]

# Setup the callback
h <- curl::new_handle(ssl_ctx_function = function(ssl_ctx){
  ssl_ctx_add_cert_to_store(ssl_ctx, mycert)
}, verbose = TRUE, forbid_reuse = TRUE)

# Perform the request
req <- curl::curl_fetch_memory('https://localhost', handle = h)

# Example 2 using a custom verify function
verify_cb <- function(cert){
  id <- cert$pubkey$fingerprint
  cat("Server cert from:", as.character(id), "\n")
  TRUE # always accept cert
}
```

```
h <- curl::new_handle(ssl_ctx_function = function(ssl_ctx){
  ssl_ctx_set_verify_callback(ssl_ctx, verify_cb)
}, verbose = TRUE, forbid_reuse = TRUE)

# Perform the request
req <- curl::curl_fetch_memory('https://localhost', handle = h)

## End(Not run)
```

---

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` functions 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)

write_openssh_pem(key, 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
`key`	a private key

Details

The pkcs1 format is the old legacy format used by OpenSSH. PKCS1 does not support the new ed25519 keys, for which you need `write_openssh_pem`. For non-ssh clients, we recommend to simply use `write_pem` to export keys and certs into the recommended formats.

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")
```

Index