Package 'cyphr'

October 12, 2022

Title High Level Encryption Wrappers

Version 1.1.4

Description Encryption wrappers, using low-level support from 'sodium' and 'openssl'. 'cyphr' tries to smooth over some pain points when using encryption within applications and data analysis by wrapping around differences in function names and arguments in different encryption providing packages. It also provides high-level wrappers for input/output functions for seamlessly adding encryption to existing analyses.

License MIT + file LICENSE

URL https://github.com/ropensci/cyphr, https://docs.ropensci.org/cyphr/

BugReports https://github.com/ropensci/cyphr/issues

Imports getPass, openssl (>= 0.9.9), sodium (>= 1.2.1)

Suggests knitr, rmarkdown, testthat

RoxygenNote 7.1.1

VignetteBuilder rmarkdown, knitr

Encoding UTF-8

Language en-GB

NeedsCompilation no

Author Rich FitzJohn [aut, cre], Jai Ranganathan [ctb]

Maintainer Rich FitzJohn <rich.fitzjohn@gmail.com>

Repository CRAN

Date/Publication 2022-06-20 11:30:02 UTC

R topics documented:

`cyphr` *High Level Encryption Wrappers*

Description

Encryption wrappers, using low-level support from sodium and openssl.

Details

It is *strongly* recommended that you read *both* vignettes before attempting to use `cyphr`.

- introduction; in R: `vignette("cyphr", package = "cyphr")`
- data vignette; in R: `vignette("data", package = "cyphr")`

Author(s)

Rich FitzJohn (rich.fitzjohn@gmail.com)

`data_admin_init` *Encrypted data administration*

Description

Encrypted data administration; functions for setting up, adding users, etc.

Usage

```
data_admin_init(path_data, path_user = NULL, quiet = FALSE)

data_admin_authorise(
  path_data = NULL,
  hash = NULL,
  path_user = NULL,
  yes = FALSE,
  quiet = FALSE
)

data_admin_list_requests(path_data = NULL)

data_admin_list_keys(path_data = NULL)
```

Arguments

path_data	Path to the data set. We will store a bunch of things in a hidden directory within this path. By default in most functions we will search down the tree until we find the .cyphr directory
path_user	Path to the directory with your ssh key. Usually this can be omitted.
quiet	Suppress printing of informative messages.
hash	A vector of hashes to add. If provided, each hash can be the binary or string representation of the hash to add. Or omit to add each request.
yes	Skip the confirmation prompt? If any request is declined then the function will throw an error on exit.

Details

`data_admin_init` initialises the system; it will create a data key if it does not exist and authorise you. If it already exists and you do not have access it will throw an error.

`data_admin_authorise` authorises a key by creating a key to the data that the user can use in conjunction with their personal key.

`data_admin_list_requests` lists current requests.

`data_admin_list_keys` lists known keys that can access the data. Note that this is *not secure*; keys not listed here may still be able to access the data (if a key was authorised and moved elsewhere for example). Conversely, if the user has deleted or changed their key they will not be able to access the data despite the key being listed here.

See Also

`data_request_access()` for requesting access to the data, and and `data_key` for using the data itself. But for a much more thorough overview, see the vignette (`vignette("data", package = "cyphr")`).

Examples

```
# The workflow here does not really lend itself to an example,
# please see the vignette instead.

# First we need a set of user ssh keys.  In a non example
# environment your personal ssh keys will probably work well, but
# hopefully they are password protected so cannot be used in
# examples.  The password = FALSE argument is only for testing,
# and should not be used for data that you care about.
path_ssh_key <- tempfile()
cyphr::ssh_keygen(path_ssh_key, password = FALSE)

# Initialise the data directory, using this key path.  Ordinarily
# the path_user argument would not be needed because we would be
# using your user ssh keys:
path_data <- tempfile()
dir.create(path_data, FALSE, TRUE)
cyphr::data_admin_init(path_data, path_user = path_ssh_key)

# Now you can get the data key
key <- cyphr::data_key(path_data, path_user = path_ssh_key)

# And encrypt things with it
cyphr::encrypt_string("hello", key)

# See the vignette for more details.  This is not the best medium
# to explore this.

# Cleanup
unlink(path_ssh_key, recursive = TRUE)
unlink(path_data, recursive = TRUE)
```

---

data_request_access *User commands*

Description

User commands

Usage

```
data_request_access(path_data = NULL, path_user = NULL, quiet = FALSE)

data_key(
  path_data = NULL,
  path_user = NULL,
  test = TRUE,
  quiet = FALSE,
```

```
  cache = TRUE
)
```

Arguments

path_data	Path to the data. If not given, then we look recursively down below the working directory for a ".cyphr" directory, and use that as the data directory.
path_user	Path to the directory with your user key. Usually this can be omitted. This argument is passed in as both pub and key to keypair_openssl(). Briefly, if this argument is not given we look at the environment variables USER_PUBKEY and USER_KEY - if set then these must refer to path of your public and private keys. If these environment variables are not set then we fall back on ~/.ssh/id_rsa.pub and ~/.ssh/id_rsa, which should work in most environments. Alternatively, provide a path to a directory where the file id_rsa.pub and id_rsa can be found.
quiet	Suppress printing of informative messages.
test	Test that the encryption is working? (Recommended)
cache	Cache the key within the session. This will be useful if you are using ssh keys that have passwords, as if the key is found within the cache, then you will not have to re-enter your password. Using cache = FALSE neither looks for the key in the cache, nor saves it.

Examples

```
# The workflow here does not really lend itself to an example,
# please see the vignette.

# Suppose that Alice has created a data directory:
path_alice <- tempfile()
cyphr::ssh_keygen(path_alice, password = FALSE)
path_data <- tempfile()
dir.create(path_data, FALSE, TRUE)
cyphr::data_admin_init(path_data, path_user = path_alice)

# If Bob can also write to the data directory (e.g., it is a
# shared git repo, on a shared drive, etc), then he can request
# access
path_bob <- tempfile()
cyphr::ssh_keygen(path_bob, password = FALSE)
hash <- cyphr::data_request_access(path_data, path_user = path_bob)

# Alice can authorise Bob
cyphr::data_admin_authorise(path_data, path_user = path_alice, yes = TRUE)

# After which Bob can get the data key
cyphr::data_key(path_data, path_user = path_bob)

# See the vignette for more details.  This is not the best medium
# to explore this.
```

```
# Cleanup
unlink(path_alice, recursive = TRUE)
unlink(path_bob, recursive = TRUE)
unlink(path_data, recursive = TRUE)
```

`encrypt` *Easy encryption and decryption*

Description

Wrapper functions for encryption. These functions wrap expressions that produce or consume a file and arrange to encrypt (for producing functions) or decrypt (for consuming functions). The forms with a trailing underscore (`encrypt_`, `decrypt_`) do not use any non-standard evaluation and may be more useful for programming.

Usage

```
encrypt(expr, key, file_arg = NULL, envir = parent.frame())

decrypt(expr, key, file_arg = NULL, envir = parent.frame())

encrypt_(expr, key, file_arg = NULL, envir = parent.frame())

decrypt_(expr, key, file_arg = NULL, envir = parent.frame())
```

Arguments

`expr`	A single expression representing a function call that would be called for the side effect of creating or reading a file.
`key`	A `cyphr_key` object describing the encryption approach to use.
`file_arg`	Optional hint indicating which argument to `expr` is the filename. This is done automatically for some built-in functions.
`envir`	Environment in which `expr` is to be evaluated.

Details

These functions will not work for all functions. For example `pdf/dev.off` will create a file but we can't wrap those up (yet!). Functions that *modify* a file (e.g., appending) also will not work and may cause data loss.

Examples

```
# To do anything we first need a key:
key <- cyphr::key_sodium(sodium::keygen())

# Encrypted write.csv - note how any number of arguments to
```

```
# write.csv will be passed along
path <- tempfile(fileext = ".csv")
cyphr::encrypt(write.csv(iris, path, row.names = FALSE), key)

# The new file now exists, but you would not be able to read it
# with read.csv because it is now binary data.
file.exists(path)

# Wrap the read.csv call with cyphr::decrypt()
dat <- cyphr::decrypt(read.csv(path, stringsAsFactors = FALSE), key)
head(dat)

file.remove(path)

# If you have a function that is not supported you can specify the
# filename argument directly.  For example, with "write.dcf" the
# filename argument is called "file"; we can pass that along
path <- tempfile()
cyphr::encrypt(write.dcf(list(a = 1), path), key, file_arg = "file")

# Similarly for decryption:
cyphr::decrypt(read.dcf(path), key, file_arg = "file")
```

`encrypt_data` *Encrypt and decrypt data and other things*

Description

Encrypt and decrypt raw data, objects, strings and files. The core functions here are `encrypt_data` and `decrypt_data` which take raw data and decrypt it, writing either to file or returning a raw vector. The other functions encrypt and decrypt arbitrary R objects (`encrypt_object`, `decrypt_object`), strings (`encrypt_string`, `decrypt_string`) and files (`encrypt_file`, `decrypt_file`).

Usage

```
encrypt_data(data, key, dest = NULL)

encrypt_object(object, key, dest = NULL, rds_version = NULL)

encrypt_string(string, key, dest = NULL)

encrypt_file(path, key, dest = NULL)

decrypt_data(data, key, dest = NULL)

decrypt_object(data, key)

decrypt_string(data, key)

decrypt_file(path, key, dest = NULL)
```

Arguments

data	(for encrypt_data, decrypt_data, decrypt_object, decrypt_string) a raw vector with the data to be encrypted or decrypted. For the decryption functions this must be data derived by encrypting something or you will get an error.
key	A cyphr_key object describing the encryption approach to use.
dest	The destination filename for the encrypted or decrypted data, or NULL to return a raw vector. This is not used by decrypt_object or decrypt_string which always return an object or string.
object	(for encrypt_object) an arbitrary R object to encrypt. It will be serialised to raw first (see serialize).
rds_version	RDS serialisation version to use (see serialize. The default in R version 3.3 and below is version 2 - in the R 3.4 series version 3 was introduced and is becoming the default. Version 3 format serialisation is not understood by older versions so if you need to exchange data with older R versions, you will need to use rds_version = 2. The default argument here (NULL) will ensure the same serialisation is used as R would use by default.
string	(for encrypt_string) a scalar character vector to encrypt. It will be converted to raw first with charToRaw.
path	(for encrypt_file) the name of a file to encrypt. It will first be read into R as binary (see readBin).

Examples

```
key <- key_sodium(sodium::keygen())
# Some super secret data we want to encrypt:
x <- runif(10)
# Convert the data into a raw vector:
data <- serialize(x, NULL)
data
# Encrypt the data; without the key above we will never be able to
# decrypt this.
data_enc <- encrypt_data(data, key)
data_enc
# Our random numbers:
unserialize(decrypt_data(data_enc, key))
# Same as the never-encrypted version:
x

# This can be achieved more easily using `encrypt_object`:
data_enc <- encrypt_object(x, key)
identical(decrypt_object(data_enc, key), x)

# Encrypt strings easily:
str_enc <- encrypt_string("secret message", key)
str_enc
decrypt_string(str_enc, key)
```

keypair_openssl *Asymmetric encryption with openssl*

Description

Wrap a pair of openssl keys. You should pass your private key and the public key of the person that you are communicating with.

Usage

```
keypair_openssl(
  pub,
  key,
  envelope = TRUE,
  password = NULL,
  authenticated = TRUE
)
```

Arguments

pub	An openssl public key. Usually this will be the path to the key, in which case it may either the path to a public key or be the path to a directory containing a file `id_rsa.pub`. If `NULL`, then your public key will be used (found via the environment variable `USER_PUBKEY`, then `~/.ssh/id_rsa.pub`). However, it is not that common to use your own public key - typically you want either the sender of a message you are going to decrypt, or the recipient of a message you want to send.
key	An openssl private key. Usually this will be the path to the key, in which case it may either the path to a private key or be the path to a directory containing a file. You may specify `NULL` here, in which case the environment variable `USER_KEY` is checked and if that is not defined then `~/.ssh/id_rsa` will be used.
envelope	A logical indicating if "envelope" encryption functions should be used. If so, then we use `openssl::encrypt_envelope()` and `openssl::decrypt_envelope()`. If `FALSE` then we use `openssl::rsa_encrypt()` and `openssl::rsa_decrypt()`. See the openssl docs for further details. The main effect of this is that using `envelope = TRUE` will allow you to encrypt much larger data than `envelope = FALSE`; this is because openssl asymmetric encryption can only encrypt data up to the size of the key itself.
password	A password for the private key. If `NULL` then you will be prompted interactively for your password, and if a string then that string will be used as the password (but be careful in scripts!)
authenticated	Logical, indicating if the result should be signed with your public key. If `TRUE` then your key will be verified on decryption. This provides tampering detection.

See Also

`keypair_sodium()` for a similar function using sodium keypairs

Examples

```
# Note this uses password = FALSE for use in examples only, but
# this should not be done for any data you actually care about.

# Note that the vignette contains much more information than this
# short example and should be referred to before using these
# functions.

# Generate two keypairs, one for Alice, and one for Bob
path_alice <- tempfile()
path_bob <- tempfile()
cyphr::ssh_keygen(path_alice, password = FALSE)
cyphr::ssh_keygen(path_bob, password = FALSE)

# Alice wants to send Bob a message so she creates a key pair with
# her private key and bob's public key (she does not have bob's
# private key).
pair_alice <- cyphr::keypair_openssl(pub = path_bob, key = path_alice)

# She can then encrypt a secret message:
secret <- cyphr::encrypt_string("hi bob", pair_alice)
secret

# Bob wants to read the message so he creates a key pair using
# Alice's public key and his private key:
pair_bob <- cyphr::keypair_openssl(pub = path_alice, key = path_bob)

cyphr::decrypt_string(secret, pair_bob)

# Clean up
unlink(path_alice, recursive = TRUE)
unlink(path_bob, recursive = TRUE)
```

---

keypair_sodium *Asymmetric encryption with sodium*

Description

Wrap a pair of sodium keys for asymmetric encryption. You should pass your private key and the public key of the person that you are communicating with.

Usage

```
keypair_sodium(pub, key, authenticated = TRUE)
```

Arguments

pub	A sodium public key. This is either a raw vector of length 32 or a path to file containing the contents of the key (written by `writeBin()`).
key	A sodium private key. This is either a raw vector of length 32 or a path to file containing the contents of the key (written by `writeBin()`).
authenticated	Logical, indicating if authenticated encryption (via `sodium::auth_encrypt()` / `sodium::auth_decrypt()`) should be used. If `FALSE` then `sodium::simple_encrypt()` / `sodium::simple_decrypt()` will be used. The difference is that with `authenticated = TRUE` the message is signed with your private key so that tampering with the message will be detected.

Details

NOTE: the order here (pub, key) is very important; if the wrong order is used you cannot decrypt things. Unfortunately because sodium keys are just byte sequences there is nothing to distinguish the public and private keys so this is a pretty easy mistake to make.

See Also

`keypair_openssl()` for a similar function using openssl keypairs

Examples

```
# Generate two keypairs, one for Alice, and one for Bob
key_alice <- sodium::keygen()
pub_alice <- sodium::pubkey(key_alice)
key_bob <- sodium::keygen()
pub_bob <- sodium::pubkey(key_bob)

# Alice wants to send Bob a message so she creates a key pair with
# her private key and bob's public key (she does not have bob's
# private key).
pair_alice <- cyphr::keypair_sodium(pub = pub_bob, key = key_alice)

# She can then encrypt a secret message:
secret <- cyphr::encrypt_string("hi bob", pair_alice)
secret

# Bob wants to read the message so he creates a key pair using
# Alice's public key and his private key:
pair_bob <- cyphr::keypair_sodium(pub = pub_alice, key = key_bob)

cyphr::decrypt_string(secret, pair_bob)
```

key_openssl *Symmetric encryption with openssl*

Description

Wrap an openssl symmetric (aes) key. This can be used with the functions `encrypt_data()` and `decrypt_data()`, along with the higher level wrappers `encrypt()` and `decrypt()`. With a symmetric key, everybody uses the same key for encryption and decryption.

Usage

```
key_openssl(key, mode = "cbc")
```

Arguments

key	An openssl aes key (i.e., an object of class `aes`).
mode	The encryption mode to use. Options are `cbc`, `ctr` and `gcm` (see the `openssl` package for more details)

Examples

```
# Create a new key
key <- cyphr::key_openssl(openssl::aes_keygen())
key

# With this key encrypt a string
secret <- cyphr::encrypt_string("my secret string", key)
# And decrypt it again:
cyphr::decrypt_string(secret, key)
```

key_sodium *Symmetric encryption with sodium*

Description

Wrap a sodium symmetric key. This can be used with the functions `encrypt_data()` and `decrypt_data()`, along with the higher level wrappers `encrypt()` and `decrypt()`. With a symmetric key, everybody uses the same key for encryption and decryption.

Usage

```
key_sodium(key)
```

Arguments

key	A sodium key (i.e., generated with `sodium::keygen()`

Examples

```
# Create a new key
key <- cyphr::key_sodium(sodium::keygen())
key

# With this key encrypt a string
secret <- cyphr::encrypt_string("my secret string", key)
# And decrypt it again:
cyphr::decrypt_string(secret, key)
```

`rewrite_register` *Register functions to work with encrypt/decrypt*

Description

Add information about argument rewriting so that they can be used with encrypt and decrypt.

Usage

```
rewrite_register(package, name, arg, fn = NULL)
```

Arguments

`package`	The name of the package with the function to support (as a scalar character). If your function has no package (e.g., a function you are working on outside of a package, use "" as the name).
`name`	The name of the function to support.
`arg`	The name of the argument in the target function that refers to the file that should be encrypted or decrypted. This is the value you would pass through to `file_arg` in encrypt.
`fn`	Optional (and should be rare) argument used to work around functions that pass all their arguments through to a second function as dots. This is how `read.csv` works. If needed this function is a length-2 character vector in the form "package", "name" with the actual function that is used. But this should be very rare!

Details

If your package uses cyphr, it might be useful to add this as an `.onLoad()` hook.

Examples

```
# The saveRDS function is already supported.  But if we wanted to
# support it we could look at the arguments for the function:
args(saveRDS)
# The 'file' argument is the one that refers to the filename, so
# we'd write:
cyphr::rewrite_register("base", "saveRDS", "file")
```

```
# It's non-API but you can see what is supported in the package by
# looking at
ls(cyphr:::db)
```

`session_key_refresh` *Refresh the session key*

Description

Refresh the session key, invalidating all keys created by `key_openssl()`, `keypair_openssl()`, `key_sodium()` and `keypair_sodium()`.

Usage

```
session_key_refresh()
```

Details

Running this function will invalidate *all* keys loaded with the above functions. It should not be needed very often.

Examples

```
# Be careful - if you run this then all keys loaded from file will
# no longer work until reloaded
if (FALSE) {
  cyphr::session_key_refresh()
}
```

`ssh_keygen` *Create ssh keypairs*

Description

Create openssl key pairs in the manner of `ssh-keygen`(1). In general this should not be used (generate keys yourself with `ssh-keygen` at the command line. However this is useful for testing and demonstration so I have included it to make that easier. Once a keypair has been generated it can be used with `keypair_openssl()`.

Usage

```
ssh_keygen(path = tempfile(), password = TRUE, use_shell = FALSE)
```

Arguments

path	A directory in which to create a keypair. If the path does not exist it will be created.
password	The password for the key. The default will prompt interactively (but without echoing the password). Other valid options are FALSE (no password) or a string.
use_shell	Try to use ssh-keygen (the shell utility) rather than functions in the openssl package. This will be necessary on at least very old versions of OS/X (Yosemite and older at least) where the keys generated by the openssl package cannot be read by the system ssh commands (e.g., ssh-add).

Value

The path, invisibly. This is useful in the case where path is tempfile().

Examples

```
# Generate a new key in a temporary directory:
path <- cyphr::ssh_keygen(password = FALSE)
dir(path) # will contain id_rsa and id_rsa.pub

# This key can now be used via keypair_openssl:
key <- cyphr::keypair_openssl(path, path)
secret <- cyphr::encrypt_string("hello", key)
cyphr::decrypt_string(secret, key)

# Cleanup
unlink(path, recursive = TRUE)
```

Index