Package: rmonocypher (via r-universe)

October 21, 2024

Type Package

Title Easy Encryption of R Objects using Strong Modern Cryptography

Version 0.1.7.9000

Maintainer Mike Cheng <mikefc@coolbutuseless.com>

Description Easy-to-use encryption of R objects using modern cryptography. Objects are serialized and then encrypted using 'XChaCha20-Poly1305' (<https://en.wikipedia.org/wiki/ChaCha20-Poly1305>) which follows RFC 8439 for authenticated encryption (<https://en.wikipedia.org/wiki/Authenticated_encryption>). Cryptographic functions are provided by the 'monocypher' 'C' library (<https://monocypher.org>).

URL https://github.com/coolbutuseless/rmonocypher

BugReports https://github.com/coolbutuseless/rmonocypher/issues

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.1

Suggests knitr, rmarkdown, testthat (>= 3.0.0)

Depends R (>= 3.4.0)

Copyright This package includes the 'monocypher' library written by Loup Vaillant, Michael Savage and Fabio Scotomi. See file 'inst/LICENSE-monocypher.md' for the license for 'monocypher'.

Config/testthat/edition 3

VignetteBuilder knitr

Repository https://coolbutuseless.r-universe.dev

RemoteUrl https://github.com/coolbutuseless/rmonocypher

RemoteRef HEAD

RemoteSha 2d339f728bb5019eae2da338c5cb323612566eea

Contents

`argon2` *Generate bytes from a password using Argon2 password-based key derivation*

Description

Argon2 is a resource intensive password-based key derivation scheme. A typical application is generating an encryption key from a text password.

Usage

```
argon2(passphrase, salt = passphrase, length = 32, type = "chr")
```

Arguments

`passphrase`	A character string used to derive the random bytes
`salt`	16-byte raw vector or 32-character hexadecimal string. A salt is data used as additional input to key derivation which helps defend against attacks that use pre-computed (i.e. rainbow) tables. Note: A salt does not need to be a secret. See https://en.wikipedia.org/wiki/Salt_(cryptography) for more details. The 'salt' may also be a non-hexadecimal string, in which case a real salt will be created by using Argon2 with a default internal salt.
`length`	Number of bytes to output. Default: 32
`type`	Should the data be returned as raw bytes? Default: "chr". Possible values "chr" or 'raw'

Value

raw vector of the requested length

Note

Using the same password with the same salt will always generate the same key. It is recommended that a random salt be used.

Technical Note

The 'C' version of the ARgon2 algorithm is configured with:

- Use the `Argon2id` variant of the algorithm
- single-threaded
- 3 iterations
- 100 megabytes of memory

See https://en.wikipedia.org/wiki/Argon2 and https://monocypher.org/manual/argon2 for more information.

Examples

```
# For the sake of convenience for novice users, a salt will be
# derived internally from the password.
argon2("my secret")

# Calling 'argon2()' without a seed is equivalent to using the password
# as the seed.  This is not the best security practice
argon2("my secret", salt = "my secret")

# Best practice is to use random bytes for the salt
# This particular key can then only be recovered if the password and
# the salt are known.
salt <- rbyte(16) # You'll want to save this value somewhere
argon2("my secret", salt = salt)
```

`decrypt` *Decrypt an ecnrypted object*

Description

Decrypt an ecnrypted object

Usage

```
decrypt(
  src,
  key = getOption("MONOCYPHER_KEY", default = NULL),
  additional_data = NULL
)
```

Arguments

src	Raw vector or filename
key	The encryption key. This may be a character string, a 32-byte raw vector or a 64-character hex string (which encodes 32 bytes). When a shorter character string is given, a 32-byte key is derived using the Argon2 key derivation function. If a key is not explicitly set by the user when calling the function, an attempt is made to fetch 'MONOCYPHER_KEY' from the session global options.
additional_data	Additional data to include in the authentication. Raw vector or character string. Default: NULL. This additional data is *not* included with the encrypted data, but represents an essential component of the message authentication. The same additional_data must be presented during both encryption and decryption for the message to be authenticated. See vignette on 'Additional Data'.

Value

Decrypted, unserialized R object

Examples

```
key <- argon2('my key')
encrypt(mtcars, key = key) |>
  decrypt(key = key)
```

---

encrypt *Save an encrypted RDS*

Description

Save an encrypted RDS

Usage

```
encrypt(
  robj,
  dst = NULL,
  key = getOption("MONOCYPHER_KEY", default = NULL),
  additional_data = NULL,
  compress = c("none", "gzip", "bzip2", "xz")
)
```

Arguments

robj	R object
dst	Either a filename or NULL. Default: NULL write results to a raw vector

key — The encryption key. This may be a character string, a 32-byte raw vector or a 64-character hex string (which encodes 32 bytes). When a shorter character string is given, a 32-byte key is derived using the Argon2 key derivation function. If a key is not explicitly set by the user when calling the function, an attempt is made to fetch 'MONOCYPHER_KEY' from the session global options.

additional_data — Additional data to include in the authentication. Raw vector or character string. Default: NULL. This additional data is *not* included with the encrypted data, but represents an essential component of the message authentication. The same additional_data must be presented during both encryption and decryption for the message to be authenticated. See vignette on 'Additional Data'.

compress — compression type. Default: 'none'. Possible values: 'none', 'gzip', 'bzip2', 'xz'

Value

Raw vector containing encrypted object written to file or returned

Examples

```
key <- argon2('my key')
encrypt(mtcars, key = key) |>
  decrypt(key = key)
```

---

encrypt_raw — *Low Level Encryption/Decryption or Raw Vectors with 'Authenticated Encryption with Additional Data' (AEAD)*

Description

This is a low-level function for encrypting/decrypting data using 'Authenticated Encryption with Additional Data' (AEAD). This encryption scheme assures data confidentiality (privacy) i.e. the encrypted data is impossible to understand without the knowledge of the secret *key*.

The authenticity of the message is also assured i.e. the message is unforgeable.

Additional data can optionally be included in the encryption process. This data is not encrypted, nor is it included with the output. Instead this data is a part of the message authentication. See below for more details.

Usage

```
encrypt_raw(
  x,
  key = getOption("MONOCYPHER_KEY", default = NULL),
  additional_data = NULL
)

decrypt_raw(
```

```
  src,
  key = getOption("MONOCYPHER_KEY", default = NULL),
  additional_data = NULL
)
```

Arguments

x	Data to encrypt. Character string or raw vector.
key	The encryption key. This may be a character string, a 32-byte raw vector or a 64-character hex string (which encodes 32 bytes). When a shorter character string is given, a 32-byte key is derived using the Argon2 key derivation function. If a key is not explicitly set by the user when calling the function, an attempt is made to fetch `'MONOCYPHER_KEY'` from the session global options.
additional_data	Additional data to include in the authentication. Raw vector or character string. Default: NULL. This additional data is *not* included with the encrypted data, but represents an essential component of the message authentication. The same `additional_data` must be presented during both encryption and decryption for the message to be authenticated. See vignette on 'Additional Data'.
src	Raw vector of data to decrypt

Details

Implements authenticated encryption as documented here https://monocypher.org/manual/aead

Value

`encrypt_raw()` returns a raw vector containing the *nonce*, *mac* and the encrypted data

`decrypt_raw()` returns the decrypted data as a raw vector

Technical Notes

The encryption functions in this package implement RFC 8439 ChaCha20-Poly1305 authenticated encryption with additional data. This algorithm combines the ChaCha20 stream cipher with the Poly1305 message authentication code.

Examples

```
# Encrypt/Decrypt a string or raw vector
# Data to encrypt
dat <- "Follow the white rabbit" |> charToRaw()

# Create an encryption key
key <- argon2("my secret key") # Keep this key secret!
key

# Encrypt the data
enc <- encrypt_raw(dat, key)
enc
```

```
# Using the same key, decrypt the data
decrypt_raw(enc, key) |> rawToChar()
```

`rbyte` *Generate random bytes from the platform-specific cryptographically secure pseudorandom number generator*

Description

Generate random bytes from the platform-specific cryptographically secure pseudorandom number generator

Usage

```
rbyte(n, type = "chr")
```

Arguments

`n`	Number of random bytes to generate. Note: if the entropy pool is exhausted on your system it may not be able to provide the requested number of bytes - in this case an error is thrown.
`type`	Type of returned values - 'raw' or "chr". Default: "chr".

Value

A raw vector or a hexadecimal string

Platform notes

The method used for generating random values varies depending on the operating system (OS):

- For macOS and BSDs: `arc4random_buf()`
- For linux: `syscall(SYS_getrandom())`
- For win32: `BCryptGenRandom()`

All these random number generators are internally seeded by the OS using entropy gathered from multiple sources and are considered cryptographically secure.

Examples

```
rbyte(16, type = "chr")
rbyte(16, type = 'raw')
```

Index