Package 'secretR'

May 17, 2023

Title Manage Secrets

Version 0.0.1

Description Manage secrets, such as passwords and encrypted messages.

License GPL (>= 3)

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.3

Imports sodium

Suggests knitr,
rmarkdown

VignetteBuilder knitr

R topics documented:

`secretR-package` *Manage secrets in R*

Description

SecretR allows to hash passwords, encrypt/decrypt messages, and store results securely in files or memory. To collect passwords from users, a GUI window is used, allowing masked input and to set strong passwords polices.

Details

To this end secretR provides the class, secretr, used to represent three types of secrets:

1. 'plain': a plain text password
2. 'hash': the hash of a plain text password
3. 'cipher': a message encrypted with a password All secretrs (including plain text) are kept in memory or files in raw format.

Author(s)

Maintainer: Antonio Fasano <firstlast@gmail.com>

`cipher` *Cipher secretr*

Description

A cipher secretr is a message encrypted with a password. You normally create a cipher with `cipher`, which executes the encryption for you. If you want to execute the encryption yourself and store the result as a cipher secretr class, you want `cipher.man`.

Usage

```
cipher(msg, pass, nonce = NULL)

cipher.man(enmsg, nonce)
```

Arguments

`msg`	Raw or character message to be encrypted.
`pass`	Plain or hash secretr use as a password.
`nonce`	24 byte nonce as raw
`enmsg`	Encrypted message as raw

Details

Note that a plain password is always hashed before being used to generate a cipher. The encryption and password hashing are computed with the sodium package data_encrypt and hash functions. A cipher consists of an encrypted message and 24-bit random nonce. Password hashes have a 32-byte size.

By default, `cipher` generates a non-deterministic cipher, because its nonce is random. Pass the `nonce` arg to obtain the same cipher for the same input message. See the examples for this. If you already have the encrypted-message/nonce pair you might use `cipher.man` to generate the cipher secretr. You could do this for testing and debugging purposes, as a rule you do not need to.

Value

A cipher secretr.

Examples

```
## Generate a cipher
cipher("hello", passcode("xyz"))

## Generate your nonce or capture an existing one
n <- as.raw(sample(1:255, 24, replace = TRUE))
n <- secretr.nonce(cipher("hello", passcode("xyz")))

## cipher() is non-deterministic
identical(cipher("hello", passcode("xyz")),
          cipher("hello", passcode("xyz"))) # FALSE

## ... unless we pass a consistent nonce
identical(cipher("hello", passcode("xyz"), nonce = n),
          cipher("hello", passcode("xyz"), nonce = n))

## Plain passwords are hashed before use
identical(cipher("hello", passcode("xyz"),              nonce = n),
          cipher("hello", passcode("xyz", hash = TRUE), nonce = n))

## Generate a manual cipher
c <- cipher("hello", passcode("xyz"))
e <- secretr.open(c)
n <- secretr.nonce(c)
identical(c, cipher.man(e, n))
```

`gui` *GUI functions to query user passwords*

Description

These functions provide a cross-platform approach to query the user for passwords using a masked input, which is not standardised in the console, if at all possible. `pwset` is for initial password setting, and requires a second confirmation input. `pwget` is for later validation, hence does not require the double-user input.

Usage

```
pwset(
  prompt = "Enter password",
  penv = NULL,
  pfile = NULL,
  mask = TRUE,
  policy = pwpolicy(),
  hash = TRUE
)

pwget(
  prompt = "Enter password",
  penv = NULL,
  pfile = NULL,
```

```
  pvalue = NULL,
  hash = TRUE,
  mask = TRUE
)
```

Arguments

prompt	The first prompt. Confirmation is standardised.
penv	Optional environment to store or validate against the secretr in pcode var.
pfile	Optional file to store or validate the secretr.
mask	Masking stars if TRUE.
policy	Policy list. See related pwpolicy().
hash	Hash the user password returned or stored
pvalue	Plain or hash secretr to validate user password.

Details

Hashes are 32-byte large and obtained from sodium package hash(). If hashing is used, the original password remains unknown. If penv is not NULL, it is an environment: pwset uses this environment to store the user input, as a secretr, in a variable named pcode; pwget looks for the variable pcode in this environment to validate the user input.

By default. pwset applies the password policy defined by pwpolicy(). See this function for the default policy and for modifying it by setting the policy arg.

If more validation args are given to pwget, their values should match, such that the unique value is used, otherwise an error is raised. If no validation arg is given, it does not occurs and the typed password is returned as secretr, whose type depends on the hasharg. When validation occurs, validation sources (which can be can be 'plain' or 'hash' secretrs) are always hashed and compared with the typed password hash, therefore the default hash = TRUE should stay unchanged

Value

pwset returns a secretr object of type 'plain' or 'hash' depending on hash argument.

If no validation occurs, pwgetreturns the same values as pwset, otherwise it returns the logical result of the validation.

See Also

pwpolicy() to learn or change the default password policy.

Examples

```
## To learn about the default password policy see pwpolicy().
## Not run:
## Ask and store as a plain secretr in a file and the variable `e$pcode`
e <- new.env()
tmp <- tempfile()
pwset(pfile = tmp, penv = e, hash = FALSE)
identical(e$pcode, readSecret(tmp))
secretr.open(e$pcode, human = TRUE)

## The same example, but now with a hash secretr
h <- pwset(pfile = tmp, penv = e, hash = TRUE)
```

```
identical(h, readSecret(tmp))
secretr.open(h)

## We edit password policy with minimum 2 digits and no symbol need
pp <- pwpolicy(min.digit = 2, min.sym = 0)
p <- pwset(pfile = tmp, penv = e, hash = FALSE, policy = pp)
secretr.open(p, human = TRUE)

## If we enter the same password as in p, validation is passed
pwget(pfile = tmp, penv = e)  # T/F result
# Validation sources are always hashed, so don't set 'hash = FALSE'

## Without validation args, it makes sense to set 'hash' arg
p <- pwget(hash = FALSE)
identical(p, passcode("123456a!", hash = FALSE))
h <- pwget(hash = TRUE)
identical(h, passcode("123456a!", hash = TRUE))

## Cleanup
file.remove(tmp)


## End(Not run)
```

`passcode` *Plain and hash secretr*

Description

Plain secretrs are plain text passwords and hash secretrs are the hashes of hashed passwords. `passcode` function is used to create both type of secretrs.

Usage

```
passcode(pass, hash = FALSE)
```

Arguments

`pass`	Password as character or raw, or password hash as raw.
`hash`	If `FALSE`, keep `pass` as plain raw; if `TRUE`, generate its hash; if `NULL` `pass` is assumed to be a proper hash in raw format.

Details

The term "plain text" means here that a password is not encrypted or hashed, but its internal representation is in raw format, however it is possible to revert it back in human readable format. Hashed passwords are obtained with the sodium package hash function and have a fixed length of 32 bytes. Recall that when you set `hash = TRUE`, there is no practical way to go back to the original password.

Value

A secretr of type 'plain' or 'hash'.

(p <- passcode("xyz")) (h <- passcode("xyz", hash = TRUE)) identical(p, passcode(charToRaw("xyz"))) identical(h, passcode(.secretr2raw(h), hash = NULL))

`pwpolicy` *Password policy*

Description

`pwpolicy()` sets a customised password policy, which can be later verified by `pwpolicy.check()`. The policy is a named list whose fields are `pwpolicy` arguments. The default policy requires at least 8 characters from an US keyboard (i.e. in the ASCII range 32:126)), of which at least one should be, respectively, a letter, a digit, a symbol, and you can use spaces.

Usage

```
pwpolicy(
  min = 8,
  min.alpha = 1,
  min.digit = 1,
  min.sym = 1,
  charset = as.raw(32:126),
  debug = FALSE,
  more = NULL,
  pwdesc = NULL
)

pwpolicy.check(pass, policy = pwpolicy())
```

Arguments

`min`	Minimum number of total chars.
`min.alpha`	Minimum number of letters.
`min.digit`	Minimum number of digits.
`min.sym`	Minimum number of symbols.
`charset`	Acceptable input, that is, a vector of raw characters. Current default are the US keyboard printables.
`debug`	If `TRUE`, outputs more info on wrong passwords, but possibly disclosing senstitive data.
`more`	Callback for further checks, receiving the password and returning logical success for an accepted password.
`pwdesc`	The policy description. If you change it, modify this accordingly. There is no text wrapping, thus use "\n".
`pass`	password in character format.
`policy`	policy to check against.

Details

If you set these arguments, don't forget to set the policy description `pwdesc`, unless you set only the `debug` arg. If you don't, you get a warning, even if you don't actually change the defaults. `pwdesc` text is not wrapped, so use enough newlines ("\n") so as to avoid gigantic dialogue windows.

In `charset` the values below 32, e.g. TAB, can be used to merge passwords or other meta-purposes.

`pwpolicy()` output can be use to set `pwset()`'s `policy` argument.

Value

`pwpolicy()`: A named list with the same names and as the function arguments.

`pwpolicy.check()`: The password compliance as a logical.

Examples

```
pwpolicy.check("abcdefgh") # FALSE

## No minimum digits or symbols.
dsc <- "At least: 8 total chars and 1 letter, from a US keyboard (including spaces)."
(pp <- pwpolicy(min.digit = 0, min.sym = 0, pwdesc = dsc))
## Not run:
pwset(policy = pp)
## End(Not run)
pwpolicy.check("abcdefgh", pp) # TRUE

## Extend accetable characters to copyright char:
pwpolicy.check("abcdef1©") # FALSE
dsc <- paste(pwpolicy()$pwdesc, "\nPlus the copyright symbol.")
pp <- pwpolicy(charset = c(as.raw(32:126), charToRaw("©")), pwdesc = dsc) # or
pp <- pwpolicy(charset = c(as.raw(32:126), as.raw(0xc2), as.raw(0xa9)), pwdesc = dsc)
pp
## Not run:
pwset(policy = pp)
## End(Not run)
pwpolicy.check("abcdef1©", pp) # TRUE

## Using the callback to allow only lower case letters:
pp <- pwpolicy(more = function(pass) {
                        l <- tolower(pass) == pass
                        if(!l) message("Use lowercase")
                        l})
## Not run:
pwset(policy = pp)
## End(Not run)
```

`secretr` *Secretr class helpers*

Description

Utilities to type-testing, converting and extracting elements from secretr

Usage

```
## S3 method for class 'secretr'
print(x, ...)

is.secretr(object)

secretr.type(secretr)

secretr.nonce(cipher = NULL)

secretr.hashing(pass)
```

Arguments

x	A secretr.
...	optional arguments to 'print' methods.
object	An object to test for being a secretr.
secretr	A secretr.
cipher	A cipher secretr.
pass	A plain or hash secretr.

Value

`is.secretr` returns `TRUE` if the object is a secretr, `FALSE` otherwise.

`secretr.type` returns one of 'plain', 'hash', 'cipher' depending on the secretr type or an error is raised.

`secretr.nonce` returns the cipher's nonce in raw format.

`secretr.hashing` returns a hash secretr from a non-cipher secretr, by hashing its plain text password for a plain secretr or as-is if it is already a hash secretr.

Examples

```
p <- passcode("hello")
is.secretr(p)

secretr.type(passcode("xyz"))
secretr.type(passcode("xyz", hash = TRUE))
secretr.type(cipher("hello", passcode("xyz")))

## Extract the nonce from a cipher
secretr.nonce(cipher("hello", passcode("xyz")))

## Each cipher as a random nonce making it unique...
c1 <- cipher("hello", passcode("xyz"))
c2 <- cipher("hello", passcode("xyz"))
identical(c1, c2)  # FALSE
## ... unless you use your random nonce
my.nonce <- secretr.nonce()
c1 <- cipher("hello", passcode("xyz"), nonce = my.nonce)
c2 <- cipher("hello", passcode("xyz"), nonce = my.nonce)
identical(c1, c2)  # TRUE
```

```
p <- passcode("hello")
(h <- secretr.hashing(p))
identical(sodium::hash(p), secretr.open(h))
```

`secretr.open` *Disclose the secret in a secretr*

Description

Display the content of a secretr as-is or possibly decoding it in human-readable format

Usage

```
secretr.open(secretr, pass = NULL, human = FALSE)
```

Arguments

`secretr`	The secretr to uncover.
`pass`	If `secretr` is a cipher, this is the plain password secretr to decrypt it.
`human`	If `TRUE`, return the content as a character format.

Details

The print function does not print the content of secretr, it only displays its type. This function can uncover it.

By default this function displays the secret contained in a secretr as-is, in the internal raw format, and this is the only possibility for hash secretrs. For plain secretrs, if `human = TRUE`, the raw secret is converted in its character representation.

For ciphers, a decryption is attempted with 'pass' (which should be a plain password secretr) and, if successful, the decrypted secret is displayed in raw or character format depending on 'human' value.

Value

The secret in raw or character format, depending on `human` arg (but the latter can only be `FALSE` for hashes). For ciphers, the decrypted output or `FALSE` in case of failed decryption. Use `isFALSE`/`isTRUE` to test for un/successful decryptions.

Examples

```
c <- cipher("hello", passcode("xyz"))
secretr.open(c, passcode("xyz"), human = TRUE)
c <- cipher("hello", passcode("xyz", hash = TRUE))
secretr.open(c, passcode("xyz"), human = TRUE)
secretr.open(c, passcode("xyz"))
```

`serialise` *Read and write secrets to files*

Description

`readSecret` and `writeSecret` provide a way to obtain secretr persistence by serialising them to files.

Usage

```
writeSecret(secret, file)

readSecret(file)
```

Arguments

`secret`	A secretr object to serialise to a file
`file`	Path of file to read or write

Value

For `readSecret` the stored secretr. For `writeSecret` invisibly `NULL`.

Examples

```
p <- passcode("xyz")
t <- tempfile()
writeSecret(p, t)
f <- readSecret(t)
identical(f, p)
file.remove(t)

h <- passcode("xyz", hash = TRUE)
t <- tempfile()
writeSecret(h, t)
f <- readSecret(t)
identical(f, h)
file.remove(t)

c <- cipher("hello", passcode("xyz"))
t <- tempfile()
writeSecret(c, t)
f <- readSecret(t)
identical(f, c)
file.remove(t)
```

Index