Package: sodium (via r-universe)

December 14, 2024

Type Package

Title A Modern and Easy-to-Use Crypto Library

Version 1.4.0

Description Bindings to 'libsodium' <https://doc.libsodium.org/>: a
modern, easy-to-use software library for encryption,
decryption, signatures, password hashing and more. Sodium uses
curve25519, a state-of-the-art Diffie-Hellman function by
Daniel Bernstein, which has become very popular after it was
discovered that the NSA had backdoored Dual EC DRBG.

License MIT + file LICENSE

URL https://docs.ropensci.org/sodium/ https://github.com/r-lib/sodium

BugReports https://github.com/r-lib/sodium/issues

SystemRequirements libsodium (>= 1.0.3)

VignetteBuilder knitr

Suggests knitr, rmarkdown

RoxygenNote 7.2.3

Config/pak/sysreqs libsodium-dev

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

RemoteUrl https://github.com/r-lib/sodium

RemoteRef main

RemoteSha 7dba77f125eb26109992d995fccb686a8c1cecc4

Contents

Authenticated encryption 2
Diffie-Hellman 3
Hash functions 4
Key generation 6
Password storage 7
Signatures 8

Simple encryption 9
Sodium utilities 10
Stream ciphers 11
Symmetric encryption 12

Index **14**

Authenticated encryption

Authenticated Encryption

Description

Exchange secure messages through curve25519 authenticated encryption.

Usage

```
auth_encrypt(msg, key, pubkey, nonce = random(24))

auth_decrypt(bin, key, pubkey, nonce = attr(bin, "nonce"))
```

Arguments

`msg`	message to be encrypted
`key`	your own private key
`pubkey`	other person's public key
`nonce`	non-secret unique data to randomize the cipher
`bin`	encrypted ciphertext generated by `secure_send`

Details

Authenticated encryption implements best practices for secure messaging. It requires that both sender and receiver have a keypair and know each other's public key. Each message gets authenticated with the key of the sender and encrypted with the key of the receiver.

Even though public keys are not confidential, you should not exchange them over the same insecure channel you are trying to protect. If the connection is being tampered with, the attacker could simply replace the key with another one to hijack the interaction.

Most people share their public key by posting them on their website or on a public keyserver. Another alternative is having your public key signed by a mutually trusted third party. HTTPS does this using Certificate Authorities.

References

https://libsodium.gitbook.io/doc/public-key_cryptography/authenticated_encryption

Examples

```
# Bob's keypair:
bob_key <- keygen()
bob_pubkey <- pubkey(bob_key)

# Alice's keypair:
alice_key <- keygen()
alice_pubkey <- pubkey(alice_key)

# Bob sends encrypted message for Alice:
msg <- charToRaw("TTIP is evil")
ciphertext <- auth_encrypt(msg, bob_key, alice_pubkey)

# Alice verifies and decrypts with her key
out <- auth_decrypt(ciphertext, alice_key, bob_pubkey)
stopifnot(identical(out, msg))

# Alice sends encrypted message for Bob
msg <- charToRaw("Let's protest")
ciphertext <- auth_encrypt(msg, alice_key, bob_pubkey)

# Bob verifies and decrypts with his key
out <- auth_decrypt(ciphertext, bob_key, alice_pubkey)
stopifnot(identical(out, msg))
```

`Diffie-Hellman` *Diffie-Hellman*

Description

The Diffie-Hellman key exchange method allows two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure channel. This key can then be used to encrypt subsequent communications using a symmetric key cipher.

Usage

```
diffie_hellman(key, pubkey)
```

Arguments

`key`	your private key
`pubkey`	other person's public key

Details

Encryption methods as implemented in data_encrypt require that parties have a shared secret key. But often we wish to establish a secure channel with a party we have no prior relationship with. Diffie-hellman is a method for jointly agreeing on a shared secret without ever exchanging the secret itself. Sodium implements Curve25519, a state-of-the-art Diffie-Hellman function suitable for a wide variety of applications.

The method conists of two steps (see examples). First, both parties generate a random private key and derive the corresponding public key using pubkey. These public keys are not confidential and can be exchanged over an insecure channel. After the public keys are exchanged, both parties will be able to calculate the (same) shared secret by combining his/her own private key with the other person's public key using diffie_hellman.

After the shared secret has been established, the private and public keys are disposed, and parties can start encrypting communications based on the shared secret using e.g. data_encrypt. Because the shared secret cannot be calculated using only the public keys, the process is safe from eavesdroppers.

Value

Returns a shared secret key which can be used in e.g. data_encrypt.

References

`https://doc.libsodium.org/advanced/scalar_multiplication.html`

Examples

```
# Bob generates keypair
bob_key <- keygen()
bob_pubkey <- pubkey(bob_key)

# Alice generates keypair
alice_key <- keygen()
alice_pubkey <- pubkey(alice_key)

# After Bob and Alice exchange pubkey they can both derive the secret
alice_secret <- diffie_hellman(alice_key, bob_pubkey)
bob_secret <- diffie_hellman(bob_key, alice_pubkey)
stopifnot(identical(alice_secret, bob_secret))
```

`Hash functions` *Hash Functions*

Description

Functions to calculate cryptographic hash of a message, with optionally a key for HMAC applications. For storing passwords, use password_store instead.

Usage

```
hash(buf, key = NULL, size = 32)

scrypt(buf, salt = raw(32), size = 32)

argon2(buf, salt = raw(16), size = 32)

shorthash(buf, key)

sha512(buf, key = NULL)

sha256(buf, key = NULL)
```

Arguments

`buf`	data to be hashed
`key`	key for HMAC hashing. Optional, except for in `shorthash`.
`size`	length of the output hash. Must be between 16 and 64 (recommended is 32)
`salt`	non-confidential random data to seed the algorithm

Details

The generic `hash` function is recommended for most applications. It uses dynamic length BLAKE2b where output size can be any value between 16 bytes (128bit) and 64 bytes (512bit).

The scrypt hash function is designed to be CPU and memory expensive to protect against brute force attacks. This algorithm is also used by the password_store function.

The argon2 hash function is also designed to be CPU and memory expensive to protect against brute force attacks. Argon2 is a password-hashing function that summarizes the state of the art in the design of memory-hard functions

The `shorthash` function is a special 8 byte (64 bit) hash based on SipHash-2-4. The output of this function is only 64 bits (8 bytes). It is useful for in e.g. Hash tables, but it should not be considered collision-resistant.

Hash functions can be used for HMAC by specifying a secret `key`. They key size for `shorthash` is 16 bytes, for `sha256` it is 32 bytes and for `sha512` it is 64 bytes. For `hash` the key size can be any value between 16 and 62, recommended is at least 32.

References

`https://libsodium.gitbook.io/doc/hashing/generic_hashing`

Examples

```
# Basic hashing
msg <- serialize(iris, NULL)
hash(msg)
sha256(msg)
sha512(msg)
```

```
scrypt(msg)

# Generate keys from passphrase
passphrase <- charToRaw("This is super secret")
key <- hash(passphrase)
shortkey <- hash(passphrase, size = 16)
longkey <- hash(passphrase, size = 64)

# HMAC (hashing with key)
hash(msg, key = key)
shorthash(msg, shortkey)
sha256(msg, key = key)
sha512(msg, key = longkey)
```

`Key generation` *Keypair Generation*

Description

Functions to generate a random private key and calculate the corresponding curve25519 public key.

Usage

```
keygen(seed = random(32))

pubkey(key)
```

Arguments

`seed`	random data to seed the keygen
`key`	private key for which to calculate the public key

Details

Asymmetric methods rely on public-private keypairs. The private keys are secret and should never be shared with anyone. The public key on the other hand is not confidential and should be shared with the other parties. Public keys are typically published on the users's website or posted in public directories or keyservers.

The two main applications for public key cryptography are encryption and authentication.

In public key encryption, data that is encrypted using a public key can only be decrypted using the corresponding private key. This allows anyone to send somebody a secure message by encrypting it with the receivers public key. The encrypted message will only be readable by the owner of the corresponding private key. Basic encryption is implemented in simple_encrypt.

Authentication works the other way around. In public key authentication, the owner of the private key creates a 'signature' (an authenticated checksum) for a message in a way that allows anyone who knows the user's public key to verify that this message was indeed signed by the owner of the private key.

If both sender and receiver know each other's public key, the two methods can be combined so that each message going back and forth is signed by the sender and encrypted for the receiver. This protects both against eavesdropping and MITM tampering, creating a fully secure channel.

Examples

```
# Create keypair
key <- keygen()
pub <- pubkey(key)

# Basic encryption
msg <- serialize(iris, NULL)
ciphertext <- simple_encrypt(msg, pub)
out <- simple_decrypt(ciphertext, key)
stopifnot(identical(msg, out))
```

`Password storage` *Password Storage*

Description

Wrapper that implements best practices for storing passwords based on scrypt with a random salt.

Usage

```
password_store(password)

password_verify(hash, password)
```

Arguments

`password`	a string of length one with a password
`hash`	a hash string of length one generated by `password_store`

Details

The password_store function returns an ASCII encoded string which contains the result of a memory-hard, CPU-intensive hash function along with the automatically generated salt and other parameters required to verify the password. Use password_verify to verify a password from this string.

References

`https://doc.libsodium.org/password_hashing/`

Examples

```
# Example password
password <- "I like cookies"

# Hash is what you store in the database
hash <- password_store(password)

# To verify the password when the user logs in
stopifnot(password_verify(hash, password))
```

Signatures *Create and Verify Signatures*

Description

Cryptographic signatures can be used to verify the integrity of a message using the author's public key.

Usage

```
sig_sign(msg, key)

sig_verify(msg, sig, pubkey)

sig_keygen(seed = random(32))

sig_pubkey(key)
```

Arguments

`msg`	message to sign
`key`	private key to sign message with
`sig`	a signature generated by `signature_sign`
`pubkey`	a public key of the keypair used by the signature
`seed`	random data to seed the keygen

Details

A signature is an authenticated checksum that can be used to check that a message (any data) was created by a particular author and was not tampered with. The signature is created using a private key and can be verified from the corresponding public key.

Signatures are used when the message itself is not confidential but integrity is important. A common use is for software repositories where maintainers include a signature of the package index. This allows client package managers to verify that the binaries were not modified by intermediate parties in the distribution process.

For confidential data, use authenticated encryption (auth_encrypt) which allows for sending signed and encrypted messages in a single method.

Currently sodium requires a different type of key pairfor signatures (ed25519) than for encryption (curve25519).

References

https://doc.libsodium.org/public-key_cryptography/public-key_signatures.html

Examples

```
# Generate keypair
key <- sig_keygen()
pubkey <- sig_pubkey(key)

# Create signature
msg <- serialize(iris, NULL)
sig <- sig_sign(msg, key)
sig_verify(msg, sig, pubkey)
```

`Simple encryption` *Anonymous Public-key Encryption (Sealed Box)*

Description

Create an encrypted message (sealed box) from a curve25519 public key.

Usage

```
simple_encrypt(msg, pubkey)

simple_decrypt(bin, key)
```

Arguments

`msg`	message to be encrypted
`pubkey`	public key of the receiver
`bin`	encrypted ciphertext
`key`	private key of the receiver

Details

Simple public key encryption allows for sending anonymous encrypted messages to a recipient given its public key. Only the recipient can decrypt these messages, using its private key.

While the recipient can verify the integrity of the message, it cannot verify the identity of the sender. For sending authenticated encrypted messages, use auth_encrypt and auth_decrypt.

References

https://doc.libsodium.org/public-key_cryptography/sealed_boxes.html

Examples

```
# Generate keypair
key <- keygen()
pub <- pubkey(key)

# Encrypt message with pubkey
msg <- serialize(iris, NULL)
ciphertext <- simple_encrypt(msg, pub)

# Decrypt message with private key
out <- simple_decrypt(ciphertext, key)
stopifnot(identical(out, msg))
```

`Sodium utilities` *Sodium Utilities*

Description

The functions `bin2hex` and `hex2bin` convert between binary (raw) vectors and corresponding string in hexadecimal notation. The `random` function generates `n` crypto secure random bytes.

Usage

```
bin2hex(bin)

hex2bin(hex, ignore = ":")

random(n = 1)
```

Arguments

`bin`	raw vector with binary data to convert to hex string
`hex`	a string with hexadecimal characters to parse into a binary (raw) vector.
`ignore`	a string with characters to ignore from `hex`. See example.
`n`	number of random bytes or numbers to generate

Examples

```
# Convert raw to hex string and back
test <- charToRaw("test 123")
x <- bin2hex(test)
y <- hex2bin(x)
stopifnot(identical(test, y))
stopifnot(identical(x, paste(test, collapse = "")))
```

```
# Parse text with characters
x2 <- paste(test, collapse = ":")
y2 <- hex2bin(x2, ignore = ":")
stopifnot(identical(test, y2))
```

`Stream ciphers` *Stream Ciphers*

Description

Generate deterministic streams of random data based off a secret key and random nonce.

Usage

```
chacha20(size, key, nonce)

xchacha20(size, key, nonce)

salsa20(size, key, nonce)

xsalsa20(size, key, nonce)
```

Arguments

`size`	length of cipher stream in bytes
`key`	secret key used by the cipher
`nonce`	non-secret unique data to randomize the cipher

Details

You usually don't need to call these methods directly. For local encryption use data_encrypt. For secure communication use simple_encrypt or auth_encrypt.

Random streams form the basis for most cryptographic methods. Based a shared secret (the key) we generate a predictable random data stream of equal length as the message we need to encrypt. Then we xor the message data with this random stream, which effectively inverts each byte in the message with probabiliy 0.5. The message can be decrypted by re-generating exactly the same random data stream and xor'ing it back. See the examples.

Each stream generator requires a key and a `nonce`. Both are required to re-generate the same stream for decryption. The key forms the shared secret and should only known to the trusted parties. The `nonce` is not secret and should be stored or sent along with the ciphertext. The purpose of the `nonce` is to make a random stream unique to protect gainst re-use attacks. This way you can re-use a your key to encrypt multiple messages, as long as you never re-use the same nonce.

References

https://libsodium.gitbook.io/doc/advanced/stream_ciphers/xsalsa20

Examples

```
# Very basic encryption
myfile <- file.path(R.home(), "COPYING")
message <- readBin(myfile, raw(), file.info(myfile)$size)
passwd <- charToRaw("My secret passphrase")

# Encrypt:
key <- hash(passwd)
nonce8 <- random(8)
stream <- chacha20(length(message), key, nonce8)
ciphertext <- base::xor(stream, message)

# Decrypt:
stream <- chacha20(length(ciphertext), key, nonce8)
out <- base::xor(ciphertext, stream)
stopifnot(identical(out, message))

# Other stream ciphers
stream <- salsa20(10000, key, nonce8)
stream <- xsalsa20(10000, key, random(24))
stream <- xchacha20(10000, key, random(24))
```

`Symmetric encryption` *Symmetric Encryption and Tagging*

Description

Encryption with authentication using a 256 bit shared secret. Mainly useful for encrypting local data. For secure communication use public-key encryption (simple_encrypt and auth_encrypt).

Usage

```
data_encrypt(msg, key, nonce = random(24))

data_decrypt(bin, key, nonce = attr(bin, "nonce"))

data_tag(msg, key)
```

Arguments

`msg`	message to be encrypted
`key`	shared secret key used for both encryption and decryption
`nonce`	non-secret unique data to randomize the cipher
`bin`	encrypted ciphertext

Details

Symmetric encryption uses a secret key to encode and decode a message. This can be used to encrypt local data on disk, or as a building block for more complex methods.

Because the same `secret` is used for both encryption and decryption, symmetric encryption by itself is impractical for communication. For exchanging secure messages with other parties, use assymetric (public-key) methods (see simple_encrypt or auth_encrypt).

The `nonce` is not confidential but required for decryption, and should be stored or sent along with the ciphertext. The purpose of the `nonce` is to randomize the cipher to protect gainst re-use attacks. This way you can use one and the same secret for encrypting multiple messages.

The data_tag function generates an authenticated hash that can be stored alongside the data to be able to verify the integrity of the data later on. For public key signatures see `sig_sign` instead.

References

https://libsodium.gitbook.io/doc/public-key_cryptography/authenticated_encryption

Examples

```
# 256-bit key
key <- sha256(charToRaw("This is a secret passphrase"))
msg <- serialize(iris, NULL)

# Encrypts with random nonce
cipher <- data_encrypt(msg, key)
orig <- data_decrypt(cipher, key)
stopifnot(identical(msg, orig))

# Tag the message with your key (HMAC)
tag <- data_tag(msg, key)
```

Index

∗ **sodium**
    Sodium utilities, 10

argon2, *5*
argon2 *(Hash functions)*, 4
auth_decrypt, *9*
auth_decrypt *(Authenticated encryption)*, 2
auth_encrypt, *9*, *11–13*
auth_encrypt *(Authenticated encryption)*, 2
Authenticated encryption, 2

bin2hex *(Sodium utilities)*, 10

chacha20 *(Stream ciphers)*, 11

data_decrypt *(Symmetric encryption)*, 12
data_encrypt, *3*, *4*, *11*
data_encrypt *(Symmetric encryption)*, 12
data_tag, *13*
data_tag *(Symmetric encryption)*, 12
diffie *(Diffie-Hellman)*, 3
Diffie-Hellman, 3
diffie_hellman, *4*
diffie_hellman *(Diffie-Hellman)*, 3

hash *(Hash functions)*, 4
Hash functions, 4
hashing *(Hash functions)*, 4
helpers *(Sodium utilities)*, 10
hex2bin *(Sodium utilities)*, 10

Key generation, 6
keygen *(Key generation)*, 6

password *(Password storage)*, 7
Password storage, 7
password_store, *4*, *5*, *7*
password_store *(Password storage)*, 7
password_verify, *7*
password_verify *(Password storage)*, 7
pubkey, *4*
pubkey *(Key generation)*, 6

random *(Sodium utilities)*, 10

salsa20 *(Stream ciphers)*, 11
scrypt, *5*
scrypt *(Hash functions)*, 4
sha256 *(Hash functions)*, 4
sha512 *(Hash functions)*, 4
shorthash *(Hash functions)*, 4
sig *(Signatures)*, 8
sig_keygen *(Signatures)*, 8
sig_pubkey *(Signatures)*, 8
sig_sign *(Signatures)*, 8
sig_verify *(Signatures)*, 8
Signatures, 8
Simple encryption, 9
simple_decrypt *(Simple encryption)*, 9
simple_encrypt, *6*, *11–13*
simple_encrypt *(Simple encryption)*, 9
Sodium utilities, 10
stream *(Stream ciphers)*, 11
Stream ciphers, 11
Symmetric encryption, 12

xchacha20 *(Stream ciphers)*, 11
xor, *11*
xsalsa20 *(Stream ciphers)*, 11