Package: zucrypt (via r-universe)

October 24, 2026

Title Cryptography Without System Dependencies

Version 0.1.0

Description Cryptographic functions over raw vectors, including message digests of in-memory data, files and connections, keyed-hash message authentication codes (HMAC), Advanced Encryption Standard (AES) encryption, constant-time comparison and hexadecimal encoding, backed by a bundled and pinned copy of the 'Mbed TLS' project's cryptography library, 'TF-PSA-Crypto' (<https://github.com/Mbed-TLS/TF-PSA-Crypto>). No system cryptographic library, 'Java' or 'Python' is required at install or run time. Algorithm names are matched exactly, and failures are reported as structured conditions. The same primitives, and secure erasure of native buffers, are published to other packages as a static archive and as a registered C interface, so that a package can call them from C without going through R.

License MIT + file LICENSE

Copyright See inst/COPYRIGHTS and tools/vendor/manifest.tsv.

URL https://github.com/pedrobtz/zucrypt, https://pedrobtz.github.io/zucrypt/

BugReports https://github.com/pedrobtz/zucrypt/issues

Depends R (>= 4.1)

Imports utils

Suggests openssl, testthat (>= 3.0.0)

SystemRequirements A C99 compiler. No system cryptographic library is used, no CMake, Python or Perl is required, and nothing is downloaded or generated during installation: the backend is built from vendored sources, with an explicit object list in portable make.

Config/Needs/website knitr, rmarkdown

Config/testthat/edition 3

Config/testthat/parallel true

Encoding UTF-8

Roxygen list(markdown = TRUE)

Config/roxygen2/version 8.1.1

Config/pak/sysreqs cmake perl python3

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

Date/Publication 2026-10-16 06:59:14 UTC

RemoteUrl https://github.com/pedrobtz/zucrypt

RemoteRef HEAD

RemoteSha 9ae5f73f7a7dcb3930b3c9406362bd70fcba8eff

Contents

crypt_aes_cbc_nopad 2
crypt_equal 4
crypt_hash 5
crypt_hex 6
crypt_hmac 7
crypt_info 8
zucrypt_c_api 9

Index **12**

`crypt_aes_cbc_nopad` *AES-CBC Without Padding or Authentication*

Description

These functions provide confidentiality only. They do not authenticate anything. Ciphertext produced here can be altered by anyone who can reach it, and decryption will return the altered plaintext without complaint. Detecting that is your job, or use an authenticated format. If you must authenticate CBC yourself, the minimum is encrypt-then-MAC, done in full:

- compute `crypt_hmac()` over the IV **and** the ciphertext, `c(iv, ciphertext)`. A MAC over the ciphertext alone lets anyone change the IV, and with it the first plaintext block, without the tag noticing;
- use a MAC key independent of the encryption key;
- verify the tag with `crypt_equal()` **before** decrypting, and do not decrypt at all if it fails.

They also add and strip no padding. The input length must already be a multiple of 16 bytes.

These are interoperability tools, for reading and writing formats that specify unauthenticated CBC – the Office encryption profiles are the reason they exist. They are not a general-purpose way to encrypt something of your own; that needs an authenticated construction, a nonce policy and a key derivation step, none of which this package provides.

Usage

```
crypt_aes_cbc_encrypt_nopad(data, key, iv)

crypt_aes_cbc_decrypt_nopad(data, key, iv)
```

Arguments

`data`	A raw vector whose length is a multiple of 16 bytes.
`key`	A raw vector of exactly 16, 24 or 32 bytes, selecting AES-128, AES-192 or AES-256. This is a key, not a password.
`iv`	A raw vector of exactly 16 bytes, the initialisation vector. For encryption it must be unpredictable and must not be reused with the same key; for decryption it is whatever the format says it is.

Value

A raw vector the same length as `data`. `data` itself is never modified.

Side channels

AES runs on the CPU's AES instructions where it has them – AES-NI on x86, the Cryptography Extension on 64-bit Arm – chosen at run time, and falls back to a software implementation where it does not. `crypt_info()` says which one this machine uses, in `build_flags$aes_implementation`.

The software fallback reads lookup tables at addresses that depend on the key and the data. The backend's own security policy warns that an attacker who can observe cache timing – another process on the same machine, or, with enough precision, the network – can recover the key from that. If `aes_implementation` is `"software"` and such an attacker is in your threat model, do not use these functions for secrets that matter. Every implementation produces the same bytes; only this exposure differs.

Segmented formats

Each call starts from `iv` and treats `data` as one continuous CBC stream. A format that restarts the chaining at segment boundaries – which is what the Office Agile profiles do – is expressed as one call per segment, each with that segment's own IV. The C interface exposes the running chaining state directly for consumers that need finer control; see zucrypt_c_api.

Examples

```
key <- as.raw(rep(0x2b, 16))
iv <- as.raw(seq.int(0, 15))
plaintext <- charToRaw("sixteen bytes!! and sixteen more")

ciphertext <- crypt_aes_cbc_encrypt_nopad(plaintext, key, iv)
ciphertext

identical(crypt_aes_cbc_decrypt_nopad(ciphertext, key, iv), plaintext)

# Authentication is separate, and is not optional. The tag covers the IV
```

```
# as well as the ciphertext, under a key of its own.
mac_key <- as.raw(rep(0x5c, 32))
tag <- crypt_hmac(c(iv, ciphertext), mac_key)

# The receiver checks the tag first, and decrypts only if it matches.
crypt_equal(tag, crypt_hmac(c(iv, ciphertext), mac_key))

# A changed IV would alter the first plaintext block. Because the tag
# covers the IV, the change is caught before anything is decrypted.
tampered_iv <- iv
tampered_iv[1] <- xor(tampered_iv[1], as.raw(1))
crypt_equal(tag, crypt_hmac(c(tampered_iv, ciphertext), mac_key))
```

`crypt_equal` *Compare Two Raw Vectors Without Leaking Timing Information*

Description

Compares in time that does not depend on the contents of the buffers. Use it whenever one side is a secret: an authentication tag, a password verifier, a derived key.

Usage

```
crypt_equal(x, y)
```

Arguments

`x, y`	Raw vectors.

Value

`TRUE` if the vectors have the same length and the same contents, otherwise `FALSE`.

Length is not hidden

Vectors of different lengths return `FALSE` immediately, without comparing anything. Hiding a length difference is not something a comparison function can do, and pretending otherwise would be worse than saying so: if the length of your secret is itself secret, compare digests of the values rather than the values.

Examples

```
a <- crypt_hash(charToRaw("message"))
b <- crypt_hash(charToRaw("message"))
crypt_equal(a, b)

crypt_equal(a, crypt_hash(charToRaw("different")))
```

```
# Different lengths are simply not equal.
crypt_equal(a, raw(0))
```

---

crypt_hash *Compute a Message Digest*

Description

Hashes a raw vector, or everything a connection yields, with one of the digest algorithms this build provides.

Usage

```
crypt_hash(data, algorithm = "sha256")
```

Arguments

data	A raw vector, or a connection to read to its end (see "Files and connections" below). Character input is never accepted: this package does not guess a text encoding and never treats a string as a file name. Convert deliberately with charToRaw() or serialize(), or pass file(path) to hash a file.
algorithm	A single algorithm name: one of "sha1", "sha256", "sha384" or "sha512", and whatever crypt_info()$algorithms reports is what this build actually accepts. Matched exactly – there is no partial matching and no fallback, so a typo is an error rather than a quietly different algorithm.

Value

A raw vector of the digest, 20 bytes for "sha1" and 32, 48 or 64 for the SHA-2 family. Hex formatting is an explicit step at the call site, with crypt_hex(); see the examples.

Choosing an algorithm

"sha256" is the default and the right answer unless something external requires otherwise. "sha1" is available because document formats specify it – notably the Office encryption profiles this package exists to support – and it is not collision resistant. Do not select it for anything new.

Files and connections

A connection is read in 1 MiB chunks, so a file of any size is hashed without being held in memory, and the call can be interrupted. Pass an unopened connection, such as file(path), and it is opened in binary mode ("rb") and closed again afterwards; pass one you have already opened and it is read from where it stands to its end, and left open.

An open connection must be in binary mode. A text-mode connection can re-encode or translate line endings before the bytes arrive, which would hash something other than the file, so it is refused with zucrypt_invalid_argument. A connection that cannot be opened signals zucrypt_connection_error,

as does a non-blocking connection that runs out of data before its end, which is refused rather than hashed short. An error while reading is R's own.

What is hashed is what the connection yields: `gzfile()` gives the decompressed bytes, `file()` the bytes on disk, and `url()` the body as it downloads, so a remote file is hashed without being saved first.

See Also

`crypt_hmac()` for a keyed digest, `crypt_equal()` for comparing digests without leaking timing information, `crypt_hex()` for hex.

Examples

```
digest <- crypt_hash(charToRaw("the quick brown fox"))
digest

# Hex, when you need it, is an explicit conversion.
crypt_hex(digest)

# The digest of empty input is well defined.
crypt_hash(raw(0))

crypt_hash(charToRaw("abc"), "sha512")

# A file, read in chunks rather than all at once.
path <- tempfile()
writeBin(charToRaw("abc"), path)
identical(crypt_hash(file(path)), crypt_hash(charToRaw("abc")))
unlink(path)
```

---

crypt_hex *Encode Bytes as Hexadecimal*

Description

Converts a raw vector to one lower-case hexadecimal string, two digits per byte, with no separators and no prefix: the form `sha256sum` prints and most formats store. The conversion runs in C, so for a short digest it costs far less than the hash itself.

Usage

```
crypt_hex(x)
```

Arguments

`x`	A raw vector, at most 2^30 - 1 bytes long (a longer result would exceed R's limit on the length of a single string).

Details

Hex is an explicit step at the call site rather than an option on `crypt_hash()` and `crypt_hmac()`, so the digest functions return one type and a caller who wants text says so. It works on any bytes: a digest, a MAC, a key you need to print for a test.

Value

A character vector of length one, `2 * length(x)` characters long. `crypt_hex(raw(0))` is `""`.

See Also

`crypt_hash()` and `crypt_hmac()`, whose results this is usually applied to.

Examples

```
crypt_hex(crypt_hash(charToRaw("abc")))

crypt_hex(as.raw(c(0x00, 0x0f, 0xa0, 0xff)))

crypt_hex(raw(0))
```

---

crypt_hmac *Compute an HMAC*

Description

Computes a keyed message authentication code (RFC 2104) over a raw vector, or over everything a connection yields.

Usage

```
crypt_hmac(data, key, algorithm = "sha256")
```

Arguments

`data`	A raw vector, or a connection to read to its end (see "Files and connections" below). Character input is never accepted: this package does not guess a text encoding and never treats a string as a file name. Convert deliberately with `charToRaw()` or `serialize()`, or pass `file(path)` to hash a file.
`key`	A raw vector holding the key. Any length is accepted, including zero, as the standard specifies. This is a key, not a password: deriving a key from a password is the caller's responsibility and is not something this package does for you.
`algorithm`	A single algorithm name: one of `"sha1"`, `"sha256"`, `"sha384"` or `"sha512"`, and whatever `crypt_info()$algorithms` reports is what this build actually accepts. Matched exactly – there is no partial matching and no fallback, so a typo is an error rather than a quietly different algorithm.

Value

A raw vector the length of the underlying digest.

Files and connections

`data` may be a connection, read in chunks exactly as for `crypt_hash()`; see "Files and connections" there. The key is always a raw vector.

Verifying a MAC

Compare with `crypt_equal()`, never with `identical()` or `==`. An ordinary comparison stops at the first differing byte, and the time it takes is therefore a measurement of how much of the expected value an attacker has guessed.

Examples

```
key <- as.raw(rep(0x0b, 20))
tag <- crypt_hmac(charToRaw("Hi There"), key)
tag

# Verification, done properly.
crypt_equal(tag, crypt_hmac(charToRaw("Hi There"), key))
```

`crypt_info` *Report What This Build of 'zucrypt' Contains*

Description

Describes the installed package: its version, the C ABI it publishes, the algorithms it provides, and the vendored cryptographic backend it was compiled against.

Usage

```
crypt_info()
```

Details

Everything here is read from the compiled library, never from `tools/vendor/manifest.tsv`. The manifest is maintainer tooling and is not installed, so it could only describe the tree a build was made *from* – which is the one thing a user holding a binary package cannot check.

Value

A list with elements:

`version` the `zucrypt` package version, as a `package_version`.

`abi_version` the C ABI version published to `LinkingTo` consumers, as an integer: `1`. The archive's ABI is frozen as ABI 1 from 0.1.0; see zucrypt_c_api.

`algorithms` the digest algorithms this build provides, which is exactly the set `crypt_hash()` and `crypt_hmac()` accept.

`vendored` a data frame with one row per vendored source, giving its `source` name and the `version` the compiled library reports.

`build_flags` a named list: `random_backend`, the operating-system random source compiled in; `aes_implementation`, which AES this machine runs – `"aesni"` or `"aesce"` on a CPU with AES instructions, `"software"` otherwise, decided at run time; and `hardware_acceleration`, `TRUE` when that is a hardware path. See the "Side channels" section of crypt_aes_cbc_nopad for why it matters.

Examples

```
info <- crypt_info()
info$version
info$algorithms
info$vendored
```

---

zucrypt_c_api *Using 'zucrypt' from C*

Description

zucrypt publishes its primitives to other packages twice, because the `zu*` family consumes siblings in two different ways. Both deliver the same library; they differ in what the consumer has to carry, and in how stable they are: the static archive is the primary shape, and the registered table is experimental. See the Stability section.

Shape one, the registered function table (experimental)

For a package that can carry an `Imports:`. Nothing is linked: R's registered C-callable mechanism hands over a single versioned table of function pointers.

```
# DESCRIPTION
Imports:   zucrypt
LinkingTo: zucrypt

# NAMESPACE -- a real import, not just the Imports: field
importFrom(zucrypt, crypt_info)
```

```
#include <zucrypt-r.h>

const zucrypt_api_v1 *api = zucrypt_api();
if (api == NULL) {
    /* zucrypt does not implement the ABI version this was built against */
}
uint8_t out[ZUC_MAX_DIGEST_SIZE];
size_t n;
zuc_status st = api->hash_compute(ZUC_ALG_SHA256, data, len,
                                  out, sizeof out, &n);
```

`LinkingTo:` **alone is not enough, and neither is** `Imports:` **alone.** `LinkingTo:` supplies the headers and links nothing. `Imports:` without a real `importFrom()` in `NAMESPACE` does not load zucrypt's namespace, and an unloaded namespace has no registered callable to find. `zucrypt_api()` resolves lazily on first use rather than from your `R_init_`, which sidesteps the remaining ordering problem, but the import directive is still required.

`NULL` **means a version mismatch, and nothing else.** When zucrypt is not installed or not loaded, `R_GetCCallable()` does not return `NULL`: it raises an R error, which longjmps out of your C code. Resolve the table before you acquire anything such a jump would strand.

A table field added after the one you built against may be missing from an older zucrypt: test with `ZUCRYPT_API_HAS(api, field)` before calling it.

Shape two, the static archive (primary)

For a package that cannot carry an `Imports:` – `zuxlsx` links its siblings statically and has no runtime dependency on them by design.

```
# DESCRIPTION
LinkingTo: zucrypt
```

A `configure` and `configure.win` resolve `system.file("lib", .Platform$r_arch, package = "zucrypt")`, falling back to `system.file("lib", package = "zucrypt")`, and substitute the result into `src/Makevars.in` as a **single-quoted** `PKG_LIBS` entry – a library path containing a space, which is the norm on Windows, otherwise reaches the linker as two arguments. Then include `<zucrypt.h>` and call the functions directly.

The archive is installed to `lib/` plus the R sub-architecture – plain `lib/` on Linux and macOS, `lib/x64/` on Windows – which is why the lookup above tries the sub-architecture first. The licence of the backend compiled into it is installed as `licenses/tf-psa-crypto-LICENSE`; a binary of your package redistributes that code, so ship the notice too.

This shape owns the backend's lifetime: call `zuc_init()` before anything else and `zuc_shutdown()` when finished. It is reference counted, so nesting is safe. A call outside that window returns `ZUC_ERR_NOT_READY`.

Two consequences worth stating plainly. Your shared object contains its own copy of the backend, kept private by hidden visibility. And upgrading zucrypt does nothing for you until your package is reinstalled – including a security fix – so report the version you actually linked, by calling `zuc_get_info()` rather than reading a macro.

What the interface promises

- Every function returns a `zuc_status`. None raises an R error, allocates an R object, or calls back into R. No status is negative, so `if (st)` means "not success".
- An object the library allocated is destroyed only by a library function. No buffer you pass is retained after the call returns.
- `inst/include/zucrypt.h` compiles standalone as C99 against `<stddef.h>` and `<stdint.h>`, and names no backend type. You never have to reproduce this package's build configuration.
- Fields are appended to the table, never removed or reordered. A layout change to any other type in the header renames the registered callable instead, so an old consumer fails at `R_GetCCallable()` rather than reading a structure that has moved.
- Each AES and HMAC handle holds one key in a key store that grows on demand, so the number of live handles is bounded only by memory.
- Main thread only, in this version.

Stability

The three surfaces are at different stages:

- **The seven `crypt_*()` R functions** are stable.
- **The static archive** – `zucrypt.h`, `libzucrypt.a` and where it is installed – is *frozen* as ABI 1 from 0.1.0. The freeze followed its first consumer: `zuxlsx`'s decryption of password-protected workbooks was written against it, linked it on Linux, macOS and Windows, and decrypted a real encrypted workbook with it.
- **The registered table** (`zucrypt-r.h`) is *experimental*: no package uses it yet. It may change in any release until one does, again with every change recorded in `NEWS.md`.

`ZUCRYPT_ABI_VERSION` is `1`. Within major version 1: functions and table fields may be added; nothing is removed, reordered or given a new meaning; enumerator values are permanent, so an algorithm compiled out of a build keeps its number and reports itself unavailable; and a `ZUC_*_REQUIRED_SIZE` macro never grows, so a consumer built against an older header keeps working without being rebuilt.

A layout change to a type that `struct_size` cannot see renames the registered callable instead, so an old consumer fails at `R_GetCCallable()` rather than reading a structure that has moved.

Not promised: the numeric value of a backend status behind `ZUC_ERR_BACKEND`, the contents of an opaque handle, and thread safety beyond the main thread.

See Also

`crypt_info()`, which reports the ABI version, the backend and the algorithms this build provides.

Index

charToRaw(), *5*, *7*
crypt_aes_cbc_decrypt_nopad
(crypt_aes_cbc_nopad), 2
crypt_aes_cbc_encrypt_nopad
(crypt_aes_cbc_nopad), 2
crypt_aes_cbc_nopad, 2, *9*
crypt_equal, 4
crypt_equal(), *2*, *6*, *8*
crypt_hash, 5
crypt_hash(), *7*, *8*
crypt_hex, 6
crypt_hex(), *5*, *6*
crypt_hmac, 7
crypt_hmac(), *2*, *6*, *7*
crypt_info, 8
crypt_info(), *11*

serialize(), *5*, *7*

zucrypt-c-api (zucrypt_c_api), 9
zucrypt_c_api, *3*, 9, *9*