



Rocky Enterprise Linux 9.2 Manual Pages on command 'crypt_r.3'

\$ man crypt_r.3

CRYPT(3) BSD Library Functions Manual CRYPT(3)

NAME

crypt, crypt_r, crypt_rn, crypt_ra ? passphrase hashing

LIBRARY

Crypt Library (libcrypt, -lcrypt)

SYNOPSIS

```
#include <crypt.h>
```

```
char *
```

```
crypt(const char *phrase, const char *setting);
```

```
char *
```

```
crypt_r(const char *phrase, const char *setting,
```

```
struct crypt_data *data);
```

```
char *
```

```
crypt_rn(const char *phrase, const char *setting,
```

```
struct crypt_data *data, int size);
```

```
char *
```

```
crypt_ra(const char *phrase, const char *setting, void **data,
```

```
int *size);
```

DESCRIPTION

The `crypt`, `crypt_r`, `crypt_rn`, and `crypt_ra` functions irreversibly hash a phrase for storage in the system password database (`shadow(5)`) using a cryptographic hashing method. The result of this operation is called a hashed passphrase or just a hash. Hashing methods are described in `crypt(5)`.

setting controls which hashing method to use, and also supplies various parameters to the chosen method, most importantly a random salt which ensures that no two stored hashes are the same, even if the phrase strings are the same.

The data argument to `crypt_r` is a structure of type `struct crypt_data`.

It has at least these fields:

```
struct crypt_data {
    char output[CRYPT_OUTPUT_SIZE];
    char setting[CRYPT_OUTPUT_SIZE];
    char phrase[CRYPT_MAX_PASSPHRASE_SIZE];
    char initialized;
};
```

Upon a successful return from `crypt_r`, the hashed passphrase will be stored in `output`. Applications are encouraged, but not required, to use the `phrase` and `setting` fields to store the strings that they will pass as phrase and setting to `crypt_r`. This will make it easier to erase all sensitive data after it is no longer needed.

The `initialized` field must be set to zero before the first time a `struct crypt_data` object is first used in a call to `crypt_r()`. We recommend zeroing the entire object, not just `initialized` and not just the documented fields, before the first use. (Of course, do this before storing anything in `setting` and `phrase`.)

The data argument to `crypt_rn` should also point to a `struct crypt_data` object, and `size` should be the size of that object, cast to `int`. When used with `crypt_rn`, the entire data object (except for the `phrase` and `setting` fields) must be zeroed before its first use; this is not just a recommendation, as it is for `crypt_r`. Otherwise, the fields of the ob-

ject have the same uses that they do for `crypt_r`.

On the first call to `crypt_ra`, `data` should be the address of a `void *` variable set to `NULL`, and `size` should be the address of an `int` variable set to zero. `crypt_ra` will allocate and initialize a struct `crypt_data` object, using `malloc(3)`, and write its address and size into the variables pointed to by `data` and `size`. These can be reused in subsequent calls. After the application is done hashing passphrases, it should deallocate the struct `crypt_data` object using `free(3)`.

RETURN VALUES

Upon successful completion, `crypt`, `crypt_r`, `crypt_rn`, and `crypt_ra` return a pointer to a string which encodes both the hashed passphrase, and the settings that were used to encode it. This string is directly usable as setting in other calls to `crypt`, `crypt_r`, `crypt_rn`, and `crypt_ra`, and as prefix in calls to `crypt_gensalt`, `crypt_gensalt_rn`, and `crypt_gensalt_ra`. It will be entirely printable ASCII, and will not contain whitespace or the characters `?:`, `;;`, `*?`, `!?`, or `\?`. See `crypt(5)` for more detail on the format of hashed passphrases.

`crypt` places its result in a static storage area, which will be overwritten by subsequent calls to `crypt`. It is not safe to call `crypt` from multiple threads simultaneously.

`crypt_r`, `crypt_rn`, and `crypt_ra` place their result in the output field of their data argument. It is safe to call them from multiple threads simultaneously, as long as a separate data object is used for each thread.

Upon error, `crypt_r`, `crypt_rn`, and `crypt_ra` write an invalid hashed passphrase to the output field of their data argument, and `crypt` writes an invalid hash to its static storage area. This string will be shorter than 13 characters, will begin with a `*?`, and will not compare equal to setting.

Upon error, `crypt_rn` and `crypt_ra` return a null pointer. `crypt_r` and `crypt` may also return a null pointer, or they may return a pointer to the invalid hash, depending on how `libcrypt` was configured. (The option to return the invalid hash is for compatibility with old applications that assume that `crypt` cannot return a null pointer. See **PORTABILITY NOTES**

below.)

All four functions set `errno` when they fail.

ERRORS

EINVAL setting is invalid, or requests a hashing method that is not supported.

ERANGE phrase is too long (more than `CRYPT_MAX_PASSPHRASE_SIZE` characters; some hashing methods may have lower limits).

`crypt_rn` only: size is too small for the hashing method requested by setting.

ENOMEM Failed to allocate internal scratch memory.

`crypt_ra` only: failed to allocate memory for data.

ENOSYS or EOPNOTSUPP

Hashing passphrases is not supported at all on this installation, or the hashing method requested by setting is not supported. These error codes are not used by this version of `libcrypt`, but may be encountered on other systems.

PORTABILITY NOTES

`crypt` is included in POSIX, but `crypt_r`, `crypt_rn`, and `crypt_ra` are not part of any standard.

POSIX does not specify any hashing methods, and does not require hashed passphrases to be portable between systems. In practice, hashed passphrases are portable as long as both systems support the hashing method that was used. However, the set of supported hashing methods varies considerably from system to system.

The behavior of `crypt` on errors isn't well standardized. Some implementations simply can't fail (except by crashing the program), others return a null pointer or a fixed string. Most implementations don't set `errno`, but some do. POSIX specifies returning a null pointer and setting `errno`, but it defines only one possible error, `ENOSYS`, in the case where `crypt` is not supported at all. Some older applications are not prepared to handle null pointers returned by `crypt`. The behavior described above for

this implementation, setting `errno` and returning an invalid hashed passphrase different from setting, is chosen to make these applications fail closed when an error occurs.

Due to historical restrictions on the export of cryptographic software from the USA, `crypt` is an optional POSIX component. Applications should therefore be prepared for `crypt` not to be available, or to always fail (setting `errno` to `ENOSYS`) at runtime.

POSIX specifies that `crypt` is declared in `<unistd.h>`, but only if the macro `_XOPEN_CRYPT` is defined and has a value greater than or equal to zero. Since `libcrypt` does not provide `<unistd.h>`, it declares `crypt`, `crypt_r`, `crypt_rn`, and `crypt_ra` in `<crypt.h>` instead.

On a minority of systems (notably recent versions of Solaris), `crypt` uses a thread-specific static storage buffer, which makes it safe to call from multiple threads simultaneously, but does not prevent each call within a thread from overwriting the results of the previous one.

BUGS

Some implementations of `crypt`, upon error, return an invalid hash that is stored in a read-only location or only initialized once, which means that it is only safe to erase the buffer pointed to by the `crypt` return value if an error did not occur.

`struct crypt_data` may be quite large (32kB in this implementation of `libcrypt`; over 128kB in some other implementations). This is large enough that it may be unwise to allocate it on the stack.

Some recently designed hashing methods need even more scratch memory, but the `crypt_r` interface makes it impossible to change the size of `struct crypt_data` without breaking binary compatibility. The `crypt_rn` interface could accommodate larger allocations for specific hashing methods, but the caller of `crypt_rn` has no way of knowing how much memory to allocate. `crypt_ra` does the allocation itself, but can only make a single call to `malloc(3)`.

ATTRIBUTES

For an explanation of the terms used in this section, see `attributes(7)`.

??

Interface	Attribute	Value
crypt	Thread safety	MT-Unsafe race:crypt
crypt_r, crypt_rn	Thread safety	MT-Safe
crypt_ra		

HISTORY

A rotor-based crypt function appeared in Version 6 AT&T UNIX. The traditional DES-based crypt first appeared in Version 7 AT&T UNIX. crypt_r originates with the GNU C Library. There's also a crypt_r function on HP-UX and MKS Toolkit, but the prototypes and semantics differ. crypt_rn and crypt_ra originate with the Openwall project.

SEE ALSO

crypt_gensalt(3), getpass(3), getpwent(3), shadow(3), login(1), passwd(1), crypt(5), passwd(5), shadow(5), pam(8)

Openwall Project October 11, 2017 Openwall Project