NAME
crypt, crypt_r, crypt_rn, crypt_ra, crypt_gensalt, crypt_gensalt_rn, crypt_gensalt_ra – password hashing

SYNOPSIS
#define _XOPEN_SOURCE
#include <unistd.h>

char *crypt(const char *key, const char *setting);

#define _GNU_SOURCE
#include <crypt.h>

char *crypt_r(const char *key, const char *setting, struct crypt_data *data);

#define _OW_SOURCE
#include <crypt.h>

char *crypt_rn(const char *key, const char *setting, void *data, int size);
char *crypt_ra(const char *key, const char *setting, void **data, int *size);
char *crypt_gensalt(const char *prefix, unsigned long count, const char *input, int size);
char *crypt_gensalt_rn(const char *prefix, unsigned long count, const char *input, int size, char *output);
char *crypt_gensalt_ra(const char *prefix, unsigned long count, const char *input, int size);

DESCRIPTION
The crypt, crypt_r, crypt_rn, and crypt_ra functions calculate a cryptographic hash function of key with one of a number of supported methods as requested with setting, which is also used to pass a salt and possibly other parameters to the chosen method. The hashing methods are explained below.

Unlike crypt, the functions crypt_r, crypt_rn and crypt_ra are reentrant. They place their result and possibly their private data in a data area of size bytes as passed to them by an application and/or in memory they allocate dynamically. Some hashing algorithms may use the data area to cache precomputed intermediate values across calls. Thus, applications must properly initialize the data area before its first use. crypt_r requires that only data->initialized be reset to zero; crypt_rn and crypt_ra require that either the entire data area is zeroed or, in the case of crypt_ra, *data is NULL. When called with a NULL *data or insufficient *size for the requested hashing algorithm, crypt_ra uses realloc(3) to allocate the required amount of memory dynamically. Thus, crypt_ra has the additional requirement that *data, when non-NULL, must point to an area allocated either with a previous call to crypt_ra or with a malloc(3) family call. The memory allocated by crypt_ra should be freed with free(3).

The crypt_gensalt, crypt_gensalt_rn, and crypt_gensalt_ra functions compile a string for use as setting – with the given prefix (used to choose a hashing method), the iteration count (if supported by the chosen method) and up to size cryptographically random input bytes for use as the actual salt. If count is 0, a low default will be picked. The random bytes may be obtained from /dev/urandom. Unlike crypt_gensalt, the functions crypt_gensalt_rn and crypt_gensalt_ra are reentrant. crypt_gensalt_rn places its result in the output buffer of output_size bytes. crypt_gensalt_ra allocates memory for its result dynamically. The memory should be freed with free(3).

RETURN VALUE
Upon successful completion, the functions crypt, crypt_r, crypt_rn, and crypt_ra return a pointer to a string containing the setting that was actually used and a printable encoding of the hash function value. The entire string is directly usable as setting with other calls to crypt, crypt_r, crypt_rn, and crypt_ra and as prefix with calls to crypt_gensalt, crypt_gensalt_rn, and crypt_gensalt_ra.

The behavior of crypt on errors isn’t well standardized. Some implementations simply can’t fail (unless the process dies, in which case they obviously can’t return), others return NULL or a fixed string. Most implementations don’t set errno, but some do. SUSv2 specifies only returning NULL and setting errno as
a valid behavior, and defines only one possible error (ENOSYS, "The functionality is not supported on this implementation.") Unfortunately, most existing applications aren’t prepared to handle NULL returns from crypt. The description below corresponds to this implementation of crypt and crypt_r only, and to crypt_rn and crypt_ra. The behavior may change to match standards, other implementations or existing applications.

crypt and crypt_r may only fail (and return) when passed an invalid or unsupported setting, in which case they return a pointer to a magic string that is shorter than 13 characters and is guaranteed to differ from setting. This behavior is safe for older applications which assume that crypt can’t fail, when both setting new passwords and authenticating against existing password hashes. crypt_rn and crypt_ra return NULL to indicate failure. All four functions set errno when they fail.

The functions crypt_gensalt, crypt_gensalt_rn, and crypt_gensalt_ra return a pointer to the compiled string for setting, or NULL on error in which case errno is set.

**ERRORS**

**EINVAL**
crypt, crypt_r, crypt_rn, crypt_ra: setting is invalid or not supported by this implementation;
crypt_gensalt, crypt_gensalt_rn, crypt_gensalt_ra: prefix is invalid or not supported by this implementation; count is invalid for the requested prefix; the input size is insufficient for the smallest valid salt with the requested prefix; input is NULL.

**ERANGE**
crypt_rn: the provided data area size is insufficient for the requested hashing algorithm;
crypt_gensalt_rn: output size is too small to hold the compiled setting string.

**ENOMEM**
crypt (original glibc only): failed to allocate memory for the output buffer (which subsequent calls would re-use);
crypt_ra: *data is NULL or *size is insufficient for the requested hashing algorithm and realloc(3) failed;
crypt_gensalt_ra: failed to allocate memory for the compiled setting string.

**ENOSYS**
crypt (SUSv2): the functionality is not supported on this implementation;
crypt, crypt_r (glibc 2.0 to 2.0.1 only): the crypt add-on is not compiled in and setting requests something other than the MD5-based algorithm.

**EOPNOTSUPP**
crypt, crypt_r (glibc 2.0.2 to 2.1.3 only): the crypt add-on is not compiled in and setting requests something other than the MD5-based algorithm.

**HASHING METHODS**
The implemented hashing methods are intended specifically for processing user passwords for storage and authentication; they are at best inefficient for most other purposes.

It is important to understand that password hashing is not a replacement for strong passwords. It is always possible for an attacker with access to password hashes to try guessing candidate passwords against the hashes. There’re, however, certain properties a password hashing method may have which make these key search attacks somewhat harder.

All of the hashing methods use salts such that the same key may produce many possible hashes. Proper use of salts may defeat a number of attacks, including:

1. The ability to try candidate passwords against multiple hashes at the price of one.
2. The use of pre-hashed lists of candidate passwords.

3. The ability to determine whether two users (or two accounts of one user) have the same or different passwords without actually having to guess one of the passwords.

The key search attacks depend on computing hashes of large numbers of candidate passwords. Thus, the computational cost of a good password hashing method must be high – but of course not too high to render it impractical.

All hashing methods implemented within the `crypt`, `crypt_r`, `crypt_rn`, and `crypt_ra` interfaces use multiple iterations of an underlying cryptographic primitive specifically in order to increase the cost of trying a candidate password. Unfortunately, due to hardware improvements, the hashing methods which have a fixed cost become increasingly less secure over time.

In addition to salts, modern password hashing methods accept a variable iteration count. This makes it possible to adapt their cost to the hardware improvements while still maintaining compatibility.

The following hashing methods are or may be implemented within the described interfaces:

**Traditional DES-based**

This method is supported by almost all implementations of `crypt`. Unfortunately, it no longer offers adequate security because of its many limitations. Thus, it should not be used for new passwords unless you absolutely have to be able to migrate the password hashes to other systems.

prefix "" (empty string);

a string matching `/./0-9A-Za-z\{2\}` (see regex(7))

**Encoding syntax**

`./0-9A-Za-z\{13\}`

**Maximum password length**

8 (uses 7-bit characters)

**Effective key size**

up to 56 bits

**Hash size**

64 bits

**Salt size**

12 bits

**Iteration count**

25

**Extended BSDI-style DES-based**

This method is used on BSDI and is also available on at least NetBSD, OpenBSD, and FreeBSD due to the use of David Burren’s FreeSec library.

prefix "-"

**Encoding syntax**

`./0-9A-Za-z\{19\}`

**Maximum password length**

unlimited (uses 7-bit characters)

**Effective key size**

up to 56 bits

**Hash size**

64 bits

**Salt size**

24 bits
Iteration count
1 to \(2^{24} - 1\) (must be odd)

FreeBSD-style MD5-based
This is Poul-Henning Kamp’s MD5-based password hashing method originally developed for FreeBSD. It is currently supported on many free Unix-like systems, on Solaris 10, and it is a part of the official glibc. Its main disadvantage is the fixed iteration count, which is already too low for the currently available hardware.

Prefix  "$1$"

Encoding syntax
```
$1\{['$']1,8']$[./0-9A-Za-z]{22}
```

Maximum password length
unlimited (uses 8-bit characters)

Effective key size
limited by the hash size only

Hash size
128 bits

Salt size
6 to 48 bits

Iteration count
1000

OpenBSD-style Blowfish-based (bcrypt)

bcrypt was originally developed by Niels Provos and David Mazieres for OpenBSD and is also supported on recent versions of FreeBSD and NetBSD, on Solaris 10, and on several GNU/*/Linux distributions. It is, however, not a part of the official glibc.

While both bcrypt and the BSDI-style DES-based hashing offer a variable iteration count, bcrypt may scale to even faster hardware, doesn’t allow for certain optimizations specific to password cracking only, doesn’t have the effective key size limitation, and uses 8-bit characters in passwords.

Prefix  "$2a$"

Encoding syntax
```
$2a\{[0-9]{2}\$[./A-Za-z0-9]{53}
```

Maximum password length
72 (uses 8-bit characters)

Effective key size
limited by the hash size only

Hash size
184 bits

Salt size
128 bits

Iteration count
\(2^4\) to \(2^{99}\) (current implementations are limited to \(2^{31}\) iterations)

With bcrypt, the count passed to crypt_gensalt, crypt_gensalt_rn, and crypt_gensalt_ra is the base-2 logarithm of the actual iteration count.

PORTABILITY NOTES
Programs using any of these functions on a glibc 2.x system must be linked against libcrypt. However, many Unix-like operating systems and older versions of the GNU C Library include the crypt function in libc.
The crypt_r, crypt_rn, crypt_ra, crypt_gensalt, crypt_gensalt_rn, and crypt_gensalt_ra functions are very non-portable.

The set of supported hashing methods is implementation-dependent.

CONFORMING TO

The crypt function conforms to SVID, X/OPEN, and is available on BSD 4.3. The strings returned by crypt are not required to be portable among conformant systems.

crypt_r is a GNU extension. There’s also a crypt_r function on HP-UX and MKS Toolkit, but the prototypes and semantics differ.

crypt_gensalt is an Openwall extension. There’s also a crypt_gensalt function on Solaris 10, but the prototypes and semantics differ.

crypt_rn, crypt_ra, crypt_gensalt_rn, and crypt_gensalt_ra are Openwall extensions.

HISTORY

A rotor-based crypt function appeared in Version 6 AT&T UNIX. The "traditional" crypt first appeared in Version 7 AT&T UNIX.

The crypt_r function was introduced during glibc 2.0 development.

BUGS

The return values of crypt and crypt_gensalt point to static buffers that are overwritten by subsequent calls. These functions are not thread-safe. (crypt on recent versions of Solaris uses thread-specific data and actually is thread-safe.)

The strings returned by certain other implementations of crypt on error may be stored in read-only locations or only initialized once, which makes it unsafe to always attempt to zero out the buffer normally pointed to by the crypt return value as it would otherwise be preferable for security reasons. The problem could be avoided with the use of crypt_r, crypt_rn, or crypt_ra where the application has full control over output buffers of these functions (and often over some of their private data as well). Unfortunately, the functions aren’t (yet?) available on platforms where crypt has this undesired property.

Applications using the thread-safe crypt_r need to allocate address space for the large (over 128 KB) struct crypt_data structure. Each thread needs a separate instance of the structure. The crypt_r interface makes it impossible to implement a hashing algorithm which would need to keep an even larger amount of private data, without breaking binary compatibility. crypt_ra allows for dynamically increasing the allocation size as required by the hashing algorithm that is actually used. Unfortunately, crypt_ra is even more non-portable than crypt_r.

Multi-threaded applications or library functions which are meant to be thread-safe should use crypt_gensalt_rn or crypt_gensalt_ra rather than crypt_gensalt.

SEE ALSO


http://www.usenix.org/events/usenix99/provos.html

http://plan9.bell-labs.com/7thEdMan/vol2/password