NAME

crypt, crypt_r − password and data encryption

SYNOPSIS

#define _XOPEN_SOURCE /* See feature_test_macros(7)
*/"
#include <unistd.h>

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

#define _GNU_SOURCE /* See feature_test_macros(7)
*/"
#include <crypt.h>

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

Link with *−lcrypt*.

DESCRIPTION

crypt() is the password encryption function. It is based on the Data Encryption Standard algorithm with variations intended (among other things) to discourage use of hardware implementations of a key search.

key is a user's typed password.

salt is a two-character string chosen from the set [**a−zA−Z0−9./**]. This string is used to perturb the algorithm in one of 4096 different ways.

By taking the lowest 7 bits of each of the first eight characters of the *key*, a 56-bit key is obtained. This 56-bit key is used to encrypt repeatedly a constant string (usually a string consisting of all zeros). The returned value points to the encrypted password, a series of 13 printable ASCII characters (the first two characters represent the salt itself). The return value points to static data whose content is overwritten by each call.

Warning: the key space consists of 2^{56} equal 7.2e16 possible values. Exhaustive searches of this key space are possible using massively parallel computers. Software, such as **crack(1),** is available which will search the portion of this key space that is generally used by humans for passwords. Hence, password selection should, at minimum, avoid common words and names. The use of a passwd(1) program that checks for crackable passwords during the selection process is recommended.

The DES algorithm itself has a few quirks which make the use of the **crypt**() interface a very poor choice for anything other than password authentication. If you are planning on using the **crypt**() interface for a cryptography project, don't do it: get a good book on encryption and one of the widely available DES libraries.

crypt_r() is a reentrant version of **crypt**(). The structure pointed to by *data* is used to store result data and bookkeeping information. Other than allocating it, the only thing that the caller should do with this structure is to set *data->initialized* to zero before the first call to **crypt_r**().

RETURN VALUE

On success, a pointer to the encrypted password is returned. On error, NULL is returned.

ERRORS

EINVAL
salt has the wrong format.

ENOSYS
The **crypt**() function was not implemented, probably because of U.S.A. export restrictions.

EPERM
/proc/sys/crypto/fips_enabled has a nonzero value, and an attempt was made to use a weak encryption type, such as DES.

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()	Thread safety	MT-Safe

CONFORMING TO

crypt(): POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD. **crypt_r**() is a GNU extension.

NOTES

Availability in glibc

The **crypt**(), encrypt(3), and setkey(3) functions are part of the POSIX.1-2008 XSI Options Group for Encryption and are optional. If the interfaces are not available, then the symbolic constant **_XOPEN_CRYPT** is either not defined, or it is defined to −1 and availability can be checked at run time with sysconf(3). This may be the case if the downstream distribution has switched from glibc crypt to *libxcrypt*. When recompiling applications in such distributions, the programmer must detect if **_XOPEN_CRYPT** is not available and include *<crypt.h>* for the function prototypes; otherwise *libxcrypt* is an ABI-compatible drop-in replacement.

Features in glibc

The glibc version of this function supports additional encryption algorithms.

If *salt* is a character string starting with the characters "$*id*$" followed by a string optionally terminated by "$", then the result has the form:

$*id*$*salt*$*encrypted*

id identifies the encryption method used instead of DES and this then determines how the rest of the password string is interpreted. The following values of *id* are supported:

ID	Method
1	MD5
2a	Blowfish (not in mainline glibc; added in some Linux distributions)
5	SHA-256 (since glibc 2.7)
6	SHA-512 (since glibc 2.7)

Thus, 5*salt*$*encrypted* and 6*salt*$*encrypted* contain the password encrypted with, respectively, functions based on SHA-256 and SHA-512.

"*salt*" stands for the up to 16 characters following "$*id*$" in the salt. The "*encrypted*" part of the password string is the actual computed password. The size of this string is fixed:

MD5	22 characters
SHA-256	43 characters
SHA-512	86 characters

The characters in "*salt*" and "*encrypted*" are drawn from the set [**a−zA−Z0−9./**]. In the MD5 and SHA implementations the entire *key* is significant (instead of only the first 8 bytes in DES).

Since glibc 2.7, the SHA-256 and SHA-512 implementations support a user-supplied number of hashing rounds, defaulting to 5000. If the "$*id*$" characters in the salt are followed by "rounds=*xxx*$", where *xxx* is an integer, then the result has the form

$*id*$*rounds*=*yyy*$*salt*$*encrypted*

where *yyy* is the number of hashing rounds actually used. The number of rounds actually used is 1000 if *xxx* is less than 1000, 999999999 if *xxx* is greater than 999999999, and is equal to *xxx* otherwise.

SEE ALSO

login(1), passwd(1), encrypt(3), getpass(3), passwd(5)

COLOPHON

This page is part of release 4.16 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.