

NAME

crypt - Trapdoor encryption

LIBRARY

Crypt Library (libcrypt, -lcrypt)

SYNOPSIS

```
#include <unistd.h>
```

```
char *
```

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

```
char *
```

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

```
const char *
```

```
crypt_get_format(void);
```

```
int
```

```
crypt_set_format(const char *string);
```

DESCRIPTION

The **crypt**() function performs password hashing with additional code added to deter key search attempts. Different algorithms can be used to in the hash. Currently these include the NBS Data Encryption Standard (DES), MD5 hash, NT-Hash (compatible with Microsoft's NT scheme) and Blowfish. The algorithm used will depend upon the format of the Salt (following the Modular Crypt Format (MCF)), if DES and/or Blowfish is installed or not, and whether **crypt_set_format**() has been called to change the default.

The first argument to **crypt** is the data to hash (usually a password), in a NUL-terminated string. The second is the salt, in one of three forms:

- | | |
|-------------|--|
| Extended | If it begins with an underscore ("_") then the DES Extended Format is used in interpreting both the key and the salt, as outlined below. |
| Modular | If it begins with the string "\$digit\$" then the Modular Crypt Format is used, as outlined below. |
| Traditional | If neither of the above is true, it assumes the Traditional Format, using the entire string as the salt (or the first portion). |

All routines are designed to be time-consuming.

DES Extended Format:

The *key* is divided into groups of 8 characters (the last group is NUL-padded) and the low-order 7 bits of each character (56 bits per group) are used to form the DES key as follows: the first group of 56 bits becomes the initial DES key. For each additional group, the XOR of the encryption of the current DES key with itself and the group bits becomes the next DES key.

The *salt* is a 9-character array consisting of an underscore followed by 4 bytes of iteration count and 4 bytes of salt. These are encoded as printable characters, 6 bits per character, least significant character first. The values 0 to 63 are encoded as `"/0-9A-Za-z"`. This allows 24 bits for both *count* and *salt*.

The *salt* introduces disorder in the DES algorithm in one of 16777216 or 4096 possible ways (i.e., with 24 or 12 bits: if bit *i* of the *salt* is set, then bits *i* and *i+24* are swapped in the DES E-box output).

The DES key is used to encrypt a 64-bit constant using *count* iterations of DES. The value returned is a NUL-terminated string, 20 or 13 bytes (plus NUL) in length, consisting of the *salt* followed by the encoded 64-bit encryption.

Modular crypt:

If the salt begins with the string *\$digit\$* then the Modular Crypt Format is used. The *digit* represents which algorithm is used in encryption. Following the token is the actual salt to use in the encryption. The maximum length of the salt used depends upon the module. The salt must be terminated with the end of the string character (NUL) or a dollar sign. Any characters after the dollar sign are ignored.

Currently supported algorithms are:

1. MD5
2. Blowfish
3. NT-Hash
4. (unused)
5. SHA-256
6. SHA-512

Other crypt formats may be easily added. An example salt would be:

`4thesalt$rest`

Traditional crypt:

The algorithm used will depend upon whether `crypt_set_format()` has been called and whether a global default format has been specified. Unless a global default has been specified or `crypt_set_format()` has set the format to something else, the built-in default format is used. This is currently DES if it is

available, or SHA-512 if not.

How the salt is used will depend upon the algorithm for the hash. For best results, specify at least eight characters of salt.

The **crypt_get_format()** function returns a constant string that represents the name of the algorithm currently used. Valid values are 'des', 'blf', 'md5', 'sha256', 'sha512' and 'nth'.

The **crypt_set_format()** function sets the default encoding format according to the supplied *string*.

The **crypt_r()** function behaves identically to **crypt()**, except that the resulting string is stored in *data*, making it thread-safe.

RETURN VALUES

The **crypt()** and **crypt_r()** functions return a pointer to the encrypted value on success, and NULL on failure. Note: this is not a standard behaviour, AT&T **crypt()** will always return a pointer to a string.

The **crypt_set_format()** function will return 1 if the supplied encoding format was valid. Otherwise, a value of 0 is returned.

SEE ALSO

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

HISTORY

A rotor-based **crypt()** function appeared in Version 6 AT&T UNIX. The current style **crypt()** first appeared in Version 7 AT&T UNIX.

The DES section of the code (FreeSec 1.0) was developed outside the United States of America as an unencumbered replacement for the U.S.-only NetBSD libcrypt encryption library.

The **crypt_r()** function was added in FreeBSD 12.0.

AUTHORS

Originally written by David Burren <davidb@werj.com.au>, later additions and changes by Poul-Henning Kamp, Mark R V Murray, Michael Bretterkieber, Kris Kennaway, Brian Feldman, Paul Herman and Niels Provos.

BUGS

The **crypt()** function returns a pointer to static data, and subsequent calls to **crypt()** will modify the same data. Likewise, **crypt_set_format()** modifies static data.

The NT-hash scheme does not use a salt, and is not hard for a competent attacker to break. Its use is not recommended.