NAME

crypt — storage format for hashed passphrases and available hashing methods

DESCRIPTION

The hashing methods implemented by `crypt`(3) are designed only to process user passphrases for storage and authentication; they are not suitable for use as general-purpose cryptographic hashes.

Passphrase hashing is not a replacement for strong passphrases. It is always possible for an attacker with access to the hashed passphrases to guess and check possible cleartext passphrases. However, with a strong hashing method, guessing will be too slow for the attacker to discover a strong passphrase.

All of the hashing methods use a "salt" to perturb the hash function, so that the same passphrase may produce many possible hashes. Newer methods accept longer salt strings. The salt should be chosen at random for each user. Salt defeats a number of attacks:

1. It is not possible to hash a passphrase once and then test it against each account's stored hash; the hash calculation must be repeated for each account.
2. It is not possible to tell whether two accounts use the same passphrase without successfully guessing one of the phrases.
3. Tables of precalculated hashes of commonly used passphrases must have an entry for each possible salt, which makes them impractically large.

All of the hashing methods are also deliberately engineered to be slow; they use many iterations of an underlying cryptographic primitive to increase the cost of each guess. The newer hashing methods allow the number of iterations to be adjusted, using the "CPU time cost" parameter to `crypt_gensalt`(3). This makes it possible to keep the hash slow as hardware improves.

FORMAT OF HASHED PASSPHRASES

All of the hashing methods supported by `crypt`(3) produce a hashed passphrase which consists of four components: *`prefix`*, *`options`*, *`salt`*, and *`hash`*. The prefix controls which hashing method is to be used, and is the appropriate string to pass to `crypt_gensalt`(3) to select that method. The contents of *`options`*, *`salt`*, and *`hash`* are up to the method. Depending on the method, the *`prefix`* and *`options`* components may be empty.

The *`setting`* argument to `crypt`(3) must begin with the first three components of a valid hashed passphrase, but anything after that is ignored. This makes authentication simple: hash the input passphrase using the stored passphrase as the setting, and then compare the result to the stored passphrase.

Hashed passphrases are always entirely printable ASCII, and do not contain any whitespace or the characters '`:`', '`;`', '`*`', '`!`', or '`\`'. (These characters are used as delimiters and special markers in the `passwd`(5) and `shadow`(5) files.)

The syntax of each component of a hashed passphrase is up to the hashing method. '`$`' characters usually delimit components, and the salt and hash are usually encoded as numerals in base 64. The details of this base-64 encoding vary among hashing methods. The common "base64" encoding specified by RFC 4648 is usually *not* used.

AVAILABLE HASHING METHODS

This is a list of *all* the hashing methods supported by `crypt`(3), in decreasing order of strength. Many of the older methods are now considered too weak to use for new passphrases. The hashed passphrase format is expressed with extended regular expressions (see `regex`(7)) and does not show the division into prefix, options, salt, and hash.

yescrypt

yescrypt is a scalable passphrase hashing scheme designed by Solar Designer, which is based on Colin Percival's scrypt. Recommended for new hashes.

Prefix
`"$y$"`

Hashed passphrase format
`\$y\$[./A-Za-z0-9]+\$[./A-Za-z0-9]{,86}\$[./A-Za-z0-9]{43}`

Maximum passphrase length
unlimited

Hash size
256 bits

Salt size
up to 512 (128+ recommended) bits

CPU time cost parameter
1 to 11 (logarithmic)

gost-yescrypt

gost-yescrypt uses the output from the yescrypt hashing method in place of a hmac message. Thus, the yescrypt crypto properties are superseded by the GOST R 34.11-2012 (Streebog) hash function with a 256 bit digest. This hashing method is useful in applications that need modern passphrase hashing methods, but require to rely on the cryptographic properties of GOST algorithms. The GOST R 34.11-2012 (Streebog) hash function has been published by the IETF as RFC 6986. Recommended for new hashes.

Prefix
`"$gy$"`

Hashed passphrase format
`\$gy\$[./A-Za-z0-9]+\$[./A-Za-z0-9]{,86}\$[./A-Za-z0-9]{43}`

Maximum passphrase length
unlimited

Hash size
256 bits

Salt size
up to 512 (128+ recommended) bits

CPU time cost parameter
1 to 11 (logarithmic)

scrypt

scrypt is a password-based key derivation function created by Colin Percival, originally for the Tarsnap online backup service. The algorithm was specifically designed to make it costly to perform large-scale custom hardware attacks by requiring large amounts of memory. In 2016, the scrypt algorithm was published by IETF as RFC 7914.

Prefix
`"$7$"`

Hashed passphrase format
`\$7\$[./A-Za-z0-9]{11,97}\$[./A-Za-z0-9]{43}`

Maximum passphrase length
unlimited

Hash size
256 bits

Salt size
up to 512 (128+ recommended) bits

CPU time cost parameter
6 to 11 (logarithmic)

bcrypt

A hash based on the Blowfish block cipher, modified to have an extra-expensive key schedule. Originally developed by Niels Provos and David Mazieres for OpenBSD and also supported on recent versions of FreeBSD and NetBSD, on Solaris 10 and newer, and on several GNU/*/Linux distributions.

Prefix
`"$2b$"`

Hashed passphrase format
`\$2[abxy]\$[0-9]{2}\$[./A-Za-z0-9]{53}`

Maximum passphrase length
72 characters

Hash size
184 bits

Salt size
128 bits

CPU time cost parameter
4 to 31 (logarithmic)

The alternative prefix "$2y$" is equivalent to "$2b$". It exists for historical reasons only. The alternative prefixes "$2a$" and "$2x$" provide bug-compatibility with crypt_blowfish 1.0.4 and earlier, which incorrectly processed characters with the 8th bit set.

sha512crypt

A hash based on SHA-2 with 512-bit output, originally developed by Ulrich Drepper for GNU libc. Supported on Linux but not common elsewhere. Acceptable for new hashes. The default CPU time cost parameter is 5000, which is too low for modern hardware.

Prefix
`"$6$"`

Hashed passphrase format
`\$6\$(rounds=[1-9][0-9]+\$)?[^$:\n]{1,16}\$[./0-9A-Za-z]{86}`

Maximum passphrase length
unlimited

Hash size
512 bits

Salt size
6 to 96 bits

CPU time cost parameter
1000 to 999,999,999

sha256crypt

A hash based on SHA-2 with 256-bit output, originally developed by Ulrich Drepper for GNU libc. Supported on Linux but not common elsewhere. Acceptable for new hashes. The default CPU time cost parameter is 5000, which is too low for modern hardware.

Prefix
`"$5$"`

Hashed passphrase format
`\$5\$(rounds=[1-9][0-9]+\$)?[^$:\n]{1,16}\$[./0-9A-Za-z]{43}`

Maximum passphrase length
unlimited

Hash size
256 bits

Salt size
6 to 96 bits

CPU time cost parameter
1000 to 999,999,999

sha1crypt

A hash based on HMAC-SHA1. Originally developed by Simon Gerraty for NetBSD. Not as weak as the DES-based hashes below, but SHA1 is so cheap on modern hardware that it should not be used for new hashes.

Prefix
`"$sha1"`

Hashed passphrase format
`\$sha1\$[1-9][0-9]+\$[./0-9A-Za-z]{1,64}\$[./0-9A-Za-z]{8,64}[./0-9A-Za-z]{32}`

Maximum passphrase length
unlimited

Hash size
160 bits

Salt size
6 to 384 bits

CPU time cost parameter
4 to 4,294,967,295

SunMD5

A hash based on the MD5 algorithm, with additional cleverness to make precomputation difficult, originally developed by Alec David Muffet for Solaris. Not adopted elsewhere, to our knowledge. Not as weak as the DES-based hashes below, but MD5 is so cheap on modern hardware that it should not be used for new hashes.

Prefix
`"$md5"`

Hashed passphrase format
`\$md5(,rounds=[1-9][0-9]+)?\$[./0-9A-Za-z]{8}\${1,2}[./0-9A-Za-z]{22}`

Maximum passphrase length
unlimited

Hash size
128 bits

Salt size
48 bits

CPU time cost parameter
4096 to 4,294,963,199

md5crypt

A hash based on the MD5 algorithm, originally developed by Poul-Henning Kamp for FreeBSD. Supported on most free Unixes and newer versions of Solaris. Not as weak as the DES-based hashes below, but MD5 is so cheap on modern hardware that it should not be used for new hashes. CPU time cost is not adjustable.

Prefix
`"$1$"`

Hashed passphrase format
`\$1\$[^$:\n]{1,8}\$[./0-9A-Za-z]{22}`

Maximum passphrase length
unlimited

Hash size
128 bits

Salt size
6 to 48 bits

CPU time cost parameter
1000

bsdicrypt (BSDI extended DES)

A weak extension of traditional DES, which eliminates the length limit, increases the salt size, and makes the time cost tunable. It originates with BSDI and is also available on at least NetBSD, OpenBSD, and FreeBSD due to the use of David Burren's FreeSec library. It is better than bigcrypt and traditional DES, but still should not be used for new hashes.

Prefix
`"_"`

Hashed passphrase format
`_[./0-9A-Za-z]{19}`

Maximum passphrase length
unlimited (ignores 8th bit)

Hash size
64 bits

Effective key size
56 bits

Salt size
24 bits

CPU time cost parameter
1 to 16,777,215 (must be odd)

bigcrypt

A weak extension of traditional DES, available on some System V-derived Unixes. All it does is raise the length limit from 8 to 128 characters, and it does this in a crude way that allows attackers to guess chunks of a long passphrase in parallel. It should not be used for new hashes.

Prefix
`""` (empty string)

Hashed passphrase format
`[./0-9A-Za-z]{13,178}`

Maximum passphrase length
128 characters (ignores 8th bit)

Hash size
up to 1024 bits

Effective key size
up to 896 bits

Salt size
12 bits

CPU time cost parameter
25

descrypt (Traditional DES)

The original hashing method from Unix V7, based on the DES block cipher. Because DES is cheap on modern hardware, because there are only 4096 possible salts and 2∗∗56 possible hashes, and because it truncates passphrases to 8 characters, it is feasible to discover *any* passphrase hashed with this method. It should only be used if you absolutely have to generate hashes that will work on an old operating system that supports nothing else.

Prefix
`""` (empty string)

Hashed passphrase format
`[./0-9A-Za-z]{13}`

Maximum passphrase length
8 characters (ignores 8th bit)

Hash size
64 bits

Effective key size
56 bits

Salt size
12 bits

CPU time cost parameter
25

NT

The hashing method used for network authentication in some versions of the SMB/CIFS protocol. Available, for cross-compatibility's sake, on FreeBSD. Based on MD4. Has no salt or tunable cost parameter. Like traditional DES, it is so weak that *any* passphrase hashed with this method is guessable. It should only be used if you absolutely have to generate hashes that will work on an old operating system that supports nothing else.

Prefix
`"$3$"`

Hashed passphrase format
`\$3\$\$[0-9a-f]{32}`

Maximum passphrase length
unlimited

Hash size
256 bits

Salt size
0 bits

CPU time cost parameter
1

SEE ALSO

`crypt`(3), `crypt_gensalt`(3), `getpwent`(3), `passwd`(5), `shadow`(5), `pam`(8)

Niels Provos and David Mazieres, "A Future-Adaptable Password Scheme", *Proceedings of the 1999 USENIX Annual Technical Conference*, https://www.usenix.org/events/usenix99/provos.html, June 1999.

Robert Morris and Ken Thompson, "Password Security: A Case History", *Communications of the ACM*, 11, 22, http://wolfram.schneider.org/bsd/7thEdManVol2/password/password.pdf, 1979.