`crypt` — Function to check Unix passwords

Source code: Lib/crypt.py

This module implements an interface to the *crypt(3)* routine, which is a one-way hash function based upon a modified DES algorithm; see the Unix man page for further details. Possible uses include storing hashed passwords so you can check passwords without storing the actual password, or attempting to crack Unix passwords with a dictionary.

Notice that the behavior of this module depends on the actual implementation of the *crypt(3)* routine in the running system. Therefore, any extensions available on the current implementation will also be available on this module.

Availability: Unix. Not available on VxWorks.

Hashing Methods

New in version 3.3.

The `crypt` module defines the list of hashing methods (not all methods are available on all platforms):

`crypt.`**`METHOD_SHA512`**

A Modular Crypt Format method with 16 character salt and 86 character hash based on the SHA-512 hash function. This is the strongest method.

`crypt.`**`METHOD_SHA256`**

Another Modular Crypt Format method with 16 character salt and 43 character hash based on the SHA-256 hash function.

`crypt.`**`METHOD_BLOWFISH`**

Another Modular Crypt Format method with 22 character salt and 31 character hash based on the Blowfish cipher.

New in version 3.7.

`crypt.`**`METHOD_MD5`**

Another Modular Crypt Format method with 8 character salt and 22 character hash based on the MD5 hash function.

`crypt.`**`METHOD_CRYPT`**

The traditional method with a 2 character salt and 13 characters of hash. This is the weakest method.

Module Attributes

New in version 3.3.

`crypt.`**`methods`**

A list of available password hashing algorithms, as `crypt.METHOD_*` objects. This list is sorted from strongest to weakest.

Module Functions

The `crypt` module defines the following functions:

`crypt.`**`crypt`**(*`word`*, *`salt=None`*)

word will usually be a user's password as typed at a prompt or in a graphical interface. The optional *salt* is either a string as returned from `mksalt()`, one of the `crypt.METHOD_*` values (though not all may be available on all platforms), or a full encrypted password including salt, as returned by this function. If *salt* is not provided, the strongest method will be used (as returned by `methods()`).

Checking a password is usually done by passing the plain-text password as *word* and the full results of a previous `crypt()` call, which should be the same as the results of this call.

salt (either a random 2 or 16 character string, possibly prefixed with `$digit$` to indicate the method) which will be used to perturb the encryption algorithm. The characters in *salt* must be in the set `[./a-zA-Z0-9]`, with the exception of Modular Crypt Format which prefixes a `$digit$`.

Returns the hashed password as a string, which will be composed of characters from the same alphabet as the salt.

Since a few *crypt(3)* extensions allow different values, with different sizes in the *salt*, it is recommended to use the full crypted password as salt when checking for a password.

Changed in version 3.3: Accept `crypt.METHOD_*` values in addition to strings for *salt*.

`crypt.`**`mksalt`**(*`method=None`*, *, *`rounds=None`*)

Return a randomly generated salt of the specified method. If no *method* is given, the strongest method available as returned by `methods()` is used.

The return value is a string suitable for passing as the *salt* argument to `crypt()`.

rounds specifies the number of rounds for `METHOD_SHA256`, `METHOD_SHA512` and `METHOD_BLOWFISH`. For `METHOD_SHA256` and `METHOD_SHA512` it must be an integer between `1000` and `999_999_999`, the default is `5000`. For `METHOD_BLOWFISH` it

must be a power of two between 16 (2^4) and `2_147_483_648` (2^{31}), the default is `4096` (2^{12}).

New in version 3.3.

Changed in version 3.7: Added the *rounds* parameter.

Examples

A simple example illustrating typical use (a constant-time comparison operation is needed to limit exposure to timing attacks. `hmac.compare_digest()` is suitable for this purpose):

```
import pwd
import crypt
import getpass
from hmac import compare_digest as compare_hash

def login():
    username = input('Python login: ')
    cryptedpasswd = pwd.getpwnam(username)[1]
    if cryptedpasswd:
        if cryptedpasswd == 'x' or cryptedpasswd == '*':
            raise ValueError('no support for shadow passwords')
        cleartext = getpass.getpass()
        return compare_hash(crypt.crypt(cleartext, cryptedpasswd), cr
    else:
        return True
```

To generate a hash of a password using the strongest available method and check it against the original:

```
import crypt
from hmac import compare_digest as compare_hash

hashed = crypt.crypt(plaintext)
if not compare_hash(hashed, crypt.crypt(plaintext, hashed)):
    raise ValueError("hashed version doesn't validate against origina
```