Passlib Documentation

Release 1.7.4

Assurance Technologies, LLC

Oct 08, 2020

Contents

See also:

What's new in Passlib 1.7.4

CHAPTER 1

Passlib 1.7.4 documentation

Note: 2020-05-01: Passlib's public repository has moved to Heptapod!

Due to BitBucket deprecating Mercurial support, Passlib's public repository and issue tracker has been relocated. It's now located at https://foss.heptapod.net/python-libs/passlib, and is powered by Heptapod. Hosting is being graciously provided by the people at Octobus and CleverCloud!

1.1 Welcome

Passlib is a password hashing library for Python 2 & 3, which provides cross-platform implementations of over 30 password hashing algorithms, as well as a framework for managing existing password hashes. It's designed to be useful for a wide range of tasks, from verifying a hash found in /etc/shadow, to providing full-strength password hashing for multi-user application.

As a quick sample, the following code hashes and then verifies a password using the *PBKDF2-SHA256* algorithm:

```
>>> # import the hash algorithm
>>> from passlib.hash import pbkdf2_sha256

>>> # generate new salt, and hash a password
>>> hash = pbkdf2_sha256.hash("toomanysecrets")
>>> hash
'$pbkdf2-sha256$29000$N2YMIWQsBWBMae09x1jrPQ$1t8iyB2A.WF/Z5JZv.
↪lfCIhXXN33N23OSgQYThBYRfk'

>>> # verifying the password
>>> pbkdf2_sha256.verify("toomanysecrets", hash)
True
>>> pbkdf2_sha256.verify("joshua", hash)
False
```

1.2 Getting Started

This documentation is organized into two main parts: a narrative walkthrough of Passlib, and a top-down API reference.

Installation

See this page for system requirements & installation instructions.

Walkthrough & Tutorials

New users in particular will want to visit the walkthrough, as it provides introductory documentation including installation requirements, an overview of what passlib provides, and a guide for getting started quickly.

API Reference

The API reference contains a top-down reference of the `passlib` package.

Other Documentation

This section contains additional things that don't fit anywhere else, including an *FAQ* and a complete *changelog*.

1.3 Online Resources

Latest Docs:	https://passlib.readthedocs.io
Latest News:	https://foss.heptapod.net/python-libs/passlib/wikis/home
Public Repo:	https://foss.heptapod.net/python-libs/passlib
Mailing List:	https://groups.google.com/group/passlib-users
Downloads @ PyPI:	https://pypi.python.org/pypi/passlib

1.4 Hosting

Thanks to the people at Octobus and CleverCloud for providing the repository / issue tracker hosting, as well as development of Heptapod!

Thanks to ReadTheDocs for providing documentation hosting!

CHAPTER 2

Walkthrough & Tutorials

Introductory Materials

2.1 Installation

2.1.1 Supported Platforms

Passlib requires Python 2 (>= 2.6) or Python 3 (>= 3.3). It is known to work with the following Python implementations:

> **Warning: Passlib 1.8 will drop support for Python 2.x, 3.3, and 3.4**; and will require Python >= 3.5. The 1.7 series will be the last to support Python 2. (See issue 119 for rationale).

- CPython 2 – v2.6 or newer.
- CPython 3 – v3.3 or newer.
- PyPy – v2.0 or newer.
- PyPy3 – v5.3 or newer.
- Jython – v2.7 or newer.

Passlib should work with all operating systems and environments, as it contains builtin fallbacks for almost all OS-dependant features. Google App Engine is supported as well.

Changed in version 1.7: Support for Python 2.5, 3.0-3.2 was dropped. Support for PyPy 1.x was dropped.

2.1.2 Optional Libraries

- bcrypt, py-bcrypt, or bcryptor

> **Warning:** Support for `py-bcrypt` and `bcryptor` will be dropped in Passlib 1.8, as these libraries are unmaintained.

If any of these packages are installed, they will be used to provide support for the BCrypt hash algorithm. This is required if you want to handle BCrypt hashes, and your OS does not provide native BCrypt support via stdlib's `crypt` (which includes pretty much all non-BSD systems).

bcrypt is currently the recommended option – it's actively maintained, and compatible with both CPython and PyPy.

Use `pip install passlib[bcrypt]` to get the recommended bcrypt setup.

- argon2_cffi (>= 18.2.0), or argon2pure (>= 1.3)

  If any of these packages are installed, they will be used to provide support for the *argon2* hash algorithm. argon2_cffi is currently the recommended option.

  Use `pip install passlib[argon2]` to get the recommended argon2 setup.

- Cryptography

  If installed, will be used to enable encryption of TOTP secrets for storage (see *passlib.totp*).

  Use `pip install passlib[totp]` to get the recommended TOTP setup.

- fastpbkdf2

  If installed, will be used to greatly speed up *pbkdf2_hmac()*, and any pbkdf2-based hashes.

- SCrypt (>= 0.6)

  If installed, this will be used to provide support for the *scrypt* hash algorithm. If not installed, a MUCH slower builtin reference implementation will be used.

Changed in version 1.7: Added fastpbkdf2, cryptography, argon2_cffi, argon2pure, and scrypt support. Removed M2Crypto support.

2.1.3 Installation Instructions

> **Caution:** All PyPI releases are signed with the gpg key 4D8592DF4CE1ED31.

To install from PyPi using **`pip`**:

```
pip install passlib
```

To install from the source using **`setup.py`**:

```
python setup.py install
```

2.1.4 Testing

Passlib contains a comprehensive set of unittests (about 38% of the total code), which provide nearly complete coverage, and verification of the hash algorithms using multiple external sources (if detected at

runtime).

All unit tests are contained within the `passlib.tests` subpackage, and are designed to be run using the Nose unit testing library (as well as the `unittest2` library under Python 2.6).

Once Passlib and Nose have been installed, the main suite of tests may be run using:

```
nosetests --tests passlib.tests
```

By default, this runs the main battery of tests, but omits some additional ones (such as internal cross-checks, and mock-testing of features not provided natively by the host OS). To run these tests as well, set the following environmental variable:

```
PASSLIB_TEST_MODE="full" nosetests --tests passlib.tests
```

To run a quick check to confirm just basic functionality, with a pared-down set of tests:

```
PASSLIB_TEST_MODE="quick" nosetests --tests passlib.tests
```

Tests may also be run via `setup.py test` or the included `tox.ini` file. The `tox.ini` file is used to test passlib before each release, and contains a number different environment setups. These tests require tox 2.5 or later.

2.1.5 Building the Documentation

The latest copy of this documentation should always be available online at https://passlib.readthedocs.io. If you wish to generate your own copy of the documentation, you will need to:

1. Download the Passlib source, extract it, and `cd` into the source directory.
2. Install all the dependencies required via `pip install -e .[build_docs]`.
3. Run `python setup.py build_sphinx`.
4. Once Sphinx completes its run, point a web browser to the file at *SOURCE*`/build/sphinx/html/index.html` to access the Passlib documentation in html format.

2.2 Library Overview

Passlib is a collection of routines for managing password hashes such as found in unix "shadow" files, as returned by stdlib's `crypt.crypt()`, as stored in mysql and postgres, and various other places. Passlib's contents can be roughly grouped into four categories: password hashes, password contexts, two-factor authentication, and other utility functions.

2.2.1 Password Hashes

All of the hashes supported by Passlib are implemented as "hasher" classes which can be imported from the *`passlib.hash`* module. In turn, all of the hashers have a uniform interface, which is documented in the *PasswordHash Tutorial*.

A word of warning: Some the hashes in this library are marked as "insecure", and are provided for historical purposes only. Still others are specialized in ways that are not generally useful. If you are creating a new application and need to choose a password hash, please read the *New Application Quickstart Guide* first.

See also:

- *PasswordHash Tutorial* – walkthrough of using a hasher class.
- *New Application Quickstart Guide* – if you need to choose a hash.
- `passlib.ifc` – PasswordHash API reference
- `passlib.hash` – list of all hashes in Passlib.

2.2.2 Password Contexts

Mature applications frequently have to deal with tables of existing password hashes. Over time, they have to support a number of tasks:

- Add support for new algorithms, and deprecate old ones.
- Raise the time-cost settings for existing algorithms as computing power increases.
- Perform rolling upgrades of existing hashes to comply with these changes.
- Eventually, these policies must be hardcoded in the source, or time must be spent implementing a configuration language to encode them.

In these situations, loading and handling multiple hash algorithms becomes complicated and tedious. The `passlib.context` module provides a single class, `CryptContext`, which attempts to solve all of these problems (or at least relieve developers of most of the burden).

This class handles managing multiple password hash schemes, deprecation & migration of old hashes, and supports a simple configuration language that can be serialized to an INI file.

See also:

- *CryptContext Tutorial* – walkthrough of the CryptContext class
- `passlib.context` – API reference

2.2.3 Two-Factor Authentication

While not strictly connected to password hashing, modern applications frequently need to perform the related task of two-factor authentication. One of the most common protocols for doing this is TOTP (**RFC 6238**). To help get TOTP in place, the `passlib.totp` module provides a set of helper functions for securely configuring, persisting, and verifying TOTP tokens.

See also:

- *TOTP tutorial* – walkthrough of setting up TOTP integration
- `passlib.totp` – API reference

2.2.4 Application Helpers

Passlib also provides a number of pre-configured `CryptContext` instances in order to get users started quickly:

- `passlib.apps` – contains pre-configured instances for managing hashes used by Postgres, Mysql, and LDAP, and others.
- `passlib.hosts` – contains pre-configured instances for managing hashes as found in the /etc/shadow files on Linux and BSD systems.

Passlib also contains a couple of additional modules which provide support for certain application-specific tasks:

- *passlib.apache* – classes for managing htpasswd and htdigest files.
- *passlib.ext.django* – Django plugin which monkeypatches support for (almost) any hash in Passlib.

2.3 New Application Quickstart Guide

Need to quickly get password hash support added into your new application, but don't have time to wade through pages of documentation, comparing and contrasting all the different schemes? Then read on...

2.3.1 Choosing a Hash

If you'd like to set up a configuration that's right for your application, the first thing to do is choose a password hashing scheme. Passlib contains a large number of schemes, but most of them should only be used when a specific format is explicitly required.

See also:

If you already know what hash algorithm(s) you want to use, skip to the next section: *Creating and Using a CryptContext*.

2.3.1.1 The Options

There are currently four good choices[1] for secure hashing:

- *argon2*
- *bcrypt*
- *pbkdf2_sha256* / *pbkdf2_sha512*
- *sha256_crypt* / *sha512_crypt*

All four hashes share the following properties:

- No known vulnerabilities.
- Based on documented and widely reviewed algorithms.
- Public-domain or BSD-licensed reference implementations available.
- variable rounds for configuring flexible cpu cost on a per-hash basis.
- At least 96 bits of salt.
- Basic algorithm has seen heavy scrutiny and use for at least 10 years *(except for Argon2, born around 2013)*.
- In use across a number of OSes and/or a wide variety of applications.

While Argon2 is much younger than the others, it has seen heavy scrutiny, and was purpose-designed for password hashing. In the near future, it stands likely to become *the* recommended standard.

[1] As of June 2016, the most commonly used password hashes are BCrypt and PBKDF2, followed by SHA512-Crypt, with Argon2 rapidly moving up the ranks. You should make sure you are reading a current copy of the Passlib documentation, in case the state of things has changed.

2.3.1.2 Detailed Comparison of Choices

Argon2

argon2 is the newest of the four recommended hashes. It was selected as the winner of the 2013 Password Hashing Competition, and draws on the design and lessons from BCrypt, PBKDF2, and SCrypt. Despite being much newer than the others, it has seen heavy scrutiny. Since the Argon2 project had the foresight to provide not just a reference implementation, but a standard hash encoding format, these hashes should be reliably interoperatable across all implementations.

Issues: In its default configuration, Argon2 uses more memory than the other hashes. However, this is one of its hallmarks as a "memory hard" hashing algorithm, and contributes to its security. Furthermore the exact amount used is configurable. Its only main drawback is that as of 2019-01-12, it's only 6 years old. It's seen only a few minor adjustments since 2013, but as it is just now gaining widespread use, the next few years are the period in which it will likely either prove itself, or be found wanting. It's for this reason, any cryptographic algorithm less than a decade old is generally considered "young". :)

BCrypt

bcrypt is based on the well-tested Blowfish cipher. In use since 1999, it's the default hash on all BSD variants. If you want your application's hashes to be readable by the native BSD crypt() function, this is the hash to use. There is also an alternative LDAP-formatted version (*ldap_bcrypt*) available.

Issues: Neither the original Blowfish, nor the modified version which BCrypt uses, have been NIST approved; this matter of concern is what motivated the development of SHA512-Crypt. As well, its rounds parameter is logarithmically scaled, making it hard to fine-tune the amount of time taken to verify passwords; which can be an issue for applications that handle a large number of simultaneous logon attempts (e.g. web apps). Finally, BCrypt only hashes the first 72 characters of a password, and will silently truncate longer ones (Passlib's non-standard *bcrypt_sha256* works around this last issue).

PBKDF2

pbkdf2_sha512 is a custom hash format designed for Passlib. However, it directly uses the PBKDF2 key derivation function, which was standardized in 2000, and found across a wide variety of applications and platforms. Unlike the previous two hashes, PBKDF2 has a simple and portable design, which is resistant (but not immune) to collision and preimage attacks on the underlying message digest. There is also *pbkdf2_sha256*, which may be faster on 32 bit processors; as well as LDAP-formatted versions of these (*ldap_pbkdf2_sha512* and *ldap_pbkdf2_sha256*).

Issues: PBKDF2 has no major security or portability issues, and compares favorably against bcrypt. However, bcrypt has proven slightly more resistant to modern GPU-based cracking techniques.

SHA512-Crypt

sha512_crypt is based on the well-tested *md5_crypt* algorithm. In use since 2008, it's the default hash on most Linux systems; its direct ancestor `md5_crypt` has been in use since 1994 on most Unix systems. If you want your application's hashes to be readable by the native Linux crypt() function, this is the hash to use. There is also *sha256_crypt*, which may be faster on 32 bit processors; as well as LDAP-formatted versions of these (*ldap_sha512_crypt* and *ldap_sha256_crypt*).

Issues: Like *md5_crypt*, its algorithm composes the underlying message digest hash in a baroque and somewhat arbitrary set of combinations. So far this "kitchen sink" design has been successful in its

primary purpose: to prevent any attempts to create an optimized version for use in a pre-computed or brute-force search. However, this design also hampers analysis of the algorithm for future flaws.

While this algorithm is still considered secure, it has fallen out of favor in comparison to bcrypt and pbkdf2, due to its non-standard construction.

Furthermore, when compared to Argon2 and BCrypt, SHA512-Crypt and PBKDF2 have proven more susceptible to cracking using modern GPU-based techniques.

sha512_crypt is probably the best choice for Google App Engine, as Google's production servers appear to provide native support via `crypt`, which will be used by Passlib.

Note: References to this algorithm are frequently confused with a raw SHA-512 hash. While `sha512_crypt` uses the SHA-512 hash as a cryptographic primitive, the algorithm's resulting password hash is far more secure.

2.3.1.3 Making a Decision

For new applications, this decision comes down to a couple of questions:

1. Does the hash need to be natively supported by your operating system's `crypt()` api, in order to allow inter-operation with third-party applications on the host?
   - If yes, the right choice is either *bcrypt* for BSD variants, or *sha512_crypt* for Linux; since these are natively supported.
   - If no, continue...
2. Does your hosting provider allow you to install C extensions?
   - If no, you probably want to use *pbkdf2_sha256*, as this currently has the fastest pure-python backend.
   - If they allow C extensions, continue...
3. Do you want to use the latest and greatest, and don't mind increased memory usage when hashing?
   - *argon2* is a next-generation hashing algorithm, attempting to become the new standard. Its design has been being slightly tweaked since 2013, but will quite likely become *the* standard in the next few years. You'll need to install the argon2_cffi support library.
   - If you want something secure, but more battle tested, continue...
4. The top choices left are *bcrypt* and *pbkdf2_sha256*.

   Both have advantages, and their respective rough edges; though currently the balance is in favor of bcrypt (pbkdf2 can be cracked somewhat more efficiently).
   - If choosing bcrypt, we strongly recommend installing the bcrypt support library on non-BSD operating systems.
   - If choosing pbkdf2, especially on python2 < 2.7.8 and python 3 < 3.4, you will probably want to install fastpbk2 support library.

2.3.2 Creating and Using a CryptContext

Once you've chosen what password hash(es) you want to use, the next step is to define a *CryptContext* object to manage your hashes and related policy configuration. Insert the following code into your application:

```
#
# import the CryptContext class, used to handle all hashing...
#
from passlib.context import CryptContext

#
# create a single global instance for your app...
#
pwd_context = CryptContext(
    # Replace this list with the hash(es) you wish to support.
    # this example sets pbkdf2_sha256 as the default,
    # with additional support for reading legacy des_crypt hashes.
    schemes=["pbkdf2_sha256", "des_crypt"],

    # Automatically mark all but first hasher in list as deprecated.
    # (this will be the default in Passlib 2.0)
    deprecated="auto",

    # Optionally, set the number of rounds that should be used.
    # Appropriate values may vary for different schemes,
    # and the amount of time you wish it to take.
    # Leaving this alone is usually safe, and will use passlib's defaults.
    ## pbkdf2_sha256__rounds = 29000,
    )
```

To start using your CryptContext, import the context you created wherever it's needed:

```
>>> # import context from where you defined it...
>>> from myapp.model.security import pwd_context

>>> # hashing a password...
>>> hash = pwd_context.hash("somepass")
>>> hash
'$pbkdf2-sha256$29000$BSBkLEXIeS9FKMW4F.I85w
↪$SJMzqVU7fw49NDOJZHt2o9vKIfDUVM4cKlAD4MxIgD0'

>>> # verifying a password...
>>> pwd_context.verify("somepass", hash)
True
>>> pwd_context.verify("wrongpass", hash)
False
```

There's many more features packed into the context objects, read the walkthrough for more...

See also:

- *CryptContext Tutorial* – full details of using the CryptContext class
- `passlib.context` – CryptContext API reference
- `passlib.hash` – list of all hashes supported by Passlib.

Tutorials

2.4 PasswordHash Tutorial

2.4.1 Overview

Passlib supports a large number of hash algorithms, all of which can be imported from the *passlib.hash* module. While the exact options and behavior will vary between each algorithm, all of the hashes provided by Passlib use the same interface, defined by the *passlib.ifc.PasswordHash* abstract class.

The PasswordHash class provides a generic interface for interacting individually with the various hashing algorithms. It offers methods and attributes for a number of use-cases, which fall into three general categories:

- Creating & verifying hashes
- Examining the configuration of a hasher, and customizing the defaults.
- Assorting supplementary methods.

See also:

- *passlib.ifc* – API reference of all the methods and attributes of the PasswordHash class.
- *passlib.context.CryptContext* – For working with multiple hash formats at once (such a user account table with multiple existing hash formats).

2.4.2 Hashing & Verifying

While all the hashers in *passlib.hash* offer a range of methods and attributes, the main activities applications will need to perform is hashing and verifying passwords. This can be done with the *PasswordHash.hash()* and *PasswordHash.verify()* methods.

Caution: Changed in 1.7:

Prior releases used *PasswordHash.encrypt()* for hashing, which has now been renamed to *PasswordHash.hash()*. A compatibility alias is present in 1.7, but will be removed in Passlib 2.0.

2.4.2.1 Hashing

First, import the desired hash. The following example uses the *pbkdf2_sha256* class (which derives from PasswordHash):

```
>>> # import the desired hasher
>>> from passlib.hash import pbkdf2_sha256
```

Use *PasswordHash.hash()* to hash a password. This call takes care of unicode encoding, picking default rounds values, and generating a random salt:

```
>>> hash = pbkdf2_sha256.hash("password")
>>> hash
'$pbkdf2-sha256$29000$9t7be09prfXee2/NOUeotQ$Y.
↪RDnnq8vsezSZSKy1QNy6xhKPdoBIwc.0XDdRm9sJ8'
```

Note that since each call generates a new salt, the contents of the resulting hash will differ between calls (despite using the same password as input):

```
>>> hash2 = pbkdf2_sha256.hash("password")
>>> hash2
'$pbkdf2-sha256$29000$V0rJeS.FcO4dw/h/D6E0Bg$FyLs7omUppxzXkARJQSl.
↪ozcEOhgp3tNgNsKIAhKmp8'
                      ^^^^^^^^^^^^^^^^^^^^^^
```

2.4.2.2 Verifying

Subsequently, you can call *PasswordHash.verify()* to check user input against an existing hash:

```
>>> pbkdf2_sha256.verify("password", hash)
True

>>> pbkdf2_sha256.verify("joshua", hash)
False
```

2.4.2.3 Unicode & non-ASCII Characters

Sidenote regarding unicode passwords & non-ASCII characters:

For the majority of hash algorithms and use-cases, passwords should be provided as either `unicode` (or `utf-8`-encoded `bytes`).

One exception is legacy hashes that were generated using a different character encoding. In this case, passwords should be encoded using the correct encoding before they are passed to `verify()`; otherwise users may not be able to log in successfully.

For proper internationalization, applications should also take care to ensure unicode inputs are normalized to a single representation before hashing. The *passlib.utils.saslprep()* function can be used for this purpose.

2.4.3 Customizing the Configuration

2.4.3.1 The using() Method

Each hasher contains a number of *informational attributes*. many of which can be customized to change the properties of the hashes generated by *PasswordHash.hash()*. When you want to change the defaults, you don't have to modify the hasher class directly, or pass in the options to each call to `PasswordHash.hash()`.

Instead, all the hashes offer a *PasswordHash.using()* method. This is a powerful method which accepts most hash informational attributes, as well as some other hash-specific configuration keywords; and returns a subclass of the original hasher (or a object with an identical interface). The returned object inherits the defaults settings from it's parent, but integrates any values you choose to override.

Caution: Changed in 1.7:

Prior releases required you to pass custom settings to each *PasswordHash.encrypt()* call. That usage pattern is deprecated, and will be removed in Passlib 2.0; code should be switched to use *PasswordHash.using()*, as shown below.

2.4.3.2 Usage Example

As an example, if the hasher you select supports a variable number of iterations (such as *pbkdf2_sha256*), you can specify a custom value using the `rounds` keyword.

Here, the default class uses 29000 rounds:

```
>>> from passlib.hash import pbkdf2_sha256

>>> pbkdf2_sha256.default_rounds
29000

>>> pbkdf2_sha256.hash("password")
'$pbkdf2-sha256$29000$V0rJeS.FcO4dw/h/D6E0Bg$FyLs7omUppxzXkARJQSl.
↪ozcEOhgp3tNgNsKIAhKmp8'
                 ^^^^^
```

But if we call *PasswordHash.using()*, we can override this value:

```
>>> custom_pbkdf2 = pbkdf2_sha256.using(rounds=123456)
>>> custom_pbkdf2.default_rounds
123456

>>> custom_pbkdf2.hash("password")
'$pbkdf2-sha256$123456$QwjBmJPSOsf4HyNE6L239g
↪$8m1pnP69EYeOiKKb5sNSiYw9M8pJMyeW.CSm0KKO.GI'
                 ^^^^^^
```

2.4.3.3 Other Keywords

While hashes frequently have additional keywords supported by using, the basic set of settings you can customize can be found by inspecting the *PasswordHash.setting_kwds* attribute:

```
>>> pbkdf2_sha256.settings_kwds
("salt", "salt_size", "rounds")
```

For instance, the following generates pbkdf2 hashes with a 32-byte salt instead of the default 16:

```
>>> pbkdf2_sha256.using(salt_size=8).hash("password")
'$pbkdf2-sha256$29000$tPZ.r5UyZgyhNEaI8Z5z7r1X6p1zTknJ.T/nHINwbq0
↪$RlM49Qf5qRraHx.L7gq3hKIKSMLttrG1zWmWXyfXqc8'
                       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

This method is also used internally by the *CryptContext* class it order to create a custom hasher configured based on the CryptContext policy it was provided.

See also:

- *PasswordHash.using()* – API reference

2.4.4 Context Keywords

While the *PasswordHash.hash()* example above works for most hashes, a small number of algorithms require you provide external data (such as a username) every time a hash is calculated.

An example of this is the *oracle10* hash, where hashing requires a username:

```
>>> from passlib.hash import oracle10
>>> hash = oracle10.hash("secret", user="admin")
'B858CE295C95193F'
```

The difference between this and specifying something like a rounds setting (see *Customizing the Configuration* above) is that a configuration option only needs to be specified once, and is then encoded into the hash string itself… Whereas a context keyword represents something that isn't stored in the hash string, and needs to be specified every time you call *PasswordHash.hash()* **or** *PasswordHash.verify()*:

```
>>> oracle10.verify("secret", hash, user="admin")
True
```

In this example, if either the username OR password is wrong, verify() will fail:

```
>>> oracle10.verify("secret", hash, user="wronguser")
False

>>> oracle10.verify("wrongpassword", hash, user="admin")
False
```

Forgetting to include a context keywords when it's required will cause a TypeError:

```
>>> hash = oracle10.hash("password")
Traceback (most recent call last):
    <traceback omitted>
TypeError: user must be unicode or bytes, not None
```

Whether a hash requires external parameters (such as `user`) can be determined from its documentation page; but also programmatically from its *PasswordHash.context_kwds* attribute:

```
>>> oracle10.context_kwds
("user",)

>>> pbkdf2_sha256.context_kwds
()
```

2.4.5 Identifying Hashes

One of the rarer use-cases is the need to identify whether a string recognizably belongs to a given hasher class. This can be important in some cases, because attempting to call *PasswordHash.verify()* with another algorithm's hash will result in a ValueError:

```
>>> from passlib.hash import pbkdf2_sha256, md5_crypt

>>> other_hash = md5_crypt.hash("password")

>>> pbkdf2_sha256.verify("password", other_hash)
Traceback (most recent call last):
    <traceback omitted>
ValueError: not a valid pbkdf2_sha256 hash
```

This can be prevented by using the identify method, which determines whether a hash belongs to a given algorithm:

```
>>> hash = pbkdf2_sha256.hash("password")
>>> pbkdf2_sha256.identify(hash)
True

>>> pbkdf2_sha256.identify(other_hash)
False
```

See also:

In most cases where an application needs to distinguish between multiple hash formats, it will be more useful to switch to a *CryptContext* object, which automatically handles this and many similar tasks.

Todo: Document usage of *PasswordHash.needs_update()*, and how it ties into *PasswordHash.using()*.

2.4.6 Choosing the right rounds value

For hash algorithms with a variable time-cost, Passlib's *PasswordHash.default_rounds* values attempt to be secure enough for the average[1] system. But the "right" value for a given hash is dependant on the server, its cpu, its expected load, and its users. Since larger values mean increased work for an attacker...

The right `rounds` value for a given hash & server should be the largest possible value that doesn't cause intolerable delay for your users.

For most public facing services, you can generally have signin take upwards of 250ms - 400ms before users start getting annoyed. For superuser accounts, it should take as much time as the admin can stand (usually ~4x more delay than a regular account).

Passlib's `default_rounds` values are retuned periodically, starting with a rough estimate of what an "average" system is capable of, and then setting all *hash*`.default_rounds` values to take ~300ms on such a system. However, some older algorithms (e.g. *bsdi_crypt*) are weak enough that a tradeoff must be made, choosing "more secure but intolerably slow" over "fast but unacceptably insecure".

For this reason, it is strongly recommended to not use a value much lower than Passlib's default, and to use one of *recommended hashes*, as one of their chief qualifying features is the mere *existence* of rounds values which take a short enough amount of time, and yet are still considered secure.

Todo: Expand this section into a full document, including information from the following posts:

- http://stackoverflow.com/questions/13545677/python-passlib-what-is-the-best-value-for-rounds
- http://stackoverflow.com/questions/11829602/pbkdf2-and-hash-comparison

As well as maybe JS-interactive calculation helper.

---

[1] For Passlib 1.6.3, all hashes were retuned to take ~300ms on a system with a 3.0 ghz 64 bit CPU.

2.5 CryptContext Tutorial

2.5.1 Overview

The *passlib.context* module contains one main class: passlib.context.CryptContext. This class is designed to take care of many of the more frequent coding patterns which occur in applications that need to handle multiple password hashes at once:

- identifying the algorithm used by a hash, and then verify a password.
- configure the default algorithm, load in support for new algorithms, deprecate old ones, set defaults for time-cost parameters, etc.
- migrate hashes / re-hash passwords when an algorithm has been deprecated.
- load said configuration from a sysadmin configurable file.

The following sections contain a walkthrough of this class, starting with some simple examples, and working up to a complex "full-integration" example.

See also:

The *passlib.context* api reference, which lists all the options and methods supported by this class.

2.5.2 Walkthrough Outline

Todo: This tutorial doesn't yet cover the *User Categories* system; and a few other parts could use elaboration.

2.5.3 Basic Usage

At its base, the CryptContext class is just a collection of *PasswordHash* objects, imported by name from the *passlib.hash* module. The following snippet creates a new context object which supports three hash algorithms (*sha256_crypt*, *md5_crypt*, and *des_crypt*):

```
>>> from passlib.context import CryptContext
>>> myctx = CryptContext(schemes=["sha256_crypt", "md5_crypt", "des_crypt"])
```

This new object exposes a very similar set of methods to the PasswordHash interface, and hashing and verifying passwords is equally as straightforward:

```
>>> # this loads first algorithm in the schemes list (sha256_crypt),
>>> # generates a new salt, and hashes the password:
>>> hash1 = myctx.hash("joshua")
>>> hash1
```

(continues on next page)

(continued from previous page)

```
'$5$rounds=80000$HFEGd1wnFknpibRl$VZqjyYcTenv7CtOf986hxuE0pRaGXnuLXyfb7m9xL69
↪'

>>> # when verifying a password, the algorithm is identified automatically:
>>> myctx.verify("gtnw", hash1)
False
>>> myctx.verify("joshua", hash1)
True

>>> # alternately, you can explicitly pick one of the configured algorithms,
>>> # through this is rarely needed in practice:
>>> hash2 = myctx.hash("dogsnamehere", scheme="md5_crypt")
>>> hash2
'$1$e2nig/AC$stejMS1ek6W0/UogYKFao/'

>>> myctx.verify("letmein", hash2)
False
>>> myctx.verify("dogsnamehere", hash2)
True
```

If not told otherwise, the context object will use the first algorithm listed in `schemes` when creating new hashes. This default can be changed by using the `default` keyword:

```
>>> myctx = CryptContext(schemes=["sha256_crypt", "md5_crypt", "des_crypt"],
                         default="des_crypt")
>>> hash = myctx.hash("password")
>>> hash
'bIwNofDzt1LCY'

>>> myctx.identify(hash)
'des_crypt'
```

This concludes the basics of how to use a CryptContext object. The rest of the sections detail the various features it offers, which probably provide a better argument for *why* you'd want to use it.

See also:

- the *CryptContext.hash()*, *verify()*, and *identify()* methods.
- the *schemes* and *default* constructor options.

2.5.4 Using Default Settings

While creating and verifying hashes is useful enough, it's not much more than could be done by importing the objects into a list. The next feature of the `CryptContext` class is that it can store various customized settings for the different algorithms, instead of hardcoding them into each `hash()` call. As an example, the `sha256_crypt` algorithm supports a `rounds` parameter which defaults to 80000, and the `ldap_salted_md5` algorithm uses 8-byte salts by default:

```
>>> from passlib.context import CryptContext
>>> myctx = CryptContext(["sha256_crypt", "ldap_salted_md5"])

>>> # sha256_crypt using 80000 rounds...
>>> myctx.hash("password", scheme="sha256_crypt")
'$5$rounds=80000$GgU/gwNBs9SaObqs$ohY23/zm.8O0TpkGx5fxk0aeVdFpaeKo9GUkMJ0VrMC
↪'
```

(continues on next page)

(continued from previous page)

```
           ^^^^^

>>> # ldap_salted_md5 with an 8 byte salt...
>>> myctx.hash("password", scheme="ldap_salted_md5")
'{SMD5}cIYrPh5f/TeUKg9oghECB5fSeu8='
       ^^^^^^^^^^
```

Instead of having to pass `rounds=91234` or `salt_size=16` every time `encrypt()` is called, CryptContext supports setting algorithm-specific defaults which will be used every time a CryptContext method is invoked. These is done by passing the CryptContext constructor a keyword with the format *scheme__setting*:

```
>>> # this reconfigures the existing context object so that
>>> # sha256_crypt now uses 91234 rounds,
>>> # and ldap_salted_md5 will use 16 byte salts:
>>> myctx.update(sha256_crypt__default_rounds=91234,
...              ldap_salted_md5__salt_size=16)

>>> # the effect of this can be seen the next time encrypt is called:
>>> myctx.hash("password", scheme="sha256_crypt")
'$5$rounds=91234$GgU/gwNBs9SaObqs$ohY23/zm.8O0TpkGx5fxk0aeVdFpaeKo9GUkMJ0VrMC
↪'
           ^^^^^

>>> myctx.hash("password", scheme="ldap_salted_md5")
'{SMD5}NnQh2S2pjnFxwtMhjbVH59TaG6P0/l/r3RsDwPj/n/M='
       ^^^^^^^^^^^^^^^^^^^^
```

See also:

- the *CryptContext.update()* method.
- the *default_rounds* and *per-scheme setting* constructor options.

2.5.5 Loading & Saving a CryptContext

The previous example built up a `CryptContext` instance in two stages, first by calling the constructor, and then the `update()` method to make some additional changes. The same configuration could of course be done in one step:

```
>>> from passlib.context import CryptContext
>>> myctx = CryptContext(schemes=["sha256_crypt", "ldap_salted_md5"],
...                      sha256_crypt__default_rounds=91234,
...                      ldap_salted_md5__salt_size=16)
```

This is not much more useful, since these settings still have to be hardcoded somewhere in the application. This is where the CryptContext's serialization abilities come into play. As a starting point, every CryptContext object can dump its configuration as a dictionary suitable for passing back into its constructor:

```
>>> myctx.to_dict()
{'schemes': ['sha256_crypt', 'ldap_salted_md5'],
'ldap_salted_md5__salt_size': 16,
'sha256_crypt__default_rounds': 91234}
```

However, this has been taken a step further, as CryptContext objects can also dump their configuration into a ConfigParser-compatible string, allowing the configuration to be written to a file:

```
>>> cfg = print myctx.to_string()
>>> print cfg
[passlib]
schemes = sha256_crypt, ldap_salted_md5
ldap_salted_md5__salt_size = 16
sha256_crypt__default_rounds = 912345
```

This "INI" format consists of a section named "`[passlib]`", following by key/value pairs which correspond exactly to the CryptContext constructor keywords (Keywords which accepts lists of names (such as `schemes`) are automatically converted to/from a comma-separated string) This format allows CryptContext configurations to be created in a separate file (say as part of an application's larger config file), and loaded into the CryptContext at runtime. Such strings can be loaded directly when creating the context object:

```
>>> # using the special from_string() constructor to
>>> # load the exported configuration created in the previous step:
>>> myctx2 = CryptContext.from_string(cfg)

>>> # or it can be loaded from a local file:
>>> myctx3 = CryptContext.from_path("/some/path/on/local/system")
```

This allows applications to completely extract their password hashing policies from the code, and into a configuration file with other security settings.

Note: For CryptContext instances which already exist, the *`load()`* and *`load_path()`* methods can be used to replace the existing state.

See also:

- the *`to_dict()`* and *`to_string()`* methods.
- the *`CryptContext.from_string()`* and *`CryptContext.from_path()`* constructors.

2.5.6 Deprecation & Hash Migration

The final and possibly most useful feature of the *`CryptContext`* class is that it can take care of deprecating and migrating existing hashes, re-hashing them using the current default algorithm and settings. All that is required is that a few settings be added to the configuration, and that the application call one extra method whenever a user logs in.

2.5.6.1 Deprecating Algorithms

The first setting that enables the hash migration features is the `deprecated` setting. This should be a list algorithms which are no longer desirable to have around, but are included in `schemes` to provide legacy support. For example:

```
>>> # this sets a context that supports 3 algorithms, but considers
>>> # two of them (md5_crypt and des_crypt) to be deprecated...
>>> from passlib.context import CryptContext
>>> myctx = CryptContext(schemes=["sha256_crypt", "md5_crypt", "des_crypt"],
                         deprecated=["md5_crypt", "des_crypt"])
```

All of the basic methods of this object will behave normally, but after an application has verified the user entered the correct password, it can check to see if the hash has been deprecated using the *needs_update()* method:

```
>>> # assume the user's password was stored as a sha256_crypt hash,
>>> # needs_update will show that the hash is still allowed.
>>> hash = '$5$rounds=80000$zWZFpsA2egmQY8R9$xp89Vvg1HeDCJ/
↪bTDDN6qkdsCwcMM61vHtM1RNxXur.'
>>> myctx.needs_update(hash)
False

>>> # but if the user's password was stored as md5_crypt hash,
>>> # need_update will indicate that it is deprecated,
>>> # and that the original password needs to be re-hashed...
>>> hash = '$1$fmWm78VW$uWjT69xZNMHWyEQjq852d1'
>>> myctx.needs_update(hash)
True
```

Note: Internally, this is not the only thing `needs_update()` does. It also checks for other issues, such as rounds / salts which are known to be weak under certain algorithms, improperly encoded hash strings, and other configurable behaviors that are detailed later.

2.5.6.2 Integrating Hash Migration

To summarize the process described in the previous section, all the actions an application would usually need to perform can be combined into the following bit of skeleton code:

```
hash = get_hash_from_user(user)
if pass_ctx.verify(password, hash):
    if pass_ctx.needs_update(hash):
        new_hash = pass_ctx.hash(password)
        replace_user_hash(user, new_hash)
    do_successful_things()
else:
    reject_user_login()
```

Since this is a very common pattern, the CryptContext object provides a shortcut: the *verify_and_update()* method, which allows replacing the above skeleton code with the following that uses 2 fewer calls (and is much more efficient internally):

```
hash = get_hash_from_user(user)
valid, new_hash = pass_ctx.verify_and_update(password, hash)
if valid:
    if new_hash:
        replace_user_hash(user, new_hash)
    do_successful_things()
else:
    reject_user_login()
```

2.5.6.3 Settings Rounds Limitations

In addition to deprecating entire algorithms, the deprecations system also allows you to place limits on algorithms that support the variable time-cost parameter `rounds`:

As an example, take a typical system containing a number of user passwords, all stored using *sha256_crypt*. As computers get faster, the minimum number of rounds that should be used gets larger, yet the existing passwords will remain in the system hashed using their original value. To solve this, the CryptContext object lets you place minimum bounds on what `rounds` values are allowed, using the *scheme*`__min_rounds` set of keywords... any hashes whose rounds are outside this limit are considered deprecated, and in need of re-encoding using the current policy:

First, we set up a context which requires all `sha256_crypt` hashes to have at least 131072 rounds:

```
>>> from passlib.context import CryptContext
>>> myctx = CryptContext(schemes="sha256_crypt",
...                      sha256_crypt__min_rounds=131072)
```

New hashes generated by this context will always honor the minimum (just as if `default_rounds` was set to the same value):

```
>>> # plain call to encrypt:
>>> hash1 = myctx.hash("password")
'$5$rounds=131072$i6xuFK6j8r66ahGn$r.
↪7H8HUk30qiH7fIWRJFJfhWG925nRZh90aYPMdewr3'
           ^^^^^^
>>> # hashes with enough rounds won't show up as deprecated...
>>> myctx.needs_update(hash1)
False
```

If an existing hash below the minimum is tested, it will show up as needing rehashing:

```
>>> # this has only 80000 rounds:
>>> hash3 = '$5$rounds=80000$qoCFY.akJr.flB7V
↪$8cIZXLwSTzuCRLcJbgHlxqYKEK0cVCENy6nFIlROj05'
>>> myctx.needs_update(hash3)
True

>>> # and verify_and_update() will upgrade this hash automatically:
>>> myctx.verify_and_update("wrong", hash3)
(False, None)
>>> myctx.verify_and_update("password", hash3)
(True, '$5$rounds=131072$rnMqBaemVZ6QGu7v
↪$vrAVQLEbsBoxhgem8ynvAbToCae8vpzl6ZuDS3/adlA')
                  ^^^^^^
```

See also:

- the *deprecated*, *min_rounds*, and *max_rounds* constructor options.
- the *needs_update()* and *verify_and_update()* methods.

2.5.7 Undocumented Features

Todo: Document usage of the *Disabled Hash Managment* options.

2.5.8 Full Integration Example

The following is an extended example showing how to fully interface a CryptContext object into your application. The sample configuration is somewhat more ornate that would usually be needed, just to

highlight some features, but should none-the-less be secure.

2.5.8.1 Policy Configuration File

The first thing to do is setup a configuration string for the CryptContext to use. This can be a dictionary or string defined in a python config file, or (in this example), part of a large INI-formatted config file. All of the documented *Context Options* are allowed.

```
; the options file uses the INI file format,
; and passlib will only read the section named "passlib",
; so it can be included along with other application configuration.

[passlib]

; setup the context to support pbkdf2_sha256, and some other hashes:
schemes = pbkdf2_sha256, sha512_crypt, sha256_crypt, md5_crypt, des_crypt

; flag md5_crypt and des_crypt as deprecated
deprecated = md5_crypt, des_crypt

; set boundaries for the pbkdf2 rounds parameter
; (pbkdf2 hashes outside this range will be flagged as needs-updating)
pbkdf2_sha256__min_rounds = 10000
pbkdf2_sha256__max_rounds = 50000

; set the default rounds to use when hashing new passwords.
pbkdf2_sha1__default_rounds = 15000

; applications can choose to treat certain user accounts differently,
; by assigning different types of account to a 'user category',
; and setting special policy options for that category.
; this create a category named 'admin', which will have a larger default
; rounds value.
admin__pbkdf2_sha1__min_rounds = 18000
admin__pbkdf2_sha1__default_rounds = 20000
```

2.5.8.2 Initializing the CryptContext

Applications which choose to use a policy file will typically want to create the CryptContext at the module level, and then load the configuration once the application starts:

1. Within a common module in your application (e.g. `myapp.model.security`):

```
#
# create a crypt context that can be imported and used wherever is␣
↪needed...
# the instance will be configured later.
#
from passlib.context import CryptContext
user_pwd_context = CryptContext()
```

2. Within some startup function within your application:

```
#
# when the app starts, import the context from step 1 and
```

(continues on next page)

(continued from previous page)

```
# configure it... such as by loading a policy file (see above)
#

from myapp.model.security import user_pwd_context

def myapp_startup():

    #
    # ... other code ...
    #

    #
    # load configuration from some application-specified path
    # using load_path() ... or use the load() method, which can
    # load a dict or in-memory string containing the INI file.
    #
    ##user_pwd_context.load(policy_config_string)
    user_pwd_context.load_path(policy_config_path)

    #
    # if you want to reconfigure the context without restarting the␣
↪application,
    # simply repeat the above step at another point.
    #

    #
    # ... other code ...
    #
```

2.5.8.3 Encrypting New Passwords

When it comes time to create a new user's password, insert the following code in the correct function:

```
from myapp.model.security import user_pwd_context

def handle_user_creation():

    #
    # ... other code ...
    #

    # vars:
    #   'secret' containing the putative password
    #   'category' containing a category assigned to the user account
    #

    hash = user_pwd_context.hash(secret, category=category)

    #... perform appropriate actions to store hash...

    #
    # ... other code ...
    #
```

Note: In the above code, the 'category' kwd can be omitted entirely, *OR* set to a string matching a

user category specified in the policy file. In the latter case, any category-specific policy settings will be enforced.

For the purposes of this example (and the sample config file listed above), it's assumed this value will be `None` for most users, and `"admin"` for special users. This namespace is entirely up to the application, it just has to match the category names used in the config file.

See *User Categories* for more details.

2.5.8.4 Verifying & Migrating Existing Passwords

Finally, when it comes time to check a users' password, insert the following code at the correct place:

```
from myapp.model.security import user_pwd_context

def handle_user_login():

    #
    # ... other code ...
    #

    #
    # this example both checks the user's password AND upgrades deprecated␣
↪hashes...
    #
    # vars:
    #   'hash' containing the specified user's hash.
    #   'secret' containing the putative password
    #   'category' containing a category assigned to the user account
    #
    # NOTE: if the user account is missing, or has no hash,
    #       you can pass ``hash=None`` to verify_and_update()
    #       mask this from the attacker by simulating the delay
    #       a real verification would have taken.
    #       hash=None will never verify.

    ok, new_hash = user_pwd_context.verify_and_update(secret, hash,␣
↪category=category)
    if not ok:
        # ... password did not match. do mean things ...
        pass

    else:
        #... password matched ...

        if new_hash:
            # old hash was deprecated by policy.

            # ... replace hash w/ new_hash for user account ...
            pass

        # ... do successful login actions ...
```

2.6 TOTP Tutorial

2.6.1 Overview

The *passlib.totp* module provides a set of classes for adding two-factor authentication (2FA) support into your application, using the widely supported TOTP specification (**RFC 6238**).

This module is based around the *TOTP* class, which supports a wide variety of use-cases, including:

- Creating & transferring configured TOTP keys to client devices.
- Generating & verifying tokens.
- Securely storing configured TOTP keys.

See also:

The *passlib.totp* API reference, which lists all details of all the classes and methods mentioned here.

2.6.2 Walkthrough

There are a number of different ways to integrate TOTP support into a server application. The following is a general outline of one of way to do this. Some details and alternate choices are omitted for brevity, see the remaining sections of this tutorial for more detailed information about these steps.

2.6.2.1 1. Generate an Application Secret

First, generate a strong application secret to use when encrypting TOTP keys for storage. Passlib offers a *generate_secret()* method to help with this:

```
>>> from passlib.totp import generate_secret
>>> generate_secret()
'pO7SwEFcUPvIDeAJr7INBj0TjsSZJr1d2ddsFL9r5eq'
```

This key should be assigned a numeric tag (e.g. "1", a timestamp, or an iso date such as "2016-11-10"); and should be stored in a file *separate* from your application's configuration. Ideally, after this file has been loaded by the TOTP constructor below, the application should give up access permissions to the file.

Example file contents:

```
2016-11-10: pO7SwEFcUPvIDeAJr7INBj0TjsSZJr1d2ddsFL9r5eq
```

This key will be used in a later step to encrypt TOTP keys for storage in your database. The sequential tag is used so that if your database (or the application secrets) are ever compromised, you can add a new application secret (with a newer tag), and gracefully migrate the compromised TOTP keys.

See also:

For more details see *Application Secrets* (below).

2.6.2.2 2. TOTP Factory Initialization

When your application is being initialized, create a TOTP factory which is configured for your application, and is set up to use the application secrets defined in step 1. You can also set a default issuer here, instead of having to provide one explicitly in step 4:

```
>>> from passlib.totp import TOTP
>>> TotpFactory = TOTP.using(secrets_path='/path/to/secret/file/in/step/1',
...                          issuer="myapp.example.org")
```

The `TotpFactory` object returned by *`TOTP.using()`* is actually a subclass of *`TOTP`* itself, and has the same methods and attributes. The main difference is that (because an application secret has been provided), the TOTP key will automatically be encrypted / decrypted when serializing the object to disk.

See also:

For more details see *Creating TOTP Instances* (below).

2.6.2.3 3. Rate-Limiting & Cache Initialization

As part of your application initialization, it **critically important** to set up infrastructure to rate limit how many token verification attempts a user / ip address is allowed to make, otherwise TOTP can be bypassed.

See also:

For more details see *Why Rate-Limiting is Critical* (below)

It's also **strongly recommended** to set up a per-user cache which can store the last matched TOTP counter (an integer) for a period of a few minutes (e.g. using dogpile.cache, memcached, redis, etc). This cache is used by later steps to protect your application during a narrow window of time where TOTP would otherwise be vulnerable to a replay attack.

See also:

For more details see *Preventing Token Reuse* (below)

2.6.2.4 4. Setting up TOTP for a User

To set up TOTP for a new user: create a new TOTP object and key using *`TOTP.new()`*. This can then be rendered into a provisioning URI, and transferred to the user's TOTP client of choice.

Rendering to a provisioning URI using *`TOTP.to_uri()`* requires picking an "issuer" string to uniquely identify your application, and a "label" string to uniquely identify the user. The following example creates a new TOTP instance with a new key, and renders it to a URI, plugging in application-specific information.

Using the `TotpFactory` object set up in step 2:

```
>>> totp = TotpFactory.new()
>>> uri = totp.to_uri(issuer="myapp.example.org", label="username")
>>> uri
'otpauth://totp/username?secret=D6RZI4ROAUQKJNAWQKYPN7W7LNV43GOT&
↪issuer=myapp.example.org'
```

This URI is generally passed to a QRCode renderer, though as fallback it's recommended to also display the key using *`TOTP.pretty_key()`*.

See also:

For more details, and more about QR Codes, see *Configuring Clients* (below).

2.6.2.5 5. Storing the TOTP object

Before enabling TOTP for the user's account, it's good practice to first have the user successfully verify a token (per step 6); thus confirming their client h as been correctly configured.

Once this is done, you can store the TOTP object in your database. This can be done via the *TOTP.to_json()* method:

```
>>> totp.to_json()
'{"enckey":{"c":14,"k":"FLEQC3VO6SIT3T7GN2GIG6ONPXADG5CZ","s":
↪"UL2J4MZG4SONHOWXLKFQ","t":"1","v":1},"type":"totp","v":1}'
```

Note that if there is no application secret configured, the key will not be encrypted, and instead look like this:

```
>>> totp.to_json()
'{"key":"D6RZI4ROAUQKJNAWQKYPN7W7LNV43GOT","type":"totp","v":1}'
```

To ensure you always save an encrypted token, you can use `totp.to_json(encrypted=True)`.

See also:

For more details see *Storing TOTP instances*

2.6.2.6 6. Verifying a Token

Whenever attempting to verify a token provided by the user, first load the serialized TOTP object from the database (stored step 5), as well as the last counter value from the cache (set up in step 3). You should use these values to call the *TOTP.verify()* method.

If verify() succeeds, it will return a *TotpMatch* object. This object contains information about the match, including *TotpMatch.counter* (a time-dependant integer tied to this token), and *TotpMatch.cache_seconds* (minimum time this counter should be cached).

If verify() fails, it will raise one of the *passlib.exc.TokenError* subclasses indicating what went wrong. This will be one of three cases: the token was malformed (e.g. too few digits), the token was invalid (didn't match), or a recent token was reused.

A skeleton example of how this should function:

```
>>> from passlib.exc import TokenError, MalformedTokenError

>>> # pull information from your application
>>> token = # ... token string provided by user ...
>>> source = # ... load totp json string from database ...
>>> last_counter = # ... load counter value from cache ...

>>> # ... check attempt rate limit for this account / address (per step 3␣
↪above) ...

>>> # using the TotpFactory object defined in step 2, invoke verify
>>> try:
...     match = TotpFactory.verify(token, source, last_counter=last_counter)
... except MalformedTokenError as err:
...     # --- malformed token ---
...     # * inform user, e.g. by displaying str(err)
... except TokenError as err:
...     # --- invalid or reused token ---
...     # * add to rate limit counter
...     # * inform user, e.g. by displaying str(err)
... else:
...     # --- successful match ---
```

(continues on next page)

(continued from previous page)

```
...     # * reset rate-limit counter
...     # * store 'match.counter' in per-user cache for at least 'match.
↪cache_seconds'
```

See also:

For more details see *Verifying Tokens* (below)

Alternate Caching Strategy

As an alternative to storing `match.counter` in the cache, applications using a cache such as memcached may wish to simply set a key based on `user + token` for `match.cache_seconds`, and reject any tokens coming in for that user who are marked in the cache.

In that case, they should run the tokens through *TOTP.normalize_token()* first, to make sure the token strings are normalized before comparison. In this case, the skeleton example can be amended to:

```
>>> # pull information from your application
>>> token = # ... token string provided by user ...
>>> source = # ... load totp json string from database ...
>>> user_id = # ... user identifier for cache

>>> # ... check attempt rate limit for this account / address (per step 3␣
↪above) ...

>>> # check token format
>>> try:
...     token = TotpFactory.normalize_token(token)
... except MalformedTokenError as err:
...     # --- malformed token ---
...     # * inform user, e.g. by displaying str(err)
...     return

>>> # check if token has been used, using app-defined present_in_cache()␣
↪helper
>>> cache_key = "totp-token-%s-%s" % (user_id, token)
>>> if present_in_cache(cache_key):
...     # * add to rate limit counter
...     # * present 'token already used' message
...     return

>>> # using the TotpFactory object defined in step 2, invoke verify
>>> try:
...     match = TotpFactory.verify(token, source)
... except TokenError as err:
...     # --- invalid token ---
...     # * add to rate limit counter
...     # * inform user, e.g. by displaying str(err)
... else:
...     # --- successful match ---
...     # * reset rate-limit counter
...     # * set 'cache_key' in per-user cache for at least 'match.cache_
↪seconds'
```

2.6.2.7 7. Reserializing Existing Objects

An organization's security policy may require that a developer periodically change the application secret key used to decrypt/encrypt TOTP objects. Alternately, the application secret may become compromised.

In either case, a new application secret will need to be created, and a new tag assigned (per step 1). Any deprecated secret(s) will need to be retained in the collection passed to the `TotpFactory`, in order to be able to decrypt existing TOTP objects.

Note: You can verify which secret is will be used to encrypt new keys by inspecting `tag = TotpFactory.wallet.default_tag`.

Once the new secret has been added, you will need to update all the serialized TOTP objects in the database, decrypting them using the old secret, and encrypting them with the new one.

This can be done in a few ways. The following skeleton example gives a simple loop that can be used, which would ideally be run in a process that's separate from your normal application:

```
>>> # presuming query_user_totp() queries your database for all user rows,
>>> # and update_user_totp() updates a specific row.
>>> for user_id, totp_source in query_user_totp():
>>>     totp = TotpFactory.from_source(totp_source)
>>>     if totp.changed:
>>>         update_user_totp(user_id, totp.to_json())
```

This uses the *TOTP.changed* attribute, which is set to `True` if *TOTP.from_source()* (or other constructor) detects the source data is encrypted with an old secret, is using outdated encryption settings, or is stored in deprecated serialization format.

Some refinements that may need to be made for specific situations:

- For applications with a large number of users, it may be faster to accumulate `(user_id, totp.to_json())` pairs in a buffer, and do a bulk SQL update once every 100-1000 rows.
- Depending on the dbapi layer in use, it may take care of JSON serialization for you, in which case you'll need to use `totp.to_dict()` instead of `totp.to_json()`.

Once all references to a deprecated secret have been replaced, it can be removed from the secrets file.

See also:

For more details see *Step 1* (above), or *Application Secrets* (below)

2.6.3 Creating TOTP Instances

2.6.3.1 Direct Creation

Creating TOTP instances is straightforward: The *TOTP* class can be called directly to constructor a TOTP instance from it's component configuration:

```
>>> from passlib.totp import TOTP
>>> totp = TOTP(key='GVDOQ7NP6XPJWE4CWCLFFSXZH6DTAZWM', digits=9)
>>> totp.generate()
'29387414'
```

You can also use a number of the alternate constructors, such as *TOTP.new()* or *TOTP.from_source()*:

```
>>> # create new instance w/ automatically generated key
>>> totp = TOTP.new()

>>> # or deserializing it from a string (e.g. the output of TOTP.to_json)
>>> totp = TOTP.from_source('{"key":"D6RZI4ROAUQKJNAWQKYPN7W7LNV43GOT","type
↪":"totp","v":1}')
```

Once created, you can inspect the object for it's configuration and key:

```
>>> otp.base32_key
'GVDOQ7NP6XPJWE4CWCLFFSXZH6DTAZWM'
>>> otp.alg
"sha1"
>>> otp.period
30
```

If you want a non-standard alg or period, you can specify it via the constructor. You can also create TOTP instances from an existing key (see the *TOTP* constructor's `key` and `format` options for more details):

```
>>> otp2 = TOTP(new=True, period=60, alg="sha256")
>>> otp2.alg
'sha256'
>>> otp2.period
60

>>> otp3 = TOTP(key='GVDOQ7NP6XPJWE4CWCLFFSXZH6DTAZWM')
```

2.6.3.2 Using a Factory

Most applications will have some default configuration which they want all TOTP instances to have. This includes application secrets (for encrypting TOTP keys for storage), or setting a default issuer label (for rendering URIs).

Instead of having to call the *TOTP* constructor each time and provide all these options, you can use the *TOTP.using()* method. This method takes in a number of the same options as the TOTP constructor, and returns a *TOTP* subclass which has these options pre-programmed in as defaults:

```
>>> # here we create a TOTP factory with a random encryption secret and a␣
↪default issuer
>>> from passlib.totp import TOTP, generate_secret
>>> TotpFactory = TOTP.using(issuer="myapp.example.org", secrets={"1":␣
↪generate_secret()})
```

Since this object is a subclass of *TOTP*, you can use all it's normal methods. The difference is that it will integrate the information provided by using():

```
>>> totp = TotpFactory.new()
>>> totp.issuer
'myapp.example.org'

>>> totp.to_json()
'{"enckey":{"c":14,"k":"FLEQC3VO6SIT3T7GN2GIG6ONPXADG5CZ","s":
↪"UL2J4MZG4SONHOWXLKFQ","t":"1","v":1},"type":"totp","v":1}'
```

In typical usage, a server application will want to create a TotpFactory as part of it's initialization, and then use that class for all operations, instead of referencing *TOTP* directly.

See also:

- *Configuring Clients* for details about the `issuer` option
- *Storing TOTP instances* for details about storage and key encryption

2.6.4 Configuring Clients

Once a TOTP instance & key has been generated on the server, it needs to be transferred to the client TOTP program for installation. This can be done by having the user manually type the key into their TOTP client, but an easier method is to render the TOTP configuration to a URI stored in a QR Code.

2.6.4.1 Rendering URIs

The KeyUriFormat is a de facto standard for encoding TOTP keys & configuration information into a string. Once the URI is rendered as a QR Code, it can easily be imported into many smartphone clients (such as Authy and Google Authenticator) via the smartphone's camera.

When transferring the TOTP configuration this way, you will need to provide unique identifiers for both your application, and the user's account. This allows TOTP clients to distinguish this key from the others in it's database. This can be done via the `issuer` and `label` parameters of the *`TOTP.to_uri()`* method.

The `issuer` string should be a globally unique label for your application (e.g. it's domain name). Since the issuer string shouldn't change across users, you can create a customized TOTP factory, and provide it with a default issuer. *(If you skip this step, the issuer will need to be provided at every `TOTP.to_uri()` call)*:

```
>>> from passlib.totp import TOTP
>>> TotpFactory = TOTP.using(issuer="myapp.example.org")
```

Once this is done, rendering to a provisioning URI just requires picking a `label` for the URI. This label should identify the user within your application (e.g. their login or their email):

```
>>> # assume an existing TOTP instance has been created
>>> totp = TotpFactory.new()

>>> # serialize the object to a URI, along with label for user
>>> uri = totp.to_uri(label="demo-user")
>>> uri
'otpauth://totp/demo-user?secret=GVDOQ7NP6XPJWE4CWCLFFSXZH6DTAZWM&
↪issuer=myapp.example.org'
```

2.6.4.2 Rendering QR Codes

This URI can then be encoded as a QR Code, using various python & javascript qrcode libraries. As an example, the following uses PyQrCode to render the URI to the console as a text-based QR code:

```
>>> import pyqrcode
>>> uri = totp.to_uri(label="demo-user")
>>> print(pyqrcode.create(uri).terminal(quiet_zone=1))
... very large ascii-art qrcode here...
```

As a fallback to the QR Code, it's recommended to alternately / also display the key itself, so that users with camera-less TOTP clients can still enter it. The *TOTP.pretty_key()* method is provided to help with this:

```
>>> totp.pretty_key()
'D6RZ-I4RO-AUQK-JNAW-QKYP-N7W7-LNV4-3GOT'
```

Note that if you use a non-default `alg`, `digits`, or `period` values, these should also be displayed next to the key.

2.6.4.3 Parsing URIs

On the client side, passlib offers the *TOTP.from_uri()* constructor creating a TOTP object from a provisioning URI. This can also be useful for testing URI encoding & output during development:

```
>>> # create new TOTP instance from a provisioning uri:
>>> from passlib.totp import TOTP
>>> totp = TOTP.from_uri('otpauth://totp/demo-user?
↪secret=GVDOQ7NP6XPJWE4CWCLFFSXZH6DTAZWM&issuer=myapp.example.org')
>>> otp.base32_key
'GVDOQ7NP6XPJWE4CWCLFFSXZH6DTAZWM'
>>> otp.alg
"sha1"
>>> otp.period
30
>>> otp.generate().token
'897453'
```

2.6.5 Storing TOTP instances

Once a TOTP object has been created, it inevitably needs to be stored in a database. Using *to_uri()* to serialize it to a URI has a few disadvantages - it always includes an issuer & a label (wasting storage space), and it stores the key in an unencrypted format.

2.6.5.1 JSON Serialization

To help with this passlib offers a way to serialize TOTP objects to and from a simple JSON format, which can optionally encrypt the keys for storage.

To serialize a TOTP object to a string, use *TOTP.to_json()*:

```
>>> from passlib.totp import TOTP
>>> totp = TOTP.new()
>>> data = totp.to_json()
>>> data
'{"key":"GVDOQ7NP6XPJWE4CWCLFFSXZH6DTAZWM","type":"totp","v":1}'
```

This string can be stored in a database, and then deserialized as needed using the *TOTP.from_json()* constructor:

```
>>> totp2 = TOTP.from_json(data)
>>> totp2.base32_key
'GVDOQ7NP6XPJWE4CWCLFFSXZH6DTAZWM'
```

There are also corresponding *TOTP.to_dict()* and *TOTP.from_dict()* methods for applications that want to serialize the object without converting it all the way into a JSON string.

Caution: The above procedure should only be used for development purposes, as it will NOT encrypt the keys; and the IETF **strongly recommends** encrypting the keys for storage (RFC-6238 sec 5.1). Encrypting the keys is covered below.

2.6.5.2 Application Secrets

The one thing lacking about the example above is that the resulting data contained the plaintext key. If the server were compromised, the TOTP keys could be used directly to impersonate the user. To solve this, Passlib offers a method for providing an application-wide secret that *TOTP.to_json()* will use to encrypt keys.

Per *Step 1* of the walkthrough (above), applications can use the *generate_secret()* helper to create new secrets. All existing secrets (the current one, and any deprecated / compromised ones) should be assigned an identifying tag, and stored in a dict or file.

Ideally, these secrets should be stored in a location which the application's process does not have access to once it has been initialized. Once this data is loaded, applications can create a factory function using *TOTP.using()*, and provide these secrets as part of it's arguments. This can take the form of a file path, a loaded string, or a dictionary:

```
>>> # load from dict
>>> from passlib.totp import TOTP
>>> TotpFactory = TOTP.using(secrets={"1": "
↪'pO7SwEFcUPvIDeAJr7INBj0TjsSZJr1d2ddsFL9r5eq'"})

>>> # load from filepath
>>> TotpFactory = TOTP.using(secrets_path="/path/to/secret/file")
```

The `secrets` and `secrets_path` values can be anything accepted by the *AppWallet* constructor (the internal class that's used to load & store the application secrets in memory). An instance of this object is accessible for inspection from the `TOTP.wallet` attribute of each factory:

```
>>> TotpFactory.wallet
<passlib.totp.AppWallet at 0x2ba5310>
```

2.6.5.3 Encrypting Keys

Once you have a TOTP factory configured with one or more application secrets, any objects you create through the factory will automatically have access to the application secrets, and will use them to encrypt the key when serializing to json.

Assuming `TotpFactory` is set up from the previous step, contrast the output of this with the plain JSON serialization example above:

```
>>> totp = TotpFactory.new()
>>> data = totp.to_json()
>>> data
'{"enckey":{"c":14,"k":"FLEQC3VO6SIT3T7GN2GIG6ONPXADG5CZ","s":
↪"UL2J4MZG4SONHOWXLKFQ","t":"1","v":1},"type":"totp","v":1}'
```

This data can be stored in the database like normal, but will require access to the application secret in order to decrypt:

```
>>> data = '{"enckey":{"c":14,"k":"FLEQC3VO6SIT3T7GN2GIG6ONPXADG5CZ","s":
↪"UL2J4MZG4SONHOWXLKFQ","t":"1","v":1},"type":"totp","v":1}'
>>> totp = TotpFactory.from_source(data)
>>> totp.base32_key
'FLEQC3VO6SIT3T7GN2GIG6ONPXADG5CZ'
```

Whereas trying to decode without a secret configured will result in:

```
>>> totp = TOTP.from_source(data)
...
TypeError: no application secrets present, can't decrypt TOTP key
```

Note that when loading TOTP objects this way, you can check the *TOTP.changed* attr to see if the object needs to be re-serialized (e.g. deprecated secret, too few encryption rounds, deprecated serialization format).

2.6.6 Generating Tokens (Client-Side Only)

Finally, the whole point of TOTP: generating and verifying tokens. The TOTP protocol generates a new time & key -dependant token every <period> seconds (usually 30).

Generating a totp token is done with the *TOTP.generate()* method, which returns a *TotpToken* instance. This object looks and acts like a tuple of `(token, expire_time)`, but offers some additional informational attributes:

```
>>> from passlib import totp
>>> otp = TOTP(key='GVDOQ7NP6XPJWE4CWCLFFSXZH6DTAZWM')

>>> # generate a TOTP token for the current timestamp
>>> # (your output will vary based on system time)
>>> otp.generate()
<TotpToken token='589720' expire_time=1475342400>

>>> # to get just the token, not the TotpToken instance...
>>> otp.generate().token
'359275'

>>> # you can generate a token for a specific time as well...
>>> otp.generate(time=1475338840).token
'359275'
```

See also:

For more details, see the *TOTP.generate()* method.

2.6.7 Verifying Tokens

In order for successful authentication, the user must generate the token on the client, and provide it to your server before the *TOTP.period* ends.

Since this there will always be a little transmission delay (and sometimes client clock drift) TOTP verification usually uses a small verification window, allowing a user to enter a token a few seconds after the period has ended. This window is usually kept as small as possible, and in passlib defaults to 30 seconds.

2.6.7.1 Match & Verify

To verify a token a user has provided, you can use the *TOTP.match()* method. If unsuccessful, a *passlib.exc.TokenError* subclass will be raised. If successful, this will return a *TotpMatch* instance, with details about the match. This object acts like a tuple of `(counter, timestamp)`, but offers some additional informational attributes:

```
>>> # NOTE: all of the following was done at a fixed time, to make these
>>> #       examples repeatable. in real-world use, you would omit the 'time
↪' parameter
>>> #       from all these calls.

>>> # assuming TOTP key & config was deserialized from database store
>>> from passlib import totp
>>> otp = TOTP(key='GVDOQ7NP6XPJWE4CWCLFFSXZH6DTAZWM')

>>> # user provides malformed token:
>>> otp.match('359', time=1475338840)
...
MalformedTokenError: Token must have exactly 6 digits

>>> # user provides token that isn't valid w/in time window:
>>> otp.match('123456', time=1475338840)
...
InvalidTokenError: Token did not match

>>> # user provides correct token
>>> otp.match('359275', time=1475338840)
<TotpMatch counter=49177961 time=1475338840 cache_seconds=60>
```

As a further optimization, the *TOTP.verify()* method allows deserializing and matching a token in a single step. Not only does this save a little code, it has a signature much more similar to that of Passlib's *passlib.ifc.PasswordHash.verify()*.

Typically applications will provide the TOTP key in whatever format it's stored by the server. This will usually be a JSON string (as output by *TOTP.to_json()*), but can be any format accepted by *TOTP.from_source()*. As an example:

```
>>> # application loads json-serialized TOTP key
>>> from passlib.totp import TOTP
>>> totp_source = '{"v": 1, "type": "totp", "key": "otxl2f5cctbprpzx"}'

>>> # parse & match the token in a single call
>>> match = TOTP.verify('123456', totp_source)
```

See also:

For more details, see the *TOTP.match()* and *TOTP.verify()* methods.

2.6.7.2 Preventing Token Reuse

Even if an attacker is able to observe a user entering a TOTP token, it will do them no good once `period + window` seconds have passed (typically 60). This is because the current time will now have advanced far enough that `TOTP.match()` will *never* match against the stolen token.

However, this leaves a small window in which the attacker can observe and replay a token, successfully impersonating the user. To prevent this, applications are strongly encouraged to record the latest *TotpMatch.counter* value that's returned by the `TOTP.match()` method.

This value should be stored per-user in a temporary cache for at least `period + window` seconds. (This is typically 60 seconds, but for an exact value, applications may check the *TotpMatch.cache_seconds* value returned by the `TOTP.match()` method).

Any subsequent calls to verify should check this cache, and pass in that value to `TOTP.match()`'s "last_counter" parameter (or `None` if no value found). Doing so will ensure that tokens can only be used once, preventing replay attacks.

As an example:

```
>>> # NOTE: all of the following was done at a fixed time, to make these
>>> #       examples repeatable. in real-world use, you would omit the 'time
↪' parameter
>>> #       from all these calls.

>>> # assuming TOTP key & config was deserialized from database store
>>> from passlib.totp import TOTP
>>> otp = TOTP(key='GVDOQ7NP6XPJWE4CWCLFFSXZH6DTAZWM')

>>> # retrieve per-user counter from cache
>>> last_counter = ...consult application cache...

>>> # if user provides valid value, a TotpMatch object will be returned.
>>> # (if they provide an invalid value, a TokenError will be raised).
>>> match = otp.match('359275', last_counter=last_counter, time=1475338830)
>>> match.counter
49177961
>>> match.cache_seconds
60

>>> # application should now cache the new 'match.counter' value
>>> # for at least 'match.cache_seconds'.

>>> # now that last_counter has been properly updated: say that
>>> # 10 seconds later attacker attempts to re-use token user just entered:
>>> last_counter = 49177961
>>> match = otp.match('359275', last_counter=last_counter, time=1475338840)
...
UsedTokenError: Token has already been used, please wait for another.
```

See also:

For more details, see the *TOTP.match()* method; for more examples, see Step 6 above.

2.6.7.3 Why Rate-Limiting is Critical

The *TOTP.match()* method offers a `window` parameter, expanding the search range to account for the client getting slightly out of sync.

While it's tempting to be user-friendly, and make this window as large as possible, there is a security downside: Since any token within the window will be treated as valid, the larger you make the window, the more likely it is that an attacker will be able to guess the correct token by random luck.

Because of this, **it's critical for applications implementing OTP to rate-limit the number of attempts on an account**, since an unlimited number of attempts guarantees an attacker will be able to guess any given token.

The Gory Details

For TOTP, the formula is `odds = guesses * (1 + 2 * window / period) / 10**digits`; where `window` in this case is the *TOTP.match()* window (measured in seconds), and `period` is the number of seconds before the token is rotated.

This formula can be inverted to give the maximum window we want to allow for a given configuration, rate limit, and desired odds: `max_window = floor((odds * 10**digits / guesses - 1) * period / 2)`.

For example (assuming TOTP with 7 digits and 30 second period), if you want an attacker's odds to be no better than 1 in 10000, and plan to lock an account after 4 failed attempts – the maximum window you should use would be `floor((1/10000 * 10**6 / 4 - 1) * 30 / 2)` or 360 seconds.

CHAPTER 3

API Reference

The reference section contains documentation of Passlib's public API. These chapters are focused on providing detailed reference of the individual functions and classes; they will generally be cross-linked to any related walkthrough documentation (which tries to provide a higher-level synthetic view).

Note: Primary modules:

The primary modules that will be of interest are:

- *passlib.hash*
- *passlib.context*
- *passlib.totp*
- *passlib.exc*

Caution: Internal modules:

The following modules are mainly used internally, may change structure between releases, and are documented mainly for completeness:

- *passlib.crypto*
- *passlib.registry*
- *passlib.utils*

Alphabetical module list:

3.1 passlib.apache - Apache Password Files

This module provides utilities for reading and writing Apache's htpasswd and htdigest files; though the use of two helper classes.

Changed in version 1.6: The api for this module was updated to be more flexible, and to have less ambiguous method names. The old method and keyword names are deprecated, and will be removed in Passlib 1.8.

Changed in version 1.7: These classes will now preserve blank lines and "#" comments when updating htpasswd files; previous releases would throw a parse error.

3.1.1 Htpasswd Files

The `HTpasswdFile` class allows managing of htpasswd files. A quick summary of its usage:

```
>>> from passlib.apache import HtpasswdFile

>>> # when creating a new file, set to new=True, add entries, and save.
>>> ht = HtpasswdFile("test.htpasswd", new=True)
>>> ht.set_password("someuser", "really secret password")
>>> ht.save()

>>> # loading an existing file to update a password
>>> ht = HtpasswdFile("test.htpasswd")
>>> ht.set_password("someuser", "new secret password")
>>> ht.save()

>>> # examining file, verifying user's password
>>> ht = HtpasswdFile("test.htpasswd")
>>> ht.users()
[ "someuser" ]
>>> ht.check_password("someuser", "wrong password")
False
>>> ht.check_password("someuser", "new secret password")
True

>>> # making in-memory changes and exporting to string
>>> ht = HtpasswdFile()
>>> ht.set_password("someuser", "mypass")
>>> ht.set_password("someuser", "anotherpass")
>>> print ht.to_string()
someuser:$apr1$T4f7D9ly$EobZDROnHblCNPCtrgh5i/
anotheruser:$apr1$vBdPWvh1$GrhfbyGvN/7HalW5cS9XB1
```

Warning: `HtpasswdFile` currently defaults to using `apr_md5_crypt`, as this is the only htpasswd hash guaranteed to be portable across operating systems. However, for security reasons Passlib 1.7 will default to using the strongest algorithm available on the host platform (e.g. `bcrypt` or `sha256_crypt`). Applications that are relying on the old behavior should specify `HtpasswdFile(default_scheme="portable")` (new in Passlib 1.6.3).

class passlib.apache.**HtpasswdFile**(*path=None*, *new=False*, *autosave=False*, ...)

class for reading & writing Htpasswd files.

The class constructor accepts the following arguments:

Parameters

- **path** (*filepath*) – Specifies path to htpasswd file, use to implicitly load from and save to.

  This class has two modes of operation:

  1. It can be "bound" to a local file by passing a `path` to the class constructor. In this case it will load the contents of the file when created, and the *load()* and *save()* methods will automatically load from and save to that file if they are called without arguments.
  2. Alternately, it can exist as an independant object, in which case *load()* and *save()* will require an explicit path to be provided whenever they are called. As well, `autosave` behavior will not be available.

     This feature is new in Passlib 1.6, and is the default if no `path` value is provided to the constructor.

  This is also exposed as a readonly instance attribute.

- **new** (*bool*) – Normally, if *path* is specified, *HtpasswdFile* will immediately load the contents of the file. However, when creating a new htpasswd file, applications can set `new=True` so that the existing file (if any) will not be loaded.

  New in version 1.6: This feature was previously enabled by setting `autoload=False`. That alias has been deprecated, and will be removed in Passlib 1.8

- **autosave** (*bool*) – Normally, any changes made to an *HtpasswdFile* instance will not be saved until *save()* is explicitly called. However, if `autosave=True` is specified, any changes made will be saved to disk immediately (assuming *path* has been set).

  This is also exposed as a writeable instance attribute.

- **encoding** (*str*) – Optionally specify character encoding used to read/write file and hash passwords. Defaults to `utf-8`, though `latin-1` is the only other commonly encountered encoding.

  This is also exposed as a readonly instance attribute.

- **default_scheme** (*str*) – Optionally specify default scheme to use when encoding new passwords.

  This can be any of the schemes with builtin Apache support, OR natively supported by the host OS's `crypt.crypt()` function.

  – Builtin schemes include `"bcrypt"` (apache 2.4+), `"apr_md5_crypt"`, and `"des_crypt"`.

  – Schemes commonly supported by Unix hosts include `"bcrypt"`, `"sha256_crypt"`, and `"des_crypt"`.

  In order to not have to sort out what you should use, passlib offers a number of aliases, that will resolve to the most appropriate scheme based on your needs:

  – `"portable"`, `"portable_apache_24"` – pick scheme that's portable across hosts running apache >= 2.4. **This will be the default as of Passlib 2.0**.

  – `"portable_apache_22"` – pick scheme that's portable across hosts running apache >= 2.4. **This is the default up to Passlib 1.9**.

  – **`"host"`, `"host_apache_24"` – pick strongest scheme supported by** apache >= 2.4 and/or host OS.

– **"host_apache_22" – pick strongest scheme supported by** apache >= 2.2 and/or host OS.

New in version 1.6: This keyword was previously named `default`. That alias has been deprecated, and will be removed in Passlib 1.8.

Changed in version 1.6.3: Added support for `"bcrypt"`, `"sha256_crypt"`, and `"portable"` alias.

Changed in version 1.7: Added apache 2.4 semantics, and additional aliases.

- **context** (*CryptContext*) – `CryptContext` instance used to create and verify the hashes found in the htpasswd file. The default value is a pre-built context which supports all of the hashes officially allowed in an htpasswd file.

  This is also exposed as a readonly instance attribute.

  Warning: This option may be used to add support for non-standard hash formats to an htpasswd file. However, the resulting file will probably not be usable by another application, and particularly not by Apache.

- **autoload** – Set to `False` to prevent the constructor from automatically loaded the file from disk.

  Deprecated since version 1.6: This has been replaced by the *new* keyword. Instead of setting `autoload=False`, you should use `new=True`. Support for this keyword will be removed in Passlib 1.8.

- **default** – Change the default algorithm used to hash new passwords.

  Deprecated since version 1.6: This has been renamed to *default_scheme* for clarity. Support for this alias will be removed in Passlib 1.8.

3.1.1.1 Loading & Saving

load(*path=None*, *force=True*)
Load state from local file. If no path is specified, attempts to load from `self.path`.

Parameters

- **path** (*str*) – local file to load from
- **force** (*bool*) – if `force=False`, only load from `self.path` if file has changed since last load.

  Deprecated since version 1.6: This keyword will be removed in Passlib 1.8; Applications should use *load_if_changed()* instead.

load_if_changed()
Reload from `self.path` only if file has changed since last load

load_string(*data*)
Load state from unicode or bytes string, replacing current state

save(*path=None*)
Save current state to file. If no path is specified, attempts to save to `self.path`.

to_string()
Export current state as a string of bytes

3.1.1.2 Inspection

users()
Return list of all users in database

check_password(*user*, *password*)
Verify password for specified user. If algorithm marked as deprecated by CryptContext, will automatically be re-hashed.

Returns

- `None` if user not found.
- `False` if user found, but password does not match.
- `True` if user found and password matches.

Changed in version 1.6: This method was previously called `verify`, it was renamed to prevent ambiguity with the `CryptContext` method. The old alias is deprecated, and will be removed in Passlib 1.8.

get_hash(*user*)
Return hash stored for user, or `None` if user not found.

Changed in version 1.6: This method was previously named `find`, it was renamed for clarity. The old name is deprecated, and will be removed in Passlib 1.8.

3.1.1.3 Modification

set_password(*user*, *password*)
Set password for user; adds user if needed.

Returns

- `True` if existing user was updated.
- `False` if user account was added.

Changed in version 1.6: This method was previously called `update`, it was renamed to prevent ambiguity with the dictionary method. The old alias is deprecated, and will be removed in Passlib 1.8.

delete(*user*)
Delete user's entry.

Returns

- `True` if user deleted.
- `False` if user not found.

3.1.1.4 Alternate Constructors

classmethod from_string(*data*, ***kwds*)
create new object from raw string.

Parameters

- **data** (*unicode or bytes*) – database to load, as single string.
- ****kwds** – all other keywords are the same as in the class constructor

3.1.1.5 Attributes

path
Path to local file that will be used as the default for all *load()* and *save()* operations. May be written to, initialized by the *path* constructor keyword.

autosave
Writeable flag indicating whether changes will be automatically written to *path*.

3.1.1.6 Errors

Raises **ValueError** – All of the methods in this class will raise a `ValueError` if any user name contains a forbidden character (one of `:\r\n\t\x00`), or is longer than 255 characters.

3.1.2 Htdigest Files

The `HtdigestFile` class allows management of htdigest files in a similar fashion to *HtpasswdFile*.

class passlib.apache.**HtdigestFile**(*path*, *default_realm=None*, *new=False*, *autosave=False*, ...)
class for reading & writing Htdigest files.

The class constructor accepts the following arguments:

Parameters

- **path** (*filepath*) – Specifies path to htdigest file, use to implicitly load from and save to.

  This class has two modes of operation:

  1. It can be "bound" to a local file by passing a `path` to the class constructor. In this case it will load the contents of the file when created, and the *load()* and *save()* methods will automatically load from and save to that file if they are called without arguments.
  2. Alternately, it can exist as an independant object, in which case *load()* and *save()* will require an explicit path to be provided whenever they are called. As well, `autosave` behavior will not be available.

     This feature is new in Passlib 1.6, and is the default if no `path` value is provided to the constructor.

  This is also exposed as a readonly instance attribute.

- **default_realm** (*str*) – If `default_realm` is set, all the *HtdigestFile* methods that require a realm will use this value if one is not provided explicitly. If unset, they will raise an error stating that an explicit realm is required.

  This is also exposed as a writeable instance attribute.

  New in version 1.6.

- **new** (*bool*) – Normally, if *path* is specified, *HtdigestFile* will immediately load the contents of the file. However, when creating a new htpasswd file, applications can set `new=True` so that the existing file (if any) will not be loaded.

  New in version 1.6: This feature was previously enabled by setting `autoload=False`. That alias has been deprecated, and will be removed in Passlib 1.8

- **autosave** (*bool*) – Normally, any changes made to an *HtdigestFile* instance will not be saved until *save()* is explicitly called. However, if `autosave=True` is specified, any changes made will be saved to disk immediately (assuming *path* has been set).

  This is also exposed as a writeable instance attribute.

- **encoding** (*str*) – Optionally specify character encoding used to read/write file and hash passwords. Defaults to `utf-8`, though `latin-1` is the only other commonly encountered encoding.

  This is also exposed as a readonly instance attribute.

- **autoload** – Set to `False` to prevent the constructor from automatically loaded the file from disk.

  Deprecated since version 1.6: This has been replaced by the *new* keyword. Instead of setting `autoload=False`, you should use `new=True`. Support for this keyword will be removed in Passlib 1.8.

3.1.2.1 Loading & Saving

load(*path=None*, *force=True*)
Load state from local file. If no path is specified, attempts to load from `self.path`.

Parameters

- **path** (*str*) – local file to load from
- **force** (*bool*) – if `force=False`, only load from `self.path` if file has changed since last load.

  Deprecated since version 1.6: This keyword will be removed in Passlib 1.8; Applications should use *load_if_changed()* instead.

load_if_changed()
Reload from `self.path` only if file has changed since last load

load_string(*data*)
Load state from unicode or bytes string, replacing current state

save(*path=None*)
Save current state to file. If no path is specified, attempts to save to `self.path`.

to_string()
Export current state as a string of bytes

3.1.2.2 Inspection

realms()
Return list of all realms in database

users(*realm=None*)
Return list of all users in specified realm.

- uses `self.default_realm` if no realm explicitly provided.
- returns empty list if realm not found.

check_password(*user*[, *realm*], *password*)
Verify password for specified user + realm.

If `self.default_realm` has been set, this may be called with the syntax `check_password(user, password)`, otherwise it must be called with all three arguments: `check_password(user, realm, password)`.

Returns

- `None` if user or realm not found.
- `False` if user found, but password does not match.
- `True` if user found and password matches.

Changed in version 1.6: This method was previously called `verify`, it was renamed to prevent ambiguity with the `CryptContext` method. The old alias is deprecated, and will be removed in Passlib 1.8.

get_hash(*user*, *realm=None*)
Return `htdigest` hash stored for user.

- uses `self.default_realm` if no realm explicitly provided.
- returns `None` if user or realm not found.

Changed in version 1.6: This method was previously named `find`, it was renamed for clarity. The old name is deprecated, and will be removed in Passlib 1.8.

3.1.2.3 Modification

set_password(*user*[, *realm*], *password*)
Set password for user; adds user & realm if needed.

If `self.default_realm` has been set, this may be called with the syntax `set_password(user, password)`, otherwise it must be called with all three arguments: `set_password(user, realm, password)`.

Returns

- `True` if existing user was updated
- `False` if user account added.

delete(*user*, *realm=None*)
Delete user's entry for specified realm.

if realm is not specified, uses `self.default_realm`.

Returns

- `True` if user deleted,
- `False` if user not found in realm.

delete_realm(*realm*)
Delete all users for specified realm.

if realm is not specified, uses `self.default_realm`.

Returns number of users deleted (0 if realm not found)

3.1.2.4 Alternate Constructors

classmethod from_string(*data*, ***kwds*)
create new object from raw string.

Parameters

- **data** (*unicode or bytes*) – database to load, as single string.
- ****kwds** – all other keywords are the same as in the class constructor

3.1.2.5 Attributes

default_realm
The default realm that will be used if one is not provided to methods that require it. By default this is None, in which case an explicit realm must be provided for every method call. Can be written to.

path
Path to local file that will be used as the default for all *load()* and *save()* operations. May be written to, initialized by the *path* constructor keyword.

autosave
Writeable flag indicating whether changes will be automatically written to *path*.

3.1.2.6 Errors

Raises **ValueError** – All of the methods in this class will raise a ValueError if any user name or realm contains a forbidden character (one of :\r\n\t\x00), or is longer than 255 characters.

3.2 passlib.apps - Helpers for various applications

This module contains a number of preconfigured *CryptContext* instances that are provided by Passlib for easily handling the hash formats used by various applications.

3.2.1 Usage Example

The CryptContext class itself has a large number of features, but to give an example of how to quickly use the instances in this module:

Each of the objects in this module can be imported directly:

```
>>> # as an example, this imports the custom_app_context object,
>>> # a helper to let new applications *quickly* add password hashing.
>>> from passlib.apps import custom_app_context
```

Hashing a password is simple (and salt generation is handled automatically):

```
>>> hash = custom_app_context.hash("toomanysecrets")
>>> hash
'$5$rounds=84740$fYChCy.52EzebF51$9bnJrmTf2FESI93hgIBFF4qAfysQcKoB0veiI0ZeYU4'
```

Verifying a password against an existing hash is just as quick:

```
>>> custom_app_context.verify("toomanysocks", hash)
False
>>> custom_app_context.verify("toomanysecrets", hash)
True
```

See also:

the *CryptContext Tutorial* and *CryptContext Reference* for more information about the CryptContext class.

3.2.2 Django

The following objects provide pre-configured `CryptContext` instances for handling Django password hashes, as used by Django's `django.contrib.auth` module. They recognize all the *builtin Django hashes* supported by the particular Django version.

Note: These objects may not match the hashes in your database if a third-party library has been used to patch Django to support alternate hash formats. This includes the django-bcrypt plugin, or Passlib's builtin *`django extension`*. As well, Django 1.4 introduced a very configurable "hashers" framework, and individual deployments may support additional hashes and/or have other defaults.

`passlib.apps.`**`django10_context`**
The object replicates the password hashing policy for Django 1.0-1.3. It supports all the Django 1.0 hashes, and defaults to *`django_salted_sha1`*.

New in version 1.6.

`passlib.apps.`**`django14_context`**
The object replicates the stock password hashing policy for Django 1.4. It supports all the Django 1.0 & 1.4 hashes, and defaults to *`django_pbkdf2_sha256`*. It treats all Django 1.0 hashes as deprecated.

New in version 1.6.

`passlib.apps.`**`django16_context`**
The object replicates the stock password hashing policy for Django 1.6. It supports all the Django 1.0-1.6 hashes, and defaults to *`django_pbkdf2_sha256`*. It treats all Django 1.0 hashes as deprecated.

New in version 1.6.2.

`passlib.apps.`**`django_context`**
This alias will always point to the latest preconfigured Django context supported by Passlib, and as such should support all historical hashes built into Django.

Changed in version 1.6.2: This now points to *`django16_context`*.

3.2.3 LDAP

Passlib provides two contexts related to ldap hashes:

`passlib.apps.`**`ldap_context`**
This object provides a pre-configured `CryptContext` instance for handling LDAPv2 password hashes. It recognizes all the *standard ldap hashes*.

It defaults to using the `{SSHA}` password hash. For times when there should be another default, using code such as the following:

```
>>> from passlib.apps import ldap_context
>>> ldap_context = ldap_context.replace(default="ldap_salted_md5")

>>> # the new context object will now default to {SMD5}:
>>> ldap_context.hash("password")
'{SMD5}T9f89F591P3fFh1jz/YtW4aWD5s='
```

passlib.apps.**ldap_nocrypt_context**
This object recognizes all the standard ldap schemes that `ldap_context` does, *except* for the `{CRYPT}`-based schemes.

3.2.4 MySQL

This module provides two pre-configured `CryptContext` instances for handling MySQL user passwords:

passlib.apps.**mysql_context**
This object should recognize the new *mysql41* hashes, as well as any legacy *mysql323* hashes.

It defaults to mysql41 when generating new hashes.

This should be used with MySQL version 4.1 and newer.

passlib.apps.**mysql3_context**
This object is for use with older MySQL deploys which only recognize the *mysql323* hash.

This should be used only with MySQL version 3.2.3 - 4.0.

3.2.5 PHPass

PHPass is a PHP password hashing library, and hashes derived from it are found in a number of PHP applications. It is found in a wide range of PHP applications, including Drupal and Wordpress.

passlib.apps.**phpass_context**
This object following the standard PHPass logic: it supports *bcrypt*, *bsdi_crypt*, and implements an custom scheme called the "phpass portable hash" *phpass* as a fallback.

BCrypt is used as the default if support is available, otherwise the Portable Hash will be used as the default.

Changed in version 1.5: Now uses Portable Hash as fallback if BCrypt isn't available. Previously used BSDI-Crypt as fallback (per original PHPass implementation), but it was decided PHPass is in fact more secure.

passlib.apps.**phpbb3_context**
This object supports phpbb3 password hashes, which use a variant of *phpass*.

3.2.6 PostgreSQL

passlib.apps.**postgres_context**
This object should recognize password hashes stores in PostgreSQL's `pg_shadow` table; which are all assumed to follow the *postgres_md5* format.

Note that the username must be provided whenever hashing or verifying a postgres hash:

```
>>> from passlib.apps import postgres_context

>>> # hashing a password...
>>> postgres_context.hash("somepass", user="dbadmin")
'md578ed0f0ab2be0386645c1b74282917e7'

>>> # verifying a password...
>>> postgres_context.verify("somepass", 'md578ed0f0ab2be0386645c1b74282917e7',␣
↪user="dbadmin")
True
>>> postgres_context.verify("wrongpass", 'md578ed0f0ab2be0386645c1b74282917e7',␣
↪user="dbadmin")
```

(continues on next page)

(continued from previous page)

```
False

>>> # forgetting the user will result in an error:
>>> postgres_context.hash("somepass")
Traceback (most recent call last):
    <traceback omitted>
TypeError: user must be unicode or bytes, not None
```

3.2.7 Roundup

The Roundup Issue Tracker has long supported a series of different methods for encoding passwords. The following contexts are available for reading Roundup password hash fields:

`passlib.apps.`**`roundup10_context`**
: This object should recognize all password hashes used by Roundup 1.4.16 and earlier: *ldap_hex_sha1* (the default), *ldap_hex_md5*, *ldap_des_crypt*, and *roundup_plaintext*.

`passlib.apps.`**`roundup15_context`**
: Roundup 1.4.17 adds support for *ldap_pbkdf2_sha1* as its preferred hash format. This context supports all the *roundup10_context* hashes, but adds that hash as well (and uses it as the default).

`passlib.apps.`**`roundup_context`**
: this is an alias for the latest version-specific roundup context supported by passlib, currently the `roundup15_context`.

3.2.8 Custom Applications

`passlib.apps.`**`custom_app_context`**
: This `CryptContext` object is provided for new python applications to quickly and easily add password hashing support. It comes preconfigured with:

  - Support for *sha256_crypt* and *sha512_crypt*
  - Defaults to SHA256-Crypt under 32 bit systems, SHA512-Crypt under 64 bit systems.
  - Large number of `rounds`, for increased time-cost to hedge against attacks.

  For applications which want to quickly add a password hash, all they need to do is import and use this object, per the *usage example* at the top of this page.

  See also:

  The *New Application Quickstart Guide* for additional details.

3.3 `passlib.context` - CryptContext Hash Manager

This page provides a complete reference of all the methods and options supported by the `CryptContext` class and helper utilities.

See also:

- *CryptContext Tutorial* – overview of this class and walkthrough of how to use it.

3.3.1 The CryptContext Class

class passlib.context.**CryptContext**(*schemes=None*, ***kwds*)

Helper for hashing passwords using different algorithms.

At its base, this is a proxy object that makes it easy to use multiple *PasswordHash* objects at the same time. Instances of this class can be created by calling the constructor with the appropriate keywords, or by using one of the alternate constructors, which can load directly from a string or a local file. Since this class has so many options and methods, they have been broken out into subsections:

- ***Constructor Keywords* – all the keywords this class accepts.**
  - *Context Options* – options affecting the Context itself.
  - *Algorithm Options* – options controlling the wrapped hashes.
- *Primary Methods* – the primary methods most applications need.
- *Hash Migration* – methods for automatically replacing deprecated hashes.
- *Alternate Constructors* – creating instances from strings or files.
- *Changing the Configuration* – altering the configuration of an existing context.
- *Examining the Configuration* – programmatically examining the context's settings.
- *Saving the Configuration* – exporting the context's current configuration.
- *Configuration Errors* – overview of errors that may be thrown by CryptContext constructor

3.3.1.1 Constructor Keywords

The *CryptContext* class accepts the following keywords, all of which are optional. The keywords are divided into two categories: *context options*, which affect the CryptContext itself; and *algorithm options*, which place defaults and limits on the algorithms used by the CryptContext.

Context Options

Options which directly affect the behavior of the CryptContext instance:

schemes List of algorithms which the instance should support.

The most important option in the constructor, This option controls what hashes can be used by the *hash()* method, which hashes will be recognized by *verify()* and *identify()*, and other effects throughout the instance. It should be a sequence of names, drawn from the hashes in *passlib.hash*. Listing an unknown name will cause a ValueError. You can use the *schemes()* method to get a list of the currently configured algorithms. As an example, the following creates a CryptContext instance which supports the *sha256_crypt* and *des_crypt* schemes:

```
>>> from passlib.context import CryptContext
>>> myctx = CryptContext(schemes=["sha256_crypt", "des_crypt"])
>>> myctx.schemes()
("sha256_crypt", "des_crypt")
```

Note: The order of the schemes is sometimes important, as *identify()* will run through the schemes from first to last until an algorithm "claims" the hash. So plaintext algorithms and the like should be listed at the end.

See also:

the *Basic Usage* example in the tutorial.

default Specifies the name of the default scheme.

This option controls which of the configured schemes will be used as the default when creating new hashes. This parameter is optional; if omitted, the first non-deprecated algorithm in `schemes` will be used. You can use the *default_scheme()* method to retrieve the name of the current default scheme. As an example, the following demonstrates the effect of this parameter on the *hash()* method:

```
>>> from passlib.context import CryptContext
>>> myctx = CryptContext(schemes=["sha256_crypt", "md5_crypt"])

>>> # hash() uses the first scheme
>>> myctx.default_scheme()
'sha256_crypt'
>>> myctx.hash("password")
'$5$rounds=80000$R5ZIZRTNPgbdcWq5$fT/Oeqq/apMa/0fbx8YheYWS6Z3XLTxCzEtutsk2cJ1'

>>> # but setting default causes the second scheme to be used.
>>> myctx.update(default="md5_crypt")
>>> myctx.default_scheme()
'md5_crypt'
>>> myctx.hash("password")
'$1$Rr0C.KI8$Kvciy8pqfL9BQ2CJzEzfZ/'
```

See also:

the *Basic Usage* example in the tutorial.

deprecated List of algorithms which should be considered "deprecated".

This has the same format as `schemes`, and should be a subset of those algorithms. The main purpose of this method is to flag schemes which need to be rehashed when the user next logs in. This has no effect on the *Primary Methods*; but if the special *Hash Migration* methods are passed a hash belonging to a deprecated scheme, they will flag it as needed to be rehashed using the `default` scheme.

This may also contain a single special value, `["auto"]`, which will configure the CryptContext instance to deprecate *all* supported schemes except for the default scheme.

New in version 1.6: Added support for the `["auto"]` value.

See also:

Deprecation & Hash Migration in the tutorial

`truncate_error`

By default, some algorithms will truncate large passwords (e.g. *bcrypt* truncates ones larger than 72 bytes). Such hashes accept a `truncate_error=True` option to make them raise a *PasswordTruncateError* instead.

This can also be set at the CryptContext level, and will passed to all hashes that support it.

New in version 1.7.

`min_verify_time`

If specified, unsuccessful *verify()* calls will be penalized, and take at least this may seconds before the method returns. May be an integer or fractional number of seconds.

Deprecated since version 1.6: This option has not proved very useful, is ignored by 1.7, and will be removed in version 1.8.

Changed in version 1.7: Per deprecation roadmap above, this option is now ignored.

harden_verify

Companion to min_verify_time, currently ignored.

New in version 1.7.

Deprecated since version 1.7.1: This option is ignored by 1.7.1, and will be removed in 1.8 along with min_verify_time.

Algorithm Options

All of the other options that can be passed to a *CryptContext* constructor affect individual hash algorithms. All of the following keys have the form *scheme__key*, where *scheme* is the name of one of the algorithms listed in schemes, and *option* one of the parameters below:

*scheme*__rounds

Set the number of rounds required for this scheme when generating new hashes (using *hash()*). Existing hashes which have a different number of rounds will be marked as deprecated.

This essentially sets default_rounds, min_rounds, and max_rounds all at once. If any of those options are also specified, they will override the value specified by rounds.

New in version 1.7: Previous releases of Passlib treated this as an alias for default_rounds.

*scheme*__default_rounds

Sets the default number of rounds to use with this scheme when generating new hashes (using *hash()*).

If not set, this will fall back to the an algorithm-specific *default_rounds*. For hashes which do not support a rounds parameter, this option is ignored. As an example:

```
>>> from passlib.context import CryptContext

>>> # no explicit default_rounds set, so hash() uses sha256_crypt's default␣
↪(80000)
>>> myctx = CryptContext(["sha256_crypt"])
>>> myctx.hash("fooey")
'$5$rounds=80000$60Y7mpmAhUv6RDvj$AdseAOq6bKUZRDRTr/2QK1t38qm3P6sYeXhXKnBAmg0
↪'
           ^^^^^

>>> # but if a default is specified, it will be used instead.
>>> myctx = CryptContext(["sha256_crypt"], sha256_crypt__default_
↪rounds=77123)
>>> myctx.hash("fooey")
'$5$rounds=77123$60Y7mpmAhUv6RDvj$AdseAOq6bKUZRDRTr/2QK1t38qm3P6sYeXhXKnBAmg0
↪'
           ^^^^^
```

See also:

the *Using Default Settings* example in the tutorial.

*scheme*__vary_rounds

Deprecated since version 1.7: This option has been deprecated as of Passlib 1.7, and will be removed in Passlib 2.0. The (very minimal) security benefit it provides was judged to not be worth code complexity it requires.

Instead of using a fixed rounds value (such as specified by default_rounds, above); this option will cause each call to *hash()* to vary the default rounds value by some amount.

This can be an integer value, in which case each call will use a rounds value within the range `default_rounds +/- vary_rounds`. It may also be a floating point value within the range 0.0 .. 1.0, in which case the range will be calculated as a proportion of the current default rounds (`default_rounds +/- default_rounds*vary_rounds`). A typical setting is `0.1` to `0.2`.

As an example of how this parameter operates:

```
>>> # without vary_rounds set, hash() uses the same amount each time:
>>> from passlib.context import CryptContext
>>> myctx = CryptContext(schemes=["sha256_crypt"],
...                      sha256_crypt__default_rounds=80000)
>>> myctx.hash("fooey")
'$5$rounds=80000$60Y7mpmAhUv6RDvj$AdseAOq6bKUZRDRTr/2QK1t38qm3P6sYeXhXKnBAmg0
↪'
>>> myctx.hash("fooey")
'$5$rounds=80000$60Y7mpmAhUv6RDvj$AdseAOq6bKUZRDRTr/2QK1t38qm3P6sYeXhXKnBAmg0
↪'
           ^^^^^

>>> # but if vary_rounds is set, each one will be randomized
>>> # (in this case, within the range 72000 .. 88000)
>>> myctx = CryptContext(schemes=["sha256_crypt"],
...                      sha256_crypt__default_rounds=80000,
...                      sha256_crypt__vary_rounds=0.1)
>>> myctx.hash("fooey")
'$5$rounds=83966$bMpgQxN2hXo2kVr4$jL4Q3ov41UPgSbO7jYL0PdtsOg5koo4mCa.UEF3zan.
↪'
>>> myctx.hash("fooey")
'$5$rounds=72109$43BBHC/hYPHzL69c$VYvVIdKn3Zdnvu0oJHVlo6rr0WjiMTGmlrZrrH.GxnA
↪'
           ^^^^^
```

Note: This is not a *needed* security measure, but it lets some of the less-significant digits of the rounds value act as extra salt bits; and helps foil any attacks targeted at a specific number of rounds of a hash.

---

scheme`__min_rounds`, *scheme*`__max_rounds`

These options place a limit on the number of rounds allowed for a particular scheme.

For one, they limit what values are allowed for `default_rounds`, and clip the effective range of the `vary_rounds` parameter. More importantly though, they proscribe a minimum strength for the hash, and any hashes which don't have sufficient rounds will be flagged as needing rehashing by the *Hash Migration* methods.

Note: These are configurable per-context limits. A warning will be issued if they exceed any hard limits set by the algorithm itself.

See also:

the *Settings Rounds Limitations* example in the tutorial.

scheme`__other-option`

Finally, any other options are assumed to correspond to one of the that algorithm's `hash() settings`, such as setting a `salt_size`.

See also:

the *Using Default Settings* example in the tutorial.

Global Algorithm Options

`all__option`

The special scheme `all` permits you to set an option, and have it act as a global default for all the algorithms in the context. For instance, `all__vary_rounds=0.1` would set the `vary_rounds` option for all the schemes where it was not overridden with an explicit *scheme*`__vary_rounds` option.

Deprecated since version 1.7: This special scheme is deprecated as of Passlib 1.7, and will be removed in Passlib 2.0. It's only legitimate use was for `vary_rounds`, which is also being removed in Passlib 2.0.

User Categories

category`__context__`*option*, *category*`__scheme__`*option*

Passing keys with this format to the *CryptContext* constructor allows you to specify conditional context and algorithm options, controlled by the `category` parameter supported by most CryptContext methods.

These options are conditional because they only take effect if the *category* prefix of the option matches the value of the `category` parameter of the CryptContext method being invoked. In that case, they override any options specified without a category prefix (e.g. *admin__sha256_crypt__min_rounds* would override *sha256_crypt__min_rounds*). The category prefix and the value passed into the `category` parameter can be any string the application wishes to use, the only constraint is that `None` indicates the default category.

Motivation: Policy limits such as default rounds values and deprecated schemes generally have to be set globally. However, it's frequently desirable to specify stronger options for certain accounts (such as admin accounts), choosing to sacrifice longer hashing time for a more secure password. The user categories system allows for this. For example, a CryptContext could be set up as follows:

```
>>> # A context object can be set up as follows:
>>> from passlib.context import CryptContext
>>> myctx = CryptContext(schemes=["sha256_crypt"],
...                      sha256_crypt__default_rounds=77000,
...                      staff__sha256_crypt__default_rounds=88000)

>>> # In this case, calling hash() with ``category=None`` would result
>>> # in a hash that used 77000 sha256-crypt rounds:
>>> myctx.hash("password", category=None)
'$5$rounds=77000$sj3XI0AbKlEydAKt$BhFvyh4.IoxaUeNlW6rvQ.O0w8BtgLQMYorkCOMzf84'
           ^^^^^

>>> # But if the application passed in ``category="staff"`` when an administrative
>>> # account set their password, 88000 rounds would be used:
>>> myctx.hash("password", category="staff")
'$5$rounds=88000$w7XIdKfTI9.YLwmA$MIzGvs6NU1QOQuuDHhICLmDsdW/t94Bbdfxdh/6NJl7'
           ^^^^^
```

3.3.1.2 Primary Methods

The main interface to the CryptContext object deliberately mirrors the *PasswordHash* interface, since its central purpose is to act as a container for multiple password hashes. Most applications will only need to make use two methods

in a CryptContext instance:

CryptContext.**hash**(*secret*, *scheme=None*, *category=None*, ***kwds*)
run secret through selected algorithm, returning resulting hash.

Parameters

- **secret** (*unicode or bytes*) – the password to hash.
- **scheme** (*str or None*) – Optional scheme to use. Scheme must be one of the ones configured for this context (see the *schemes* option). If no scheme is specified, the configured default will be used.

  Deprecated since version 1.7: Support for this keyword is deprecated, and will be removed in Passlib 2.0.
- **category** (*str or None*) – Optional *user category*. If specified, this will cause any category-specific defaults to be used when hashing the password (e.g. different default scheme, different default rounds values, etc).
- ****kwds** – All other keyword options are passed to the selected algorithm's *PasswordHash.hash()* method.

Returns The secret as encoded by the specified algorithm and options. The return value will always be a str.

Raises **TypeError, ValueError** –

- If any of the arguments have an invalid type or value. This includes any keywords passed to the underlying hash's *PasswordHash.hash()* method.

See also:

the *Basic Usage* example in the tutorial

CryptContext.**encrypt**(**args*, ***kwds*)
Legacy alias for *hash()*.

Deprecated since version 1.7: This method was renamed to hash() in version 1.7. This alias will be removed in version 2.0, and should only be used for compatibility with Passlib 1.3 - 1.6.

CryptContext.**verify**(*secret*, *hash*, *scheme=None*, *category=None*, ***kwds*)
verify secret against an existing hash.

If no scheme is specified, this will attempt to identify the scheme based on the contents of the provided hash (limited to the schemes configured for this context). It will then check whether the password verifies against the hash.

Parameters

- **secret** (*unicode or bytes*) – the secret to verify
- **hash** (*unicode or bytes*) – hash string to compare to

  if None is passed in, this will be treated as "never verifying"
- **scheme** (*str*) – Optionally force context to use specific scheme. This is usually not needed, as most hashes can be unambiguously identified. Scheme must be one of the ones configured for this context (see the *schemes* option).

  Deprecated since version 1.7: Support for this keyword is deprecated, and will be removed in Passlib 2.0.

- **category** (*str or None*) – Optional *user category* string. This is mainly used when generating new hashes, it has little effect when verifying; this keyword is mainly provided for symmetry.
- ****kwds** – All additional keywords are passed to the appropriate handler, and should match its *context_kwds*.

Returns `True` if the password matched the hash, else `False`.

Raises

- **ValueError** –
  - if the hash did not match any of the configured *schemes()*.
  - if any of the arguments have an invalid value (this includes any keywords passed to the underlying hash's *PasswordHash.verify()* method).
- **TypeError** –
  - if any of the arguments have an invalid type (this includes any keywords passed to the underlying hash's *PasswordHash.verify()* method).

See also:

the *Basic Usage* example in the tutorial

`CryptContext.`**identify**(*hash*, *category=None*, *resolve=False*, *required=False*, *unconfigured=False*)

Attempt to identify which algorithm the hash belongs to.

Note that this will only consider the algorithms currently configured for this context (see the *schemes* option). All registered algorithms will be checked, from first to last, and whichever one positively identifies the hash first will be returned.

Parameters

- **hash** (*unicode or bytes*) – The hash string to test.
- **category** (*str or None*) – Optional *user category*. Ignored by this function, this parameter is provided for symmetry with the other methods.
- **resolve** (*bool*) – If `True`, returns the hash handler itself, instead of the name of the hash.
- **required** (*bool*) – If `True`, this will raise a ValueError if the hash cannot be identified, instead of returning `None`.

Returns The handler which first identifies the hash, or `None` if none of the algorithms identify the hash.

`CryptContext.`**dummy_verify**(*elapsed=0*)

Helper that applications can call when user wasn't found, in order to simulate time it would take to hash a password.

Runs verify() against a dummy hash, to simulate verification of a real account password.

Parameters **elapsed** – Deprecated since version 1.7.1: this option is ignored, and will be removed in passlib 1.8.

New in version 1.7.

"crypt"-style methods

Additionally, the main interface offers wrappers for the two Unix "crypt" style methods provided by all the *PasswordHash* objects:

CryptContext.**genhash**(*secret*, *config*, *scheme=None*, *category=None*, ***kwds*)
Generate hash for the specified secret using another hash.

Deprecated since version 1.7: This method will be removed in version 2.0, and should only be used for compatibility with Passlib 1.3 - 1.6.

CryptContext.**genconfig**(*scheme=None*, *category=None*, ***settings*)
Generate a config string for specified scheme.

Deprecated since version 1.7: This method will be removed in version 2.0, and should only be used for compatibility with Passlib 1.3 - 1.6.

3.3.1.3 Hash Migration

Applications which want to detect and regenerate deprecated hashes will want to use one of the following methods:

CryptContext.**verify_and_update**(*secret*, *hash*, *scheme=None*, *category=None*, ***kwds*)
verify password and re-hash the password if needed, all in a single call.

This is a convenience method which takes care of all the following: first it verifies the password (*verify()*), if this is successfull it checks if the hash needs updating (*needs_update()*), and if so, re-hashes the password (*hash()*), returning the replacement hash. This series of steps is a very common task for applications which wish to update deprecated hashes, and this call takes care of all 3 steps efficiently.

Parameters

- **secret** (*unicode or bytes*) – the secret to verify
- **hash** – hash string to compare to.

  if None is passed in, this will be treated as "never verifying"
- **scheme** (*str*) – Optionally force context to use specific scheme. This is usually not needed, as most hashes can be unambiguously identified. Scheme must be one of the ones configured for this context (see the *schemes* option).

  Deprecated since version 1.7: Support for this keyword is deprecated, and will be removed in Passlib 2.0.
- **category** (*str or None*) – Optional *user category*. If specified, this will cause any category-specific defaults to be used if the password has to be re-hashed.
- ****kwds** – all additional keywords are passed to the appropriate handler, and should match that hash's *PasswordHash.context_kwds*.

Returns

This function returns a tuple containing two elements: `(verified, replacement_hash)`. The first is a boolean flag indicating whether the password verified, and the second an optional replacement hash. The tuple will always match one of the following 3 cases:

- `(False, None)` indicates the secret failed to verify.
- `(True, None)` indicates the secret verified correctly, and the hash does not need updating.

- `(True, str)` indicates the secret verified correctly, but the current hash needs to be updated. The `str` will be the freshly generated hash, to replace the old one.

Raises **TypeError, ValueError** – For the same reasons as *verify()*.

See also:

the *Deprecation & Hash Migration* example in the tutorial.

`CryptContext.`**needs_update**(*hash*, *scheme=None*, *category=None*, *secret=None*)

Check if hash needs to be replaced for some reason, in which case the secret should be re-hashed.

This function is the core of CryptContext's support for hash migration: This function takes in a hash string, and checks the scheme, number of rounds, and other properties against the current policy. It returns `True` if the hash is using a deprecated scheme, or is otherwise outside of the bounds specified by the policy (e.g. the number of rounds is lower than *min_rounds* configuration for that algorithm). If so, the password should be re-hashed using *hash()* Otherwise, it will return `False`.

Parameters

- **hash** (*unicode or bytes*) – The hash string to examine.
- **scheme** (*str or None*) – Optional scheme to use. Scheme must be one of the ones configured for this context (see the *schemes* option). If no scheme is specified, it will be identified based on the value of *hash*.

  Deprecated since version 1.7: Support for this keyword is deprecated, and will be removed in Passlib 2.0.
- **category** (*str or None*) – Optional *user category*. If specified, this will cause any category-specific defaults to be used when determining if the hash needs to be updated (e.g. is below the minimum rounds).
- **secret** (*unicode, bytes, or None*) – Optional secret associated with the provided `hash`. This is not required, or even currently used for anything… it's for forward-compatibility with any future update checks that might need this information. If provided, Passlib assumes the secret has already been verified successfully against the hash.

  New in version 1.6.

Returns `True` if hash should be replaced, otherwise `False`.

Raises **ValueError** – If the hash did not match any of the configured *schemes()*.

New in version 1.6: This method was previously named *hash_needs_update()*.

See also:

the *Deprecation & Hash Migration* example in the tutorial.

`CryptContext.`**hash_needs_update**(*hash*, *scheme=None*, *category=None*)

Legacy alias for *needs_update()*.

Deprecated since version 1.6: This method was renamed to `needs_update()` in version 1.6. This alias will be removed in version 2.0, and should only be used for compatibility with Passlib 1.3 - 1.5.

3.3.1.4 Disabled Hash Managment

New in version 1.7.

It's frequently useful to disable a user's ability to login by replacing their password hash with a standin that's guaranteed to never verify, against *any* password. CryptContext offers some convenience methods for this through the following API.

CryptContext.**disable**(*hash=None*)
return a string to disable logins for user, usually by returning a non-verifying string such as "!".

Parameters **hash** – Callers can optionally provide the account's existing hash. Some disabled handlers (such as unix_disabled) will encode this into the returned value, so that it can be recovered via *enable()*.

Raises **RuntimeError** – if this function is called w/o a disabled hasher (such as *unix_disabled*) included in the list of schemes.

Returns hash string which will be recognized as valid by the context, but is guaranteed to not validate against *any* password.

CryptContext.**enable**(*hash*)
inverse of *disable()* – attempts to recover original hash which was converted by a disable() call into a disabled hash – thus restoring the user's original password.

Raises **ValueError** – if original hash not present, or if the disabled handler doesn't support encoding the original hash (e.g. django_disabled)

Returns the original hash.

CryptContext.**is_enabled**(*hash*)
test if hash represents a usable password – i.e. does not represent an unusable password such as "!", which is recognized by the *unix_disabled* hash.

Raises **ValueError** – if the hash is not recognized (typically solved by adding unix_disabled to the list of schemes).

3.3.1.5 Alternate Constructors

In addition to the main class constructor, which accepts a configuration as a set of keywords, there are the following alternate constructors:

classmethod CryptContext.**from_string**(*source*, *section='passlib'*, *encoding='utf-8'*)
create new CryptContext instance from an INI-formatted string.

Parameters

- **source** (*unicode or bytes*) – string containing INI-formatted content.
- **section** (*str*) – option name of section to read from, defaults to "passlib".
- **encoding** (*str*) – optional encoding used when source is bytes, defaults to "utf-8".

Returns new *CryptContext* instance, configured based on the parameters in the *source* string.

Usage example:

```
>>> from passlib.context import CryptContext
>>> context = CryptContext.from_string('''
... [passlib]
... schemes = sha256_crypt, des_crypt
... sha256_crypt__default_rounds = 30000
... ''')
```

New in version 1.6.

See also:

to_string(), the inverse of this constructor.

classmethod CryptContext.**from_path**(*path*, *section='passlib'*, *encoding='utf-8'*)
create new CryptContext instance from an INI-formatted file.

this functions exactly the same as *from_string()*, except that it loads from a local file.

Parameters

- **path** (*str*) – path to local file containing INI-formatted config.
- **section** (*str*) – option name of section to read from, defaults to "passlib".
- **encoding** (*str*) – encoding used to load file, defaults to "utf-8".

Returns new CryptContext instance, configured based on the parameters stored in the file *path*.

New in version 1.6.

See also:

from_string() for an equivalent usage example.

CryptContext.**copy**(**kwds*)
Return copy of existing CryptContext instance.

This function returns a new CryptContext instance whose configuration is exactly the same as the original, with the exception that any keywords passed in will take precedence over the original settings. As an example:

```
>>> from passlib.context import CryptContext

>>> # given an existing context...
>>> ctx1 = CryptContext(["sha256_crypt", "md5_crypt"])

>>> # copy can be used to make a clone, and update
>>> # some of the settings at the same time...
>>> ctx2 = custom_app_context.copy(default="md5_crypt")

>>> # and the original will be unaffected by the change
>>> ctx1.default_scheme()
"sha256_crypt"
>>> ctx2.default_scheme()
"md5_crypt"
```

New in version 1.6: This method was previously named `replace()`. That alias has been deprecated, and will be removed in Passlib 1.8.

See also:

update()

3.3.1.6 Changing the Configuration

CryptContext objects can have their configuration replaced or updated on the fly, and from a variety of sources (keywords, strings, files). This is done through three methods:

CryptContext.**update**(**kwds*)
Helper for quickly changing configuration.

This acts much like the `dict.update()` method: it updates the context's configuration, replacing the original value(s) for the specified keys, and preserving the rest. It accepts any *keyword* accepted by the `CryptContext` constructor.

New in version 1.6.

See also:

copy()

CryptContext.**load**(*source*, *update=False*, *section='passlib'*, *encoding='utf-8'*)

Load new configuration into CryptContext, replacing existing config.

Parameters

- **source** – source of new configuration to load. this value can be a number of different types:
  - a `dict` object, or compatible Mapping

    the key/value pairs will be interpreted the same keywords for the *CryptContext* class constructor.
  - a `unicode` or `bytes` string

    this will be interpreted as an INI-formatted file, and appropriate key/value pairs will be loaded from the specified *section*.
  - another `CryptContext` object.

    this will export a snapshot of its configuration using *to_dict()*.
- **update** (*bool*) – By default, *load()* will replace the existing configuration entirely. If `update=True`, it will preserve any existing configuration options that are not overridden by the new source, much like the *update()* method.
- **section** (*str*) – When parsing an INI-formatted string, *load()* will look for a section named `"passlib"`. This option allows an alternate section name to be used. Ignored when loading from a dictionary.
- **encoding** (*str*) – Encoding to use when **source** is bytes. Defaults to `"utf-8"`. Ignored when loading from a dictionary.

  Deprecated since version 1.8: This keyword, and support for bytes input, will be dropped in Passlib 2.0

Raises

- **TypeError** –
  - If the source cannot be identified.
  - If an unknown / malformed keyword is encountered.
- **ValueError** – If an invalid keyword value is encountered.

Note: If an error occurs during a `load()` call, the `CryptContext` instance will be restored to the configuration it was in before the `load()` call was made; this is to ensure it is *never* left in an inconsistent state due to a load error.

New in version 1.6.

CryptContext.**load_path**(*path*, *update=False*, *section='passlib'*, *encoding='utf-8'*)

Load new configuration into CryptContext from a local file.

This function is a wrapper for *load()* which loads a configuration string from the local file *path*, instead of an in-memory source. Its behavior and options are otherwise identical to `load()` when provided with an INI-formatted string.

New in version 1.6.

3.3.1.7 Examining the Configuration

The CryptContext object also supports basic inspection of its current configuration:

`CryptContext.`**`schemes`**(*resolve=False*, *category=None*, *unconfigured=False*)
return schemes loaded into this CryptContext instance.

Parameters **`resolve`** (*bool*) – if `True`, will return a tuple of *PasswordHash* objects instead of their names.

Returns returns tuple of the schemes configured for this context via the *schemes* option.

New in version 1.6: This was previously available as `CryptContext().policy.schemes()`

See also:

the *schemes* option for usage example.

`CryptContext.`**`default_scheme`**(*category=None*, *resolve=False*, *unconfigured=False*)
return name of scheme that *hash()* will use by default.

Parameters

- **`resolve`** (*bool*) – if `True`, will return a *PasswordHash* object instead of the name.
- **`category`** (*str or None*) – Optional *user category*. If specified, this will return the catgory-specific default scheme instead.

Returns name of the default scheme.

See also:

the *default* option for usage example.

New in version 1.6.

Changed in version 1.7: This now returns a hasher configured with any CryptContext-specific options (custom rounds settings, etc). Previously this returned the base hasher from *passlib.hash*.

`CryptContext.`**`handler`**(*scheme=None*, *category=None*, *unconfigured=False*)
helper to resolve name of scheme -> *PasswordHash* object used by scheme.

Parameters

- **`scheme`** – This should identify the scheme to lookup. If omitted or set to `None`, this will return the handler for the default scheme.
- **`category`** – If a user category is specified, and no scheme is provided, it will use the default for that category. Otherwise this parameter is ignored.
- **`unconfigured`** – By default, this returns a handler object whose .hash() and .needs_update() methods will honor the configured provided by CryptContext. See `unconfigured=True` to get the underlying handler from before any context-specific configuration was applied.

Raises **`KeyError`** – If the scheme does not exist OR is not being used within this context.

Returns *PasswordHash* object used to implement the named scheme within this context (this will usually be one of the objects from *passlib.hash*)

New in version 1.6: This was previously available as `CryptContext().policy.get_handler()`

Changed in version 1.7: This now returns a hasher configured with any CryptContext-specific options (custom rounds settings, etc). Previously this returned the base hasher from *passlib.hash*.

CryptContext.**context_kwds**
return set containing union of all *contextual keywords* supported by the handlers in this context.

New in version 1.6.6.

3.3.1.8 Saving the Configuration

More detailed inspection can be done by exporting the configuration using one of the serialization methods:

CryptContext.**to_dict**(*resolve=False*)
Return current configuration as a dictionary.

Parameters **resolve** (*bool*) – if True, the schemes key will contain a list of a *PasswordHash* objects instead of just their names.

This method dumps the current configuration of the CryptContext instance. The key/value pairs should be in the format accepted by the CryptContext class constructor, in fact CryptContext(**myctx.to_dict()) will create an exact copy of myctx. As an example:

```
>>> # you can dump the configuration of any crypt context...
>>> from passlib.apps import ldap_nocrypt_context
>>> ldap_nocrypt_context.to_dict()
{'schemes': ['ldap_salted_sha1',
'ldap_salted_md5',
'ldap_sha1',
'ldap_md5',
'ldap_plaintext']}
```

New in version 1.6: This was previously available as CryptContext().policy.to_dict()

See also:

the *Loading & Saving a CryptContext* example in the tutorial.

CryptContext.**to_string**(*section='passlib'*)
serialize to INI format and return as unicode string.

Parameters **section** – name of INI section to output, defaults to "passlib".

Returns CryptContext configuration, serialized to a INI unicode string.

This function acts exactly like *to_dict()*, except that it serializes all the contents into a single human-readable string, which can be hand edited, and/or stored in a file. The output of this method is accepted by *from_string()*, *from_path()*, and *load()*. As an example:

```
>>> # you can dump the configuration of any crypt context...
>>> from passlib.apps import ldap_nocrypt_context
>>> print ldap_nocrypt_context.to_string()
[passlib]
schemes = ldap_salted_sha1, ldap_salted_md5, ldap_sha1, ldap_md5, ldap_plaintext
```

New in version 1.6: This was previously available as CryptContext().policy.to_string()

See also:

the *Loading & Saving a CryptContext* example in the tutorial.

3.3.1.9 Configuration Errors

The following errors may be raised when creating a `CryptContext` instance via any of its constructors, or when updating the configuration of an existing instance:

raises ValueError

- If a configuration option contains an invalid value (e.g. `all__vary_rounds=-1`).
- If the configuration contains valid but incompatible options (e.g. listing a scheme as both *default* and *deprecated*).

raises KeyError

- If the configuration contains an unknown or forbidden option (e.g. *scheme*`__salt`).
- If the *schemes*, *default*, or *deprecated* options reference an unknown hash scheme (e.g. `schemes=['xxx']`)

raises TypeError

- If a configuration value has the wrong type (e.g. `schemes=123`).

Note that this error shouldn't occur when loading configurations from a file/string (e.g. using *`CryptContext.from_string()`*).

Additionally, a *`PasslibConfigWarning`* may be issued if any invalid-but-correctable values are encountered (e.g. if `sha256_crypt__min_rounds` is set to less than *`sha256_crypt`* 's minimum of 1000).

Changed in version 1.6: Previous releases used Python's builtin `UserWarning` instead of the more specific `passlib.exc.PasslibConfigWarning`.

3.3.2 Other Helpers

class passlib.context.**LazyCryptContext**(*[schemes=None]*, ***kwds[, onload=None]*)

CryptContext subclass which doesn't load handlers until needed.

This is a subclass of CryptContext which takes in a set of arguments exactly like CryptContext, but won't import any handlers (or even parse its arguments) until the first time one of its methods is accessed.

Parameters

- **schemes** – The first positional argument can be a list of schemes, or omitted, just like CryptContext.
- **onload** – If a callable is passed in via this keyword, it will be invoked at lazy-load time with the following signature: `onload(**kwds) -> kwds`; where `kwds` is all the additional kwds passed to LazyCryptContext. It should perform any additional deferred initialization, and return the final dict of options to be passed to CryptContext.

  New in version 1.6.
- **create_policy** – Deprecated since version 1.6: This option will be removed in Passlib 1.8, applications should use `onload` instead.
- **kwds** – All additional keywords are passed to CryptContext; or to the *onload* function (if provided).

This is mainly used internally by modules such as *`passlib.apps`*, which define a large number of contexts, but only a few of them will be needed at any one time. Use of this class saves the memory needed to import the specified handlers until the context instance is actually accessed. As well, it allows constructing a context at *module-init* time, but using `onload()` to provide dynamic configuration at *application-run* time.

Note: This class is only useful if you're referencing handler objects by name, and don't want them imported until runtime. If you want to have the config validated before your application runs, or are passing in already-imported handler instances, you should use *CryptContext* instead.

New in version 1.4.

3.3.3 The CryptPolicy Class (deprecated)

class passlib.context.**CryptPolicy**(**args*, ***kwds*)

Deprecated since version 1.6: This class has been deprecated, and will be removed in Passlib 1.8. All of its functionality has been rolled into *CryptContext*.

This class previously stored the configuration options for the CryptContext class. In the interest of interface simplification, all of this class' functionality has been rolled into the CryptContext class itself. The documentation for this class is now focused on documenting how to migrate to the new api. Additionally, where possible, the deprecation warnings issued by the CryptPolicy methods will list the replacement call that should be used.

3.3.3.1 Constructors

CryptPolicy objects can be constructed directly using any of the keywords accepted by *CryptContext*. Direct uses of the `CryptPolicy` constructor should either pass the keywords directly into the CryptContext constructor, or to *CryptContext.update()* if the policy object was being used to update an existing context object.

In addition to passing in keywords directly, CryptPolicy objects can be constructed by the following methods:

classmethod from_path(*path*, *section='passlib'*, *encoding='utf-8'*)

create a CryptPolicy instance from a local file.

Deprecated since version 1.6.

Creating a new CryptContext from a file, which was previously done via `CryptContext(policy=CryptPolicy.from_path(path))`, can now be done via `CryptContext.from_path(path)`. See *CryptContext.from_path()* for details.

Updating an existing CryptContext from a file, which was previously done `context.policy = CryptPolicy.from_path(path)`, can now be done via `context.load_path(path)`. See *CryptContext.load_path()* for details.

classmethod from_string(*source*, *section='passlib'*, *encoding='utf-8'*)

create a CryptPolicy instance from a string.

Deprecated since version 1.6.

Creating a new CryptContext from a string, which was previously done via `CryptContext(policy=CryptPolicy.from_string(data))`, can now be done via `CryptContext.from_string(data)`. See *CryptContext.from_string()* for details.

Updating an existing CryptContext from a string, which was previously done `context.policy = CryptPolicy.from_string(data)`, can now be done via `context.load(data)`. See *CryptContext.load()* for details.

classmethod from_source(*source*, *_warn=True*)

create a CryptPolicy instance from some source.

this method autodetects the source type, and invokes the appropriate constructor automatically. it attempts to detect whether the source is a configuration string, a filepath, a dictionary, or an existing CryptPolicy instance.

Deprecated since version 1.6.

Create a new CryptContext, which could previously be done via `CryptContext(policy=CryptPolicy.from_source(source))`, should now be done using an explicit method: the *CryptContext* constructor itself, *CryptContext.from_path()*, or *CryptContext.from_string()*.

Updating an existing CryptContext, which could previously be done via `context.policy = CryptPolicy.from_source(source)`, should now be done using an explicit method: *CryptContext.update()*, or *CryptContext.load()*.

classmethod from_sources(*sources*, *_warn=True*)
create a CryptPolicy instance by merging multiple sources.

each source is interpreted as by *from_source()*, and the results are merged together.

Deprecated since version 1.6: Instead of using this method to merge multiple policies together, a *CryptContext* instance should be created, and then the multiple sources merged together via *CryptContext.load()*.

replace(**args*, ***kwds*)
create a new CryptPolicy, optionally updating parts of the existing configuration.

Deprecated since version 1.6: Callers of this method should *CryptContext.update()* or *CryptContext.copy()* instead.

3.3.3.2 Introspection

All of the informational methods provided by this class have been deprecated by identical or similar methods in the *CryptContext* class:

has_schemes()
return True if policy defines *any* schemes for use.

Deprecated since version 1.6: applications should use `bool(context.schemes())` instead. see *CryptContext.schemes()*.

schemes(*resolve=False*)
return list of schemes defined in policy.

Deprecated since version 1.6: applications should use *CryptContext.schemes()* instead.

iter_handlers()
return iterator over handlers defined in policy.

Deprecated since version 1.6: applications should use `context.schemes(resolve=True)` instead. see *CryptContext.schemes()*.

get_handler(*name=None*, *category=None*, *required=False*)
return handler as specified by name, or default handler.

Deprecated since version 1.6: applications should use *CryptContext.handler()* instead, though note that the `required` keyword has been removed, and the new method will always act as if `required=True`.

get_options(*name*, *category=None*)
return dictionary of options specific to a given handler.

Deprecated since version 1.6: this method has no direct replacement in the 1.6 api, as there is not a clearly defined use-case. however, examining the output of *CryptContext.to_dict()* should serve as the closest alternative.

handler_is_deprecated(*name*, *category=None*)
check if handler has been deprecated by policy.

Deprecated since version 1.6: this method has no direct replacement in the 1.6 api, as there is not a clearly defined use-case. however, examining the output of *CryptContext.to_dict()* should serve as the closest alternative.

get_min_verify_time(*category=None*)
get min_verify_time setting for policy.

Deprecated since version 1.6: min_verify_time option will be removed entirely in passlib 1.8

Changed in version 1.7: this method now always returns the value automatically calculated by `CryptContext.min_verify_time()`, any value specified by policy is ignored.

3.3.3.3 Exporting

iter_config(*ini=False*, *resolve=False*)
iterate over key/value pairs representing the policy object.

Deprecated since version 1.6: applications should use *CryptContext.to_dict()* instead.

to_dict(*resolve=False*)
export policy object as dictionary of options.

Deprecated since version 1.6: applications should use *CryptContext.to_dict()* instead.

to_file(*stream*, *section='passlib'*)
export policy to file.

Deprecated since version 1.6: applications should use *CryptContext.to_string()* instead, and then write the output to a file as desired.

to_string(*section='passlib'*, *encoding=None*)
export policy to file.

Deprecated since version 1.6: applications should use *CryptContext.to_string()* instead.

Note: CryptPolicy are immutable. Use the *replace()* method to mutate existing instances.

Deprecated since version 1.6.

3.4 passlib.crypto - Cryptographic Helper Functions

This module is primarily used as an internal support module. It contains cryptography utility functions used by Passlib. However, end-user applications may find some of the functions useful.

It contains the following submodules:

3.4.1 passlib.crypto.digest - Hash & Related Helpers

New in version 1.7.

This module provides various cryptographic support functions used by Passlib to implement the various password hashes it provides, as well as paper over some VM & version incompatibilities.

3.4.1.1 Hash Functions

passlib.crypto.digest.**norm_hash_name**(*name*, *format='hashlib'*)

Normalize hash function name (convenience wrapper for *lookup_hash()*).

Parameters

- **name** – Original hash function name.

  This name can be a Python hashlib digest name, a SCRAM mechanism name, IANA assigned hash name, etc. Case is ignored, and underscores are converted to hyphens.

- **format** – Naming convention to normalize to. Possible values are:

  - "hashlib" (the default) - normalizes name to be compatible with Python's hashlib.
  - "iana" - normalizes name to IANA-assigned hash function name. For hashes which IANA hasn't assigned a name for, this issues a warning, and then uses a heuristic to return a "best guess" name.

Returns Hash name, returned as native str.

passlib.crypto.digest.**lookup_hash**(*digest*, *return_unknown=False*, *required=True*)

Returns a *HashInfo* record containing information about a given hash function. Can be used to look up a hash constructor by name, normalize hash name representation, etc.

Parameters

- **digest** – This can be any of:

  - A string containing a hashlib digest name (e.g. "sha256"),
  - A string containing an IANA-assigned hash name,
  - A digest constructor function (e.g. hashlib.sha256).

  Case is ignored, underscores are converted to hyphens, and various other cleanups are made.

- **required** – By default (True), this function will throw an *UnknownHashError* if no hash constructor can be found, or if the hash is not actually available.

  If this flag is False, it will instead return a dummy HashInfo record which will defer throwing the error until it's constructor function is called. This is mainly used by *norm_hash_name()*.

- **return_unknown** – Deprecated since version 1.7.3: deprecated, and will be removed in passlib 2.0. this acts like inverse of **required**.

Returns HashInfo *HashInfo* instance containing information about specified digest.

Multiple calls resolving to the same hash should always return the same HashInfo instance.

Note: lookup_hash() supports all hashes available directly in hashlib, as well as offered through hashlib.new(). It will also fallback to passlib's builtin MD4 implementation if one is not natively available.

---

class passlib.crypto.digest.**HashInfo**
Record containing information about a given hash algorithm, as returned *lookup_hash()*.

This class exposes the following attributes:

const = None
Hash constructor function (e.g. `hashlib.sha256()`)

digest_size = None
Hash's digest size

block_size = None
Hash's block size

name = None
Canonical / hashlib-compatible name (e.g. `"sha256"`).

iana_name = None
IANA assigned name (e.g. `"sha-256"`), may be `None` if unknown.

aliases = ()
Tuple of other known aliases (may be empty)

supported
whether hash is available for use (if False, constructor will throw UnknownHashError if called)

This object can also be treated a 3-element sequence containing `(const, digest_size, block_size)`.

3.4.1.2 PKCS#5 Key Derivation Functions

passlib.crypto.digest.**pbkdf1**(*digest*, *secret*, *salt*, *rounds*, *keylen=None*)
pkcs#5 password-based key derivation v1.5

Parameters

- **digest** – digest name or constructor.
- **secret** – secret to use when generating the key. may be `bytes` or `unicode` (encoded using UTF-8).
- **salt** – salt string to use when generating key. may be `bytes` or `unicode` (encoded using UTF-8).
- **rounds** – number of rounds to use to generate key.
- **keylen** – number of bytes to generate (if omitted / `None`, uses digest's native size)

Returns raw `bytes` of generated key

Note: This algorithm has been deprecated, new code should use PBKDF2. Among other limitations, `keylen` cannot be larger than the digest size of the specified hash.

---

passlib.crypto.digest.**pbkdf2_hmac**(*digest*, *secret*, *salt*, *rounds*, *keylen=None*)
pkcs#5 password-based key derivation v2.0 using HMAC + arbitrary digest.

Parameters

- **digest** – digest name or constructor.
- **secret** – passphrase to use to generate key. may be `bytes` or `unicode` (encoded using UTF-8).

- **salt** – salt string to use when generating key. may be `bytes` or `unicode` (encoded using UTF-8).
- **rounds** – number of rounds to use to generate key.
- **keylen** – number of bytes to generate. if omitted / `None`, will use digest's native output size.

Returns raw bytes of generated key

Changed in version 1.7: This function will use the first available of the following backends:

- fastpbk2
- `hashlib.pbkdf2_hmac()` (only available in py2 >= 2.7.8, and py3 >= 3.4)
- builtin pure-python backend

See *`passlib.crypto.digest.PBKDF2_BACKENDS`* to determine which backend(s) are in use.

`passlib.crypto.digest.`**`PBKDF2_BACKENDS`**
List of the pbkdf2 backends in use (listed in order of priority).

New in version 1.7.

Note: The details of PBKDF1 and PBKDF2 are specified in **RFC 2898**.

3.4.2 `passlib.crypto.des` - DES routines

Changed in version 1.7: This module was relocated from `passlib.utils.des`; the old location will be removed in Passlib 2.0.

> **Warning:** NIST has declared DES to be "inadequate" for cryptographic purposes. These routines, and the password hashes based on them, should not be used in new applications.

This module contains routines for encrypting blocks of data using the DES algorithm. Note that these functions do not support multi-block operation or decryption, since they are designed primarily for use in password hash algorithms (such as *`des_crypt`* and *`bsdi_crypt`*).

`passlib.crypto.des.`**`expand_des_key`**(*key*)
convert DES from 7 bytes to 8 bytes (by inserting empty parity bits)

`passlib.crypto.des.`**`des_encrypt_block`**(*key*, *input*, *salt=0*, *rounds=1*)
encrypt single block of data using DES, operates on 8-byte strings.

Parameters

- **key** – DES key as 7 byte string, or 8 byte string with parity bits (parity bit values are ignored).
- **input** – plaintext block to encrypt, as 8 byte string.
- **salt** – Optional 24-bit integer used to mutate the base DES algorithm in a manner specific to *`des_crypt`* and its variants. The default value `0` provides the normal (unsalted) DES behavior. The salt functions as follows: if the `i`'th bit of `salt` is set, bits `i` and `i+24` are swapped in the DES E-box output.

- **rounds** – Optional number of rounds of to apply the DES key schedule. the default (`rounds=1`) provides the normal DES behavior, but *des_crypt* and its variants use alternate rounds values.

Raises

- **TypeError** – if any of the provided args are of the wrong type.
- **ValueError** – if any of the input blocks are the wrong size, or the salt/rounds values are out of range.

Returns resulting 8-byte ciphertext block.

`passlib.crypto.des.`**des_encrypt_int_block**(*key*, *input*, *salt=0*, *rounds=1*)

encrypt single block of data using DES, operates on 64-bit integers.

this function is essentially the same as *des_encrypt_block()*, except that it operates on integers, and will NOT automatically expand 56-bit keys if provided (since there's no way to detect them).

Parameters

- **key** – DES key as 64-bit integer (the parity bits are ignored).
- **input** – input block as 64-bit integer
- **salt** – optional 24-bit integer used to mutate the base DES algorithm. defaults to `0` (no mutation applied).
- **rounds** – optional number of rounds of to apply the DES key schedule. defaults to `1`.

Raises

- **TypeError** – if any of the provided args are of the wrong type.
- **ValueError** – if any of the input blocks are the wrong size, or the salt/rounds values are out of range.

Returns resulting ciphertext as 64-bit integer.

3.5 `passlib.exc` - Exceptions and warnings

This module contains all the custom exceptions & warnings that may be raised by Passlib.

3.5.1 Exceptions

exception `passlib.exc.`**MissingBackendError**

Error raised if multi-backend handler has no available backends; or if specifically requested backend is not available.

`MissingBackendError` derives from `RuntimeError`, since it usually indicates lack of an external library or OS feature. This is primarily raised by handlers which depend on external libraries (which is currently just *bcrypt*).

exception `passlib.exc.`**InternalBackendError**

Error raised if something unrecoverable goes wrong with backend call; such as if `crypt.crypt()` returning a malformed hash.

New in version 1.7.3.

exception passlib.exc.**PasswordValueError**
Error raised if a password can't be hashed / verified for various reasons. This exception derives from the builtin ValueError.

May be thrown directly when password violates internal invariants of hasher (e.g. some don't support NULL characters). Hashers may also throw more specific subclasses, such as PasswordSizeError.

New in version 1.7.3.

exception passlib.exc.**PasswordSizeError**(*max_size*, *msg=None*)
Error raised if a password exceeds the maximum size allowed by Passlib (by default, 4096 characters); or if password exceeds a hash-specific size limitation.

This exception derives from *PasswordValueError* (above).

Many password hash algorithms take proportionately larger amounts of time and/or memory depending on the size of the password provided. This could present a potential denial of service (DOS) situation if a maliciously large password is provided to an application. Because of this, Passlib enforces a maximum size limit, but one which should be *much* larger than any legitimate password. *PasswordSizeError* derives from ValueError.

Note: Applications wishing to use a different limit should set the PASSLIB_MAX_PASSWORD_SIZE environmental variable before Passlib is loaded. The value can be any large positive integer.

---

max_size
indicates the maximum allowed size.

New in version 1.6.

exception passlib.exc.**PasswordTruncateError**(*cls*, *msg=None*)
Error raised if password would be truncated by hash. This derives from *PasswordSizeError* (above).

Hashers such as *bcrypt* can be configured to raises this error by setting truncate_error=True.

max_size
indicates the maximum allowed size.

New in version 1.7.

exception passlib.exc.**PasslibSecurityError**
Error raised if critical security issue is detected (e.g. an attempt is made to use a vulnerable version of a bcrypt backend).

New in version 1.6.3.

exception passlib.exc.**UnknownHashError**(*message=None*, *value=None*)
Error raised by lookup_hash if hash name is not recognized. This exception derives from ValueError.

As of version 1.7.3, this may also be raised if hash algorithm is known, but has been disabled due to FIPS mode (message will include phrase "disabled for fips").

As of version 1.7.4, this may be raised if a *CryptContext* is unable to identify the algorithm used by a password hash.

New in version 1.7.

Changed in version 1.7.4: altered call signature.

3.5.1.1 TOTP Exceptions

exception passlib.exc.**TokenError**(*msg=None*, **args*, ***kwds*)
Base error raised by v:mod:*passlib.totp* when a token can't be parsed / isn't valid / etc. Derives from `ValueError`.

Usually one of the more specific subclasses below will be raised:

- *`MalformedTokenError`* – invalid chars, too few digits
- *`InvalidTokenError`* – no match found
- *`UsedTokenError`* – match found, but token already used

New in version 1.7.

exception passlib.exc.**MalformedTokenError**(*msg=None*, **args*, ***kwds*)
Error raised by *`passlib.totp`* when a token isn't formatted correctly (contains invalid characters, wrong number of digits, etc)

exception passlib.exc.**InvalidTokenError**(*msg=None*, **args*, ***kwds*)
Error raised by *`passlib.totp`* when a token is formatted correctly, but doesn't match any tokens within valid range.

exception passlib.exc.**UsedTokenError**(**args*, ***kwds*)
Error raised by *`passlib.totp`* if a token is reused. Derives from *`TokenError`*.

expire_time = None
optional value indicating when current counter period will end, and a new token can be generated.

New in version 1.7.

3.5.2 Warnings

exception passlib.exc.**PasslibWarning**
base class for Passlib's user warnings, derives from the builtin `UserWarning`.

New in version 1.6.

3.5.2.1 Minor Warnings

exception passlib.exc.**PasslibConfigWarning**
Warning issued when non-fatal issue is found related to the configuration of a *`CryptContext`* instance.

This occurs primarily in one of two cases:

- The CryptContext contains rounds limits which exceed the hard limits imposed by the underlying algorithm.
- An explicit rounds value was provided which exceeds the limits imposed by the CryptContext.

In both of these cases, the code will perform correctly & securely; but the warning is issued as a sign the configuration may need updating.

New in version 1.6.

exception passlib.exc.**PasslibHashWarning**
Warning issued when non-fatal issue is found with parameters or hash string passed to a passlib hash class.

This occurs primarily in one of two cases:

- A rounds value or other setting was explicitly provided which exceeded the handler's limits (and has been clamped by the *relaxed* flag).
- A malformed hash string was encountered which (while parsable) should be re-encoded.

New in version 1.6.

3.5.2.2 Critical Warnings

exception passlib.exc.**PasslibRuntimeWarning**
Warning issued when something unexpected happens during runtime.

The fact that it's a warning instead of an error means Passlib was able to correct for the issue, but that it's anomalous enough that the developers would love to hear under what conditions it occurred.

New in version 1.6.

exception passlib.exc.**PasslibSecurityWarning**
Special warning issued when Passlib encounters something that might affect security.

New in version 1.6.

3.6 passlib.ext.django - Django Password Hashing Plugin

Warning: This extension is a high maintenance, with an uncertain number of users. The current plan is to split this out as a separate package concurrent with Passlib 1.8, and then judge whether it should continue to be maintained in it's own right. See issue 81.

This module contains a Django plugin which overrides all of Django's password hashing functions, replacing them with wrappers around a Passlib *CryptContext* object whose configuration is controlled from Django's `settings`. While this extension's utility is diminished with the advent of Django 1.4's *hashers* framework, this plugin still has a number of uses:

- Make use of the new Django 1.4 *pbkdf2 & bcrypt formats*, even under earlier Django releases.
- Allow your application to work with any password hash format *supported* by Passlib, allowing you to import existing hashes from other systems. Common examples include SHA512-Crypt, PHPass, and BCrypt.
- Set different iterations / cost settings based on the type of user account, and automatically update hashes that use weaker settings when the user logs in.
- Mark any hash algorithms as deprecated, and automatically migrate to stronger hashes when the user logs in.

Note: This plugin should be considered "release candidate" quality. It works, and has good unittest coverage, but has seen only limited real-world use. Please report any issues. It has been tested with Django 1.8 - 3.1.

New in version 1.6.

Changed in version 1.7: Support for Django 1.0 - 1.7 was dropped; now requires Django 1.8 or newer.

Warning: As of Passlib 1.8, this module will require Django 2.2 or newer.

3.6.1 Installation

Installation is simple: once Passlib itself has been installed, just add `"passlib.ext.django"` to Django's `settings.INSTALLED_APPS`, as soon as possible after `django.contrib.auth`.

Once installed, this plugin will automatically monkeypatch Django to use a Passlib `CryptContext` instance in place of the normal Django password authentication routines (as an unfortunate side effect, this disables Django 1.4's hashers framework entirely, though the default configuration supports all the built-in Django 1.4 hashers).

3.6.2 Configuration

While this plugin will function perfectly well without setting any configuration options, you can customize it using the following options in Django's `settings.py`:

`PASSLIB_CONFIG`

This option specifies the CryptContext configuration options that will be used when the plugin is loaded.

- Its value will usually be an INI-formatted string or a dictionary, containing options to be passed to *`CryptContext`*.
- Alternately, it can be the name of any preset supported by *`get_preset_config()`*, such as `"passlib-default"` or `"django-default"`.
- Finally, it can be the special string `"disabled"`, which will disable this plugin.

At any point after this plugin has been loaded, you can serialize its current configuration to a string:

```
>>> from passlib.ext.django.models import password_context
>>> print password_context.to_string()
```

This string can then be modified, and used as the new value of `PASSLIB_CONFIG`.

Note: It is *strongly* recommended to use a configuration which will support the existing Django hashes. Dumping and then modifying one of the preset strings is a good starting point.

`PASSLIB_GET_CATEGORY`

By default, Passlib will assign users to one of three categories: `"superuser"`, `"staff"`, or `None`; based on the attributes of the `User` object. This allows `PASSLIB_CONFIG` to have per-category policies, such as a larger number of iterations for the superuser account.

This option allows overriding the function which performs this mapping, so that more fine-grained / alternate user categories can be used. If specified, the function should have the call syntax `get_category(user) -> category_string|None`.

See also:

See *User Categories* for more details.

`PASSLIB_CONTEXT`

Deprecated since version 1.6: This is a deprecated alias for `PASSLIB_CONFIG`, used by the (undocumented) version of this plugin that was released with Passlib 1.5. It should not be used by new applications.

3.6.3 Module Contents

passlib.ext.django.models.**password_context**
The CryptContext instance that drives this plugin. It can be imported and examined to inspect the current configuration, changes made to it will immediately alter how Django hashes passwords.

(Do not replace the reference with another CryptContext, it will break things; just update the context in-place).

passlib.ext.django.models.**context_changed**()
If the context is modified after loading, call this function to clear internal caches.

passlib.ext.django.utils.**get_preset_config**(*name*)
Returns configuration string for one of the preset strings supported by the PASSLIB_CONFIG setting. Currently supported presets:

- "passlib-default" - default config used by this release of passlib.
- "django-default" - config matching currently installed django version.
- "django-latest" - config matching newest django version (currently same as "django-1.6").
- "django-1.0" - config used by stock Django 1.0 - 1.3 installs
- "django-1.4" - config used by stock Django 1.4 installs
- "django-1.6" - config used by stock Django 1.6 installs

passlib.ext.django.utils.**PASSLIB_DEFAULT**
This constant contains the default configuration for PASSLIB_CONFIG. It provides the following features:

- uses *django_pbkdf2_sha256* as the default algorithm.
- supports all of the Django 1.0-1.4 *hash formats*.
- additionally supports SHA512-Crypt, BCrypt, and PHPass.
- is configured to use a larger number of rounds for the superuser account.
- is configured to automatically migrate all Django 1.0 hashes to use the default hash as soon as each user logs in.

As of Passlib 1.6, it contains the following string:

```
[passlib]

; list of schemes supported by configuration
; currently all django 1.4 hashes, django 1.0 hashes,
; and three common modular crypt format hashes.
schemes =
    django_pbkdf2_sha256, django_pbkdf2_sha1, django_bcrypt,
    django_salted_sha1, django_salted_md5, django_des_crypt, hex_md5,
    sha512_crypt, bcrypt, phpass

; default scheme to use for new hashes
default = django_pbkdf2_sha256

; hashes using these schemes will automatically be re-hashed
; when the user logs in (currently all django 1.0 hashes)
deprecated =
    django_pbkdf2_sha1, django_salted_sha1, django_salted_md5,
    django_des_crypt, hex_md5

; sets some common options, including minimum rounds for two primary hashes.
```

(continues on next page)

(continued from previous page)

```
; if a hash has less than this number of rounds, it will be re-hashed.
all__vary_rounds = 0.05
sha512_crypt__min_rounds = 80000
django_pbkdf2_sha256__min_rounds = 10000

; set somewhat stronger iteration counts for ``User.is_staff``
staff__sha512_crypt__default_rounds = 100000
staff__django_pbkdf2_sha256__default_rounds = 12500

; and even stronger ones for ``User.is_superuser``
superuser__sha512_crypt__default_rounds = 120000
superuser__django_pbkdf2_sha256__default_rounds = 15000
```

3.7 passlib.hash - Password Hashing Schemes

3.7.1 Overview

The passlib.hash module contains all the password hash algorithms built into Passlib. While each hash has its own options and output format, they all inherit from the *PasswordHash* base interface. The following pages describe each hash in detail, including its format, underlying algorithm, and known security issues.

Danger: Many of the hash algorithms listed below are *NOT* secure.

Passlib supports a wide array of hash algorithms, primarily to support legacy data and systems. If you want to choose a secure algorithm for a new application, see the *Quickstart Guide*.

See also:

PasswordHash Tutorial – for general usage examples

3.7.2 Unix Hashes

Aside from "archaic" schemes such as des_crypt, most of the password hashes supported by modern Unix flavors adhere to the *modular crypt format*, allowing them to be easily distinguished when used within the same file. Variants of this format's basic $*scheme*$*salt*$*digest* structure have also been adopted for use by other applications and password hash schemes.

3.7.2.1 Active Unix Hashes

All the following schemes are actively in use by various Unix flavors to store user passwords They all follow the modular crypt format.

passlib.hash.bcrypt - BCrypt

BCrypt was developed to replace *md5_crypt* for BSD systems. It uses a modified version of the Blowfish stream cipher. Featuring a large salt and variable number of rounds, it's currently the default password hash for many systems (notably BSD), and has no known weaknesses. It is one of the four hashes Passlib *recommends* for new applications. This class can be used directly as follows:

```
>>> from passlib.hash import bcrypt

>>> # generate new salt, hash password
>>> h = bcrypt.hash("password")
>>> h
'$2a$12$NT0I31Sa7ihGEWpka9ASYrEFkhuTNeBQ2xfZskIiiJeyFXhRgS.Sy'

>>> # the same, but with an explicit number of rounds
>>> bcrypt.using(rounds=13).hash("password")
'$2b$13$HMQTprwhaUwmir.g.ZYoXuRJhtsbra4uj.qJPHrKsX5nGlhpts0jm'

>>> # verify password
>>> bcrypt.verify("password", h)
True
>>> bcrypt.verify("wrong", h)
False
```

Note: It is strongly recommended that you install bcrypt when using this hash.

See also:

the generic *PasswordHash usage examples*

Interface

class passlib.hash.**bcrypt**

This class implements the BCrypt password hash, and follows the *PasswordHash API*.

It supports a fixed-length salt, and a variable number of rounds.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*str*) – Optional salt string. If not specified, one will be autogenerated (this is recommended). If specified, it must be 22 characters, drawn from the regexp range `[./0-9A-Za-z]`.
- **rounds** (*int*) – Optional number of rounds to use. Defaults to 12, must be between 4 and 31, inclusive. This value is logarithmic, the actual number of iterations used will be `2**rounds` – increasing the rounds by +1 will double the amount of time taken.
- **ident** (*str*) – Specifies which version of the BCrypt algorithm will be used when creating a new hash. Typically this option is not needed, as the default (`"2b"`) is usually the correct choice. If specified, it must be one of the following:
  - `"2"` - the first revision of BCrypt, which suffers from a minor security flaw and is generally not used anymore.
  - `"2a"` - some implementations suffered from rare security flaws, replaced by 2b.
  - `"2y"` - format specific to the *crypt_blowfish* BCrypt implementation, identical to `"2b"` in all but name.
  - `"2b"` - latest revision of the official BCrypt algorithm, current default.

- **truncate_error** (*bool*) – By default, BCrypt will silently truncate passwords larger than 72 bytes. Setting `truncate_error=True` will cause *hash()* to raise a *PasswordTruncateError* instead.

  New in version 1.7.

- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `rounds` that are too small or too large, and `salt` strings that are too long.

  New in version 1.6.

Changed in version 1.6: This class now supports `"2y"` hashes, and recognizes (but does not support) the broken `"2x"` hashes. (see the *crypt_blowfish bug* for details).

Changed in version 1.6: Added a pure-python backend.

Changed in version 1.6.3: Added support for `"2b"` variant.

Changed in version 1.7: Now defaults to `"2b"` variant.

Bcrypt Backends

Warning: Support for `py-bcrypt` and `bcryptor` will be dropped in Passlib 1.8, as these libraries are unmaintained.

This class will use the first available of five possible backends:

1. bcrypt, if installed.
2. py-bcrypt, if installed (DEPRECATED)
3. bcryptor, if installed (DEPRECATED).
4. stdlib's `crypt.crypt()`, if the host OS supports BCrypt (primarily BSD-derived systems).
5. A pure-python implementation of BCrypt, built into Passlib.

If no backends are available, `hash()` and `verify()` will throw *MissingBackendError* when they are invoked. You can check which backend is in use by calling `bcrypt.get_backend()`.

As of Passlib 1.6.3, a one-time check is peformed when the backend is first loaded, to detect the backend's capabilities & bugs. If this check detects a fatal bug, a *PasslibSecurityError* will be raised. This generally means you need to upgrade the external package being used as the backend (this will be detailed in the error message).

Warning: *The pure-python backend (#5) is disabled by default!*

That backend is currently too slow to be usable given the number of rounds required for security. That said, if you have no other alternative and need to use it, set the environmental variable `PASSLIB_BUILTIN_BCRYPT="enabled"` before importing Passlib.

What's "too slow"? Passlib's *rounds selection guidelines* currently require BCrypt be able to do at least 12 cost in under 300ms. By this standard the pure-python backend is 128x too slow under CPython 2.7, and 16x too slow under PyPy 1.8. (speedups are welcome!)

Format & Algorithm

Bcrypt is compatible with the *Modular Crypt Format*, and uses a number of identifying prefixes: `$2$`, `$2a$`, `$2x$`, `$2y$`, and `$2b$`. Each prefix indicates a different revision of the BCrypt algorithm; and all but the `$2b$` identifier are considered deprecated.

An example hash (of `password`) is:

```
$2b$12$GhvMmNVjRW29ulnudl.LbuAnUtN/LRfe1JsBm1Xu6LE3059z5Tr8m
```

Bcrypt hashes have the format `$2a$`*`rounds`*`$`*`saltchecksum`*, where:

- *`rounds`* is a cost parameter, encoded as 2 zero-padded decimal digits, which determines the number of iterations used via *`iterations`*`=2**`*`rounds`* (rounds is 12 in the example).
- *`salt`* is a 22 character salt string, using the characters in the regexp range `[./A-Za-z0-9]` (`GhvMmNVjRW29ulnudl.Lbu` in the example). Note that due to padding bits within the encoding, the last character should always be one of `[.Oeu]`: under some bcrypt implementations, other final characters may result in false negatives when verifying.
- *`checksum`* is a 31 character checksum, using the same characters as the salt (`AnUtN/LRfe1JsBm1Xu6LE3059z5Tr8m` in the example).

While BCrypt's basic algorithm is described in its design document[1], the OpenBSD implementation[2] is considered the canonical reference, even though it differs from the design document in a few small ways.

Security Issues

- Password Truncation.

  While not a security issue per-se, bcrypt does have one major limitation: password are truncated on the first NULL byte (if any), and only the first 72 bytes of a password are hashed... all the rest are ignored. Furthermore, bytes 55-72 are not fully mixed into the resulting hash (citation needed!). To work around both these issues, many applications first run the password through a message digest such as (HMAC-) SHA2-256. Passlib offers the premade *passlib.hash.bcrypt_sha256 - BCrypt+SHA256* to take care of this issue.

Deviations

This implementation of bcrypt differs from others in a few ways:

- Restricted salt string character set:

  BCrypt does not specify what the behavior should be when passed a salt string outside of the regexp range `[./A-Za-z0-9]`. In order to avoid this situation, Passlib strictly limits salts to the allowed character set, and will throw a `ValueError` if an invalid salt character is encountered.

- Unicode Policy:

  The underlying algorithm takes in a password specified as a series of non-null bytes, and does not specify what encoding should be used; though a `us-ascii` compatible encoding is implied by nearly all implementations of bcrypt as well as all known reference hashes.

  In order to provide support for unicode strings, Passlib will encode unicode passwords using `utf-8` before running them through bcrypt. If a different encoding is desired by an application, the password should be encoded before handing it to Passlib.

[1] the bcrypt format specification - http://www.usenix.org/event/usenix99/provos/provos_html/
[2] the OpenBSD BCrypt source - http://www.openbsd.org/cgi-bin/cvsweb/src/lib/libc/crypt/bcrypt.c

- Padding Bits

  BCrypt's base64 encoding results in the last character of the salt encoding only 2 bits of data, the remaining 4 are "padding" bits. Similarly, the last character of the digest contains 4 bits of data, and 2 padding bits. Because of the way they are coded, many BCrypt implementations will reject *all* passwords if these padding bits are not set to 0. Due to a legacy *issue* with Passlib <= 1.5.2, Passlib will print a warning if it encounters hashes with any padding bits set, and then validate the hash as if the padding bits were cleared. (This behavior will eventually be deprecated and such hashes will throw a `ValueError` instead).

- The *crypt_blowfish* 8-bit bug

  Pre-1.1 versions of the crypt_blowfish bcrypt implementation suffered from a serious flaw[3] in how they handled 8-bit passwords. The manner in which the flaw was fixed resulted in *crypt_blowfish* adding support for two new BCrypt hash identifiers:

  `$2x$`, allowing sysadmins to mark any `$2a$` hashes which were potentially generated with the buggy algorithm. Passlib 1.6 recognizes (but does not currently support generating or verifying) these hashes.

  `$2y$`, the default for crypt_blowfish 1.1-1.2, indicates the hash was generated with the canonical OpenBSD-compatible algorithm, and should match *correctly* generated `$2a$` hashes. Passlib 1.6 can generate and verify these hashes.

  As well, crypt_blowfish 1.2 modified the way it generates `$2a$` hashes, so that passwords containing the byte value 0xFF are hashed in a manner incompatible with either the buggy or canonical algorithms. Passlib does not support this algorithmic variant either, though it should be *very* rarely encountered in practice.

  (crypt_blowfish 1.3 switched to the `$2b$` standard as the default)

  Changed in version 1.6.3: Passlib will now throw a *`PasslibSecurityError`* if an attempt is made to use any backend which is vulnerable to this bug.

- The 'BSD wraparound' bug

  OpenBSD <= 5.4, and most bcrypt libraries derived from it's source, are vulnerable to a 'wraparound' bug[4], where passwords larger than 254 characters will be incorrectly hashed using only the first few characters of the string, resulting in a severely weakened hash.

  OpenBSD 5.5 fixed this flaw, and introduced the `$2b$` hash identifier to indicate the hash was generated with the correct algorithm.

  py-bcrypt <= 0.4 is known to be vulnerable to this, as well as the os_crypt backend (if running on a vulnerable operating system).

  Passlib 1.6.3 adds the following:

  - Support for the `$2b$` hash format (though for backward compat it has not been made the default yet).
  - Detects if the active backend is vulnerable to the bug, issues a warning, and enables a workaround so that vulnerable passwords will still be hashed correctly. (This does mean that existing hashes suffering this vulnerability will no longer verify using their correct password).

`passlib.hash.sha256_crypt` - SHA-256 Crypt

SHA-256 Crypt and SHA-512 Crypt were developed in 2008 by Ulrich Drepper[1], designed as the successor to *`md5_crypt`*. They include fixes and advancements such as variable rounds, and use of NIST-approved cryptographic primitives. The design involves repeated composition of the underlying digest algorithm, using various arbitrary permutations of inputs. SHA-512 / SHA-256 Crypt are currently the default password hash for many systems (notably

[3] The flaw in pre-1.1 crypt_blowfish is described here - CVE-2011-2483
[4] The wraparound flaw is described here - http://www.openwall.com/lists/oss-security/2012/01/02/4
[1] Ulrich Drepper's SHA-256/512-Crypt specification, reference implementation, and test vectors - sha-crypt specification

Linux), and have no known weaknesses. SHA-256 Crypt is one of the four hashes Passlib *recommends* for new applications. This class can be used directly as follows:

```
>>> from passlib.hash import sha256_crypt

>>> # generate new salt, hash password
>>> hash = sha256_crypt.hash("password")
>>> hash
'$5$rounds=80000$wnsT7Yr92oJoP28r$cKhJImk5mfuSKV9b3mumNzlbstFUplKtQXXMo4G6Ep5'

>>> # same, but with explict number of rounds
>>> sha256_crypt.using(rounds=12345).hash("password")
'$5$rounds=12345$q3hvJE5mn5jKRsW.$BbbYTFiaImz9rTy03GGi.Jf9YY5bmxN0LU3p3uI1iUB'

>>> # verify password
>>> sha256_crypt.verify("password", hash)
True
>>> sha256_crypt.verify("letmein", hash)
False
```

See also:

- *password hash usage* – for more usage examples
- *sha512_crypt* – the companion 512-bit version of this hash.

Interface

class passlib.hash.**sha256_crypt**

This class implements the SHA256-Crypt password hash, and follows the *PasswordHash API*.

It supports a variable-length salt, and a variable number of rounds.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*str*) – Optional salt string. If not specified, one will be autogenerated (this is recommended). If specified, it must be 0-16 characters, drawn from the regexp range `[./0-9A-Za-z]`.
- **rounds** (*int*) – Optional number of rounds to use. Defaults to 535000, must be between 1000 and 999999999, inclusive.

  Note: per the official specification, when the rounds parameter is set to 5000, it may be omitted from the hash string.

- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `rounds` that are too small or too large, and `salt` strings that are too long.

  New in version 1.6.

Note: This class will use the first available of two possible backends:

- stdlib `crypt()`, if the host OS supports SHA256-Crypt (most Linux systems).

- a pure python implementation of SHA256-Crypt built into Passlib.

You can see which backend is in use by calling the `get_backend()` method.

Format & Algorithm

An example sha256-crypt hash (of the string `password`) is:

```
$5$rounds=80000$wnsT7Yr92oJoP28r$cKhJImk5mfuSKV9b3mumNzlbstFUplKtQXXMo4G6Ep5
```

An sha256-crypt hash string has the format `$5$rounds=`*`rounds`*`$`*`salt`*`$`*`checksum`*, where:

- `$5$` is the prefix used to identify sha256-crypt hashes, following the *Modular Crypt Format*
- *`rounds`* is the decimal number of rounds to use (80000 in the example).
- *`salt`* is 0-16 characters drawn from `[./0-9A-Za-z]`, providing a 96-bit salt (`wnsT7Yr92oJoP28r` in the example).
- *`checksum`* is 43 characters drawn from the same set, encoding a 256-bit checksum (`cKhJImk5mfuSKV9b3mumNzlbstFUplKtQXXMo4G6Ep5` in the example).

The official implementation allows omitting the `rounds` section when it's set to 5000, resulting in an alternate hash format: `$5$`*`salt`*`$`*`checksum`*. (Passlib supports this via the `implicit_rounds` constructor parameter).

The algorithm used by SHA256-Crypt is laid out in detail in the specification document linked to below[1].

Security Issues

- The algorithm's initialization stage contains a loop which varies linearly with the square of the password size; and further loops which vary linearly with the password size * rounds.
  - This means an attacker could provide a maliciously large password at the login screen to attempt a DOS on a publically visible login. For example, a 32kib password would require hashing 1gib of data. Passlib mitigates this by limiting the maximum password size to 4k by default.
  - An attacker could also theoretically determine a password's size by observing the time taken on a successful login, and then attempting verification themselves to find the size password which has an equivalent delay. This has not been applied in practice, probably due to the fact that (for normal passwords < 64 bytes), the contribution of the password size to the overall time taken is below the observable noise level when evesdropping on the timings of successful logins for a single user.

Deviations

This implementation of sha256-crypt differs from the specification, and other implementations, in a few ways:

- Zero-Padded Rounds:

  The specification does not specify how to deal with zero-padding within the rounds portion of the hash. No existing examples or test vectors have zero padding, and allowing it would result in multiple encodings for the same configuration / hash. To prevent this situation, Passlib will throw an error if the rounds parameter in a hash has leading zeros.

- Restricted salt string character set:

  The underlying algorithm can unambiguously handle salt strings which contain any possible byte value besides `\x00` and `$`. However, Passlib strictly limits salts to the *hash64* character set, as nearly all implementations

of sha256-crypt generate and expect salts containing those characters, but may have unexpected behaviors for other character values.

- Unicode Policy:

  The underlying algorithm takes in a password specified as a series of non-null bytes, and does not specify what encoding should be used; though a `us-ascii` compatible encoding is implied by nearly all implementations of sha256-crypt as well as all known reference hashes.

  In order to provide support for unicode strings, Passlib will encode unicode passwords using `utf-8` before running them through sha256-crypt. If a different encoding is desired by an application, the password should be encoded before handing it to Passlib.

`passlib.hash.sha512_crypt` - SHA-512 Crypt

Defined by the same specification as *sha256_crypt*, SHA512-Crypt is identical to SHA256-Crypt in almost every way, including design and security issues. The only difference is the doubled digest size; while this provides some increase in security, it's also a bit slower 32 bit operating systems.

See also:

- *password hash usage* – for examples of how to use this class via the common hash interface.
- *sha256_crypt* – the companion 256-bit version of this hash.

Interface

class passlib.hash.**sha512_crypt**

This class implements the SHA512-Crypt password hash, and follows the *PasswordHash API*.

It supports a variable-length salt, and a variable number of rounds.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*str*) – Optional salt string. If not specified, one will be autogenerated (this is recommended). If specified, it must be 0-16 characters, drawn from the regexp range `[./0-9A-Za-z]`.
- **rounds** (*int*) – Optional number of rounds to use. Defaults to 656000, must be between 1000 and 999999999, inclusive.

  Note: per the official specification, when the rounds parameter is set to 5000, it may be omitted from the hash string.

- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `rounds` that are too small or too large, and `salt` strings that are too long.

  New in version 1.6.

Note: This class will use the first available of two possible backends:

- stdlib `crypt()`, if the host OS supports SHA512-Crypt (most Linux systems).

- a pure python implementation of SHA512-Crypt built into passlib.

You can see which backend is in use by calling the `get_backend()` method.

Format & Algorithm

SHA512-Crypt is defined by the same specification as SHA256-Crypt. The format and algorithm are exactly the same, except for the following notable differences:

- it uses the *modular crypt prefix* `$6$`, whereas SHA256-Crypt uses `$5$`.
- it uses the SHA-512 message digest in place of the SHA-256 message digest.
- its output hash is correspondingly larger in size, with an 86-character encoded checksum, instead of 43 characters.

See *sha256_crypt* for the format and algorithm descriptions, as well as security notes.

Special note should be made of the following fallback helper, which is not an actual hash scheme, but implements the "disabled account marker" found in many Linux & BSD password files:

`passlib.hash.unix_disabled` - Unix Disabled Account Helper

This class does not provide an encryption scheme, but instead provides a helper for handling disabled password fields as found in unix `/etc/shadow` files. This class is mainly useful only for plugging into a *CryptContext* instance. It can be used directly as follows:

```
>>> from passlib.hash import unix_disabled

>>> # 'hashing' a password always results in "!" or "*"
>>> unix_disabled.hash("password")
'!'

>>> # verifying will fail for all passwords and hashes
>>> unix_disabled.verify("password", "!")
False
>>> unix_disabled.verify("letmein", "*NOPASSWORD*")
False

>>> # this class should identify all strings which aren't
>>> # valid Unix crypt() output, while leaving MCF hashes alone
>>> unix_disabled.identify('!')
True
>>> unix_disabled.identify('')
True
>>> unix_disabled.identify("$1$somehash")
False
```

Interface

class passlib.hash.**unix_disabled**

This class provides disabled password behavior for unix shadow files, and follows the *PasswordHash API*.

This class does not implement a hash, but instead matches the "disabled account" strings found in `/etc/shadow` on most Unix variants. "encrypting" a password will simply return the disabled account marker. It will reject all passwords, no matter the hash string. The *hash()* method supports one optional keyword:

Parameters **marker** (*str*) – Optional marker string which overrides the platform default used to indicate a disabled account.

If not specified, this will default to `"*"` on BSD systems, and use the Linux default `"!"` for all other platforms. (`unix_disabled.default_marker` will contain the default value)

New in version 1.6: This class was added as a replacement for the now-deprecated *unix_fallback* class, which had some undesirable features.

Deprecated Interface

class passlib.hash.**unix_fallback**

This class provides the fallback behavior for unix shadow files, and follows the *PasswordHash API*.

This class does not implement a hash, but instead provides fallback behavior as found in /etc/shadow on most unix variants. If used, should be the last scheme in the context.

- this class will positively identify all hash strings.
- for security, passwords will always hash to `!`.
- it rejects all passwords if the hash is NOT an empty string (`!` or `*` are frequently used).
- by default it rejects all passwords if the hash is an empty string, but if `enable_wildcard=True` is passed to verify(), all passwords will be allowed through if the hash is an empty string.

Deprecated since version 1.6: This has been deprecated due to its "wildcard" feature, and will be removed in Passlib 1.8. Use *unix_disabled* instead.

Deviations

According to the Linux `shadow` man page, an empty string is treated as a wildcard by Linux, allowing all passwords. For security purposes, this behavior is NOT supported; empty strings are treated the same as `!` or `*`.

3.7.2.2 Deprecated Unix Hashes

The following schemes are supported by various Unix systems using the modular crypt format, but are no longer considered secure, and have been deprecated in favor of the *Active Unix Hashes* (above).

- *passlib.hash.bsd_nthash* - FreeBSD's MCF-compatible encoding of *nthash* digests

passlib.hash.md5_crypt - MD5 Crypt

Danger: This algorithm is not considered secure by modern standards. It should only be used when verifying existing hashes, or when interacting with applications that require this format. For new code, see the list of *recommended hashes*.

This algorithm was developed for FreeBSD in 1994 by Poul-Henning Kamp, to replace the aging *passlib.hash.des_crypt*. It has since been adopted by a wide variety of other Unix flavors, and is found in many other contexts as well. Due to its origins, it's sometimes referred to as "FreeBSD MD5 Crypt". Security-wise it should now be

considered weak, and most Unix flavors have since replaced it with stronger schemes (such as *sha512_crypt* and *bcrypt*).

This is also referred to on Cisco IOS systems as a "type 5" hash. The format and algorithm are identical, though Cisco seems to require 4 salt characters instead of the full 8 characters used by most systems[3].

The `md5_crypt` class can be can be used directly as follows:

```
>>> from passlib.hash import md5_crypt

>>> # generate new salt, hash password
>>> h = md5_crypt.hash("password")
>>> h
'$1$3azHgidD$SrJPt7B.9rekpmwJwtON31'

>>> # verify the password
>>> md5_crypt.verify("password", h)
True
>>> md5_crypt.verify("secret", h)
False

>>> # hash password using cisco-compatible 4-char salt
>>> md5_crypt.using(salt_size=4).hash("password")
'$1$wu98$9UuD3hvrwehnqyF1D548N0'
```

See also:

- *password hash usage* – for more usage examples
- *apr_md5_crypt* – Apache's variant of this algorithm.

Interface

class passlib.hash.**md5_crypt**

This class implements the MD5-Crypt password hash, and follows the *PasswordHash API*.

It supports a variable-length salt.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*str*) – Optional salt string. If not specified, one will be autogenerated (this is recommended). If specified, it must be 0-8 characters, drawn from the regexp range `[./0-9A-Za-z]`.
- **salt_size** (*int*) – Optional number of characters to use when autogenerating new salts. Defaults to 8, but can be any value between 0 and 8. (This is mainly needed when generating Cisco-compatible hashes, which require `salt_size=4`).
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `salt` strings that are too long.

  New in version 1.6.

Note: This class will use the first available of two possible backends:

[3] Note about Cisco Type 5 salt size - http://serverfault.com/a/46399.

- stdlib `crypt()`, if the host OS supports MD5-Crypt (most Unix systems).
- a pure python implementation of MD5-Crypt built into Passlib.

You can see which backend is in use by calling the `get_backend()` method.

Format

An example md5-crypt hash (of the string `password`) is `$1$5pZSV9va$azfrPr6af3Fc7dLblQXVa0`.

An md5-crypt hash string has the format `$1$`*`salt`*`$`*`checksum`*, where:

- `$1$` is the prefix used to identify md5-crypt hashes, following the *Modular Crypt Format*
- *`salt`* is 0-8 characters drawn from the regexp range `[./0-9A-Za-z]`; providing a 48-bit salt (`5pZSV9va` in the example).
- *`checksum`* is 22 characters drawn from the same character set as the salt; encoding a 128-bit checksum (`azfrPr6af3Fc7dLblQXVa0` in the example).

Algorithm

The MD5-Crypt algorithm[1] calculates a checksum as follows:

1. A password string and salt string are provided.

   (The salt should not include the magic prefix, it should match the string referred to as *`salt`* in the format section, above).
2. If needed, the salt should be truncated to a maximum of 8 characters.
3. Start MD5 digest B.
4. Add the password to digest B.
5. Add the salt to digest B.
6. Add the password to digest B.
7. Finish MD5 digest B.
8. Start MD5 digest A.
9. Add the password to digest A.
10. Add the constant string `$1$` to digest A. (The Apache variant of MD5-Crypt uses `$apr1$` instead, this is the only change made by this variant).
11. Add the salt to digest A.
12. For each block of 16 bytes in the password string, add digest B to digest A.
13. For the remaining N bytes of the password string, add the first N bytes of digest B to digest A.
14. For each bit in the binary representation of the length of the password string; starting with the lowest value bit, up to and including the largest-valued bit that is set to `1`:
    a. If the current bit is set 1, add the first character of the password to digest A.
    b. Otherwise, add a NULL character to digest A.

[1] The authoritative reference for MD5-Crypt is Poul-Henning Kamp's original FreeBSD implementation - http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/lib/libcrypt/crypt.c?rev=1.2

(If the password is the empty string, step 14 is omitted entirely).

15. Finish MD5 digest A.
16. For 1000 rounds (round values 0..999 inclusive), perform the following steps:
    a. Start MD5 Digest C for the round.
    b. If the round is odd, add the password to digest C.
    c. If the round is even, add the previous round's result to digest C (for round 0, add digest A instead).
    d. If the round is not a multiple of 3, add the salt to digest C.
    e. If the round is not a multiple of 7, add the password to digest C.
    f. If the round is even, add the password to digest C.
    g. If the round is odd, add the previous round's result to digest C (for round 0, add digest A instead).
    h. Use the final value of MD5 digest C as the result for this round.
17. Transpose the 16 bytes of the final round's result in the following order: `12,6,0,13,7,1,14,8,2,15,9,3,5,10,4,11`.
18. Encode the resulting 16 byte string into a 22 character *hash64*-encoded string (the 2 msb bits encoded by the last hash64 character are used as 0 padding). This results in the portion of the md5 crypt hash string referred to as *checksum* in the format section.

Security Issues

MD5-Crypt has a couple of issues which have weakened severely:

- It relies on the MD5 message digest, for which theoretical pre-image attacks exist[2].
- More seriously, its fixed number of rounds (combined with the availability of high-throughput MD5 implementations) means this algorithm is increasingly vulnerable to brute force attacks. It is this issue which has motivated its replacement by new algorithms such as *bcrypt* and *sha512_crypt*.

Deviations

Passlib's implementation of md5-crypt differs from the reference implementation (and others) in two ways:

- Restricted salt string character set:

  The underlying algorithm can unambiguously handle salt strings which contain any possible byte value besides `\x00` and `$`. However, Passlib strictly limits salts to the *hash64* character set, as nearly all implementations of md5-crypt generate and expect salts containing those characters, but may have unexpected behaviors for other character values.

- Unicode Policy:

  The underlying algorithm takes in a password specified as a series of non-null bytes, and does not specify what encoding should be used; though a `us-ascii` compatible encoding is implied by nearly all implementations of md5-crypt as well as all known reference hashes.

  In order to provide support for unicode strings, Passlib will encode unicode passwords using `utf-8` before running them through md5-crypt. If a different encoding is desired by an application, the password should be encoded before handing it to Passlib.

[2] Security issues with MD5 - http://en.wikipedia.org/wiki/MD5#Security.

passlib.hash.sha1_crypt - SHA-1 Crypt

SHA1-Crypt is a hash algorithm introduced by NetBSD in 2004. It's based on a variation of the PBKDF1 algorithm, and supports a large salt and variable number of rounds.

See also:

password hash usage – for examples of how to use this class via the common hash interface.

Interface

class passlib.hash.**sha1_crypt**

This class implements the SHA1-Crypt password hash, and follows the *PasswordHash API*.

It supports a variable-length salt, and a variable number of rounds.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*str*) – Optional salt string. If not specified, an 8 character one will be autogenerated (this is recommended). If specified, it must be 0-64 characters, drawn from the regexp range `[./0-9A-Za-z]`.
- **salt_size** (*int*) – Optional number of bytes to use when autogenerating new salts. Defaults to 8 bytes, but can be any value between 0 and 64.
- **rounds** (*int*) – Optional number of rounds to use. Defaults to 480000, must be between 1 and 4294967295, inclusive.
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `rounds` that are too small or too large, and `salt` strings that are too long.

  New in version 1.6.

Note: This class will use the first available of two possible backends:

- stdlib `crypt()`, if the host OS supports sha1-crypt (NetBSD).
- a pure python implementation of sha1-crypt built into Passlib.

You can see which backend is in use by calling the `get_backend()` method.

Format

An example hash (of `password`) is `$sha1$40000$jtNX3nZ2$hBNaIXkt4wBI2o5rsi8KejSjNqIq`. An sha1-crypt hash string has the format `$sha1$`*`rounds`*`$`*`salt`*`$`*`checksum`*, where:

- `$sha1$` is the prefix used to identify sha1-crypt hashes, following the *Modular Crypt Format*
- *rounds* is the decimal number of rounds to use (40000 in the example).
- *salt* is 0-64 characters drawn from `[./0-9A-Za-z]` (`jtNX3nZ2` in the example).
- *checksum* is 28 characters drawn from the same set, encoding a 168-bit checksum. (`hBNaIXkt4wBI2o5rsi8KejSjNqIq/` in the example).

Algorithm

The checksum is calculated using a modified version of PBKDF1[3], replacing its use of the SHA1 message digest with HMAC-SHA1, (which does not suffer from the current vulnerabilities that SHA1 itself does, as well as providing some of the advancements made in PBKDF2).

- first, the HMAC-SHA1 digest of *salt*`$sha1$`*rounds* is generated, using the password as the HMAC-SHA1 key.
- then, for *rounds*`-1` iterations, the previous HMAC-SHA1 digest is fed back through HMAC-SHA1, again using the password as the HMAC-SHA1 key.
- the checksum is then rendered into hash-64 format using an ordering that roughly corresponds to big-endian encoding of 24-bit chunks (see `passlib.hash.sha1_crypt._chk_offsets` for exact byte order).

Deviations

This implementation of sha1-crypt differs from the NetBSD implementation in a few ways:

- Default Rounds:

  The NetBSD implementation randomly varies the actual number of rounds when generating a new configuration string, in order to decrease predictability. This feature is provided by Passlib to *all* hashes, via the `CryptContext` class, and so it omitted from this implementation.

- Zero-Padded Rounds:

  The specification does not specify how to deal with zero-padding within the rounds portion of the hash. No existing examples or test vectors have zero padding, and allowing it would result in multiple encodings for the same configuration / hash. To prevent this situation, Passlib will throw an error if the rounds in a hash have leading zeros.

- Restricted salt string character set:

  The underlying algorithm can unambiguously handle salt strings which contain any possible byte value besides `\x00` and `$`. However, Passlib strictly limits salts to the *hash64* character set, as nearly all implementations of sha1-crypt generate and expect salts containing those characters.

- Unicode Policy:

  The underlying algorithm takes in a password specified as a series of non-null bytes, and does not specify what encoding should be used; though a `us-ascii` compatible encoding is implied by nearly all known reference hashes.

  In order to provide support for unicode strings, Passlib will encode unicode passwords using `utf-8` before running them through sha1-crypt. If a different encoding is desired by an application, the password should be encoded before handing it to Passlib.

`passlib.hash.sun_md5_crypt` - Sun MD5 Crypt

This algorithm was developed by Alec Muffett[1] for Solaris, as a replacement for the aging *des_crypt*. It was introduced in Solaris 9u2. While based on the MD5 message digest, it has very little at all in common with the *md5_crypt* algorithm. It supports 32 bit variable rounds and an 8 character salt.

See also:

[3] rfc defining PBKDF1 & PBKDF2 - http://tools.ietf.org/html/rfc2898 -
[1] Overview of & motivations for the algorithm - http://dropsafe.crypticide.com/article/1389

password hash usage – for examples of how to use this class via the common hash interface.

Note: The original Solaris implementation has some hash encoding quirks which may not be properly accounted for in Passlib. Until more user feedback and sample hashes have been gathered, *caveat emptor*.

Interface

class passlib.hash.**sun_md5_crypt**

This class implements the Sun-MD5-Crypt password hash, and follows the *PasswordHash API*.

It supports a variable-length salt, and a variable number of rounds.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*str*) – Optional salt string. If not specified, a salt will be autogenerated (this is recommended). If specified, it must be drawn from the regexp range `[./0-9A-Za-z]`.
- **salt_size** (*int*) – If no salt is specified, this parameter can be used to specify the size (in characters) of the autogenerated salt. It currently defaults to 8.
- **rounds** (*int*) – Optional number of rounds to use. Defaults to 34000, must be between 0 and 4294963199, inclusive.
- **bare_salt** (*bool*) – Optional flag used to enable an alternate salt digest behavior used by some hash strings in this scheme. This flag can be ignored by most users. Defaults to `False`. (see *Bare Salt Issue* for details).
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `rounds` that are too small or too large, and `salt` strings that are too long.

  New in version 1.6.

Format

An example hash (of `passwd`) is `$md5,rounds=5000$GUBv0xjJ$$mSwgIswdjlTY0YxV7HBVm0`. A sun-md5-crypt hash string has the format `$md5,rounds=`*rounds*`$`*salt*`$$`*checksum*, where:

- `$md5,` is the prefix used to identify the hash.
- *rounds* is the decimal number of rounds to use (5000 in the example).
- *salt* is 0-8 salt characters drawn from `[./0-9A-Za-z]` (`GUBv0xjJ` in the example).
- *checksum* is 22 characters drawn from the same set, encoding a 128-bit checksum (`mSwgIswdjlTY0YxV7HBVm0` in the example).

An alternate format, `$md5$`*salt*`$$`*checksum* is used when the rounds value is 0.

There also exists some hashes which have only a single `$` between the salt and the checksum; these have a slightly different checksum calculation (see *Bare Salt Issue* for details).

Note: Solaris seems to deviate from the *Modular Crypt Format* in that it considers `,` to indicate the end of the identifier in addition to `$`.

Algorithm

The algorithm used is based around the MD5 message digest and the "Muffett Coin Toss" algorithm.

1. Given a password, the number of rounds, and a salt string.
2. an initial MD5 digest is created from the concatenation of the password, and the configuration string (using the format `$md5,rounds=`*`rounds`*`$`*`salt`*`$`, or `$md5$`*`salt`*`$` if rounds is 0).

   (See *Bare Salt Issue* for details about an issue affecting this step)
3. **for rounds+4096 iterations, a new digest is created:**
   i. a buffer is initialized, containing the previous round's MD5 digest (for the first round, the digest from step 2 is used).
   ii. `MuffetCoinToss(rounds, previous digest)` is called, resulting in a 0 or 1.
   iii. If step 3.ii results in a 1, a constant data string is added to the buffer; if the result is a 0, the string is not added for this round. The constant data string is a 1517 byte excerpt from Hamlet[2] (`To be, or not to be...all my sins remember'd.\n`), including an appended null character.
   iv. the current iteration as a zero-indexed integer is converted to a string (not zero-padded) and added to the buffer.
   v. the output for this iteration is the MD5 digest of the buffer's contents.
4. The final digest is then encoded into `hash64` format using the same transposed byte order that *`md5_crypt`* uses, and returned.

Muffet Coin Toss

The Muffet Coin Toss algorithm is as follows: Given the current round number, and a 16 byte MD5 digest, it returns a 0 or 1, using the following formula:

Note: All references below to a specific bit of the digest should be interpreted mod 128. All references below to a specific byte of the digest should be interpreted mod 16.

---

1. A 8-bit integer *`X`* is generated from the following formula: for each *`i`* in 0..7 inclusive:
   - let *`A`* be the *`i`*'th byte of the digest, as an 8-bit int.
   - let *`B`* be the *`i`*`+3`'rd byte of the digest, as an 8-bit int.
   - let *`R`* be *`A`* shifted right by *`B`* `% 5` bits.
   - let *`V`* be the *`R`*'th byte of the digest.
   - if the *`A`* `% 8`'th bit of *`B`* is 1, divide *`V`* by 2.
   - use the *`V`*'th bit of the digest as the *`i`*'th bit of *`X`*.
2. Another 8-bit integer, *`Y`*, is generated exactly the same manner as *`X`*, except that:
   - *`A`* is the *`i`*`+8`'th byte of the digest,
   - *`B`* is the *`i`*`+11`'th byte of the digest.
3. if bit *`round`* of the digest is 1, *`X`* is divided by 2.
4. if bit *`round`*`+64` of the digest is 1, *`Y`* is divided by 2.

[2] The source of Hamlet's speech, used byte-for-byte as the constant data - http://www.ibiblio.org/pub/docs/books/gutenberg/etext98/2ws2610.txt

5. the final result is *X*'th bit of the digest XORed against *Y*'th bit of the digest.

Bare Salt Issue

According to the only existing documentation of this algorithm[1], its hashes were supposed to have the format `$md5$salt$checksum`, and include only the bare string `$md5$salt` in the salt digest step (see *step 2*, above).

However, almost all hashes encountered in production environments have the format `$md5$salt$$checksum` (note the double `$$`). Unfortunately, it is not merely a cosmetic difference: hashes of this format incorporate the first `$` after the salt within the salt digest step, so the resulting checksum is different.

The documentation hints that this stems from a bug within the production implementation's parser. This bug causes the implementation to return `$$`-format hashes when passed a configuration string that ends with `$`. It returns the intended original format & checksum only if there is at least one letter after the `$`, e.g. `$md5$salt$x`.

Passlib attempts to accommodate both formats using the special `bare_salt` keyword. It is set to `True` to indicate a configuration or hash string which contains only a single `$`, and does not incorporate it into the hash calculation. The `$$` hash is encountered more often in production since it seems the Solaris salt generator always appends a `$`; because of this `bare_salt=False` was chosen as the default, so that hashes will be generated which by default conform to what users are used to.

Deviations

Passlib's implementation of Sun-MD5-Crypt deliberately deviates from the official implementation in the following ways:

- Unicode Policy:

  The underlying algorithm takes in a password specified as a series of non-null bytes, and does not specify what encoding should be used; though a `us-ascii` compatible encoding is implied by all known reference hashes.

  In order to provide support for unicode strings, Passlib will encode unicode passwords using `utf-8` before running them through sun-md5-crypt. If a different encoding is desired by an application, the password should be encoded before handing it to Passlib.

- Rounds encoding

  The underlying scheme implicitly allows rounds to have zero padding (e.g. `$md5,rounds=001$abc$`), and also allows 0 rounds to be specified two ways (`$md5$abc$` and `$md5,rounds=0$abc$`). Allowing either of these would result in multiple possible checksums for the same password & salt. To prevent ambiguity, Passlib will throw a `ValueError` if the rounds value is zero-padded, or specified explicitly as 0 (e.g. `$md5,rounds=0$abc$`).

Given the lack of documentation, lack of test vectors, and known bugs which accompany the original Solaris implementation, Passlib may not accurately be able to generate and verify all hashes encountered in a Solaris environment. Issues of concern include:

- Some hashes found on the web use a `$` in place of the `,`. It is unclear whether this is an accepted alternate format or just a typo, nor whether this is supposed to affect the checksum in the resulting hash string.
- The current implementation needs addition test vectors; especially ones which contain an explicitly specific number of rounds.
- More information is needed about the parsing / formatting issue described in the *Bare Salt Issue* section.

3.7.2.3 Archaic Unix Hashes

The following schemes are supported by certain Unix systems, but are considered particularly archaic: Not only do they predate the modular crypt format, but they're based on the outmoded DES block cipher, and are woefully insecure:

passlib.hash.des_crypt - DES Crypt

Danger: This algorithm is dangerously insecure by modern standards. It is trivially broken, and should not be used if at all possible. For new code, see the list of *recommended hashes*.

This class implements the original DES-based Unix Crypt algorithm. While no longer in active use in most places, it is supported for legacy purposes by many Unix flavors. It can used directly as follows:

```
>>> from passlib.hash import des_crypt

>>> # generate new salt, hash password
>>> hash = des_crypt.hash("password")
'JQMuyS6H.AGMo'

>>> # verify the password
>>> des_crypt.verify("password", hash)
True
>>> des_crypt.verify("letmein", hash)
False
```

See also:

the generic *PasswordHash usage examples*

Interface

class passlib.hash.**des_crypt**

This class implements the des-crypt password hash, and follows the *PasswordHash API*.

It supports a fixed-length salt.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*str*) – Optional salt string. If not specified, one will be autogenerated (this is recommended). If specified, it must be 2 characters, drawn from the regexp range `[./0-9A-Za-z]`.
- **truncate_error** (*bool*) – By default, des_crypt will silently truncate passwords larger than 8 bytes. Setting `truncate_error=True` will cause *hash()* to raise a *PasswordTruncateError* instead.

  New in version 1.7.
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `salt` strings that are too long.

  New in version 1.6.

Note: This class will use the first available of two possible backends:

- stdlib `crypt()`, if the host OS supports DES-Crypt (most Unix systems).
- a pure Python implementation of DES-Crypt built into Passlib.

You can see which backend is in use by calling the `get_backend()` method.

Format

A des-crypt hash string consists of 13 characters, drawn from `[./0-9A-Za-z]`. The first 2 characters form a *hash64*-encoded 12 bit integer used as the salt, with the remaining characters forming a hash64-encoded 64-bit integer checksum.

A des-crypt configuration string is also accepted by this module, consists of only the first 2 characters, corresponding to the salt only.

An example hash (of the string `password`) is `JQMuyS6H.AGMo`, where the salt is `JQ`, and the checksum `MuyS6H.AGMo`.

Algorithm

The checksum is formed by a modified version of the DES cipher in encrypt mode:

1. Given a password string and a salt string.
2. The 2 character salt string is decoded to a 12-bit integer salt value; The salt string uses little-endian *hash64* encoding.
3. If the password is less than 8 bytes, it's NULL padded at the end to 8 bytes.
4. The lower 7 bits of the first 8 bytes of the password are used to form a 56-bit integer; with the first byte providing the most significant 7 bits, and the 8th byte providing the least significant 7 bits.

   The remainder of the password (if any) is ignored.
5. 25 repeated rounds of modified DES encryption are performed; starting with a null input block, and using the 56-bit integer from step 4 as the DES key.

   The salt is used to to mutate the normal DES encrypt operation by swapping bits *i* and *i*+24 in the DES E-Box output if and only if bit *i* is set in the salt value. Thus, if the salt is set to 0, normal DES encryption is performed. (This was intended to prevent optimized implementations of regular DES encryption to be useful in attacking this algorithm).
6. The 64-bit result of the last round of step 5 is then lsb-padded with 2 zero bits.
7. The resulting 66-bit integer is encoded in big-endian order using the *hash64-big* format.

Security Issues

DES-Crypt is no longer considered secure, for a variety of reasons:

- Its use of the DES stream cipher, which is vulnerable to practical pre-image attacks, and considered broken, as well as having too-small key and block sizes.
- The 12-bit salt is considered to small to defeat rainbow-table attacks (most modern algorithms provide at least a 48-bit salt).

- The fact that it only uses the lower 7 bits of the first 8 bytes of the password results in a dangerously small keyspace which needs to be searched.

Deviations

This implementation of des-crypt differs from others in a few ways:

- Minimum salt string:

  Some implementations of des-crypt permit empty and single-character salt strings. However, behavior in these cases varies wildly; with implementations returning everything from errors to incorrect hashes that never validate. To avoid all this, Passlib will throw an "invalid salt" if the provided salt string is not at least 2 characters.

- Restricted salt string character set:

  The underlying algorithm expects salt strings to use the *hash64* character set to encode a 12-bit integer. Many implementations of des-crypt will accept a salt containing other characters, but vary wildly in how they are handled, including errors and implementation-specific value mappings. To avoid all this, Passlib will throw an "invalid salt" if the salt string contains any non-standard characters.

- Unicode Policy:

  The original des-crypt algorithm was designed for 7-bit `us-ascii` encoding only (as evidenced by the fact that it discards the 8th bit of all password bytes).

  In order to provide support for unicode strings, Passlib will encode unicode passwords using `utf-8` before running them through des-crypt. If a different encoding is desired by an application, the password should be encoded before handing it to Passlib.

`passlib.hash.bsdi_crypt` - BSDi Crypt

Danger: This algorithm is not considered secure by modern standards. It should only be used when verifying existing hashes, or when interacting with applications that require this format. For new code, see the list of *recommended hashes*.

This algorithm was developed by BSDi for their BSD/OS distribution. It's based on *des_crypt*, and contains a larger salt and a variable number of rounds. This algorithm is also known as "Extended DES Crypt". It class can be used directly as follows:

```
>>> from passlib.hash import bsdi_crypt

>>> # generate new salt, hash password
>>> hash = bsdi_crypt.hash("password")
>>> hash
'_7C/.Bf/4gZk10RYRs4Y'

>>> # same, but with explict number of rounds
>>> bsdi_crypt.using(rounds=10001).hash("password")
'_FQ0.amG/zwCMip7DnBk'

>>> # verify password
>>> bsdi_crypt.verify("password", hash)
True
>>> bsdi_crypt.verify("secret", hash)
False
```

See also:

the generic *PasswordHash usage examples*

Interface

class passlib.hash.**bsdi_crypt**

This class implements the BSDi-Crypt password hash, and follows the *PasswordHash API*.

It supports a fixed-length salt, and a variable number of rounds.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*str*) – Optional salt string. If not specified, one will be autogenerated (this is recommended). If specified, it must be 4 characters, drawn from the regexp range `[./0-9A-Za-z]`.
- **rounds** (*int*) – Optional number of rounds to use. Defaults to 5001, must be between 1 and 16777215, inclusive.
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `rounds` that are too small or too large, and `salt` strings that are too long.

  New in version 1.6.

Changed in version 1.6: `hash()` will now issue a warning if an even number of rounds is used (see *Security Issues* regarding weak DES keys).

Note: This class will use the first available of two possible backends:

- stdlib `crypt()`, if the host OS supports BSDi-Crypt (primarily BSD-derived systems).
- a pure Python implementation of BSDi-Crypt built into Passlib.

You can see which backend is in use by calling the `get_backend()` method.

Format

An example hash (of the string `password`) is `_EQ0.jzhSVeUyoSqLupI`. A bsdi_crypt hash string consists of a 20 character string of the form *_roundssaltchecksum*. All characters except the underscore prefix are drawn from `[./0-9A-Za-z]`.

- _ - the underscore is used to distinguish this scheme from others, such as des-crypt.
- *rounds* is the number of rounds, stored as a 4 character *hash64*-encoded 24-bit integer (`EQ0.` in the example).
- *salt* is the salt, stored as as a 4 character hash64-encoded 24-bit integer (`jzhS` in the example).
- *checksum* is the checksum, stored as an 11 character hash64-encoded 64-bit integer (`VeUyoSqLupI` in the example).

A bsdi_crypt configuration string is also accepted by this module; and has the same format as the hash string, but with the checksum portion omitted.

Algorithm

The checksum is formed by a modified version of the DES cipher in encrypt mode:

1. Given a password string, a salt string, and rounds string.
2. The 4 character rounds string is decoded to a 24-bit integer rounds value; The rounds string uses little-endian *hash64* encoding.
3. The 4 character salt string is decoded to a 24-bit integer salt value; The salt string uses little-endian *hash64* encoding.
4. The password is NULL-padded on the end to the smallest non-zero multiple of 8 bytes.
5. The lower 7 bits of the first 8 bytes of the password are used to form a 56-bit integer; with the first byte providing the most significant 7 bits, and the 8th byte providing the least significant 7 bits. This is the DES key.
6. For each additional block of 8 bytes in the padded password:
   a. The current DES key is encrypted using a single round of normal DES, with itself as the input block.
   b. Step 5 is repeated for the current 8-byte block, and xored against the existing DES key.
7. Repeated rounds of (modified) DES encryption are performed; starting with a null input block, and using the 56-bit integer from step 5/6 as the DES key.

   The salt is used to to mutate the normal DES encrypt operation by swapping bits `i` and `i+24` in the DES E-Box output if and only if bit `i` is set in the salt value.

   The number of rounds is controlled by the value decoded in step 2.
8. The 64-bit result of the last round of step 7 is then lsb-padded with 2 zero bits.
9. The resulting 66-bit integer is encoded in big-endian order using the *hash64-big* format.

Security Issues

BSDi Crypt should not be considered sufficiently secure, for a number of reasons:

- Its use of the DES stream cipher, which is vulnerable to practical pre-image attacks, and considered broken, as well as having too-small key and block sizes.
- The 24-bit salt is too small to defeat rainbow-table attacks (most modern algorithms provide at least a 48-bit salt).
- The fact that it only uses the lower 7 bits of each byte of the password restricts the keyspace which needs to be searched.
- Additionally, even *rounds* values are slightly weaker still, as they may reveal the hash used one of the weak DES keys[3]. This information could theoretically allow an attacker to perform a brute-force attack on a reduced keyspace and against only 1-2 rounds of DES. (This issue is mitigated by the fact that few passwords are both valid *and* result in a weak key).

This algorithm is none-the-less stronger than `des_crypt` itself, since it supports variable rounds, a larger salt size, and uses all the bytes of the password.

[3] DES weak keys - https://en.wikipedia.org/wiki/Weak_key#Weak_keys_in_DES

Deviations

This implementation of bsdi-crypt differs from others in one way:

- Unicode Policy:

  The original bsdi-crypt algorithm was designed for 7-bit `us-ascii` encoding only (as evidenced by the fact that it discards the 8th bit of all password bytes).

  In order to provide support for unicode strings, Passlib will encode unicode passwords using `utf-8` before running them through bsdi-crypt. If a different encoding is desired by an application, the password should be encoded before handing it to Passlib.

`passlib.hash.bigcrypt` - BigCrypt

Danger: This algorithm is dangerously insecure by modern standards. It is trivially broken, and should not be used if at all possible. For new code, see the list of *recommended hashes*.

This class implements BigCrypt (a modified version of DES-Crypt) commonly found on HP-UX, Digital Unix, and OSF/1. The main difference between it and *des_crypt* is that BigCrypt uses all the characters of a password, not just the first 8, and has a variable length hash.

See also:

password hash usage – for examples of how to use this class via the common hash interface.

Interface

class passlib.hash.**bigcrypt**

This class implements the BigCrypt password hash, and follows the *PasswordHash API*.

It supports a fixed-length salt.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*str*) – Optional salt string. If not specified, one will be autogenerated (this is recommended). If specified, it must be 22 characters, drawn from the regexp range `[./0-9A-Za-z]`.
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `salt` strings that are too long.

  New in version 1.6.

Format

An example hash (of the string `passphrase`) is `S/8NbAAlzbYO66hAa9XZyWy2`. A bigcrypt hash string has the format *saltchecksum_1checksum_2...checksum_n* for some integer *n*>0, where:

- *salt* is the salt, stored as a 2 character *hash64*-encoded 12-bit integer (`S/` in the example).

- each *checksum_i* is a separate checksum, stored as an 11 character *hash64-big*-encoded 64-bit integer (`8NbAAlzbYO6` and `6hAa9XZyWy2` in the example).
- the integer `n` (the number of checksums) is determined by the formula *n*=`min(1, (len(`*secret*`)+7)//8)`.

Note: This hash format lacks any magic prefix that can be used to unambiguously identify it. Out of context, certain `bigcrypt` hashes may be confused with that of two other algorithms:

- *des_crypt* - BigCrypt hashes of passwords with < 8 characters are exactly the same as the Des-Crypt hash of the same password.
- *crypt16* - BigCrypt hashes of passwords with 9 to 16 characters have the same size and character set as Crypt-16 hashes; though the actual algorithms are different.

Algorithm

The bigcrypt algorithm is designed to re-use the original des-crypt algorithm:

1. Given a password string and a salt string.
2. The password is NULL padded at the end to the smallest non-zero multiple of 8 bytes.
3. The lower 7 bits of the first 8 characters of the password are used to form a 56-bit integer; with the first character providing the most significant 7 bits, and the 8th character providing the least significant 7 bits.
4. The 2 character salt string is decoded to a 12-bit integer salt value; The salt string uses little-endian *hash64* encoding.
5. 25 repeated rounds of modified DES encryption are performed; starting with a null input block, and using the 56-bit integer from step 3 as the DES key.

   The salt is used to to mutate the normal DES encrypt operation by swapping bits *i* and *i*+`24` in the DES E-Box output if and only if bit *i* is set in the salt value.
6. The 64-bit result of the last round of step 5 is then lsb-padded with 2 zero bits.
7. The resulting 66-bit integer is encoded in big-endian order using the *hash64-big* format. This forms the first checksum segment.
8. For each additional block of 8 bytes in the padded password (from step 2), an additional checksum is generated by repeating steps 3..7, with the following changes:
   a. Step 3 uses the specified 8 bytes of the password, instead of the first 8 bytes.
   b. Step 4 uses the first two characters from the previous checksum as the salt for the next checksum.
9. The final checksum string is the concatenation of the checksum segments generated from steps 7 and 8, in order.

Note: Because of the chained structure, bigcrypt has the property that the first 13 characters of any bigcrypt hash form a valid *des_crypt* hash of the same password; and bigcrypt hashes of any passwords less than 9 characters will be identical to des-crypt.

Security Issues

BigCrypt is dangerously flawed:

- It suffers from all the flaws of *des_crypt*.
- Since each checksum component in its hash is essentially a separate des-crypt checksum, they can be attacked in parallel.
- It reveals information about the length of the encoded password (to within 8 characters), further reducing the keyspace that needs to be searched for each of the individual segments.
- The last checksum typically contains only a few characters of the passphrase, and once cracked, can be used to narrow the overall keyspace.

Deviations

This implementation of bigcrypt differs from others in two ways:

- Maximum Password Size:

  This implementation currently accepts arbitrarily large passwords, producing arbitrarily large hashes. Other implementation have various limits on maximum password length (commonly, 128 chars), and discard the remaining part of the password.

  Thus, while Passlib should be able to verify all existing bigcrypt hashes, other systems may require hashes generated by Passlib to be truncated to their specific maximum length.

- Unicode Policy:

  The original bigcrypt algorithm was designed for 7-bit `us-ascii` encoding only (as evidenced by the fact that it discards the 8th bit of all password bytes).

  In order to provide support for unicode strings, Passlib will encode unicode passwords using `utf-8` before running them through bigcrypt. If a different encoding is desired by an application, the password should be encoded before handing it to Passlib.

passlib.hash.crypt16 - Crypt16

Danger: This algorithm is dangerously insecure by modern standards. It is trivially broken, and should not be used if at all possible. For new code, see the list of *recommended hashes*.

This class implements the Crypt16 password hash, commonly found on Ultrix and Tru64. It's a minor modification of *des_crypt*, which allows passwords of up to 16 characters.

See also:

password hash usage – for examples of how to use this class via the common hash interface.

Interface

class passlib.hash.**crypt16**

This class implements the crypt16 password hash, and follows the *PasswordHash API*.

It supports a fixed-length salt.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*str*) – Optional salt string. If not specified, one will be autogenerated (this is recommended). If specified, it must be 2 characters, drawn from the regexp range `[./0-9A-Za-z]`.
- **truncate_error** (*bool*) – By default, crypt16 will silently truncate passwords larger than 16 bytes. Setting `truncate_error=True` will cause *hash()* to raise a *PasswordTruncateError* instead.

  New in version 1.7.
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `salt` strings that are too long.

  New in version 1.6.

Format

An example hash (of the string `passphrase`) is `aaX/UmCcBrceQ0kQGGWKTbuE`. A crypt16 hash string has the format *saltchecksum_1checksum_2*, where:

- *salt* is the salt, stored as a 2 character *hash64*-encoded 12-bit integer (`aa` in the example).
- each *checksum_i* is a separate checksum, stored as an 11 character *hash64-big*-encoded 64-bit integer (`X/UmCcBrceQ` and `0kQGGWKTbuE` in the example).

Note: This hash is frequently confused with the *bigcrypt* hash algorithm, as it has the same size and uses the same character set as a `bigcrypt` hash of a password with 9 to 16 characters; though the actual algorithms are different.

Algorithm

The crypt16 algorithm uses a weakened version of the des-crypt algorithm:

1. Given a password string and a salt string.
2. The 2 character salt string is decoded to a 12-bit integer salt value; The salt string uses little-endian *hash64* encoding.
3. If the password is larger than 16 bytes, the end is truncated to 16 bytes. If the password is smaller than 16 bytes, the end is NULL padded to 16 bytes.
4. The lower 7 bits of the first 8 characters of the password are used to form a 56-bit integer; with the first character providing the most significant 7 bits, and the 8th character providing the least significant 7 bits.
5. 20 repeated rounds of modified DES encryption are performed; starting with a null input block, and using the 56-bit integer from step 4 as the DES key.

   The salt value from step 2 is used to to mutate the normal DES encrypt operation by swapping bits *i* and *i*+24 in the DES E-Box output if and only if bit *i* is set in the salt value.
6. The 64-bit result of the last round of step 5 is then lsb-padded with 2 zero bits.
7. The resulting 66-bit integer is encoded in big-endian order using the *hash64-big* format. This is the first checksum segment.
8. The second checksum segment is created by repeating steps 4..7 using the second 8 bytes of the padding password (from step 3). The only difference is that step 5 uses only 5 rounds.

9. The final checksum string is the concatenation of the two checksum segments, in order.

Security Issues

Crypt16 is dangerously flawed:

- It suffers from all the flaws of *des_crypt*.
- Compared to des-crypt, its smaller number of rounds makes it even *more* vulnerable to brute-force attacks.
- For a given salt, passwords under 9 characters all have the same 2nd checksum. Given the 12-bit salt size, all such 2nd checksums can be easily pre-computed; making an attack easier, and giving away information about password size.
- Since both checksums use the same salt, they can be attacked at once (by doing 5 rounds, checking the result against checksum 2, doing 15 rounds more, and checking the result against checksum 1).

Deviations

This implementation of crypt16 deviates from public documentation of the format in one way:

- Unicode Policy:

  The original crypt16 algorithm was designed for 7-bit `us-ascii` encoding only (as evidenced by the fact that it discards the 8th bit of all password bytes).

  In order to provide support for unicode strings, Passlib will encode unicode passwords using `utf-8` before running them through crypt16. If a different encoding is desired by an application, the password should be encoded before handing it to Passlib.

3.7.3 Other "Modular Crypt" Hashes

The *modular crypt format* is a loose standard for password hash strings which started life under the Unix operating system, and is used by many of the Unix hashes (above). However, it's it's basic `$scheme$hash` format has also been adopted by a number of application-specific hash algorithms:

3.7.3.1 Active Hashes

While most of these schemes are generally application-specific, and are not natively supported by any Unix OS, they can be used compatibly along side other modular crypt format hashes:

`passlib.hash.argon2` - Argon2

New in version 1.7.

This hash provides support for the Argon2[1] password hash. Argon2(i) is a state of the art memory-hard password hash, and the winner of the 2013 Password Hashing Competition[2]. It has seen active development and analysis in subsequent years, and while young, and is intended to replace *pbkdf2_sha256*, *bcrypt*, and *scrypt*.

It is one of the four hashes Passlib *recommends* for new applications. This class can be used directly as follows:

[1] the Argon2 homepage - https://github.com/P-H-C/phc-winner-argon2
[2] 2012 Password Hashing Competition - https://password-hashing.net/

```
>>> from passlib.hash import argon2

>>> # generate new salt, hash password
>>> h = argon2.hash("password")
>>> h
'$argon2i$v=19$m=512,t=2,p=2$aI2R0hpDyLm3ltLa+1/rvQ$LqPKjd6n8yniKtAithoR7A'

>>> # the same, but with an explicit number of rounds
>>> argon2.using(rounds=4).hash("password")
'$argon2i$v=19$m=512,t=4,p=2$eM+ZMyYkpDRGaI3xXmuNcQ$c5DeJg3eb5dskVt1mDdxfw'

>>> # verify password
>>> argon2.verify("password", h)
True
>>> argon2.verify("wrong", h)
False
```

See also:

the generic *PasswordHash usage examples*

Interface

class passlib.hash.**argon2**

This class implements the Argon2 password hash[1], and follows the *PasswordHash API*.

Argon2 supports a variable-length salt, and variable time & memory cost, and a number of other configurable parameters.

The `replace()` method accepts the following optional keywords:

Parameters

- **type** (*str*) – Specify the type of argon2 hash to generate. Can be one of "ID", "I", "D".

  This defaults to "ID" if supported by the backend, otherwise "I".
- **salt** (*str*) – Optional salt string. If specified, the length must be between 0-1024 bytes. If not specified, one will be auto-generated (this is recommended).
- **salt_size** (*int*) – Optional number of bytes to use when autogenerating new salts.
- **rounds** (*int*) – Optional number of rounds to use. This corresponds linearly to the amount of time hashing will take.
- **time_cost** (*int*) – An alias for **rounds**, for compatibility with underlying argon2 library.
- **memory_cost** (*int*) – Defines the memory usage in kibibytes. This corresponds linearly to the amount of memory hashing will take.
- **parallelism** (*int*) – Defines the parallelization factor. *NOTE: this will affect the resulting hash value.*
- **digest_size** (*int*) – Length of the digest in bytes.
- **max_threads** (*int*) – Maximum number of threads that will be used. -1 means unlimited; otherwise hashing will use `min(parallelism, max_threads)` threads.

  Note: This option is currently only honored by the argon2pure backend.

- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a ValueError. If relaxed=True, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include rounds that are too small or too large, and salt strings that are too long.

Changed in version 1.7.2: Added the "type" keyword, and support for type "D" and "ID" hashes. (Prior versions could verify type "D" hashes, but not generate them).

Todo:

- Support configurable threading limits.

Argon2 Backends

This class will use the first available of two possible backends:

1. argon2_cffi, if installed. (this is the recommended option).
2. argon2pure, if installed.

If no backends are available, hash() and verify() will throw *MissingBackendError* when they are invoked. You can check which backend is in use by calling argon2.get_backend().

Format & Algorithm

The Argon2 hash format is defined by the argon2 reference implementation. It's compatible with the *PHC Format* and *Modular Crypt Format*, and uses $argon2i$, $argon2d$, or $argon2id$ as the identifying prefixes for all its strings. An example hash (of password) is:

```
$argon2i$v=19$m=512,t=3,p=2$c29tZXNhbHQ$SqlVijFGiPG+935vDSGEsA
```

This string has the format $argon2*X*$v=*V*$m=*M*,t=*T*,p=*P*$*salt*$*digest*, where:

- *X* is either i, d, or id; depending on the argon2 variant (i in the example).
- *V* is an integer representing the argon2 revision. the value (when rendered into hexidecimal) matches the argon2 version (in the example, v=19 corresponds to 0x13, or Argon2 v1.3).
- *M* is an integer representing the variable memory cost, in kibibytes (512kib in the example).
- *T* is an integer representing the variable time cost, in linear iterations. (3 in the example).
- *P* is a parallelization parameter, which controls how much of the hash calculation is parallelization (2 in the example).
- *salt* - this is the base64-encoded version of the raw salt bytes passed into the Argon2 function (c29tZXNhbHQ in the example).
- *digest* - this is the base64-encoded version of the raw derived key bytes returned from the Argon2 function. Argon2 supports a variable checksum size, though the hashes in passlib will typically be 16 bytes, resulting in a 22 byte digest (SqlVijFGiPG+935vDSGEsA in the example).

All integer values are encoded uses ascii decimal, with no leading zeros. All byte strings are encoded using the standard base64 encoding, but without any trailing padding ("=") chars.

Note: The v=*version*$ segment was added in Argon2 v1.3; older version Argon2 v1.0 hashes may not include this portion.

The Argon2 specification also supports an optional `,data=data` suffix following `p=parallelism`; but this is not consistently or fully supported.

The algorithm used by all of these schemes is deliberately identical and simple: The password is encoded into UTF-8 if not already encoded, and handed off to the Argon2 function. A specified number of bytes (16 byte default in passlib) returned result are encoded as the checksum.

See https://github.com/P-H-C/phc-winner-argon2 for the canonical description of the Argon2 hash.

Security Issues

Argon2 is relatively new compared to other password hash algorithms, having started life in 2013, and thus may still harbor some undiscovered issues. That said, it's one of *very* few which were designed explicitly with password hashing in mind; and draws strongly on the lessons of the algorithms before it. As of the release of Passlib 1.7, it has no known major security issues.

Deviations

- This implementation currently encodes all unicode passwords using UTF-8 before hashing, other implementations may vary, or offer a configurable encoding; though UTF-8 is assumed to be the default.

`passlib.hash.bcrypt_sha256` - BCrypt+SHA256

New in version 1.6.2.

BCrypt was developed to replace *md5_crypt* for BSD systems. It uses a modified version of the Blowfish stream cipher. It does, however, truncate passwords to 72 bytes, and some other minor quirks (see *BCrypt Password Truncation* for details). This class works around that issue by first running the password through HMAC-SHA2-256. This class can be used directly as follows:

```
>>> from passlib.hash import bcrypt_sha256

>>> # generate new salt, hash password
>>> h = bcrypt_sha256.hash("password")
>>> h
'$bcrypt-sha256$v=2,t=2b,r=12$n79VH.0Q2TMWmt3Oqt9uku$Kq4Noyk3094Y2QlB8NdRT8SvGiI4ft2'

>>> # the same, but with an explicit number of rounds
>>> bcrypt_sha256.using(rounds=13).hash("password")
'$bcrypt-sha256$v=2,t=2b,r=13$AmytCA45b12VeVg0YdDT3.$IZTbbJKgJlD5IJoCWhuDUqYjnJwNPlO'

>>> # verify password
>>> bcrypt_sha256.verify("password", h)
True
>>> bcrypt_sha256.verify("wrong", h)
False
```

Note: It is strongly recommended that you install bcrypt when using this hash. See *passlib.hash.bcrypt - BCrypt* for more details.

Interface

class passlib.hash.**bcrypt_sha256**

This class implements a composition of BCrypt + HMAC_SHA256, and follows the *PasswordHash API*.

It supports a fixed-length salt, and a variable number of rounds.

The *hash()* and *genconfig()* methods accept all the same optional keywords as the base *bcrypt* hash.

New in version 1.6.2.

Changed in version 1.7: Now defaults to "2b" bcrypt variant; though supports older hashes generated using the "2a" bcrypt variant.

Changed in version 1.7.3: For increased security, updated to use HMAC-SHA256 instead of plain SHA256. Now only supports the "2b" bcrypt variant. Hash format updated to "v=2".

Format

Bcrypt-SHA256 is compatible with the *Modular Crypt Format*, and uses $bcrypt-sha256$ as the identifying prefix for all it's strings. An example hash (of password) is:

```
$bcrypt-sha256$v=2,t=2b,r=12$n79VH.0Q2TMWmt3Oqt9uku$Kq4Noyk3094Y2QlB8NdRT8SvGiI4ft2
```

Version 1 of this format had the format $bcrypt-sha256$*type*,*rounds*$*salt*$*digest*. Passlib 1.7.3 introduced version 2 of this format, which changed the algorithm slightly (see below), and adjusted the format to indicate a version: $bcrypt-sha256$v=2,t=*type*,r=*rounds*$*salt*$*digest*, where:

- *type* is the BCrypt variant in use (always 2b under version 2; though 2a was allowed under version 1).
- *rounds* is a cost parameter, encoded as decimal integer, which determines the number of iterations used via *iterations*=2***rounds* (rounds is 12 in the example).
- *salt* is a 22 character salt string, using the characters in the regexp range [./A-Za-z0-9] (n79VH.0Q2TMWmt3Oqt9uku in the example).
- *digest* is a 31 character digest, using the same characters as the salt (Kq4Noyk3094Y2QlB8NdRT8SvGiI4ft2 in the example).

Algorithm

The algorithm this hash uses is as follows:

- first the password is encoded to UTF-8 if not already encoded.
- the next step is to hash the password before handing it off to bcrypt:
  - Under version 2 of this algorithm (the default as of passlib 1.7.3), the password is run through HMAC-SHA2-256, with the HMAC key set to the bcrypt salt (encoded as a 22 character ascii salt string).
  - Under the older version 1 of this algorithm, the password was instead run through plain SHA2-256.

  In either case, this generates a 32 byte digest.
- this hash is then encoded using base64, resulting in a 44-byte result (including the trailing padding =). For the example "password" and the salt "n79VH.0Q2TMWmt3Oqt9uku", the output from this stage would be b"7CwRr5rxo2JZcVmSDAi/2JPTkvkAdNy20Cz2LwYC0fw=" (for version 2).

- this base64 string is then passed on to the underlying bcrypt algorithm as the new password to be hashed. See *passlib.hash.bcrypt - BCrypt* for details on it's operation. For the example in the prior line, the resulting bcrypt digest component would be `"Kq4Noyk3094Y2QlB8NdRT8SvGiI4ft2"`.

passlib.hash.phpass - PHPass' Portable Hash

This algorithm is used primarily by PHP software which uses PHPass[1], a PHP library similar to Passlib. The PHPass Portable Hash is a custom password hash used by PHPass as a fallback when none of its other hashes are available. Due to its reliance on MD5, and the simplistic implementation, other hash algorithms should be used if possible.

See also:

password hash usage – for examples of how to use this class via the common hash interface.

Interface

class passlib.hash.**phpass**

This class implements the PHPass Portable Hash, and follows the *PasswordHash API*.

It supports a fixed-length salt, and a variable number of rounds.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*str*) – Optional salt string. If not specified, one will be autogenerated (this is recommended). If specified, it must be 8 characters, drawn from the regexp range `[./0-9A-Za-z]`.
- **rounds** (*int*) – Optional number of rounds to use. Defaults to 19, must be between 7 and 30, inclusive. This value is logarithmic, the actual number of iterations used will be `2**rounds`.
- **ident** (*str*) – phpBB3 uses `H` instead of `P` for its identifier, this may be set to `H` in order to generate phpBB3 compatible hashes. it defaults to `P`.
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `rounds` that are too small or too large, and `salt` strings that are too long.

  New in version 1.6.

Format

An example hash (of `password`) is `$P$8ohUJ.1sdFw09/bMaAQPTGDNi2BIUt1`. A phpass portable hash string has the format `$P$`*`roundssaltchecksum`*, where:

- `$P$` is the prefix used to identify phpass hashes, following the *Modular Crypt Format*.
- *`rounds`* is a single character encoding a 6-bit integer representing the number of rounds used. This is logarithmic, the real number of rounds is `2**rounds`. (in the example, rounds is encoded as `8`, or 2**13 iterations).
- *`salt`* is eight characters drawn from `[./0-9A-Za-z]`, providing a 48-bit salt (`ohUJ.1sd` in the example).
- *`checksum`* is 22 characters drawn from the same set, encoding the 128-bit checksum (`Fw09/bMaAQPTGDNi2BIUt1` in the example).

[1] PHPass homepage, which describes the Portable Hash algorithm - http://www.openwall.com/phpass/

Note: Note that phpBB3 databases uses the alternate prefix `$H$`, both prefixes are recognized by this implementation, and the checksums are the same.

Algorithm

PHPass uses a straightforward algorithm to calculate the checksum:

1. an initial result is generated from the MD5 digest of the salt string + the secret.
2. for `2**rounds` iterations, a new result is created from the MD5 digest of the last result + the password.
3. the last result is then encoded according to the format described above.

Deviations

This implementation of phpass differs from the specification in one way:

- Unicode Policy:

  The underlying algorithm takes in a password specified as a series of non-null bytes, and does not specify what encoding should be used; though a `us-ascii` compatible encoding is implied by nearly all known reference hashes.

  In order to provide support for unicode strings, Passlib will encode unicode passwords using `utf-8` before running them through phpass. If a different encoding is desired by an application, the password should be encoded before handing it to Passlib.

`passlib.hash.pbkdf2_digest` - Generic PBKDF2 Hashes

Passlib provides three custom hash schemes based on the PBKDF2[1] algorithm which are compatible with the *modular crypt format*:

- *pbkdf2_sha1*
- *pbkdf2_sha256*
- *pbkdf2_sha512*

Security-wise, PBKDF2 is currently one of the leading key derivation functions, and has no known security issues. Though the original PBKDF2 specification uses the SHA-1 message digest, it is not vulnerable to any of the known weaknesses of SHA-1[2], and can be safely used. However, for those still concerned, SHA-256 and SHA-512 versions are offered as well. PBKDF2-SHA512 is one of the four hashes Passlib *recommends* for new applications.

All of these classes can be used directly as follows:

```
>>> from passlib.hash import pbkdf2_sha256

>>> # generate new salt, hash password
>>> hash = pbkdf2_sha256.hash("password")
>>> hash
'$pbkdf2-sha256$6400$0ZrzXitFSGltTQnBWOsdAw
↪$Y11AchqV4b0sUisdZd0Xr97KWoymNE0LNNrnEgY4H9M'
```

(continues on next page)

[1] The specification for the PBKDF2 algorithm - http://tools.ietf.org/html/rfc2898#section-5.2, part of **RFC 2898**.

[2] While SHA1 has fallen to collision attacks, HMAC-SHA1 as used by PBKDF2 is still considered secure - http://www.schneier.com/blog/archives/2005/02/sha1_broken.html.

(continued from previous page)

```
>>> # same, but with an explicit number of rounds and salt length
>>> pbkdf2_sha256.using(rounds=8000, salt_size=10).hash("password")
'$pbkdf2-sha256$8000$XAuBMIYQQogxRg$tRRlz8hYn63B9LYiCd6PRo6FMiunY9ozmMMI3srxeRE'

>>> # verify the password
>>> pbkdf2_sha256.verify("password", hash)
True
>>> pbkdf2_sha256.verify("wrong", hash)
False
```

See also:

- *password hash usage* – for more usage examples
- *ldap_pbkdf2_{digest}* – alternate LDAP-compatible versions of these hashes.

Interface

class passlib.hash.**pbkdf2_sha256**

This class implements a generic `PBKDF2-HMAC-SHA256`-based password hash, and follows the *Password-Hash API*.

It supports a variable-length salt, and a variable number of rounds.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*bytes*) – Optional salt bytes. If specified, the length must be between 0-1024 bytes. If not specified, a 16 byte salt will be autogenerated (this is recommended).
- **salt_size** (*int*) – Optional number of bytes to use when autogenerating new salts. Defaults to 16 bytes, but can be any value between 0 and 1024.
- **rounds** (*int*) – Optional number of rounds to use. Defaults to 29000, but must be within `range(1,1<<32)`.
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `rounds` that are too small or too large, and `salt` strings that are too long.

  New in version 1.6.

class passlib.hash.**pbkdf2_sha512**

except for the choice of message digest, this class is the same as *pbkdf2_sha256*.

class passlib.hash.**pbkdf2_sha1**

except for the choice of message digest, this class is the same as *pbkdf2_sha256*.

Format & Algorithm

An example `pbkdf2_sha256` hash (of `password`):

```
$pbkdf2-sha256$6400$.6UI/S.nXIk8jcbdHx3Fhg$98jZicV16ODfEsEZeYPGHU3kbrUrvUEXOPimVSQDD44
```

All of the pbkdf2 hashes defined by passlib follow the same format, `$pbkdf2-digest$rounds$salt$checksum`.

- `$pbkdf2-digest$` is used as the *Modular Crypt Format* identifier (`$pbkdf2-sha256$` in the example).
- *digest* - this specifies the particular cryptographic hash used in conjunction with HMAC to form PBKDF2's pseudorandom function for that particular hash (`sha256` in the example).
- *rounds* - the number of iterations that should be performed. this is encoded as a positive decimal number with no zero-padding (`6400` in the example).
- *salt* - this is the *adapted base64 encoding* of the raw salt bytes passed into the PBKDF2 function.
- *checksum* - this is the *adapted base64 encoding* of the raw derived key bytes returned from the PBKDF2 function. Each scheme uses the digest size of its specific hash algorithm (*digest*) as the size of the raw derived key. This is enlarged by approximately 4/3 by the base64 encoding, resulting in a checksum size of 27, 43, and 86 for each of the respective algorithms listed above.

The algorithm used by all of these schemes is deliberately identical and simple: The password is encoded into UTF-8 if not already encoded, and run through *pbkdf2_hmac()* along with the decoded salt, the number of rounds, and a prf built from HMAC + the respective message digest. The result is then encoded using *ab64_encode()*.

passlib.hash.scram - SCRAM Hash

New in version 1.6.

SCRAM is a password-based challenge response protocol defined by **RFC 5802**. While Passlib does not provide an implementation of SCRAM, applications which use SCRAM on the server side frequently need a way to store user passwords in a secure format that can be used to authenticate users over SCRAM.

To accomplish this, Passlib provides the following *Modular Crypt Format*-compatible password hash scheme which uses the `$scram$` identifier. This format encodes a salt, rounds settings, and one or more *pbkdf2_hmac()* digests... one digest for each of the hash algorithms the server wishes to support over SCRAM.

Since this format is PBKDF2-based, it has equivalent security to Passlib's other *pbkdf2 hashes*, and can be used to authenticate users using either the normal *PasswordHash API* or the SCRAM-specific class methods documented below.

Note: If you aren't working with the SCRAM protocol, you probably don't need to use this hash format.

Usage

This class can be used like any other Passlib hash, as follows:

```
>>> from passlib.hash import scram

>>> # generate new salt, hash password against default list of algorithms
>>> hash = scram.hash("password")
>>> hash
'$scram$6400$.Z/znnNOKWUsBaCU$sha-1=cRseQyJpnuPGn3e6d6u6JdJWk.0,sha-256=5G
cjEbRaUIIci1r6NAMdI9OPZbxl9S5CFR6la9CHXYc,sha-512=.DHbIm82ajXbFR196Y.9Ttbs
gzvGjbMeuWCtKve8TPjRMNoZK9EGyHQ6y0lW9OtWdHZrDZbBUhB9ou./VI2mlw'

>>> # same, but with an explicit number of rounds
>>> scram.using(rounds=8000).hash("password")
```

(continues on next page)

(continued from previous page)

```
'$scram$8000$Y0zp/R/DeO89h/De$sha-1=eE8dq1f1P1hZm21lfzsr3CMbiEA,sha-256=Nf
kaDFMzn/yHr/HTv7KEFZqaONo6psRu5LBBFLEbZ.o,sha-512=XnGG11X.J2VGSG1qTbkR3FVr
9j5JwsnV5Fd094uuC.GtVDE087m8e7rGoiVEgXnduL48B2fPsUD9grBjURjkiA'

>>> # verify password
>>> scram.verify("password", hash)
True
>>> scram.verify("secret", hash)
False
```

See the generic *PasswordHash usage examples* for more details on how to use the common hash interface.

Additionally, this class provides a number of useful methods for SCRAM-specific actions:

- You can override the default list of digests, and/or the number of iterations:

```
>>> hash = scram.using(rounds=1000, algs="sha-1,sha-256,md5").hash("password")
>>> hash
'$scram$1000$RsgZo7T2/l8rBUBI$md5=iKsH555d3ctn795Za4S7bQ,sha-1=dRcE2AUjALLF
tX5DstdLCXZ9Afw,sha-256=WYE/LF7OntriUUdFXIrYE19OY2yL0N5qsQmdPNFn7JE'
```

- Given a scram hash, you can use a single call to extract all the information the SCRAM needs to authenticate against a specific mechanism:

```
>>> # this returns (salt_bytes, rounds, digest_bytes)
>>> scram.extract_digest_info(hash, "sha-1")
('F\xc8\x19\xa3\xb4\xf6\xfe_+\x05@H',
 1000,
 'u\x17\x04\xd8\x05#\x00\xb2\xc5\xb5~C\xb2\xd7K\tv}\x01\xfc')
```

- Given a scram hash, you can extract the list of digest algorithms it contains information for (`sha-1` will always be present):

```
>>> scram.extract_digest_algs(hash)
["md5", "sha-1", "sha-256"]
```

- This class also provides a standalone helper which can calculate the `SaltedPassword` portion of the SCRAM protocol, taking care of the SASLPrep step as well:

```
>>> scram.derive_digest("password", b'\x01\x02\x03', 1000, "sha-1")
b'k\x086vg\xb3\xfciz\xb4\xb4\xe2JRZ\xaet\xe4`\xe7'
```

Interface

Note: This hash format is new in Passlib 1.6, and its SCRAM-specific API may change in the next few releases, depending on user feedback.

---

class passlib.hash.**scram**

This class provides a format for storing SCRAM passwords, and follows the *PasswordHash API*.

It supports a variable-length salt, and a variable number of rounds.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*bytes*) – Optional salt bytes. If specified, the length must be between 0-1024 bytes. If not specified, a 12 byte salt will be autogenerated (this is recommended).
- **salt_size** (*int*) – Optional number of bytes to use when autogenerating new salts. Defaults to 12 bytes, but can be any value between 0 and 1024.
- **rounds** (*int*) – Optional number of rounds to use. Defaults to 100000, but must be within `range(1,1<<32)`.
- **algs** (*list of strings*) – Specify list of digest algorithms to use.

  By default each scram hash will contain digests for SHA-1, SHA-256, and SHA-512. This can be overridden by specify either be a list such as `["sha-1", "sha-256"]`, or a comma-separated string such as `"sha-1, sha-256"`. Names are case insensitive, and may use `hashlib` or IANA hash names.
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `rounds` that are too small or too large, and `salt` strings that are too long.

  New in version 1.6.

In addition to the standard *PasswordHash API* methods, this class also provides the following methods for manipulating Passlib scram hashes in ways useful for pluging into a SCRAM protocol stack:

classmethod extract_digest_info(*hash*, *alg*)
return (salt, rounds, digest) for specific hash algorithm.

Parameters

- **hash** (*str*) – `scram` hash stored for desired user
- **alg** (*str*) – Name of digest algorithm (e.g. `"sha-1"`) requested by client.

  This value is run through *norm_hash_name()*, so it is case-insensitive, and can be the raw SCRAM mechanism name (e.g. `"SCRAM-SHA-1"`), the IANA name, or the hashlib name.

Raises **KeyError** – If the hash does not contain an entry for the requested digest algorithm.

Returns A tuple containing `(salt, rounds, digest)`, where *digest* matches the raw bytes returned by SCRAM's `Hi()` function for the stored password, the provided *salt*, and the iteration count (*rounds*). *salt* and *digest* are both raw (unencoded) bytes.

classmethod extract_digest_algs(*hash*, *format='iana'*)
Return names of all algorithms stored in a given hash.

Parameters

- **hash** (*str*) – The `scram` hash to parse
- **format** (*str*) – This changes the naming convention used by the returned algorithm names. By default the names are IANA-compatible; possible values are `"iana"` or `"hashlib"`.

Returns Returns a list of digest algorithms; e.g. `["sha-1"]`

classmethod derive_digest(*password*, *salt*, *rounds*, *alg*)
helper to create SaltedPassword digest for SCRAM.

This performs the step in the SCRAM protocol described as:

```
SaltedPassword  := Hi(Normalize(password), salt, i)
```

Parameters

- **password** (*unicode or utf-8 bytes*) – password to run through digest
- **salt** (*bytes*) – raw salt data
- **rounds** (*int*) – number of iterations.
- **alg** (*str*) – name of digest to use (e.g. `"sha-1"`).

Returns raw bytes of `SaltedPassword`

Format & Algorithm

An example scram hash (of the string `password`) is:

```
$scram$6400$.Z/znnNOKWUsBaCU$sha-1=cRseQyJpnuPGn3e6d6u6JdJWk.0,sha-256=5G
cjEbRaUIIci1r6NAMdI9OPZbxl9S5CFR6la9CHXYc,sha-512=.DHbIm82ajXbFR196Y.9Ttb
sgzvGjbMeuWCtKve8TPjRMNoZK9EGyHQ6y0lW9OtWdHZrDZbBUhB9ou./VI2mlw
```

An scram hash string has the format `$scram$`*rounds*`$`*salt*`$`*alg1*`=`*digest1*`,`*alg2*`=`*digest2*`,...`, where:

- `$scram$` is the prefix used to identify Passlib scram hashes, following the *Modular Crypt Format*
- *rounds* is the number of decimal rounds to use (6400 in the example), zero-padding not allowed. this value must be in `range(1, 2**32)`.
- *salt* is a base64 salt string (`.Z/znnNOKWUsBaCU` in the example), encoded using *ab64_encode()*.
- *alg* is a lowercase IANA hash function name[2], which should match the digest in the SCRAM mechanism name.
- *digest* is a base64 digest for the specific algorithm, encoded using *ab64_encode()*. Digests for `sha-1`, `sha-256`, and `sha-512` are present in the example.
- There will always be one or more *alg*`=`*digest* pairs, separated by a comma. Per the SCRAM specification, the algorithm `sha-1` should always be present.

There is also an alternate format (`$scram$`*rounds*`$`*salt*`$`*alg*`,...`) which is used to represent a configuration string that doesn't contain any digests. An example would be:

```
$scram$6400$.Z/znnNOKWUsBaCU$sha-1,sha-256,sha-512
```

The algorithm used to calculate each digest is:

```
pbkdf2(salsprep(password).encode("utf-8"), salt, rounds, alg_digest_size, "hmac-"+alg)
```

...as laid out in the SCRAM specification[1]. All digests should verify against the same password, or the hash is considered malformed.

Note: This format is similar in spirit to the LDAP storage format for SCRAM hashes, defined in **RFC 5803**, except that it encodes everything into a single string, and does not have any storage requirements (outside of the ability to store 512+ character ascii strings).

---

[2] The official list of IANA-assigned hash function names - http://www.iana.org/assignments/hash-function-text-names

[1] The SCRAM protocol is laid out in **RFC 5802**.

Security

The security of this hash is only as strong as the weakest digest used by this hash. Since the SCRAM[1] protocol requires SHA1 always be supported, this will generally be the weakest link, since the other digests will generally be stronger ones (e.g. SHA2-256).

None-the-less, since PBKDF2 is sufficiently collision-resistant on its own, any pre-image weaknesses found in SHA1 should be mitigated by the PBKDF2-HMAC-SHA1 wrapper; and should have no flaws outside of brute-force attacks on PBKDF2-HMAC-SHA1.

`passlib.hash.scrypt` - SCrypt

New in version 1.7.

This is a custom hash scheme provided by Passlib which allows storing password hashes generated using the SCrypt[1] key derivation function, and is designed as the of a new generation of "memory hard" functions.

Warning: Be careful when using this algorithm, as the memory and CPU requirements needed to achieve adequate security are generally higher than acceptable for heavily used production systems[2]. This is because (unlike many password hashes), increasing the rounds value of scrypt will increase the *memory* required as well as the time.

Unless you know what you're doing, **You probably want** *argon2* **instead.**

This class can be used directly as follows:

```
>>> from passlib.hash import scrypt

>>> # generate new salt, hash password
>>> h = scrypt.hash("password")
>>> h
'$scrypt$ln=16,r=8,p=1$aM15713r3Xsvxbi31lqr1Q
↪$nFNh2CVHVjNldFVKDHDlm4CbdRSCdEBsjjJxD+iCs5E'

>>> # the same, but with an explicit number of rounds
>>> scrypt.using(rounds=8).hash("password")
'$scrypt$ln=8,r=8,p=1$WKs1xljLudd6z9kbY0wpJQ$yCR4iDZYDKv+iEJj6yHY0lv/epnfB6f/
↪w1EbXrsJOuQ'

>>> # verify password
>>> scrypt.verify("password", h)
True
>>> scrypt.verify("wrong", h)
False
```

Note: It is strongly recommended that you install scrypt when using this hash.

See also:

the generic *PasswordHash usage examples*

[1] the SCrypt KDF homepage - http://www.tarsnap.com/scrypt.html

[2] posts discussing security implications of scrypt's tying memory cost to calculation time - http://blog.ircmaxell.com/2014/03/why-i-dont-recommend-scrypt.html, http://security.stackexchange.com/questions/26245/is-bcrypt-better-than-scrypt, http://security.stackexchange.com/questions/4781/do-any-security-experts-recommend-bcrypt-for-password-storage

Interface

class passlib.hash.**scrypt**

This class implements an SCrypt-based password[1] hash, and follows the *PasswordHash API*.

It supports a variable-length salt, a variable number of rounds, as well as some custom tuning parameters unique to scrypt (see below).

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*str*) – Optional salt string. If specified, the length must be between 0-1024 bytes. If not specified, one will be auto-generated (this is recommended).
- **salt_size** (*int*) – Optional number of bytes to use when autogenerating new salts. Defaults to 16 bytes, but can be any value between 0 and 1024.
- **rounds** (*int*) – Optional number of rounds to use. Defaults to 16, but must be within `range(1,32)`.

  Warning: Unlike many hash algorithms, increasing the rounds value will increase both the time *and memory* required to hash a password.

- **block_size** (*int*) – Optional block size to pass to scrypt hash function (the `r` parameter). Useful for tuning scrypt to optimal performance for your CPU architecture. Defaults to 8.
- **parallelism** (*int*) – Optional parallelism to pass to scrypt hash function (the `p` parameter). Defaults to 1.
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `rounds` that are too small or too large, and `salt` strings that are too long.

Note: The underlying scrypt hash function has a number of limitations on it's parameter values, which forbids certain combinations of settings. The requirements are:

- `linear_rounds = 2**<some positive integer>`
- `linear_rounds < 2**(16 * block_size)`
- `block_size * parallelism <= 2**30-1`

Todo: This class currently does not support configuring default values for `block_size` or `parallelism` via a *CryptContext* configuration.

Scrypt Backends

This class will use the first available of two possible backends:

1. Python stdlib's `hashlib.scrypt()` method (only present for Python 3.6+ and OpenSSL 1.1+)
2. The C-accelerated scrypt package, if installed.

3. A pure-python implementation of SCrypt, built into Passlib.

Warning: If `hashlib.scrypt()` is not present on your system, it is strongly recommended to install the external scrypt package. The pure-python backend is intended as a reference and last-resort implementation only; it is 10-100x too slow to be usable in production at a secure `rounds` cost.

Changed in version 1.7.2: Added support for using stdlib's `hashlib.scrypt()`

Format & Algorithm

This Scrypt hash format is compatible with the *PHC Format* and *Modular Crypt Format*, and uses `$scrypt$` as the identifying prefix for all its strings. An example hash (of `password`) is:

```
$scrypt$ln=16,r=8,p=1$aM15713r3Xsvxbi31lqr1Q$nFNh2CVHVjNldFVKDHDlm4CbdRSCdEBsjjJxD+iCs
```

This string has the format `$scrypt$ln=`*`logN`*`,r=`*`R`*`,p=`*`P`*`$`*`salt`*`$`*`checksum`*, where:

- *`logN`* is the exponent for calculating SCRYPT's cost parameter (N), encoded as a decimal digit, (logN is 16 in the example, corresponding to n = 2**16 = 65536).
- *`R`* is the value of SCRYPT's block size parameter (r), encoded as a decimal digit, (r is 8 in the example).
- *`P`* is the value of SCRYPT's parallel count parameter (p), encoded as a decimal digit, (p is 1 in the example).
- *`salt`* - this base64 encoded salt bytes passed into the SCRYPT function (`aM15713r3Xsvxbi31lqr1Q` in the example).
- *`checksum`* - this is the base64 encoded derived key bytes returned from the SCRYPT function. This hash currently always uses 32 bytes, resulting in a 43-character checksum. (`nFNh2CVHVjNldFVKDHDlm4CbdRSCdEBsjjJxD+iCs5E` in the example).

All byte strings are encoded using the standard base64 encoding, but without any trailing padding ("=") chars. The password is encoded into UTF-8 if not already encoded, and run throught the SCRYPT function; along with the salt, and the values of n, r, and p. The first 32 bytes of the returned result are encoded as the checksum.

See http://www.tarsnap.com/scrypt.html for the canonical description of the scrypt kdf.

Security Issues

SCrypt is the first in a class of "memory-hard" key derivation functions. Initially, it looked very promising as a replacement for BCrypt, PBKDF2, and SHA512-Crypt. However, the fact that it's `N` parameter controls both time *and* memory cost means the two cannot be varied completely independantly. This eventually proved to be problematic, as `N` values required for even BCrypt levels of security resulting in memory requirements that were unacceptable on most production systems.

See also:

`argon2`, a next generation memory-hard KDF designed as the successor to SCrypt.

3.7.3.2 Deprecated Hashes

The following are some additional application-specific hashes which are still occasionally seen, use the modular crypt format, but are rarely used or weak enough that they have been deprecated:

passlib.hash.apr_md5_crypt - Apache's MD5-Crypt variant

Danger: This algorithm is not considered secure by modern standards. It should only be used when verifying existing hashes, or when interacting with applications that require this format. For new code, see the list of *recommended hashes*.

This hash is a variation of *md5_crypt*, primarily used by the Apache webserver in `htpasswd` files. It contains only minor changes to the MD5-Crypt algorithm, and should be considered just as weak as MD5-Crypt itself.

See also:

- *password hash usage* – for examples of how to use this class via the common hash interface.
- *passlib.apache* – routines for manipulating `htpasswd` files.

Interface

class passlib.hash.**apr_md5_crypt**

This class implements the Apr-MD5-Crypt password hash, and follows the *PasswordHash API*.

It supports a variable-length salt.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*str*) – Optional salt string. If not specified, one will be autogenerated (this is recommended). If specified, it must be 0-8 characters, drawn from the regexp range `[./0-9A-Za-z]`.
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `salt` strings that are too long.

  New in version 1.6.

Format & Algorithm

This format and algorithm of Apache's MD5-Crypt is identical to the original MD5-Crypt, except for two changes:

1. The encoded string uses `$apr1$` as its prefix, while md5-crypt uses `$1$`.
2. The algorithm uses `$apr1$` as a constant in the step where md5-crypt uses `$1$` in its calculation of digest B (see the *md5-crypt algorithm*). Because of this change, even raw checksums generated by apr-md5-crypt and md5-crypt are not compatible with each other.

See *md5_crypt* for the format & algorithm descriptions, as well as security notes.

passlib.hash.cta_pbkdf2_sha1 - Cryptacular's PBKDF2 hash

This class provides an implementation of Cryptacular's PBKDF2-HMAC-SHA1 hash format[1]. PBKDF2 is a key derivation function[2] that is ideally suited as the basis for a password hash, as it provides variable length salts, variable

[1] The reference for this hash format - https://bitbucket.org/dholth/cryptacular/.
[2] The specification for the PBKDF2 algorithm - http://tools.ietf.org/html/rfc2898#section-5.2.

number of rounds.

See also:

- *password hash usage* – for examples of how to use this class via the common hash interface.
- *dlitz_pbkdf2_sha1* for another hash which looks almost exactly like this one.

Interface

class passlib.hash.**cta_pbkdf2_sha1**

This class implements Cryptacular's PBKDF2-based crypt algorithm, and follows the *PasswordHash API*.

It supports a variable-length salt, and a variable number of rounds.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*bytes*) – Optional salt bytes. If specified, it may be any length. If not specified, a one will be autogenerated (this is recommended).
- **salt_size** (*int*) – Optional number of bytes to use when autogenerating new salts. Defaults to 16 bytes, but can be any value between 0 and 1024.
- **rounds** (*int*) – Optional number of rounds to use. Defaults to 60000, must be within `range(1,1<<32)`.
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `rounds` that are too small or too large, and `salt` strings that are too long.

  New in version 1.6.

Format & Algorithm

A example hash (of `password`) is:

```
$p5k2$2710$oX9ZZOcNgYoAsYL-8bqxKg==$AU2JLf2rNxWoZxWxRCluY0u6h6c=
```

All of this scheme's hashes have the format `$p5k2$`*rounds*`$`*salt*`$`*checksum*, where:

- `$p5k2$` is used as the *Modular Crypt Format* identifier.
- *rounds* is the number of PBKDF2 iterations to perform, stored as lowercase hexadecimal number with no zero-padding (in the example: `2710` or 10000 iterations).
- *salt* is the salt string encoding using base64 (with `-_` as the high values). `oX9ZZOcNgYoAsYL-8bqxKg==` in the example.
- *checksum* is 28 characters encoding the resulting 20-byte PBKDF2 derived key using base64 (with `-_` as the high values). `AU2JLf2rNxWoZxWxRCluY0u6h6c=` in the example.

In order to generate the checksum, the password is first encoded into UTF-8 if it's unicode. The salt is decoded from its base64 representation. PBKDF2 is called using the encoded password, the full salt, the specified number of rounds, and using HMAC-SHA1 as its pseudorandom function. 20 bytes of derived key are requested, and the resulting key is encoded and used as the checksum portion of the hash.

passlib.hash.dlitz_pbkdf2_sha1 - Dwayne Litzenberger's PBKDF2 hash

Warning: Due to a small flaw, this hash is not as strong as other PBKDF1-HMAC-SHA1 based hashes. It should probably not be used for new applications.

This class provides an implementation of Dwayne Litzenberger's PBKDF2-HMAC-SHA1 hash format[1]. PBKDF2 is a key derivation function[2] that is ideally suited as the basis for a password hash, as it provides variable length salts, variable number of rounds.

See also:

- *password hash usage* – for examples of how to use this class via the common hash interface.
- *cta_pbkdf2_sha1* for another hash which looks almost exactly like this one.

Interface

class passlib.hash.**dlitz_pbkdf2_sha1**

This class implements Dwayne Litzenberger's PBKDF2-based crypt algorithm, and follows the *PasswordHash API*.

It supports a variable-length salt, and a variable number of rounds.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*str*) – Optional salt string. If specified, it may be any length, but must use the characters in the regexp range `[./0-9A-Za-z]`. If not specified, a 16 character salt will be autogenerated (this is recommended).
- **salt_size** (*int*) – Optional number of bytes to use when autogenerating new salts. Defaults to 16 bytes, but can be any value between 0 and 1024.
- **rounds** (*int*) – Optional number of rounds to use. Defaults to 60000, must be within `range(1,1<<32)`.
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `rounds` that are too small or too large, and `salt` strings that are too long.

  New in version 1.6.

Format & Algorithm

A example hash (of `password`) is:

```
$p5k2$2710$.pPqsEwHD7MiECU0$b8TQ5AMQemtlaSgegw5Je.JBE3QQhLbO.
```

All of this scheme's hashes have the format `$p5k2$`*rounds*`$`*salt*`$`*checksum*, where:

- `$p5k2$` is used as the *Modular Crypt Format* identifier.

[1] The reference for this hash format - http://www.dlitz.net/software/python-pbkdf2/.
[2] The specification for the PBKDF2 algorithm - http://tools.ietf.org/html/rfc2898#section-5.2.

- *rounds* is the number of PBKDF2 iterations to perform, stored as lowercase hexadecimal number with no zero-padding (in the example: `2710` or 10000 iterations).
- *salt* is the salt string, which can be any number of characters, drawn from the *hash64 charset* (`.pPqsEwHD7MiECU0` in the example).
- *checksum* is 32 characters, which encode the resulting 24-byte PBKDF2 derived key using *ab64_encode()* (`b8TQ5AMQemtlaSgegw5Je.JBE3QQhLbO` in the example).

In order to generate the checksum, the password is first encoded into UTF-8 if it's unicode. Then, the entire configuration string (all of the hash except the checksum, ie `$p5k2$`*rounds*`$`*salt*) is used as the PBKDF2 salt. PBKDF2 is called using the encoded password, the full salt, the specified number of rounds, and using HMAC-SHA1 as its pseudorandom function. 24 bytes of derived key are requested, and the resulting key is encoded and used as the checksum portion of the hash.

Security Issues

- *Extra Block:* This hash generates 24 bytes using PBKDF2-HMAC-SHA1. Since SHA1 has a digest size of only 20 bytes, this means an second PBKDF2 block must be generated for each *dlitz_pbkdf2_sha1* hash. While a normal user has to calculate both blocks, a dedicated attacker would only have to calculate the first block when brute-forcing, taking half the time. That means this hash is half as strong as other PBKDF2-HMAC-SHA1 based hashes (given a fixed amount of time spent by the user).

3.7.4 LDAP / RFC2307 Hashes

All of the following hashes use a variant of the password hash format used by LDAPv2. Originally specified in **RFC 2307** and used by OpenLDAP[1], the basic format `{SCHEME}HASH` has seen widespread adoption in a number of programs.

3.7.4.1 Standard LDAP Schemes

`passlib.hash.ldap_`*digest* - RFC2307 Standard Digests

Passlib provides support for all the standard LDAP hash formats specified by **RFC 2307**. This includes `{MD5}`, `{SMD5}`, `{SHA}`, `{SSHA}`. These schemes range from somewhat to very insecure, and should not be used except when required. These classes all wrap the underlying hashlib implementations, and are can be used directly as follows:

```
>>> from passlib.hash import ldap_salted_md5 as lsm

>>> # hash password
>>> hash = lsm.hash("password")
>>> hash
'{SMD5}OqsUXNHIhHbznxrqHoIM+ZT8DmE='

>>> # verify password
>>> lms.verify("password", hash)
True
>>> lms.verify("secret", hash)
False
```

See also:

- *password hash usage* – for more usage examples

[1] OpenLDAP homepage - http://www.openldap.org/.

- *ldap_{crypt}* – LDAP `{CRYPT}` wrappers for common Unix hash algorithms.
- *passlib.apps* – for a list of *premade ldap contexts*.

Plain Hashes

Warning: These hashes should not be considered secure in any way, as they are nothing but raw MD5 & SHA-1 digests, which are extremely vulnerable to brute-force attacks.

class passlib.hash.**ldap_md5**
This class stores passwords using LDAP's plain MD5 format, and follows the *PasswordHash API*.

The *hash()* and *genconfig()* methods have no optional keywords.

class passlib.hash.**ldap_sha1**
This class stores passwords using LDAP's plain SHA1 format, and follows the *PasswordHash API*.

The *hash()* and *genconfig()* methods have no optional keywords.

Format

These hashes have the format *prefixchecksum*.

- *prefix* is `{MD5}` for ldap_md5, and `{SHA}` for ldap_sha1.
- *checksum* is the base64 encoding of the raw message digest of the password, using the appropriate digest algorithm.

An example ldap_md5 hash (of `password`) is `{MD5}X03MO1qnZdYdgyfeuILPmQ==`. An example ldap_sha1 hash (of `password`) is `{SHA}W6ph5Mm5Pz8GgiULbPgzG37mj9g=`.

Salted Hashes

class passlib.hash.**ldap_salted_md5**
This class stores passwords using LDAP's salted MD5 format, and follows the *PasswordHash API*.

It supports a 4-16 byte salt.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*bytes*) – Optional salt string. If not specified, one will be autogenerated (this is recommended). If specified, it may be any 4-16 byte string.
- **salt_size** (*int*) – Optional number of bytes to use when autogenerating new salts. Defaults to 4 bytes for compatibility with the LDAP spec, but some systems use larger salts, and Passlib supports any value between 4-16.
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `salt` strings that are too long.

  New in version 1.6.

Changed in version 1.6: This format now supports variable length salts, instead of a fix 4 bytes.

class passlib.hash.**ldap_salted_sha1**

This class stores passwords using LDAP's "Salted SHA1" format, and follows the *PasswordHash API*.

It supports a 4-16 byte salt.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*bytes*) – Optional salt string. If not specified, one will be autogenerated (this is recommended). If specified, it may be any 4-16 byte string.
- **salt_size** (*int*) – Optional number of bytes to use when autogenerating new salts. Defaults to 4 bytes for compatibility with the LDAP spec, but some systems use larger salts, and Passlib supports any value between 4-16.
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `salt` strings that are too long.

  New in version 1.6.

Changed in version 1.6: This format now supports variable length salts, instead of a fix 4 bytes.

class passlib.hash.**ldap_salted_sha256**

This class stores passwords using LDAP's "Salted SHA2-256" format, and follows the *PasswordHash API*.

It supports a 4-16 byte salt.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*bytes*) – Optional salt string. If not specified, one will be autogenerated (this is recommended). If specified, it may be any 4-16 byte string.
- **salt_size** (*int*) – Optional number of bytes to use when autogenerating new salts. Defaults to 8 bytes for compatibility with the LDAP spec, but Passlib supports any value between 4-16.
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `salt` strings that are too long.

New in version 1.7.3.

class passlib.hash.**ldap_salted_sha512**

This class stores passwords using LDAP's "Salted SHA2-512" format, and follows the *PasswordHash API*.

It supports a 4-16 byte salt.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*bytes*) – Optional salt string. If not specified, one will be autogenerated (this is recommended). If specified, it may be any 4-16 byte string.
- **salt_size** (*int*) – Optional number of bytes to use when autogenerating new salts. Defaults to 8 bytes for compatibility with the LDAP spec, but Passlib supports any value between 4-16.

- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a ValueError. If relaxed=True, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include salt strings that are too long.

New in version 1.7.3.

These hashes have the format *prefixdata*.

- *prefix* is {SMD5} for ldap_salted_md5, and {SSHA} for ldap_salted_sha1.
- *data* is the base64 encoding of *checksumsalt*; and in turn *salt* is a multi-byte binary salt, and *checksum* is the raw digest of the the string *passwordsalt*, using the appropriate digest algorithm.

Format

An example hash (of password) is {SMD5}jNoSMNY0cybfuBWiaGlFw3Mfi/U=. After decoding, this results in a raw salt string s\x1f\x8b\xf5, and a raw MD5 checksum of \x8c\xda\x120\xd64s&\xdf\xb8\x15\xa2hiE\xc3.

An example hash (of password) is {SSHA}pKqkNr1tq3wtQqk+UcPyA3HnA2NsU5NJ. After decoding, this results in a raw salt string lS\x93I, and a raw SHA1 checksum of \xa4\xaa\xa46\xbdm\xab|-B\xa9>Q\xc3\xf2\x03q\xe7\x03c.

Security Issues

The LDAP salted hashes should not be considered very secure.

- They use only a single round of digests with known collision and pre-image attacks (SHA1 & MD5).
- They currently use only 32 bits of entropy in their salt, which is only borderline sufficient to defeat rainbow tables, and cannot (portably) be increased.
- The SHA2 salted hashes (SSHA256, SSHA512) are only marginally better. they use the newer SHA2 hash; and 64 bits of entropy in their salt.

Plaintext

class passlib.hash.**ldap_plaintext**

This class stores passwords in plaintext, and follows the *PasswordHash API*.

This class acts much like the generic passlib.hash.plaintext handler, except that it will identify a hash only if it does NOT begin with the {XXX} identifier prefix used by RFC2307 passwords.

The *hash()*, *genhash()*, and *verify()* methods all require the following additional contextual keyword:

Parameters **encoding** (*str*) – This controls the character encoding to use (defaults to utf-8).

This encoding will be used to encode unicode passwords under Python 2, and decode bytes hashes under Python 3.

Changed in version 1.6: The encoding keyword was added.

This handler does not hash passwords at all, rather it encoded them into UTF-8. The only difference between this class and *plaintext* is that this class will NOT recognize any strings that use the {SCHEME}HASH format.

Deviations

- The salt size for the salted digests appears to vary between applications. While OpenLDAP is fixed at 4 bytes, some systems appear to use 8 or more. As of 1.6, Passlib can accept and generate strings with salts between 4-16 bytes, though various servers may differ in what they can handle.

The following schemes are explicitly defined by RFC 2307, and are supported by OpenLDAP.

- *passlib.hash.ldap_md5* - MD5 digest
- *passlib.hash.ldap_sha1* - SHA1 digest
- *passlib.hash.ldap_salted_md5* - salted MD5 digest
- *passlib.hash.ldap_salted_sha1* - salted SHA1 digest
- *passlib.hash.ldap_salted_sha256* - salted SHA256 digest
- *passlib.hash.ldap_salted_sha512* - salted SHA512 digest

passlib.hash.ldap_*crypt* - LDAP crypt() Wrappers

Passlib provides support for all the standard LDAP hash formats specified by **RFC 2307**. One of these, identified by RFC 2307 as the `{CRYPT}` scheme, is somewhat different from the others. Instead of specifying a password hashing scheme, it's supposed to wrap the host OS's `crypt()`. Being host-dependant, the actual hashes supported by this scheme may differ greatly between host systems. In order to provide uniform support across platforms, Passlib defines a corresponding `ldap_`*`crypt-scheme`* class for each of the *standard unix hashes*. These classes all wrap the underlying implementations documented elsewhere in Passlib, and can be used directly as follows:

```
>>> from passlib.hash import ldap_md5_crypt

>>> # hash password
>>> hash = ldap_md5_crypt.hash("password")
>>> hash
'{CRYPT}$1$gwvn5BO0$3dyk8j.UTcsNUPrLMsU6/0'

>>> # verify password
>>> ldap_md5_crypt.verify("password", hash)
True
>>> ldap_md5_crypt.verify("secret", hash)
False

>>> # determine if the underlying crypt() algorithm is supported
>>> # by your host OS, or if the builtin Passlib implementation is being used.
>>> # "os_crypt" - host supported; "builtin" - passlib version
>>> ldap_md5_crypt.get_backend()
"os_crypt"
```

See also:

- *password hash usage* – for more usage examples
- *ldap_{digest}* – for the other standard LDAP hashes.
- *passlib.apps* – for a list of *premade ldap contexts*.

Interface

class passlib.hash.**ldap_des_crypt**

class passlib.hash.**ldap_bsdi_crypt**

class passlib.hash.**ldap_md5_crypt**

class passlib.hash.**ldap_bcrypt**

class passlib.hash.**ldap_sha1_crypt**

class passlib.hash.**ldap_sha256_crypt**

class passlib.hash.**ldap_sha512_crypt**

All of these classes have the same interface as their corresponding underlying hash (e.g. *des_crypt*, *md5_crypt*, etc).

- *passlib.hash.ldap_plaintext* - LDAP-Aware Plaintext Handler

3.7.4.2 Non-Standard LDAP Schemes

None of the following schemes are actually used by LDAP, but follow the LDAP format:

passlib.hash.ldap_*other* - Non-Standard RFC2307 Hashes

This section as a catch-all for a number of password hash formats supported by Passlib which use **RFC 2307** style encoding, but are not part of any standard.

See also:

- *password hash usage* – for examples of how to use these classes via the common hash interface.
- *LDAP / RFC2307 Hashes* for a full list of RFC 2307 style hashes.

Hexadecimal Digests

All of the digests specified in RFC 2307 use base64 encoding. The following are non-standard versions which use hexadecimal encoding, as is found in some applications.

class passlib.hash.**ldap_hex_md5**

hexadecimal version of *ldap_md5*, this is just the md5 digest of the password.

an example hash (of `password`) is `{MD5}5f4dcc3b5aa765d61d8327deb882cf99`.

class passlib.hash.**ldap_hex_sha1**

hexadecimal version of *ldap_sha1*, this is just the sha1 digest of the password.

an example hash (of `password`) is `{SHA}5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8`.

Other Hashes

class passlib.hash.**roundup_plaintext**

RFC 2307 specifies plaintext passwords should be stored without any identifying prefix. This class implements an alternate method used by the Roundup Issue Tracker[1], which (when storing plaintext passwords) uses the identifying prefix `{plaintext}`.

[1] Roundup Issue Tracker homepage - http://www.roundup-tracker.org.

an example hash (of `password`) is `{plaintext}password`.

- *passlib.hash.ldap_hex_md5* - Hex-encoded MD5 Digest
- *passlib.hash.ldap_hex_sha1* - Hex-encoded SHA1 Digest

passlib.hash.ldap_pbkdf2_*digest* - Generic PBKDF2 Hashes

Passlib provides three custom hash schemes based on the PBKDF2[1] algorithm which are compatible with the *ldap hash format*: `ldap_pbkdf2_sha1`, `ldap_pbkdf2_sha256`, `ldap_pbkdf2_sha512`. They feature variable length salts, variable rounds.

See also:

These classes are simply wrappers around the *MCF-Compatible Simple PBKDF2 Hashes*.

Interface

class passlib.hash.**ldap_pbkdf2_sha1**
this is the same as *pbkdf2_sha1*, except that it uses `{PBKDF2}` as its identifying prefix instead of `$pdkdf2$`.

class passlib.hash.**ldap_pbkdf2_sha256**
this is the same as *pbkdf2_sha256*, except that it uses `{PBKDF2-SHA256}` as its identifying prefix instead of `$pdkdf2-sha256$`.

class passlib.hash.**ldap_pbkdf2_sha512**
this is the same as *pbkdf2_sha512*, except that it uses `{PBKDF2-SHA512}` as its identifying prefix instead of `$pdkdf2-sha512$`.

passlib.hash.atlassian_pbkdf2_sha1 - Atlassian's PBKDF2-based Hash

This class provides an implementation of the PBKDF2 based hash used by Atlassian in Jira and other products. Note that unlike the most PBKDF2 hashes supported by Passlib, this one uses a fixed number of rounds (10000). That is currently a sufficient amount, but it cannot be altered; so this scheme should only be used to read existing hashes, and not used in new applications.

See also:

- *password hash usage* – for examples of how to use this class via the common hash interface.
- *passlib.hash.pbkdf2_{digest}* – for some other PBKDF2-based hashes.

Interface

class passlib.hash.**atlassian_pbkdf2_sha1**
This class implements the PBKDF2 hash used by Atlassian.

It supports a fixed-length salt, and a fixed number of rounds.

The *using()* method accepts the following optional keywords:

Parameters

[1] The specification for the PBKDF2 algorithm - http://tools.ietf.org/html/rfc2898#section-5.2, part of **RFC 2898**.

- **salt** (*bytes*) – Optional salt bytes. If specified, the length must be exactly 16 bytes. If not specified, a salt will be autogenerated (this is recommended).
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `salt` strings that are too long.

  New in version 1.6.

Format & Algorithm

All of this scheme's hashes have the format `{PKCS5S2}`*data*, where *data* is a 64 character base64 encoded string; which (when decoded), contains a 16 byte salt, and a 32 byte checksum.

A example hash (of `password`) is:

```
{PKCS5S2}DQIXJU038u4P7FdsuFTY/+35bm41kfjZa57UrdxHp2Mu3qF2uy+ooD+jF5t1tb8J
```

Once decoded, the salt value (in hexadecimal octets) is:

```
0d0217254d37f2ee0fec576cb854d8ff
```

and the checksum value (in hexadecimal octets) is:

```
edf96e6e3591f8d96b9ed4addc47a7632edea176bb2fa8a03fa3179b75b5bf09
```

When calculating the checksum: the password is encoded into UTF-8 if not already encoded. Using the specified salt, and a fixed 10000 rounds, PBKDF2-HMAC-SHA1 is used to generate a 32 byte key, which appended to the salt and encoded in base64.

passlib.hash.fshp - Fairly Secure Hashed Password

Note: While the SHA-2 variants of PBKDF1 have no critical security vulnerabilities, PBKDF1 itself has been deprecated in favor of its successor, PBKDF2. Furthermore, FSHP has been listed as insecure by its author (for unspecified reasons); so this scheme should probably only be used to support existing hashes.

The Fairly Secure Hashed Password (FSHP) scheme[1] is a cross-platform hash based on PBKDF1[2], and uses an LDAP-style hash format. It features a variable length salt, variable rounds, and support for cryptographic hashes from SHA-1 up to SHA-512. This class supports the standard Passlib options for rounds and salt, as well as a special digest keyword for selecting the variant of FSHP to use. It can be used directly as follows:

```
>>> from passlib.hash import fshp

>>> # generate new salt, hash password
>>> hash = fshp.hash("password")
>>> hash
'{FSHP1|16|16384}PtoqcGUetmVEy/uR8715TNqKa8+teMF9qZO1lA9lJNUm1EQBLPZ+qPRLeEPHqy6C'

>>> # the same, but with an explicit number of rounds, larger salt, and specific␣
↪variant
>>> fshp.using(rounds=40000, salt_size=32, variant="sha512").hash("password")
```

(continues on next page)

[1] The FSHP homepage contains implementations in a wide variety of programming languages – https://github.com/bdd/fshp-is-not-secure-anymore.

[2] rfc defining PBKDF1 & PBKDF2 - http://tools.ietf.org/html/rfc2898 -

(continued from previous page)

```
'{FSHP3|32|40000}cB8yE/CuADSgUTQZjWy+YTf/cvbU11D/rHNKiUiB6z4dIaO77U/rmNW
pgZcZllZbCra5GJ8ZfFRNwCHirPqvYTAnbaQQeFQbWym/frRrRev3buoygFQRYexl4091Pc5m'

>>> # verify password
>>> fshp.verify("password", hash)
True
>>> fshp.verify("secret", hash)
False
```

See also:

the generic *PasswordHash usage examples*

Interface

class passlib.hash.**fshp**

This class implements the FSHP password hash, and follows the *PasswordHash API*.

It supports a variable-length salt, and a variable number of rounds.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** – Optional raw salt string. If not specified, one will be autogenerated (this is recommended).
- **salt_size** – Optional number of bytes to use when autogenerating new salts. Defaults to 16 bytes, but can be any non-negative value.
- **rounds** – Optional number of rounds to use. Defaults to 480000, must be between 1 and 4294967295, inclusive.
- **variant** – Optionally specifies variant of FSHP to use.
  - 0 - uses SHA-1 digest (deprecated).
  - 1 - uses SHA-2/256 digest (default).
  - 2 - uses SHA-2/384 digest.
  - 3 - uses SHA-2/512 digest.
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a ValueError. If relaxed=True, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include rounds that are too small or too large, and salt strings that are too long.

  New in version 1.6.

Format & Algorithm

All of this scheme's hashes have the format: {FSHP*variant*|*saltsize*|*rounds*}*data*. A example hash (of password) is:

```
{FSHP1|16|16384}PtoqcGUetmVEy/uR8715TNqKa8+teMF9qZO1lA9lJNUm1EQBLPZ+qPRLeEPHqy6C
```

- *variant* is a decimal integer identifying the version of FSHP; in particular, which cryptographic hash function should be used to calculate the checksum. `1` in the example. (see the class description above for a list of possible values).
- *saltsize* is a decimal integer identifying the number of bytes in the salt. `16` in the example.
- *rounds* is a decimal integer identifying the number of rounds to apply when calculating the checksum (see below). `16384` in the example.
- *data* is a base64-encoded string which, when decoded, contains a salt string of the specified size, followed by the checksum. In the example, the data portion decodes to a salt value (in hexadecimal octets) of:

  ```
  3eda2a70651eb66544cbfb91f3bd794c
  ```

  and a checksum value (in hexadecimal octets) of:

  ```
  da8a6bcfad78c17da993b5940f6524d526d444012cf67ea8f44b7843c7ab2e82
  ```

FSHP is basically just a wrapper around PBKDF1: The checksum is calculated using *pbkdf1()*, passing in the password, the decoded salt string, the number of rounds, and hash function specified by the variant identifier. FSHP has one quirk in that the password is passed in as the pbkdf1 salt, and the salt is passed in as the pbkdf1 password.

Security Issues

- A minor issue is that FSHP swaps the location the password and salt from what is described in the PBKDF1 standard. This issue is mainly noted in order to dismiss it: while the swap permits an attacker to pre-calculate part of the initial digest, the impact of this is negligible when a large number of rounds is used.
- Since PBKDF1 is based on repeated composition of a hash, it is vulnerable to any first-preimage attacks on the underlying hash. This has led to the deprecation of using SHA-1 or earlier hashes with PBKDF1. In contrast, its successor PBKDF2 was designed to mitigate this weakness (among other things), and enjoys much stronger preimage resistance when used with the same cryptographic hashes.

Deviations

- Unicode Policy:

  The official FSHP python implementation takes in a password specified as a series of bytes, and does not specify what encoding should be used; though a `us-ascii` compatible encoding is implied by the implementation, as well as all known reference hashes.

  In order to provide support for unicode strings, Passlib will encode unicode passwords using `utf-8` before running them through FSHP. If a different encoding is desired by an application, the password should be encoded before handing it to Passlib.

- *passlib.hash.roundup_plaintext* - Roundup-specific LDAP Plaintext Handler

3.7.5 SQL Database Hashes

The following schemes are used by various SQL databases to encode their own user accounts. These schemes have encoding and contextual requirements not seen outside those specific contexts:

3.7.5.1 passlib.hash.mssql2000 - MS SQL 2000 password hash

Danger: This algorithm is not considered secure by modern standards. It should only be used when verifying existing hashes, or when interacting with applications that require this format. For new code, see the list of *recommended hashes*.

New in version 1.6.

This class implements the hash algorithm used by Microsoft SQL Server 2000 to store its user account passwords, until it was replaced by a slightly more secure variant (*mssql2005*) in MSSQL 2005. This class can be used directly as follows:

```
>>> from passlib.hash import mssql2000 as m20

>>> # hash password
>>> h = m20.hash("password")
>>> h

↪'0x0100200420C4988140FD3920894C3EDC188E94F428D57DAD5905F6CC1CBAF950CAD4C63F272B2C91E4DEEB5E6444
↪'

>>> # verify correct password
>>> m20.verify("password", h)
True
>>> m20.verify("letmein", h)
False
```

See also:

- *password hash usage* – for more usage examples
- *mssql2005* – the successor to this hash.

Interface

class passlib.hash.**mssql2000**

This class implements the password hash used by MS-SQL 2000, and follows the *PasswordHash API*.

It supports a fixed-length salt.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*bytes*) – Optional salt string. If not specified, one will be autogenerated (this is recommended). If specified, it must be 4 bytes in length.
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a ValueError. If relaxed=True, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include salt strings that are too long.

Format & Algorithm

MSSQL 2000 hashes are usually presented as a series of 92 upper-case hexadecimal characters, prefixed by 0x. An example MSSQL 2000 hash (of "password"):

```
0x0100200420C4988140FD3920894C3EDC188E94F428D57DAD5905F6CC1CBAF950CAD4C63F272B2C91E4DEEB5E6444
```

This encodes 46 bytes of raw data, consisting of:

- a 2-byte constant `0100`
- 4 byte of salt (`200420C4` in the example)
- the first 20 byte digest (`988140FD3920894C3EDC188E94F428D57DAD5905` in the example).
- a second 20 byte digest (`F6CC1CBAF950CAD4C63F272B2C91E4DEEB5E6444` in the example).

The first digest is generated by encoding the unicode password using `UTF-16-LE`, and calculating `SHA1(encoded_secret + salt)`.

The second digest is generated the same as the first, except that the password is converted to upper-case first.

Only the second digest is used when verifying passwords (and hence the hash is case-insensitive). The first digest is presumably for forward-compatibility: MSSQL 2005 removed the second digest, and thus became case sensitive.

Note: MSSQL 2000 hashes do not actually have a native textual format, as they are stored as raw bytes in an SQL table. However, when external programs deal with them, MSSQL generally encodes raw bytes as upper-case hexadecimal, prefixed with `0x`. This is the representation Passlib uses.

Security Issues

This algorithm is reasonably weak, and shouldn't be used for any purpose besides manipulating existing MSSQL 2000 hashes, due to the following flaws:

- The fact that it is case insensitive greatly reduces the keyspace that must be searched by brute-force or pre-computed attacks.
- Its simplicity, and years of research on high-speed SHA1 implementations, makes efficient brute force attacks much more feasible.

3.7.5.2 `passlib.hash.mssql2005` - MS SQL 2005 password hash

Danger: This algorithm is not considered secure by modern standards. It should only be used when verifying existing hashes, or when interacting with applications that require this format. For new code, see the list of *recommended hashes*.

New in version 1.6.

This class implements the hash algorithm used by Microsoft SQL Server 2005 to store its user account passwords, replacing the slightly less secure *mssql2000* variant. This class can be used directly as follows:

```
>>> from passlib.hash import mssql2005 as m25

>>> # hash password
>>> h = m25.hash("password")
>>> h
'0x01006ACDF9FF5D2E211B392EEF1175EFFE13B3A368CE2F94038B'

>>> # verify password
```

(continues on next page)

(continued from previous page)

```
>>> m25.verify("password", h)
True
>>> m25.verify("letmein", h)
False
```

See also:

- *password hash usage* – for more usage examples
- *mssql2000* – the predecessor to this hash.

Interface

class passlib.hash.**mssql2005**

This class implements the password hash used by MS-SQL 2005, and follows the *PasswordHash API*.

It supports a fixed-length salt.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*bytes*) – Optional salt string. If not specified, one will be autogenerated (this is recommended). If specified, it must be 4 bytes in length.
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a ValueError. If relaxed=True, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include salt strings that are too long.

Format & Algorithm

MSSQL 2005 hashes are usually presented as a series of 52 upper-case hexadecimal characters, prefixed by 0x. An example MSSQL 2005 hash (of "password"):

```
0x01006ACDF9FF5D2E211B392EEF1175EFFE13B3A368CE2F94038B
```

This encodes 26 bytes of raw data, consisting of:

- a 2-byte constant 0100
- 4 byte of salt (6ACDF9FF in the example)
- 20 byte digest (5D2E211B392EEF1175EFFE13B3A368CE2F94038B in the example).

The digest is generated by encoding the unicode password using UTF-16-LE, and calculating SHA1(encoded_secret + salt).

This format and algorithm is identical to *mssql2000*, except that this hash omits the 2nd case-insensitive digest used by MSSQL 2000.

Note: MSSQL 2005 hashes do not actually have a native textual format, as they are stored as raw bytes in an SQL table. However, when external programs deal with them, MSSQL generally encodes raw bytes as upper-case hexadecimal, prefixed with 0x. This is the representation Passlib uses.

Security Issues

This algorithm is reasonably weak, and shouldn't be used for any purpose besides manipulating existing MSSQL 2005 hashes. This mainly due to its simplicity, and years of research on high-speed SHA1 implementations, which makes efficient brute force attacks feasible.

3.7.5.3 passlib.hash.mysql323 - MySQL 3.2.3 password hash

Danger: This algorithm is not considered secure by modern standards. It should only be used when verifying existing hashes, or when interacting with applications that require this format. For new code, see the list of *recommended hashes*.

This class implements the first of MySQL's password hash functions, used to store its user account passwords. Introduced in MySQL 3.2.3 under the function `PASSWORD()`, this function was renamed to `OLD_PASSWORD()` under MySQL 4.1, when a newer password hash algorithm was introduced (see *mysql41*). Users will most likely find the frontends provided by *passlib.apps* to be more useful than accessing this class directly. That aside, this class can be used as follows:

```
>>> from passlib.hash import mysql323

>>> # hash password
>>> mysql323.hash("password")
'5d2e19393cc5ef67'

>>> # verify correct password
>>> mysql323.verify("password", '5d2e19393cc5ef67')
True
>>> mysql323.verify("secret", '5d2e19393cc5ef67')
False
```

See also:

- *password hash usage* – for more usage examples
- *passlib.apps* – for a list of predefined *mysql contexts*.

Interface

class passlib.hash.**mysql323**

This class implements the MySQL 3.2.3 password hash, and follows the *PasswordHash API*.

It has no salt and a single fixed round.

The *hash()* and *genconfig()* methods accept no optional keywords.

Format & Algorithm

A mysql-323 password hash consists of 16 hexadecimal digits, directly encoding the 64 bit checksum. MySQL always uses lower-case letters, and so does Passlib (though Passlib will recognize upper case letters as well).

The algorithm used is extremely simplistic, for details, see the source implementation in the footnotes[1].

[1] Source of implementation used by Passlib - http://djangosnippets.org/snippets/1508/

Security Issues

Lacking any sort of salt, ignoring all whitespace, and having a simplistic algorithm that amounts to little more than a checksum, this is not secure, and should not be used for *any* purpose but verifying existing MySQL 3.2.3 - 4.0 password hashes.

3.7.5.4 `passlib.hash.mysql41` - MySQL 4.1 password hash

> **Danger: This algorithm is not considered secure by modern standards.** It should only be used when verifying existing hashes, or when interacting with applications that require this format. For new code, see the list of *recommended hashes*.

This class implements the second of MySQL's password hash functions, used to store its user account passwords. Introduced in MySQL 4.1.1 under the function `PASSWORD()`, it replaced the previous algorithm (*`mysql323`*) as the default used by MySQL, and is still in active use under MySQL 5. Users will most likely find the frontends provided by *`passlib.apps`* to be more useful than accessing this class directly.

See also:

- *password hash usage* – for examples of how to use this class via the common hash interface.
- *`passlib.apps`* for a list of *premade mysql contexts*.

Interface

class passlib.hash.**mysql41**

This class implements the MySQL 4.1 password hash, and follows the *PasswordHash API*.

It has no salt and a single fixed round.

The *`hash()`* and *`genconfig()`* methods accept no optional keywords.

Format & Algorithm

A mysql-41 password hash consists of an asterisk `*` followed by 40 hexadecimal digits, directly encoding the 160 bit checksum. An example hash (of `password`) is `*2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19`. MySQL always uses upper-case letters, and so does Passlib (though Passlib will recognize lower-case letters as well).

The checksum is calculated simply, as the SHA1 hash of the SHA1 hash of the password, which is then encoded into hexadecimal.

Security Issues

Lacking any sort of salt, and using only 2 rounds of the common SHA1 message digest, it's not very secure, and should not be used for *any* purpose but verifying existing MySQL 4.1+ password hashes.

3.7.5.5 passlib.hash.postgres_md5 - PostgreSQL MD5 password hash

Danger: This algorithm is not considered secure by modern standards. It should only be used when verifying existing hashes, or when interacting with applications that require this format. For new code, see the list of *recommended hashes*.

This class implements the md5-based hash algorithm used by PostgreSQL to store its user account passwords. This scheme was introduced in PostgreSQL 7.2; prior to this PostgreSQL stored its password in plain text. Users will most likely find the frontend provided by *passlib.apps* to be more useful than accessing this class directly. That aside, this class can be used directly as follows:

```
>>> from passlib.hash import postgres_md5

>>> # hash password using specified username
>>> hash = postgres_md5.hash("password", user="username")
>>> hash
'md55a231fcdb710d73268c4f44283487ba2'

>>> # verify correct password
>>> postgres_md5.verify("password", hash, user="username")
True
>>> # verify correct password w/ wrong username
>>> postgres_md5.verify("password", hash, user="somebody")
False
>>> # verify incorrect password
>>> postgres_md5.verify("password", hash, user="username")
False
```

See also:

the generic *PasswordHash usage examples*

Interface

class passlib.hash.**postgres_md5**

This class implements the Postgres MD5 Password hash, and follows the *PasswordHash API*.

It does a single round of hashing, and relies on the username as the salt.

The *hash()*, *genhash()*, and *verify()* methods all require the following additional contextual keywords:

Parameters **user** (*str*) – name of postgres user account this password is associated with.

Format & Algorithm

Postgres-MD5 hashes all have the format `md5`*`checksum`*, where *checksum* is 32 hexadecimal digits, encoding a 128-bit checksum. This checksum is the MD5 message digest of the password concatenated with the username.

Security Issues

This algorithm it not suitable for *any* use besides manipulating existing PostgreSQL account passwords, due to the following flaws:

- Its use of the username as a salt value means that common usernames (e.g. `admin`, `root`, `postgres`) will occur more frequently as salts, weakening the effectiveness of the salt in foiling pre-computed tables.
- Since the keyspace of `user+password` is still a subset of ascii characters, existing MD5 lookup tables have an increased chance of being able to reverse common hashes.
- Its simplicity makes high-speed brute force attacks much more feasible[3] .

3.7.5.6 `passlib.hash.oracle10` - Oracle 10g password hash

Danger: This algorithm is dangerously insecure by modern standards. It is trivially broken, and should not be used if at all possible. For new code, see the list of *recommended hashes*.

This class implements the hash algorithm used by the Oracle Database up to version 10g Rel.2. It was superseded by a newer algorithm in *Oracle 11*. This class can be used directly as follows (note that this class requires a username for all encrypt/verify operations):

```
>>> from passlib.hash import oracle10 as oracle10

>>> # hash password using specified username
>>> hash = oracle10.hash("password", user="username")
>>> hash
'872805F3F4C83365'

>>> # verify correct password
>>> oracle10.verify("password", hash, user="username")
True
>>> # verify correct password w/ wrong username
>>> oracle10.verify("password", hash, user="somebody")
False
>>> # verify incorrect password
>>> oracle10.verify("letmein", hash, user="username")
False
```

See also:

the generic *PasswordHash usage examples*

Warning: This implementation has not been compared very carefully against the official implementation or reference documentation, and its behavior may not match under various border cases. *caveat emptor*.

Interface

class passlib.hash.**oracle10**

This class implements the password hash used by Oracle up to version 10g, and follows the *PasswordHash API*.

It does a single round of hashing, and relies on the username as the salt.

The *hash()*, *genhash()*, and *verify()* methods all require the following additional contextual keywords:

Parameters **user** (*str*) – name of oracle user account this password is associated with.

[3] Blog post demonstrating brute-force attack http://pentestmonkey.net/blog/cracking-postgres-hashes/.

Format & Algorithm

Oracle10 hashes all consist of a series of 16 hexadecimal digits, representing the resulting checksum. Oracle10 hashes can be formed by the following procedure:

1. Concatenate the username and password together.
2. Convert the result to upper case
3. Encoding the result in a multi-byte format[1] such that ascii characters (eg: `USER`) are represented with additional null bytes inserted (eg: `\x00U\x00S\x00E\x00R`).
4. Right-pad the result with null bytes, to bring the total size to an integer multiple of 8. this is the final input string.
5. The input string is then encoded using DES in CBC mode. The string `\x01\x23\x45\x67\x89\xAB\xCD\xEF` is used as the DES key, and a block of null bytes is used as the CBC initialization vector. All but the last block of ciphertext is discarded.
6. The input string is then run through DES-CBC a second time; this time the last block of ciphertext from step 5 is used as the DES key, a block of null bytes is still used as the CBC initialization vector. All but the last block of ciphertext is discarded.
7. The last block of ciphertext of step 6 is converted to a hexadecimal string, and returned as the checksum.

Security Issues

This algorithm it not suitable for *any* use besides manipulating existing Oracle10 account passwords, due to the following flaws[2]:

- Its use of the username as a salt value means that common usernames (e.g. `system`) will occur more frequently as salts, weakening the effectiveness of the salt in foiling pre-computed tables.
- The fact that it is case insensitive, and simply concatenates the username and password, greatly reduces the keyspace that must be searched by brute-force or pre-computed attacks.
- Its simplicity, and decades of research on high-speed DES implementations, makes efficient brute force attacks much more feasible.

Deviations

Passlib's implementation of the Oracle10g hash may deviate from the official implementation in unknown ways, as there is no official documentation. There is only one known issue:

- Unicode Policy

  Lack of testing (and test vectors) leaves it unclear as to how Oracle 10g handles passwords containing non-7bit ascii. In order to provide support for unicode strings, Passlib will encode unicode passwords using `utf-16-be`[1] before running them through the Oracle10g algorithm. This behavior may be altered in the future, if further testing reveals another behavior is more in line with the official representation. This note applies as well to any provided username, as they are run through the same policy.

[1] The exact encoding used in step 3 of the algorithm is not clear from known references. Passlib uses `utf-16-be`, as this is both compatible with existing test vectors, and supports unicode input.

[2] Whitepaper analyzing flaws in this algorithm - http://www.isg.rhul.ac.uk/~ccid/publications/oracle_passwd.pdf.

3.7.5.7 passlib.hash.oracle11 - Oracle 11g password hash

This class implements the hash algorithm introduced in version 11g of the Oracle Database. It supersedes the *Oracle 10* password hash. This class can be can be used directly as follows:

```
>>> from passlib.hash import oracle11 as oracle11

>>> # generate new salt, hash password
>>> hash = oracle11.hash("password")
>>> hash
'S:4143053633E59B4992A8EA17D2FF542C9EDEB335C886EED9C80450C1B4E6'

>>> # verify password
>>> oracle11.verify("password", hash)
True
>>> oracle11.verify("secret", hash)
False
```

See also:

the generic *PasswordHash usage examples*

Warning: This implementation has not been compared very carefully against the official implementation or reference documentation, and its behavior may not match under various border cases. *caveat emptor*.

Interface

class passlib.hash.**oracle11**

This class implements the Oracle11g password hash, and follows the *PasswordHash API*.

It supports a fixed-length salt.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*str*) – Optional salt string. If not specified, one will be autogenerated (this is recommended). If specified, it must be 20 hexadecimal characters.
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `salt` strings that are too long.

  New in version 1.6.

Format & Algorithm

An example oracle11 hash (of the string `password`) is:

```
S:4143053633E59B4992A8EA17D2FF542C9EDEB335C886EED9C80450C1B4E6
```

An oracle11 hash string has the format `S:`*checksumsalt*, where:

- `S:` is the prefix used to identify oracle11 hashes (as distinct from oracle10 hashes, which have no constant prefix).

- *checksum* is 40 hexadecimal characters; encoding a 160-bit checksum.

  (4143053633E59B4992A8EA17D2FF542C9EDEB335 in the example)

- *salt* is 20 hexadecimal characters; providing a 80-bit salt (C886EED9C80450C1B4E6 in the example).

The Oracle 11 hash has a very simple algorithm: The salt is decoded from its hexadecimal representation into binary, and the SHA-1 digest of *passwordraw_salt* is then encoded into hexadecimal, and returned as the checksum.

Deviations

Passlib's implementation of the Oracle11g hash may deviate from the official implementation in unknown ways, as there is no official documentation. There is only one known issue:

- Unicode Policy

  Lack of testing (and test vectors) leaves it unclear as to how Oracle 11g handles passwords containing non-7bit ascii. In order to provide support for unicode strings, Passlib will encode unicode passwords using utf-8 before running them through Oracle11. This behavior may be altered in the future, if further testing reveals another behavior is more in line with the official representation.

3.7.6 MS Windows Hashes

The following hashes are used in various places by Microsoft Windows. As they were designed for "internal" use, they generally contain no identifying markers, identifying them is pretty much context-dependant.

3.7.6.1 passlib.hash.lmhash - LanManager Hash

> **Danger: This algorithm is not considered secure by modern standards.** It should only be used when verifying existing hashes, or when interacting with applications that require this format. For new code, see the list of *recommended hashes*.

New in version 1.6.

This class implements the LanManager Hash (aka *LanMan* or *LM* hash). It was used by early versions of Microsoft Windows to store user passwords, until it was supplanted (though not entirely replaced) by the *nthash* algorithm in Windows NT. It continues to crop up in production due to its integral role in the legacy NTLM authentication protocol. This class can be used directly as follows:

```
>>> from passlib.hash import lmhash

>>> # hash password
>>> h = lmhash.hash("password")
>>> h
'e52cac67419a9a224a3b108f3fa6cb6d'

>>> # verify correct password
>>> lmhash.verify("password", h)
True
>>> # verify incorrect password
>>> lmhash.verify("secret", h)
False
```

See also:

the generic *PasswordHash usage examples*

Interface

class passlib.hash.**lmhash**

This class implements the Lan Manager Password hash, and follows the *PasswordHash API*.

It has no salt and a single fixed round.

The *using()* method accepts a single optional keyword:

Parameters **truncate_error** (*bool*) – By default, this will silently truncate passwords larger than 14 bytes. Setting truncate_error=True will cause *hash()* to raise a *PasswordTruncateError* instead.

New in version 1.7.

The *hash()* and *verify()* methods accept a single optional keyword:

Parameters **encoding** (*str*) – This specifies what character encoding LMHASH should use when calculating digest. It defaults to cp437, the most common encoding encountered.

Note that while this class outputs digests in lower-case hexadecimal, it will accept upper-case as well.

Issues with Non-ASCII Characters

Passwords containing only ascii characters should hash and compare correctly across all LMhash implementations. However, due to historical issues, no two LMhash implementations handle non-ascii characters in quite the same way. While Passlib makes every attempt to behave as close to correct as possible, the meaning of "correct" is dependant on the software you are interoperating with. If you think you will have passwords containing non-ascii characters, please read the *Deviations* section (below) for details about the known interoperability issues. It's a mess of codepages.

Format & Algorithm

A LM hash consists of 32 hexadecimal digits, which encode the 16 byte digest. An example hash (of password) is e52cac67419a9a224a3b108f3fa6cb6d.

The digest is calculated as follows:

1. First, the password should be converted to uppercase, and encoded using the "OEM Codepage" of the Windows release that the host / target server is running[2].

   For pure-ASCII passwords, this step can be performed using the us-ascii encoding (as most OEM Codepages are ASCII-compatible). However, for passwords with non-ASCII characters, this step is fraught with compatibility issues and border cases (see *Deviations* for details).

2. The password is then truncated to 14 bytes, or the end NULL padded to 14 bytes; as appropriate.

3. The first 7 bytes of the truncated password from step 2 are used as a key to DES encrypt the constant KGS!@#$%, resulting in the first 8 bytes of the final digest.

4. Step 3 is repeated using the second 7 bytes of the password from step 2, resulting in the second 8 bytes of the final digest.

[2] The OEM codepage used by specific Window XP (and earlier) releases can be found at http://msdn.microsoft.com/nl-nl/goglobal/cc563921%28en-us%29.aspx.

5. The combined digests from 3 and 4 are then encoded to hexadecimal.

Security Issues

Due to a myriad of flaws, and the existence high-speed password cracking software dedicated to LMHASH, this algorithm should be considered broken. The major flaws include:

- It has no salt, making hashes easily pre-computable.
- It limits the password to 14 characters, and converts the password to uppercase before hashing, greatly reducing the keyspace.
- By breaking the password into two independent chunks, they can be attacked independently and simultaneously.
- The independence of the chunks reveals significant information about the original password: The second 8 bytes of the digest are the same for all passwords < 8 bytes; and for passwords of 8-9 characters, the second chunk can be broken *much* faster, revealing part of the password, and reducing the likely keyspace for the first chunk.

Deviations

Passlib's implementation differs from others in a few ways, all related to the handling of non-ASCII characters.

- Unicode Policy:

  Officially, unicode passwords should be encoded using the "OEM Codepage" used[2] by the specific release of Windows that the host or target server is running. Common encodings include `cp437` (used by the English edition of Windows XP), `cp580` (used by many Western European editions of XP), and `cp866` (used by many Eastern European editions of XP). Complicating matters further, some third-party implementations are known to use encodings such as `latin-1` and `utf-8`, which cause non-ASCII characters to hash in a manner incompatible with the canonical MS Windows implementation.

  Thus if an application wishes to provide support for non-ASCII passwords, it must decide which encoding to use.

  Passlib uses `cp437` as it's default encoding for unicode strings. However, if your database used a different encoding, you will need to either first encode the passwords into bytes, or override the default encoding via `lmhash.hash(secret, encoding="some-other-codec")`

  All known encodings are `us-ascii`-compatible, so for ASCII passwords, the default should be sufficient.

- Upper Case Conversion:

  Note: Future releases of Passlib may change this behavior as new information and code is integrated.

  Once critical step in the LMHASH algorithm is converting the password to upper case. While ASCII characters are uppercased as normal, non-ASCII characters are converted in implementation-dependant ways:

  Windows systems encode the password first, and then convert it to uppercase using an codepage-specific table. For the most part these tables seem to agree with the Unicode specification, but there are some codepoints where they deviate (for example, Unicode uppercases U+00B5 -> U+039C, but `cp437` leaves it unchanged[3]).

  In contrast, most third-party implementations (Passlib included) perform the uppercase conversion first using the Unicode specification, and then encode the password second; despite the non-ASCII border cases where the resulting hash would not match the official Windows hash.

[3] Online discussion dealing with upper-case encoding issues - http://www.openwall.com/lists/john-dev/2011/08/01/2.

3.7.6.2 passlib.hash.nthash - Windows' NT-HASH

Danger: This algorithm is dangerously insecure by modern standards. It is trivially broken, and should not be used if at all possible. For new code, see the list of *recommended hashes*.

New in version 1.6.

This class implements the NT-HASH algorithm, used by Microsoft Windows NT and successors to store user account passwords, supplanting the much weaker *lmhash* algorithm. This class can be used directly as follows:

```
>>> from passlib.hash import nthash

>>> # hash password
>>> h = nthash.hash("password")
>>> h
'8846f7eaee8fb117ad06bdd830b7586c'

>>> # verify password
>>> nthash.verify("password", h)
True
>>> nthash.verify("secret", h)
False
```

See also:

the generic *PasswordHash usage examples*

Interface

class passlib.hash.**nthash**

This class implements the NT Password hash, and follows the *PasswordHash API*.

It has no salt and a single fixed round.

The *hash()* and *genconfig()* methods accept no optional keywords.

Note that while this class outputs lower-case hexadecimal digests, it will accept upper-case digests as well.

Format & Algorithm

A nthash consists of 32 hexadecimal digits, which encode the digest. An example hash (of `password`) is `8846f7eaee8fb117ad06bdd830b7586c`.

The digest is calculated by encoding the secret using `UTF-16-LE`, taking the MD4 digest, and then encoding that as hexadecimal.

FreeBSD Variant

For cross-compatibility, FreeBSD's `crypt()` supports storing NTHASH digests in a manner compatible with the *Modular Crypt Format*, to enable administrators to store user passwords in a manner compatible with the SMB/CIFS protocol. This is accomplished by assigning NTHASH digests the identifier `$3$`, and prepending the identifier to the normal (lowercase) NTHASH digest. An example digest (of `password`) is `$3$$8846f7eaee8fb117ad06bdd830b7586c` (note the doubled `$$`).

passlib.hash.**bsd_nthash**

This object supports FreeBSD's representation of NTHASH (which is compatible with the *Modular Crypt Format*), and follows the *PasswordHash API*.

It has no salt and a single fixed round.

The *hash()* and *genconfig()* methods accept no optional keywords.

Changed in version 1.6: This hash was named nthash under previous releases of Passlib.

Security Issues

This algorithm should be considered *completely* broken:

- It has no salt.
- The MD4 message digest has been severely compromised by collision and preimage attacks.
- Brute-force and pre-computed attacks exist targeting MD4 hashes in general, and the encoding used by NTHASH in particular.

3.7.6.3 passlib.hash.msdcc - Windows' Domain Cached Credentials

Danger: This algorithm is not considered secure by modern standards. It should only be used when verifying existing hashes, or when interacting with applications that require this format. For new code, see the list of *recommended hashes*.

New in version 1.6.

This class implements the DCC (Domain Cached Credentials) hash, used by Windows to cache and verify remote credentials when the relevant server is unavailable. It is known by a number of other names, including "mscache" and "mscash" (Microsoft CAched haSH). Security wise it is not particularly strong, as it's little more than *nthash* salted with a username. It was replaced by *msdcc2* in Windows Vista. This class can be used directly as follows:

```
>>> from passlib.hash import msdcc

>>> # hash password using specified username
>>> hash = msdcc.hash("password", user="Administrator")
>>> hash
'25fd08fa89795ed54207e6e8442a6ca0'

>>> # verify correct password
>>> msdcc.verify("password", hash, user="Administrator")
True
>>> # verify correct password w/ wrong username
>>> msdcc.verify("password", hash, user="User")
False
>>> # verify incorrect password
>>> msdcc.verify("letmein", hash, user="Administrator")
False
```

See also:

- *password hash usage* – for more usage examples
- *msdcc2* – the successor to this hash

Interface

class passlib.hash.**msdcc**

This class implements Microsoft's Domain Cached Credentials password hash, and follows the *PasswordHash API*.

It has a fixed number of rounds, and uses the associated username as the salt.

The *hash()*, *genhash()*, and *verify()* methods have the following optional keywords:

Parameters **user** (*str*) – String containing name of user account this password is associated with. This is required to properly calculate the hash.

This keyword is case-insensitive, and should contain just the username (e.g. `Administrator`, not `SOMEDOMAIN\Administrator`).

Note that while this class outputs lower-case hexadecimal digests, it will accept upper-case digests as well.

Format & Algorithm

Much like `lmhash` and `nthash`, MS DCC hashes consists of a 16 byte digest, usually encoded as 32 hexadecimal characters. An example hash (of `"password"` with the account `"Administrator"`) is `25fd08fa89795ed54207e6e8442a6ca0`.

The digest is calculated as follows:

1. The password is encoded using `UTF-16-LE`.
2. The MD4 digest of step 1 is calculated. (The result of this step is identical to the *nthash* of the password).
3. The unicode username is converted to lowercase, and encoded using `UTF-16-LE`. This should be just the plain username (e.g. `User` not `SOMEDOMAIN\\User`)
4. The username from step 3 is appended to the digest from step 2; and the MD4 digest of the result is calculated.
5. The result of step 4 is encoded into hexadecimal, this is the DCC hash.

Security Issues

This algorithm is should not be used for any purpose besides manipulating existing DCC v1 hashes, due to the following flaws:

- Its use of the username as a salt value (and lower-case at that), means that common usernames (e.g. `Administrator`) will occur more frequently as salts, weakening the effectiveness of the salt in foiling pre-computed tables.
- The MD4 message digest has been severely compromised by collision and preimage attacks.
- Efficient brute-force attacks on MD4 exist.

3.7.6.4 passlib.hash.msdcc2 - Windows' Domain Cached Credentials v2

New in version 1.6.

This class implements the DCC2 (Domain Cached Credentials version 2) hash, used by Windows Vista and newer to cache and verify remote credentials when the relevant server is unavailable. It is known by a number of other names, including "mscache2" and "mscash2" (Microsoft CAched haSH). It replaces the weaker *msdcc v1* hash used by previous releases of Windows. Security wise it is not particularly weak, but due to its use of the username as a salt,

it should probably not be used for anything but verifying existing cached credentials. This class can be used directly as follows:

```
>>> from passlib.hash import msdcc2

>>> # hash password using specified username
>>> hash = msdcc2.hash("password", user="Administrator")
>>> hash
'4c253e4b65c007a8cd683ea57bc43c76'

>>> # verify correct password
>>> msdcc2.verify("password", hash, user="Administrator")
True
>>> # verify correct password w/ wrong username
>>> msdcc2.verify("password", hash, user="User")
False
>>> # verify incorrect password
>>> msdcc2.verify("letmein", hash, user="Administrator")
False
```

See also:

- *password hash usage* – for more usage examples
- *msdcc* – the predecessor to this hash

Interface

class passlib.hash.**msdcc2**

This class implements version 2 of Microsoft's Domain Cached Credentials password hash, and follows the *PasswordHash API*.

It has a fixed number of rounds, and uses the associated username as the salt.

The *hash()*, *genhash()*, and *verify()* methods have the following extra keyword:

Parameters **user** (*str*) – String containing name of user account this password is associated with. This is required to properly calculate the hash.

This keyword is case-insensitive, and should contain just the username (e.g. `Administrator`, not `SOMEDOMAIN\Administrator`).

Format & Algorithm

Much like `lmhash`, `nthash`, and `msdcc`, MS DCC v2 hashes consists of a 16 byte digest, usually encoded as 32 hexadecimal characters. An example hash (of `"password"` with the account `"Administrator"`) is `4c253e4b65c007a8cd683ea57bc43c76`.

The digest is calculated as follows:

1. The password is encoded using `UTF-16-LE`.
2. The MD4 digest of step 1 is calculated. (The result of this is identical to the *nthash* digest of the password).
3. The unicode username is converted to lowercase, and encoded using `UTF-16-LE`. This should be just the plain username (e.g. `User` not `SOMEDOMAIN\\User`)
4. The username from step 3 is appended to the digest from step 2; and the MD4 digest of the result is calculated (The result of this is identical to the *msdcc* digest).

5. *PBKDF2-HMAC-SHA1* is then invoked, using the result of step 4 as the secret, the username from step 3 as the salt, 10240 rounds, and resulting in a 16 byte digest.
6. The result of step 5 is encoded into hexadecimal; this is the DCC2 hash.

Security Issues

This hash is essentially *msdcc v1* with a fixed-round PBKDF2 function wrapped around it. The number of rounds of PBKDF2 is currently sufficient to make this a semi-reasonable way to store passwords, but the use of the lowercase username as a salt, and the fact that the rounds can't be increased, means this hash is not particularly future-proof, and should not be used for new applications.

Deviations

- Max Password Size

  Windows appears to enforce a maximum password size, but the actual value of this limit is unclear; sources report it to be set at assorted values from 26 to 128 characters, and it may in fact vary between Windows releases. The one consistent piece of information is that passwords above the limit are simply not allowed (rather than truncated ala *des_crypt*). Because of this, Passlib does not currently enforce a size limit: any hashes this class generates should be correct, provided Windows is willing to accept a password of that size.

3.7.7 Cisco Hashes

Cisco IOS

The following hashes are used in various places on Cisco IOS, and are usually referred to by a Cisco-assigned "type" code:

3.7.7.1 passlib.hash.cisco_type7 - Cisco "Type 7" hash

Danger: This is not a hash, this is a reversible plaintext encoding. **This format can be trivially decoded**.

New in version 1.6.

This class implements the "Type 7" password encoding used Cisco IOS. This is not actually a true hash, but a reversible XOR Cipher encoding the plaintext password. Type 7 strings are (and were designed to be) plaintext equivalent; the goal was to protect from "over the shoulder" eavesdropping, and little else. They can be trivially decoded. This class can be used directly as follows:

```
>>> from passlib.hash import cisco_type7

>>> # encode password
>>> h = cisco_type7.hash("password")
>>> h
'044B0A151C36435C0D'

>>> # verify password
>>> cisco_type7.verify("password", h)
True
>>> pm.verify("letmein", h)
```

(continues on next page)

(continued from previous page)

```
False

>>> # to demonstrate this is an encoding, not a real hash,
>>> # this class supports decoding the resulting string:
>>> cisco_type7.decode(h)
"password"
```

See also:

the generic *PasswordHash usage examples*

Note: This implementation should work correctly for most cases, but may not fully implement some edge cases (see *Deviations* below). Please report any issues encountered.

Interface

class passlib.hash.**cisco_type7**

This class implements the "Type 7" password encoding used by Cisco IOS, and follows the *PasswordHash API*. It has a simple 4-5 bit salt, but is nonetheless a reversible encoding instead of a real hash.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*int*) – This may be an optional salt integer drawn from `range(0,16)`. If omitted, one will be chosen at random.
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `salt` values that are out of range.

Note that while this class outputs digests in upper-case hexadecimal, it will accept lower-case as well.

This class also provides the following additional method:

classmethod decode(*hash*, *encoding='utf-8'*)

decode hash, returning original password.

Parameters

- **hash** – encoded password
- **encoding** – optional encoding to use (defaults to `UTF-8`).

Returns password as unicode

Format & Algorithm

The Cisco Type 7 encoding consists of two decimal digits (encoding the salt), followed a series of hexadecimal characters, two for every byte in the encoded password. An example encoding (of `"password"`) is `044B0A151C36435C0D`. This has a salt/offset of 4 (`04` in the example), and encodes password via `4B0A151C36435C0D`.

Note: The following description may not be entirely correct with respect to the official algorithm, see the *Deviations* section for details.

The algorithm is a straightforward XOR Cipher:

1. The algorithm relies on the following `ascii`-encoded 53-byte constant:

   ```
   "dsfd;kfoA,.iyewrkldJKDHSUBsgvca69834ncxv9873254k;fg87"
   ```

2. A integer salt should be generated from the range 0 .. 15. The first two characters of the encoded string are the zero-padded decimal encoding of the salt.

3. The remaining characters of the encoded string are generated as follows: For each byte in the password (starting with the 0th byte), the *i*'th byte of the password is encoded as follows:

   a. let `j=(i + salt) % 53`

   b. XOR the *i*'th byte of the password with the *j*'th byte of the magic constant.

   c. encode the resulting byte as uppercase hexadecimal, and append to the encoded string.

Deviations

This implementation differs from the official one in a few ways. It may be updated as more information becomes available.

- Unicode Policy:

  Type 7 encoding is primarily used with `ASCII` passwords, how it handles other characters is not known.

  In order to provide support for unicode strings, Passlib will encode unicode passwords using `UTF-8` before running them through this algorithm. If a different encoding is desired by an application, the password should be encoded before handing it to Passlib.

- Magic Constant:

  Other implementations contain a truncated 26-byte constant instead of the 53-byte constant listed above. However, it is likely those implementations were merely incomplete, as they exhibit other issues as well after the 26th byte is reached (throwing an error, truncating the password, outputing garbage), and only worked for shorter passwords.

- Salt Range:

  All known test vectors contain salt values in `range(0,16)`. However, the algorithm itself should be able to handle any salt value in `range(0,53)` (the size of the key). For maximum compatibility with other implementations, Passlib will accept `range(0,53)`, but only generate salts in `range(0,16)`.

- While this implementation handles all known test vectors, and tries to make sense of the disparate implementations, the actual algorithm has not been published by Cisco, so there may be other unknown deviations.

- *passlib.hash.md5_crypt* – "Type 5" hashes are actually just the standard Unix MD5-Crypt hash, the format is identical.

- *passlib.hash.cisco_type7* – "Type 7" isn't actually a hash, but a reversible encoding designed to obscure passwords from idle view.

- "Type 8" hashes are based on PBKDF2-HMAC-SHA256; but not currently supported by passlib (issue 87).

- "Type 9" hashes are based on scrypt; but not currently supported by passlib (issue 87).

Cisco PIX & ASA

Separately from this, Cisco PIX & ASA firewalls have their own hash formats, generally identified by the "format" parameter in the `username user password hash format` config line they occur in. The following are known & handled by passlib:

3.7.7.2 passlib.hash.cisco_pix - Cisco PIX MD5 hash

Danger: This algorithm is not considered secure by modern standards. It should only be used when verifying existing hashes, or when interacting with applications that require this format. For new code, see the list of *recommended hashes*.

New in version 1.6.

Overview

Todo: Caveat Emptor

Passlib's implementations of *cisco_pix* and *cisco_asa* both need verification. For those with access to Cisco PIX and ASA systems, verifying Passlib's reference vectors would be a great help (see issue 51). In the mean time, there are no guarantees that passlib correctly replicates the official implementation.

Changed in version 1.7.1: A number of *bugs* were fixed after expanding the reference vectors, and testing against an ASA 9.6 system.

The *cisco_asa* class implements the "encrypted" password hash algorithm commonly found on Cisco ASA systems. The companion *cisco_pix* class implements the older variant found on Cisco PIX. Aside from internal differences, and slightly different limitations, the two hashes have the same format, and in some cases the same output.

These classes can be used directly to generate or verify a hash for a specific user. Specifying the user account name is required for this hash:

```
>>> from passlib.hash import cisco_asa

>>> # hash password using specified username
>>> hash = cisco_asa.hash("password", user="user")
>>> hash
'A5XOy94YKDPXCo7U'

>>> # verify correct password
>>> cisco_asa.verify("password", hash, user="user")
True

>>> # verify correct password w/ wrong username
>>> cisco_asa.verify("password", hash, user="other")
False

>>> # verify incorrect password
>>> cisco_asa.verify("letmein", hash, user="user")
False
```

The main "enable" password can be hashes / verified just by omitting the `user` parameter, or setting `user=""`:

```
>>> # hash password without associated user account
>>> hash2 = cisco_asa.hash("password")
>>> hash2
'NuLKvvWGg.x9HEKO'

>>> # verify password without associated user account
>>> cisco_asa.verify("password", hash2)
True
```

See also:

the generic *PasswordHash usage examples*

Interface

class passlib.hash.**cisco_pix**

This class implements the password hash used by older Cisco PIX firewalls, and follows the *PasswordHash API*. It does a single round of hashing, and relies on the username as the salt.

This class only allows passwords <= 16 bytes, anything larger will result in a *PasswordSizeError* if passed to `hash()`, and be silently rejected if passed to `verify()`.

The *hash()*, *genhash()*, and *verify()* methods all support the following extra keyword:

Parameters **user** (*str*) – String containing name of user account this password is associated with.

This is *required* in order to correctly hash passwords associated with a user account on the Cisco device, as it is used to salt the hash.

Conversely, this *must* be omitted or set to `""` in order to correctly hash passwords which don't have an associated user account (such as the "enable" password).

New in version 1.6.

Changed in version 1.7.1: Passwords > 16 bytes are now rejected / throw error instead of being silently truncated, to match Cisco behavior. A number of *bugs* were fixed which caused prior releases to generate unverifiable hashes in certain cases.

class passlib.hash.**cisco_asa**

This class implements the password hash used by Cisco ASA/PIX 7.0 and newer (2005). Aside from a different internal algorithm, it's use and format is identical to the older *cisco_pix* class.

For passwords less than 13 characters, this should be identical to `cisco_pix`, but will generate a different hash for most larger inputs (See the *Format & Algorithm* section for the details).

This class only allows passwords <= 32 bytes, anything larger will result in a *PasswordSizeError* if passed to `hash()`, and be silently rejected if passed to `verify()`.

New in version 1.7.

Changed in version 1.7.1: Passwords > 32 bytes are now rejected / throw error instead of being silently truncated, to match Cisco behavior. A number of *bugs* were fixed which caused prior releases to generate unverifiable hashes in certain cases.

Note: These hash algorithms have a context-sensitive peculiarity. They take in an optional username to salt the hash, but have specific restrictions...

- The username *must* be provided in order to correctly hash passwords associated with a user account on the Cisco device.

- Conversely, the username *must not* be provided (or must be set to `""`) in order to correctly hash passwords which don't have an associated user account (such as the "enable" password).

Format & Algorithm

Cisco PIX & ASA hashes consist of a 12 byte digest, encoded as a 16 character *HASH64*-encoded string. An example hash (of `"password"`, with user `""`) is `"NuLKvvWGg.x9HEKO"`.

The PIX / ASA digests are calculated as follows:

1. The password is encoded using `UTF-8` (though entering non-ASCII characters is subject to interface-specific issues, and may lead to problems such as double-encoding).

   If the result is greater than 16 bytes (for PIX), or 32 bytes (for ASA), the password is not allowed – it will be rejected when set, and simplify not verify during authentication.

2. If the hash is associated with a user account, append the first four bytes of the user account name to the end of the password. If the hash is NOT associated with a user account (e.g. it's the "enable" password), this step should be omitted.

   If the user account is 1-3 bytes, it is repeated until all 4 bytes are filled up (e.g. "usr" becomes "usru").

   For `cisco_asa`, this step is omitted if the password is 28 bytes or more.

3. The password+user string is truncated, or right-padded with NULLs, until it's 16 bytes in size.

   For `cisco_asa`, if the password+user string is 16 or more bytes, a padding size of 32 is used instead.

4. Run the result of step 3 through MD5.

5. Discard every 4th byte of the 16-byte MD5 hash, starting with the 4th byte.

6. Encode the 12-byte result using *HASH64*.

Changed in version 1.7.1: Updated to reflect current understanding of the algorithm.

Security Issues

This algorithm is not suitable for *any* use besides manipulating existing Cisco PIX hashes, due to the following flaws:

- Its use of the username as a salt value (and only the first four characters at that), means that common usernames (e.g. `admin`, `cisco`) will occur more frequently as salts, weakening the effectiveness of the salt in foiling pre-computed tables.
- Its truncation of the `password+user` combination to 16 characters additionally limits the keyspace, and the effectiveness of the username as a salt; making pre-computed and brute force attacks much more feasible.
- Since the keyspace of `password+user` is still a subset of ascii characters, existing MD5 lookup tables have an increased chance of being able to reverse common hashes.
- Its simplicity, and the weakness of MD5, makes high-speed brute force attacks much more feasible.
- Furthermore, it discards of 1/4 of MD5's already small 16 byte digest, making collisions much more likely.

Deviations

This implementation tries to adhere to the canonical Cisco implementation, but without an official specification, there may be other unknown deviations. The following are known issues:

- Unicode Policy:

  ASA documentation[4] indicates it uses UTF-8 encoding, and Passlib does as well. However, some ASA interfaces have issues such as: ASDM may double-encode unicode characters, and SSH connections may drop non-ASCII characters entirely.

- How usernames are added is not entirely pinned down. Under ASA, 3-character usernames have their last character repeated to make a string of length 4. It is currently assumed that a similar repetition would be applied to usernames of 1-2 characters, and that this applies to PIX as well; though neither assumption has been confirmed.

- **Passlib 1.7.1 Bugfix**: Prior releases of Passlib had a number of issues with their implementation of the PIX & ASA algorithms. As of 1.7.1, the reference vectors were greatly expanded, and then tested against an ASA 9.6 system. This revealed a number of errors in passlib's implementation, which under the following conditions would create hashes that were unverifiable on a Cisco system:

  – PIX and ASA: Usernames containing 1-3 characters were not appended correctly (step 2, above).

  – ASA omits the user entirely (step 2, above) for passwords with >= 28 characters, not >= 27. Non-enable passwords of exactly 27 characters were previous hashed incorrectly.

  – ASA's padding size decision (step 3, above) is made after the user has been appended, not before. This caused prior releases to incorrectly hash non-enable passwords of length 13-15.

  Anyone relying on cisco_asa or cisco_pix should upgrade to Passlib 1.7.1 or newer to avoid these issues.

3.7.7.3 `passlib.hash.cisco_asa` - Cisco ASA MD5 hash

Danger: This algorithm is not considered secure by modern standards. It should only be used when verifying existing hashes, or when interacting with applications that require this format. For new code, see the list of *recommended hashes*.

New in version 1.7.

Todo: Caveat Emptor

Passlib's implementations of *cisco_pix* and *cisco_asa* both need verification. For those with access to Cisco PIX and ASA systems, verifying Passlib's reference vectors would be a great help (see issue 51). In the mean time, there are no guarantees that passlib correctly replicates the official implementation.

Changed in version 1.7.1: A number of *bugs* were fixed after expanding the reference vectors, and testing against an ASA 9.6 system.

The `cisco_asa` class provides support for Cisco ASA "encrypted" hash format. This is a revision of the older `cisco_pix` hash; and the usage and format is the same.

See the *cisco_pix* documentation page for combined details of both these classes.

- *passlib.hash.cisco_pix* – PIX "encrypted" hashes use a simple unsalted MD5-based algorithm.
- *passlib.hash.cisco_asa* – ASA "encrypted" hashes use a similar algorithm to PIX, with some minor improvements.
- ASA "nt-encrypted" hashes are the same as *passlib.hash.nthash*, except that they use base64 encoding rather than hexadecimal.

[4] Character set used by ASA 8.4 - http://www.cisco.com/c/en/us/td/docs/security/asa/asa84/configuration/guide/asa_84_cli_config/ref_cli.html#Supported_Character_Sets

- ASA 9.5 added support for "pbkdf2" hashes (based on PBKDF2-HMAC-SHA512); which aren't currently supported by passlib (issue 87).

3.7.8 Other Hashes

The following schemes are used in various contexts, but have formats or uses which cannot be easily placed in one of the above categories:

3.7.8.1 passlib.hash.django_*digest* - Django-specific Hashes

The Django web framework provides a module for storing user accounts and passwords (`django.contrib.auth`). This module's password hashing code supports a few simple salted digests, stored using the format *`id$salt$checksum`* (where *`id`* is an identifier assigned by Django). Passlib provides support for all the hashes used up to and including Django 1.10.

Django 1.10 Hashes

Argon2

Django 1.10 added a wrapper for Argon2:

class passlib.hash.**django_argon2**
This class implements Django 1.10's Argon2 wrapper, and follows the *PasswordHash API*.

This is identical to `argon2` itself, but with the Django-specific prefix `"argon2"` prepended.

See *argon2* for more details, the usage and behavior is identical.

This should be compatible with the hashes generated by Django 1.10's `Argon2PasswordHasher` class.

New in version 1.7.

This hash has the exact same structure as *passlib.hash.argon2*, except that it has the prefix `argon2` added. For example, the django_argon2 hash...:

```
argon2$argon2i$v=19$m=256,t=1,p=1$c29tZXNhbHQ$AJFIsNZTMKTAewB4+ETN1A
```

...corresponds to the argon2 hash:

```
argon2i$v=19$m=256,t=1,p=1$c29tZXNhbHQ$AJFIsNZTMKTAewB4+ETN1A
```

Django 1.6 Hashes

Django 1.6 added one new hash:

Bcrypt SHA256

class passlib.hash.**django_bcrypt_sha256**(*ident=None*, ***kwds*)
This class implements Django 1.6's Bcrypt+SHA256 hash, and follows the *PasswordHash API*.

It supports a variable-length salt, and a variable number of rounds.

While the algorithm and format is somewhat different, the api and options for this hash are identical to `bcrypt` itself, see *bcrypt* for more details.

New in version 1.6.2.

This hash has the exact same structure as *bcrypt*, except that it has the prefix `bcrypt$` added. For example, the django_bcrypt_sha256 hash...:

```
bcrypt_sha256$$2a$06$/3OeRpbOf8/l6nPPRdZPp.nRiyYqPobEZGdNRBWihQhiFDh1ws1tu
```

...has the same structure as the bcrypt hash.:

```
$2a$06$/3OeRpbOf8/l6nPPRdZPp.nRiyYqPobEZGdNRBWihQhiFDh1ws1tu
```

That said, the hash is calculated slightly differently... the password is run through sha256(), the result encoded to lowercase hexadecimal, and then handed to bcrypt proper. This very similar to passlib's *bcrypt_sha256*, and addresses the same set of issues.

Django 1.4 Hashes

Django 1.4 introduced a new "hashers" framework, as well as three new modern large-salt variable-cost hash algorithms:

- *django_pbkdf2_sha256* - a PBKDF2-HMAC-SHA256 based hash.
- *django_pbkdf2_sha1* - a PBKDF2-HMAC-SHA1 based hash.
- *django_bcrypt* - a wrapper around *bcrypt*.

These classes can be used directly as follows:

```
>>> from passlib.hash import django_pbkdf2_sha256 as handler

>>> # hash password
>>> h = handler.hash("password")
>>> h
'pbkdf2_sha256$10000$s1w0UXDd00XB$+4ORmyvVWAQvoAEWlDgN34vlaJx1ZTZpa1pCSRey2Yk='

>>> # verify password
>>> handler.verify("password", h)
True
>>> handler.verify("eville", h)
False
```

See also:

the generic *PasswordHash usage examples*

Interface

class passlib.hash.**django_pbkdf2_sha256**

This class implements Django's PBKDF2-HMAC-SHA256 hash, and follows the *PasswordHash API*.

It supports a variable-length salt, and a variable number of rounds.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*str*) – Optional salt string. If not specified, a 12 character one will be autogenerated (this is recommended). If specified, may be any series of characters drawn from the regexp range `[0-9a-zA-Z]`.
- **salt_size** (*int*) – Optional number of characters to use when autogenerating new salts. Defaults to 12, but can be any positive value.
- **rounds** (*int*) – Optional number of rounds to use. Defaults to 29000, but must be within `range(1,1<<32)`.
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `rounds` that are too small or too large, and `salt` strings that are too long.

This should be compatible with the hashes generated by Django 1.4's `PBKDF2PasswordHasher` class.

New in version 1.6.

class passlib.hash.**django_pbkdf2_sha1**

This class implements Django's PBKDF2-HMAC-SHA1 hash, and follows the *PasswordHash API*.

It supports a variable-length salt, and a variable number of rounds.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*str*) – Optional salt string. If not specified, a 12 character one will be autogenerated (this is recommended). If specified, may be any series of characters drawn from the regexp range `[0-9a-zA-Z]`.
- **salt_size** (*int*) – Optional number of characters to use when autogenerating new salts. Defaults to 12, but can be any positive value.
- **rounds** (*int*) – Optional number of rounds to use. Defaults to 131000, but must be within `range(1,1<<32)`.
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `rounds` that are too small or too large, and `salt` strings that are too long.

This should be compatible with the hashes generated by Django 1.4's `PBKDF2SHA1PasswordHasher` class.

New in version 1.6.

passlib.hash.**django_bcrypt**

This class implements Django 1.4's BCrypt wrapper, and follows the *PasswordHash API*.

This is identical to `bcrypt` itself, but with the Django-specific prefix `"bcrypt$"` prepended. See *passlib.hash.bcrypt - BCrypt* for more details, the usage and behavior is identical.

This should be compatible with the hashes generated by Django 1.4's `BCryptPasswordHasher` class.

New in version 1.6.

Format

An example `django_pbkdf2_sha256` hash (of `password`) is:

```
pbkdf2_sha256$10000$s1w0UXDd00XB$+4ORmyvVWAQvoAEWlDgN34vlaJx1ZTZpa1pCSRey2Yk=
```

Both of Django's PBKDF2 hashes have the same basic format, *ident*$*rounds*$*salt*$*checksum*, where:

- *ident* is an identifier (pbkdf2_sha256 in the case of the example).
- *rounds* is a variable cost parameter encoded in decimal.
- *salt* consists of (usually 12) alphanumeric digits (s1w0UXDd00XB in the example).
- *checksum* is the base64 encoding the PBKDF2 digest.

The digest portion is generated by passing the utf-8 encoded password, the ascii-encoded salt string, and the number of rounds into PBKDF2 using the HMAC-SHA256 prf; and generated a 32 byte checksum, which is then encoding using base64.

The other PBKDF2 wrapper functions similarly.

Django 1.0 Hashes

Warning: All of the following hashes are very susceptible to brute-force attacks; since they are simple single-round salted digests. They should not be used for any purpose besides manipulating existing Django password hashes.

Django 1.0 supports some basic salted digests, as well as some legacy hashes:

- *django_salted_sha1* - simple salted SHA1 digest, Django 1.0-1.3's default.
- *django_salted_md5* - simple salted MD5 digest.
- *django_des_crypt* - support for legacy *des_crypt* hashes, shoehorned into Django's hash format.

These classes can be used directly as follows:

```
>>> from passlib.hash import django_salted_sha1 as handler

>>> # hash password
>>> h = handler.hash("password")
>>> h
'sha1$c6218$161d1ac8ab38979c5a31cbaba4a67378e7e60845'

>>> # verify password
>>> handler.verify("password", h)
True
>>> handler.verify("eville", h)
False
```

See also:

the generic *PasswordHash usage examples*

Interface

class passlib.hash.**django_salted_md5**

This class implements Django's Salted MD5 hash, and follows the *PasswordHash API*.

It supports a variable-length salt, and uses a single round of MD5.

The *hash()* and *genconfig()* methods accept the following optional keywords:

Parameters

- **salt** (*str*) – Optional salt string. If not specified, a 12 character one will be autogenerated (this is recommended). If specified, may be any series of characters drawn from the regexp range `[0-9a-zA-Z]`.
- **salt_size** (*int*) – Optional number of characters to use when autogenerating new salts. Defaults to 12, but can be any positive value.

This should be compatible with the hashes generated by Django 1.4's `MD5PasswordHasher` class.

class passlib.hash.**django_salted_sha1**

This class implements Django's Salted SHA1 hash, and follows the *PasswordHash API*.

It supports a variable-length salt, and uses a single round of SHA1.

The *hash()* and *genconfig()* methods accept the following optional keywords:

Parameters

- **salt** (*str*) – Optional salt string. If not specified, a 12 character one will be autogenerated (this is recommended). If specified, may be any series of characters drawn from the regexp range `[0-9a-zA-Z]`.
- **salt_size** (*int*) – Optional number of characters to use when autogenerating new salts. Defaults to 12, but can be any positive value.

This should be compatible with Django 1.4's `SHA1PasswordHasher` class.

Format

An example `django_salted_sha1` hash (of `password`) is:

```
sha1$f8793$c4cd18eb02375a037885706d414d68d521ca18c7
```

Both of Django's salted hashes have the same basic format, *ident*`$`*salt*`$`*checksum*, where:

- *ident* is an identifier (`sha1` in the case of the example, `md5` for `django_salted_md5`).
- *salt* consists of (usually 5) lowercase hexadecimal digits (`f8793` in the example).
- *checksum* is lowercase hexadecimal encoding of the checksum.

The checksum is generated by concatenating the salt digits followed by the password, and hashing them using the specified digest (MD5 or SHA-1). The digest is then encoded to hexadecimal. If the password is unicode, it is converted to `utf-8` first.

Security Issues

Django's salted hashes should not be considered very secure.

- They use only a single round of digests with known collision and pre-image attacks (SHA1 & MD5).
- While it could be increased, they currently use only 20 bits of entropy in their salt, which is borderline insufficient to defeat rainbow tables.
- They digest the encoded hexadecimal salt, not the raw bytes, increasing the odds that a particular salt+password string will be present in a pre-computed tables of ascii digests.

Des Crypt Wrapper

class passlib.hash.**django_des_crypt**

This class implements Django's *des_crypt* wrapper, and follows the *PasswordHash API*.

It supports a fixed-length salt.

The *hash()* and *genconfig()* methods accept the following optional keywords:

Parameters

- **salt** (*str*) – Optional salt string. If not specified, one will be autogenerated (this is recommended). If specified, it must be 2 characters, drawn from the regexp range `[./0-9A-Za-z]`.
- **truncate_error** (*bool*) – By default, django_des_crypt will silently truncate passwords larger than 8 bytes. Setting `truncate_error=True` will cause *hash()* to raise a *PasswordTruncateError* instead.

  New in version 1.7.

This should be compatible with the hashes generated by Django 1.4's `CryptPasswordHasher` class. Note that Django only supports this hash on Unix systems (though `django_des_crypt` is available cross-platform under Passlib).

Changed in version 1.6: This class will now accept hashes with empty salt strings, since Django 1.4 generates them this way.

Format

An example `django_des_crypt` hash (of `password`) is `crypt$cd1a4$cdlRbNJGImptk`; the general format is the same as the salted hashes: *ident*`$`*salt*`$`*checksum*, where:

- *ident* is the identifier `crypt`.
- *salt* is 5 lowercase hexadecimal digits (`cd1a4` in the example).
- *checksum* is a `des_crypt` hash (`cdlRbNJGImptk` in the example).

It should be noted that this class essentially just shoe-horns *des_crypt* into a format compatible with the Django salted hashes (above). It has a few quirks, such as the fact that only the first two characters of the salt are used by `des_crypt`, and they are in turn duplicated as the first two characters of the checksum.

For security issues relating to `django_des_crypt`, see *des_crypt*.

Other Hashes

class passlib.hash.**django_disabled**

This class provides disabled password behavior for Django, and follows the *PasswordHash API*.

This class does not implement a hash, but instead claims the special hash string `"!"` which Django uses to indicate an account's password has been disabled.

- newly encrypted passwords will hash to `"!"`.
- it rejects all passwords.

Note: Django 1.6 prepends a randomly generated 40-char alphanumeric string to each unusuable password. This class recognizes such strings, but for backwards compatibility, still returns `"!"`.

See https://code.djangoproject.com/ticket/20079 for why Django appends an alphanumeric string.

Changed in version 1.6.2: added Django 1.6 support

Changed in version 1.7: started appending an alphanumeric string.

Note: Some older (pre-1.0) versions of Django encoded passwords using *hex_md5*, though this has been deprecated by Django, and should become increasingly rare.

3.7.8.2 passlib.hash.grub_pbkdf2_sha512 - Grub's PBKDF2 Hash

This class provides an implementation of Grub's PBKDF2-HMAC-SHA512 password hash[1], as generated by the **grub-mkpasswd-pbkdf2** command, and may be found in Grub2 configuration files. PBKDF2 is a key derivation function[2] that is ideally suited as the basis for a password hash, as it provides variable length salts, variable number of rounds.

See also:

- *password hash usage* – for examples of how to use this class via the common hash interface.
- *passlib.hash.pbkdf2_{digest}* – for some other PBKDF2-based hashes.

Interface

class passlib.hash.**grub_pbkdf2_sha512**

This class implements Grub's pbkdf2-hmac-sha512 hash, and follows the *PasswordHash API*.

It supports a variable-length salt, and a variable number of rounds.

The *using()* method accepts the following optional keywords:

Parameters

- **salt** (*bytes*) – Optional salt bytes. If specified, the length must be between 0-1024 bytes. If not specified, a 64 byte salt will be autogenerated (this is recommended).
- **salt_size** (*int*) – Optional number of bytes to use when autogenerating new salts. Defaults to 64 bytes, but can be any value between 0 and 1024.
- **rounds** (*int*) – Optional number of rounds to use. Defaults to 19000, but must be within `range(1,1<<32)`.
- **relaxed** (*bool*) – By default, providing an invalid value for one of the other keywords will result in a `ValueError`. If `relaxed=True`, and the error can be corrected, a *PasslibHashWarning* will be issued instead. Correctable errors include `rounds` that are too small or too large, and `salt` strings that are too long.

  New in version 1.6.

Format & Algorithm

A example hash (of `password`) is

[1] Information about Grub's password hashes - http://grub.enbug.org/Authentication.
[2] The specification for the PBKDF2 algorithm - http://tools.ietf.org/html/rfc2898#section-5.2.

```
grub.pbkdf2.sha512.10000.4483972AD2C52E1F590B3E2260795FDA9CA0B07B
96FF492814CA9775F08C4B59CD1707F10B269E09B61B1E2D11729BCA8D62B7827
B25B093EC58C4C1EAC23137.DF4FCB5DD91340D6D31E33423E4210AD47C7A4DF9
FA16F401663BF288C20BF973530866178FE6D134256E4DBEFBD984B652332EED3
ACAED834FEA7B73CAE851D
```

All of this scheme's hashes have the format `grub.pbkdf2.sha512.`*`rounds`*`.`*`salt`*`.`*`checksum`*, where *`rounds`* is the number of iteration stored in decimal, *`salt`* is the salt string encoded using upper-case hexadecimal, and *`checksum`* is the resulting 64-byte derived key, also encoded in upper-case hexadecimal. It can be identified by the prefix `grub.pdkdf2.sha512.`.

The algorithm used is the same as *pbkdf2_sha1*: the password is encoded into UTF-8 if not already encoded, and passed through *pbkdf1()* along with the decoded salt, and the number of rounds. The result is then encoded into hexadecimal.

3.7.8.3 passlib.hash.hex_*digest* - Generic Hexadecimal Digests

Danger: Using a single round of any cryptographic hash (especially without a salt) is so insecure that it's barely better than plaintext. Do not use these schemes in new applications.

Some existing applications store passwords by storing them using hexadecimal-encoded message digests, such as MD5 or SHA1. Such schemes are *extremely* vulnerable to pre-computed brute-force attacks, and should not be used in new applications. However, for the sake of backwards compatibility when converting existing applications, Passlib provides wrappers for few of the common hashes. These classes all wrap the underlying hashlib implementations, and can be used directly as follows:

```
>>> from passlib.hash import hex_sha1 as hex_sha1

>>> # hash password
>>> h = hex_sha1.hash("password")
>>> h
'5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8'

>>> # verify correct password
>>> hex_sha1.verify("password", h)
True

>>> # verify incorrect password
>>> hex_sha1.verify("secret", h)
False
```

See also:

the generic *PasswordHash usage examples*

Interface

class passlib.hash.**hex_md4**

class passlib.hash.**hex_md5**

class passlib.hash.**hex_sha1**

class passlib.hash.**hex_sha256**

class passlib.hash.**hex_sha512**

Each of these classes implements a plain hexadecimal encoded message digest, using the relevant digest function from `hashlib`, and following the *PasswordHash API*.

They support no settings or other keywords.

Note: Oracle VirtualBox's **`VBoxManager internalcommands passwordhash`** command uses *hex_sha256*.

Format & Algorithm

All of these classes just report the result of the specified digest, encoded as a series of lowercase hexadecimal characters; though upper case is accepted as input.

3.7.8.4 `passlib.hash.plaintext` - Plaintext

This class stores passwords in plaintext. This is, of course, ridiculously insecure; it is provided for backwards compatibility when migrating existing applications. *It should not be used* for any other purpose. This class should always be the last algorithm checked, as it will recognize all hashes. It can be used directly as follows:

```
>>> from passlib.hash import plaintext as plaintext

>>> # "encrypt" password
>>> plaintext.hash("password")
'password'

>>> # verify password
>>> plaintext.verify("password", "password")
True
>>> plaintext.verify("secret", "password")
False
```

See also:

- *password hash usage* – for more usage examples
- *ldap_plaintext* – on LDAP systems, this format is probably more appropriate for storing plaintext passwords.

Interface

class passlib.hash.**plaintext**

This class stores passwords in plaintext, and follows the *PasswordHash API*.

The *hash()*, *genhash()*, and *verify()* methods all require the following additional contextual keyword:

Parameters **`encoding`** (*str*) – This controls the character encoding to use (defaults to `utf-8`).

This encoding will be used to encode `unicode` passwords under Python 2, and decode `bytes` hashes under Python 3.

Changed in version 1.6: The `encoding` keyword was added.

3.8 passlib.hosts - OS Password Handling

This module provides some preconfigured *CryptContext* instances for hashing & verifying password hashes tied to user accounts of various operating systems. While (most) of the objects are available cross-platform, their use is oriented primarily towards Linux and BSD variants.

See also:

for Microsoft Windows, see the list of *MS Windows Hashes* in *passlib.hash*.

3.8.1 Usage Example

The CryptContext class itself has a large number of features, but to give an example of how to quickly use the instances in this module:

Each of the objects in this module can be imported directly:

```
>>> # as an example, this imports the linux_context object,
>>> # which is configured to recognized most hashes found in Linux /etc/shadow files.
>>> from passlib.apps import linux_context
```

Hashing a password is simple (and salt generation is handled automatically):

```
>>> hash = linux_context.hash("toomanysecrets")
>>> hash
'$5$rounds=84740$fYChCy.52EzebF51$9bnJrmTf2FESI93hgIBFF4qAfysQcKoB0veiI0ZeYU4'
```

Verifying a password against an existing hash is just as quick:

```
>>> linux_context.verify("toomanysocks", hash)
False
>>> linux_context.verify("toomanysecrets", hash)
True
```

You can also identify hashes::

```
>>> linux_context.identify(hash)
'sha512_crypt'
```

Or encrypt using a specific algorithm::

```
>>> linux_context.schemes()
('sha512_crypt', 'sha256_crypt', 'md5_crypt', 'des_crypt', 'unix_disabled')
>>> linux_context.hash("password", scheme="des_crypt")
'2fmLLcoHXuQdI'
>>> linux_context.identify('2fmLLcoHXuQdI')
'des_crypt'
```

See also:

the *CryptContext Tutorial* and *CryptContext Reference* for more information about the CryptContext class.

3.8.2 Unix Password Hashes

Passlib provides a number of pre-configured CryptContext instances which can identify and manipulate all the formats used by Linux and BSD. See the *modular crypt identifier list* for a complete list of which hashes are supported

by which operating system.

3.8.2.1 Predefined Contexts

Passlib provides `CryptContext` instances for the following Unix variants:

`passlib.hosts.`**`linux_context`**
: context instance which recognizes hashes used by the majority of Linux distributions. encryption defaults to `sha512_crypt`.

`passlib.hosts.`**`freebsd_context`**
: context instance which recognizes all hashes used by FreeBSD 8. encryption defaults to `bcrypt`.

`passlib.hosts.`**`netbsd_context`**
: context instance which recognizes all hashes used by NetBSD. encryption defaults to `bcrypt`.

`passlib.hosts.`**`openbsd_context`**
: context instance which recognizes all hashes used by OpenBSD. encryption defaults to `bcrypt`.

Note: All of the above contexts include the *unix_disabled* handler as a final fallback. This special handler treats all strings as invalid passwords, particularly the common strings `!` and `*` which are used to indicate that an account has been disabled[1].

3.8.2.2 Current Host OS

`passlib.hosts.`**`host_context`**

Platform Unix

This *CryptContext* instance should detect and support all the algorithms the native OS `crypt()` offers. The main differences between this object and `crypt()`:

- this object provides introspection about *which* schemes are available on a given system (via `host_context.schemes()`).
- it defaults to the strongest algorithm available, automatically configured to an appropriate strength for hashing new passwords.
- whereas `crypt()` typically defaults to using *des_crypt*; and provides little introspection.

As an example, this can be used in conjunction with stdlib's `spwd` module to verify user passwords on the local system:

```
>>> # NOTE/WARNING: this example requires running as root on most systems.
>>> import spwd, os
>>> from passlib.hosts import host_context
>>> hash = spwd.getspnam(os.environ['USER']).sp_pwd
>>> host_context.verify("toomanysecrets", hash)
True
```

Changed in version 1.4: This object is only available on systems where the stdlib `crypt` module is present. In version 1.3 and earlier, it was available on non-Unix systems, though it did nothing useful.

[1] Man page for Linux /etc/shadow - http://linux.die.net/man/5/shadow

3.9 passlib.ifc – Password Hash Interface

3.9.1 PasswordHash API

This module provides the `PasswordHash` abstract base class. This class defines the common methods and attributes present on all the hashes importable from the *passlib.hash* module. Additionally, the *passlib.context.CryptContext* class is deliberately designed to parallel many of this interface's methods.

See also:

PasswordHash Tutorial – Overview of this interface and how to use it.

3.9.2 Base Abstract Class

class passlib.ifc.**PasswordHash**

This class provides an abstract interface for an arbitrary password hasher.

Applications will generally not construct instances directly – most of the operations are performed via class-methods, allowing instances of a given class to be an internal detail used to implement the various operations.

While `PasswordHash` offers a number of methods and attributes, most applications will only need the two primary methods:

- *PasswordHash.hash()* - generate new salt, return hash of password.
- *PasswordHash.verify()* - verify password against existing hash.

Two additional support methods are also provided:

- *PasswordHash.using()* - create subclass with customized configuration.
- *PasswordHash.identify()* - check if hash belongs to this algorithm.

Each hash algorithm also provides a number of *informational attributes*, allowing programmatic inspection of its options and parameter limits.

See also:

PasswordHash Tutorial – Overview of this interface and how to use it.

3.9.3 Hashing & Verification Methods

Most applications will only need to use two methods: *PasswordHash.hash()* to generate new hashes, and *PasswordHash.verify()* to check passwords against existing hashes. These methods provide an easy interface for working with a password hash, and abstract away details such as salt generation, hash normalization, and hash comparison.

classmethod PasswordHash.**hash**(*secret*, ***kwds*)

Digest password using format-specific algorithm, returning resulting hash string.

For most hashes supported by Passlib, the returned string will contain: an algorithm identifier, a cost parameter, the salt string, and finally the password digest itself.

Parameters

- **secret** (*unicode or bytes*) – string containing the password to encode.

- ****kwds** – All additional keywords are algorithm-specific, and will be listed in that hash's documentation; though many of the more common keywords are listed under *PasswordHash.setting_kwds* and *PasswordHash.context_kwds*.

  Deprecated since version 1.7: Passing *PasswordHash.setting_kwds* such as `rounds` and `salt_size` directly into the *hash()* method is deprecated. Callers should instead use `handler.using(**settings).hash(secret)`. Support for the old method is is tentatively scheduled for removal in Passlib 2.0.

  Context keywords such as `user` should still be provided to `hash()`.

Returns Resulting password hash, encoded in an algorithm-specific format. This will always be an instance of `str` (i.e. `unicode` under Python 3, `ascii`-encoded `bytes` under Python 2).

Raises

- **ValueError** –
  - If a `kwd`'s value is invalid (e.g. if a `salt` string is too small, or a `rounds` value is out of range).
  - If `secret` contains characters forbidden by the hash algorithm (e.g. `des_crypt` forbids NULL characters).
- **TypeError** –
  - if `secret` is not `unicode` or `bytes`.
  - if a `kwd` argument has an incorrect type.
  - if an algorithm-specific required `kwd` is not provided.

Changed in version 1.6: Hashes now raise `TypeError` if a required keyword is missing, rather than `ValueError` like in previous releases; in order to conform with normal Python behavior.

Changed in version 1.6: Passlib is now much stricter about input validation: for example, out-of-range `rounds` values now cause an error instead of being clipped (though applications may set *relaxed=True* to restore the old behavior).

Changed in version 1.7: This method was renamed from *encrypt()*. Deprecated support for passing settings directly into `hash()`.

classmethod PasswordHash.**encrypt**(*secret*, ***kwds*)

Legacy alias for *PasswordHash.hash()*.

Deprecated since version 1.7: This method was renamed to `hash()` in version 1.7. This alias will be removed in version 2.0, and should only be used for compatibility with Passlib 1.3 - 1.6.

classmethod PasswordHash.**verify**(*secret*, *hash*, ***context_kwds*)

Verify a secret using an existing hash.

This checks if a secret matches against the one stored inside the specified hash.

Parameters

- **secret** (*unicode or bytes*) – A string containing the password to check.
- **hash** – A string containing the hash to check against, such as returned by *PasswordHash.hash()*.

  Hashes may be specified as `unicode` or `ascii`-encoded `bytes`.
- ****kwds** – Very few hashes will have additional keywords.

  The ones that do typically require external contextual information in order to calculate the digest. For these hashes, the values must match the ones passed to the original

PasswordHash.hash() call when the hash was generated, or the password will not verify.

These additional keywords are algorithm-specific, and will be listed in that hash's documentation; though the more common keywords are listed under *PasswordHash.context_kwds*. Examples of common keywords include `user`.

Returns `True` if the secret matches, otherwise `False`.

Raises

- **TypeError** –
  - if either `secret` or `hash` is not a unicode or bytes instance.
  - if the hash requires additional `kwds` which are not provided,
  - if a `kwd` argument has the wrong type.
- **ValueError** –
  - if `hash` does not match this algorithm's format.
  - if the `secret` contains forbidden characters (see *PasswordHash.hash()*).
  - if a configuration/salt string generated by *PasswordHash.genconfig()* is passed in as the value for `hash` (these strings look similar to a full hash, but typically lack the digest portion needed to verify a password).

Changed in version 1.6: This function now raises `ValueError` if `None` or a config string is provided instead of a properly-formed hash; previous releases were inconsistent in their handling of these two border cases.

See also:

- *Hashing & Verifying* tutorial for a usage example

3.9.4 Crypt Methods

Taken together, the *PasswordHash.genconfig()* and *PasswordHash.genhash()* are two tightly-coupled methods that mimic the standard Unix "crypt" interface. The first method generates salt / configuration strings from a set of settings, and the second hashes the password using the provided configuration string.

See also:

Most applications will find *PasswordHash.hash()* much more useful, as it combines the functionality of these two methods into one.

classmethod PasswordHash.**genconfig**(***setting_kwds*)

Deprecated since version 1.7: As of 1.7, this method is deprecated, and slated for complete removal in Passlib 2.0.

For all known real-world uses, `.hash("", **settings)` should provide equivalent functionality.

This deprecation may be reversed if a use-case presents itself in the mean time.

Returns a configuration string encoding settings for hash generation.

This function takes in all the same *PasswordHash.setting_kwds* as *PasswordHash.hash()*, fills in suitable defaults, and encodes the settings into a single "configuration" string, suitable passing to *PasswordHash.genhash()*.

Parameters ****kwds** – All additional keywords are algorithm-specific, and will be listed in that hash's documentation; though many of the more common keywords are listed under *PasswordHash.setting_kwds* Examples of common keywords include `salt` and `rounds`.

Returns A configuration string (as `str`).

Raises **ValueError, TypeError** – This function raises exceptions for the same reasons as *PasswordHash.hash()*.

Changed in version 1.7: This should now always return a full hash string, even in cases where previous releases would return a truncated "configuration only" string, or `None`.

classmethod PasswordHash.**genhash**(*secret*, *config*, ***context_kwds*)

Encrypt secret using specified configuration string.

Deprecated since version 1.7: As of 1.7, this method is deprecated, and slated for complete removal in Passlib 2.0.

This deprecation may be reversed if a use-case presents itself in the mean time.

This takes in a password and a configuration string, and returns a hash for that password.

Parameters

- **secret** (*unicode or bytes*) – string containing the password to be encrypted.
- **config** (*unicode or bytes*) – configuration string to use when hashing the secret. this can either be an existing hash that was previously returned by *PasswordHash.genhash()*, or a configuration string that was previously created by *PasswordHash.genconfig()*.

  Changed in version 1.7: `None` is no longer accepted for hashes which (prior to 1.7) lacked a configuration string format.
- ****kwds** – Very few hashes will have additional keywords.

  The ones that do typically require external contextual information in order to calculate the digest. For these hashes, the values must match the ones passed to the original *PasswordHash.hash()* call when the hash was generated, or the password will not verify.

  These additional keywords are algorithm-specific, and will be listed in that hash's documentation; though the more common keywords are listed under :*PasswordHash.context_kwds*. Examples of common keywords include `user`.

Returns Encoded hash matching specified secret, config, and kwds. This will always be a native `str` instance.

Raises **ValueError, TypeError** – This function raises exceptions for the same reasons as *PasswordHash.hash()*.

Warning: Traditionally, password verification using the "crypt" interface was done by testing if `hash == genhash(password, hash)`. This test is only reliable for a handful of algorithms, as various hash representation issues may cause false results. Applications are strongly urged to use *verify()* instead.

3.9.5 Factory Creation

One powerful method offered by the `PasswordHash` class *PasswordHash.using()*. This method allows you to quickly create subclasses of a specific hash, providing it with preconfigured defaults specific to your application:

classmethod PasswordHash.**using**(*relaxed=False*, ***settings*)
This method takes in a set of algorithm-specific settings, and returns a new handler object which uses the specified default settings instead.

Parameters ****settings** – All keywords are algorithm-specific, and will be listed in that hash's documentation; though many of the more common keywords are listed under *PasswordHash.setting_kwds*. Examples of common keywords include `rounds` and `salt_size`.

Returns A new object which adheres to `PasswordHash` api.

Raises

- **ValueError** –
  - If a keywords's value is invalid (e.g. if a `salt` string is too small, or a `rounds` value is out of range).
- **TypeError** –
  - if a `kwd` argument has an incorrect type.

New in version 1.7.

See also:

Customizing the Configuration tutorial for a usage example

3.9.6 Hash Inspection Methods

There are currently two hash inspection methods, *PasswordHash.identify()* and *PasswordHash.needs_update()*.

classmethod PasswordHash.**identify**(*hash*)
Quickly identify if a hash string belongs to this algorithm.

Parameters **hash** (*unicode or bytes*) – the candidate hash string to check

Returns

- **True if the input is a configuration string or hash string** identifiable as belonging to this scheme (even if it's malformed).
- `False` if the input does not belong to this scheme.

Raises **TypeError** – if *hash* is not a unicode or bytes instance.

Note: A small number of the hashes supported by Passlib lack a reliable method of identification (e.g. *lmhash* and *nthash* both consist of 32 hexadecimal characters, with no distinguishing features). For such hashes, this method may return false positives.

See also:

If you are considering using this method to select from multiple algorithms (e.g. in order to verify a password), you will be better served by the *CryptContext* class.

classmethod PasswordHash.**needs_update**(*hash*, *secret=None*)
check if hash's configuration is outside desired bounds, or contains some other internal option which requires updating the password hash.

Parameters

- **hash** – hash string to examine
- **secret** – optional secret known to have verified against the provided hash. (this is used by some hashes to detect legacy algorithm mistakes).

Returns whether secret needs re-hashing.

New in version 1.7.

3.9.7 General Informational Attributes

Each hash provides a handful of informational attributes, allowing programs to dynamically adapt to the requirements of different hash algorithms. The following attributes should be defined for all the hashes in passlib:

`PasswordHash.name`
Name uniquely identifying this hash.

For the hashes built into Passlib, this will always match the location where it was imported from — `passlib.hash.name` — though externally defined hashes may not adhere to this.

This should always be a `str` consisting of lowercase `a-z`, the digits `0-9`, and the underscore character _.

`PasswordHash.setting_kwds`
Tuple listing the keywords supported by *PasswordHash.using()* control hash generation, and which will be encoded into the resulting hash.

(These keywords will also be accepted by *PasswordHash.hash()* and *PasswordHash.genconfig()*, though that behavior is deprecated as of Passlib 1.7; and will be removed in Passlib 2.0).

This list commonly includes keywords for controlling salt generation, adjusting time-cost parameters, etc. Most of these settings are optional, and suitable defaults will be chosen if they are omitted (e.g. salts will be autogenerated).

While the documentation for each hash should have a complete list of the specific settings the hash uses, the following keywords should have roughly the same behavior for all the hashes that support them:

salt Specifies a fixed salt string to use, rather than randomly generating one.

This option is supported by most of the hashes in Passlib, though typically it isn't used, as random generation of a salt is usually the desired behavior.

Hashes typically require this to be a `unicode` or `bytes` instance, with additional constraints appropriate to the algorithm.

`salt_size`

Most algorithms which support the `salt` setting will autogenerate a salt when none is provided. Most of those hashes will also offer this option, which allows the caller to specify the size of salt which should be generated. If omitted, the hash's default salt size will be used.

See also:

the *salt info* attributes (below)

rounds If present, this means the hash can vary the number of internal rounds used in some part of its algorithm, allowing the calculation to take a variable amount of processor time, for increased security.

While this is almost always a non-negative integer, additional constraints may be present for each algorithm (such as the cost varying on a linear or logarithmic scale).

This value is typically omitted, in which case a default value will be used. The defaults for all the hashes in Passlib are periodically retuned to strike a balance between security and responsiveness.

See also:

the *rounds info* attributes (below)

ident If present, the class supports multiple formats for encoding the same hash. The class's documentation will generally list the allowed values, allowing alternate output formats to be selected.

Note that these values will typically correspond to different revision of the hash algorithm itself, and they may not all offer the same level of security.

truncate_error

This will be present if and only if the hash truncates passwords larger than some limit (reported via it's *truncate_size* attribute). By default, they will silently truncate passwords above their limit. Setting truncate_error=True will cause *PasswordHash.hash()* to raise a *PasswordTruncateError* instead.

relaxed By default, passing an invalid value to *PasswordHash.using()* will result in a ValueError. However, if relaxed=True then Passlib will attempt to correct the error and (if successful) issue a *PasslibHashWarning* instead. This warning may then be filtered if desired. Correctable errors include (but are not limited to): rounds and salt_size values that are too low or too high, salt strings that are too large.

New in version 1.6.

PasswordHash.**context_kwds**

Tuple listing the keywords supported by *PasswordHash.hash()*, *PasswordHash.verify()*, and *PasswordHash.genhash()*. These keywords are different from the settings kwds in that the context keywords affect the hash, but are not encoded within it, and thus must be provided each time the hash is calculated.

This list commonly includes a user account, http realm identifier, etc. Most of these keywords are required by the hashes which support them, as they are frequently used in place of an embedded salt parameter.

Most hash algorithms in Passlib will have no context keywords.

While the documentation for each hash should have a complete list of the specific context keywords the hash uses, the following keywords should have roughly the same behavior for all the hashes that support them:

user

If present, the class requires a username be specified whenever performing a hash calculation (e.g. *postgres_md5* and *oracle10*).

encoding

Some hashes have poorly-defined or host-dependant unicode behavior, and properly hashing a non-ASCII password requires providing the correct encoding (*lmhash* is perhaps the worst offender). Hashes which provide this keyword will always expose their default encoding programmatically via the PasswordHash.default_encoding attribute.

passlib.ifc.**truncate_size**

A positive integer, indicating the hash will truncate any passwords larger than this many bytes. If None (the more common case), indicates the hash will use the entire password provided.

Hashes which specify this setting will also support a truncate_error flag via their *PasswordHash.using()* method, to configure how truncation is handled.

See also:

Customizing the Configuration tutorial for a usage example

3.9.8 Salt Information Attributes

For schemes which support a salt string, `"salt"` should be listed in their *PasswordHash.setting_kwds*, and the following attributes should be defined:

`PasswordHash.`**`max_salt_size`**
: The maximum number of bytes/characters allowed in the salt. Should either be a positive integer, or `None` (indicating the algorithm has no effective upper limit).

`PasswordHash.`**`min_salt_size`**
: The minimum number of bytes/characters required for the salt. Must be an integer between 0 and *PasswordHash.max_salt_size*.

`PasswordHash.`**`default_salt_size`**
: The default salt size that will be used when generating a salt, assuming `salt_size` is not set explicitly. This is typically the same as *max_salt_size*, or a sane default if `max_salt_size=None`.

`PasswordHash.`**`salt_chars`**
: A unicode string containing all the characters permitted in a salt string.

  For most *Modular Crypt Format* hashes, this is equal to *passlib.utils.binary.HASH64_CHARS*. For the rare hashes where the `salt` parameter must be specified in bytes, this will be a placeholder `bytes` object containing all 256 possible byte values.

3.9.9 Rounds Information Attributes

For schemes which support a variable time-cost parameter, `"rounds"` should be listed in their *PasswordHash.setting_kwds*, and the following attributes should be defined:

`PasswordHash.`**`max_rounds`**
: The maximum number of rounds the scheme allows. Specifying a value beyond this will result in a `ValueError`. This will be either a positive integer, or `None` (indicating the algorithm has no effective upper limit).

`PasswordHash.`**`min_rounds`**
: The minimum number of rounds the scheme allows. Specifying a value below this will result in a `ValueError`. Will always be an integer between 0 and *PasswordHash.max_rounds*.

`PasswordHash.`**`default_rounds`**
: The default number of rounds that will be used if none is explicitly provided to *PasswordHash.hash()*. This will always be an integer between *PasswordHash.min_rounds* and *PasswordHash.max_rounds*.

`PasswordHash.`**`rounds_cost`**
: While the cost parameter `rounds` is an integer, how it corresponds to the amount of time taken can vary between hashes. This attribute indicates the scale used by the hash:

  - `"linear"` - time taken scales linearly with rounds value (e.g. *sha512_crypt*)
  - `"log2"` - time taken scales exponentially with rounds value (e.g. *bcrypt*)

Todo: document the additional *PasswordHash.using()* keywords available for setting rounds limits.

3.10 passlib.pwd – Password generation helpers

New in version 1.7.

3.10.1 Password Generation

Warning: Before using these routines, make sure your system's RNG entropy pool is secure and full. Also make sure that `genword()` or `genphrase()` is called with a sufficiently high `entropy` parameter the intended purpose of the password.

passlib.pwd.**genword**(*entropy=None*, *length=None*, *charset="ascii_62"*, *chars=None*, *returns=None*)

Generate one or more random passwords.

This function uses `random.SystemRandom` to generate one or more passwords using various character sets. The complexity of the password can be specified by size, or by the desired amount of entropy.

Usage Example:

```
>>> # generate a random alphanumeric string with 48 bits of entropy (the default)
>>> from passlib import pwd
>>> pwd.genword()
'DnBHvDjMK6'

>>> # generate a random hexadecimal string with 52 bits of entropy
>>> pwd.genword(entropy=52, charset="hex")
'310f1a7ac793f'
```

Parameters

- **entropy** – Strength of resulting password, measured in 'guessing entropy' bits. An appropriate **length** value will be calculated based on the requested entropy amount, and the size of the character set.

  This can be a positive integer, or one of the following preset strings: `"weak"` (24), `"fair"` (36), `"strong"` (48), and `"secure"` (56).

  If neither this or **length** is specified, **entropy** will default to `"strong"` (48).

- **length** – Size of resulting password, measured in characters. If omitted, the size is auto-calculated based on the **entropy** parameter.

  If both **entropy** and **length** are specified, the stronger value will be used.

- **returns** – Controls what this function returns:

  - If `None` (the default), this function will generate a single password.
  - If an integer, this function will return a list containing that many passwords.
  - If the `iter` constant, will return an iterator that yields passwords.

- **chars** – Optionally specify custom string of characters to use when randomly generating a password. This option cannot be combined with **charset**.

- **charset** – The predefined character set to draw from (if not specified by **chars**). There are currently four presets available:

- `"ascii_62"` (the default) – all digits and ascii upper & lowercase letters. Provides ~5.95 entropy per character.
- `"ascii_50"` – subset which excludes visually similar characters (`1IiLl0Oo5S8B`). Provides ~5.64 entropy per character.
- `"ascii_72"` – all digits and ascii upper & lowercase letters, as well as some punctuation. Provides ~6.17 entropy per character.
- `"hex"` – Lower case hexadecimal. Providers 4 bits of entropy per character.

Returns `unicode` string containing randomly generated password; or list of 1+ passwords if `returns=`*`int`* is specified.

`passlib.pwd.`**`genphrase`**(*entropy=None*, *length=None*, *wordset="eff_long"*, *words=None*, *sep=" "*, *returns=None*)

Generate one or more random password / passphrases.

This function uses `random.SystemRandom` to generate one or more passwords; it can be configured to generate alphanumeric passwords, or full english phrases. The complexity of the password can be specified by size, or by the desired amount of entropy.

Usage Example:

```
>>> # generate random phrase with 48 bits of entropy
>>> from passlib import pwd
>>> pwd.genphrase()
'gangly robbing salt shove'

>>> # generate a random phrase with 52 bits of entropy
>>> # using a particular wordset
>>> pwd.genword(entropy=52, wordset="bip39")
'wheat dilemma reward rescue diary'
```

Parameters

- **`entropy`** – Strength of resulting password, measured in 'guessing entropy' bits. An appropriate **length** value will be calculated based on the requested entropy amount, and the size of the word set.

  This can be a positive integer, or one of the following preset strings: `"weak"` (24), `"fair"` (36), `"strong"` (48), and `"secure"` (56).

  If neither this or **length** is specified, **entropy** will default to `"strong"` (48).

- **`length`** – Length of resulting password, measured in words. If omitted, the size is auto-calculated based on the **entropy** parameter.

  If both **entropy** and **length** are specified, the stronger value will be used.

- **`returns`** – Controls what this function returns:
  - If `None` (the default), this function will generate a single password.
  - If an integer, this function will return a list containing that many passwords.
  - If the `iter` builtin, will return an iterator that yields passwords.
- **`words`** – Optionally specifies a list/set of words to use when randomly generating a passphrase. This option cannot be combined with **wordset**.
- **`wordset`** – The predefined word set to draw from (if not specified by **words**). There are currently four presets available:

`"eff_long"` (the default)

Wordset containing 7776 english words of ~7 letters. Constructed by the EFF, it offers ~12.9 bits of entropy per word.

This wordset (and the other `"eff_"` wordsets) were created by the EFF to aid in generating passwords. See their announcement page for more details about the design & properties of these wordsets.

`"eff_short"`

Wordset containing 1296 english words of ~4.5 letters. Constructed by the EFF, it offers ~10.3 bits of entropy per word.

`"eff_prefixed"`

Wordset containing 1296 english words of ~8 letters, selected so that they each have a unique 3-character prefix. Constructed by the EFF, it offers ~10.3 bits of entropy per word.

`"bip39"`

Wordset of 2048 english words of ~5 letters, selected so that they each have a unique 4-character prefix. Published as part of Bitcoin's BIP 39, this wordset has exactly 11 bits of entropy per word.

This list offers words that are typically shorter than `"eff_long"` (at the cost of slightly less entropy); and much shorter than `"eff_prefixed"` (at the cost of a longer unique prefix).

- **sep** – Optional separator to use when joining words. Defaults to `" "` (a space), but can be an empty string, a hyphen, etc.

Returns `unicode` string containing randomly generated passphrase; or list of 1+ passphrases if `returns=`*`int`* is specified.

3.10.2 Predefined Symbol Sets

The following predefined sets are used by the generation functions above, but are exported by this module for general use:

default_charsets
Dictionary mapping charset name -> string of characters, used by *genword()*. See that function for a list of predefined charsets present in this dict.

default_wordsets
Dictionary mapping wordset name -> tuple of words, used by *genphrase()*. See that function for a list of predefined wordsets present in this dict.

(Note that this is actually a special object which will lazy-load wordsets from disk on-demand)

3.10.3 Password Strength Estimation

Passlib does not currently offer any password strength estimation routines. However, the (javascript-based) zxcvbn project is a *very* good choice.

Though there are a few different python ports of ZXCVBN library, as of 2019-11-13, zxcvbn (@ pypi) is the most up-to-date, and is endorsed by the upstream zxcvbn developers.

3.11 passlib.registry - Password Handler Registry

This module contains the code Passlib uses to track all password hash handlers that it knows about. While custom handlers can be used directly within an application, or even handed to a `CryptContext`; it is frequently useful to register them globally within a process and then refer to them by name. This module provides facilities for that, as well as programmatically querying Passlib to detect what algorithms are available.

Warning: This module is primarily used as an internal support module. Its interface has not been finalized yet, and may be changed somewhat between major releases of Passlib, as the internal code is cleaned up and simplified.

Applications should access hashes through the *passlib.hash* module where possible (new ones may also be registered by writing to that module).

3.11.1 Interface

passlib.registry.**get_crypt_handler**(*name*[, *default*])
return handler for specified password hash scheme.

this method looks up a handler for the specified scheme. if the handler is not already loaded, it checks if the location is known, and loads it first.

Parameters

- **name** – name of handler to return
- **default** – optional default value to return if no handler with specified name is found.

Raises **KeyError** – if no handler matching that name is found, and no default specified, a KeyError will be raised.

Returns handler attached to name, or default value (if specified).

passlib.registry.**list_crypt_handlers**(*loaded_only=False*)
return sorted list of all known crypt handler names.

Parameters **loaded_only** – if `True`, only returns names of handlers which have actually been loaded.

Returns list of names of all known handlers

passlib.registry.**register_crypt_handler_path**(*name*, *path*)
register location to lazy-load handler when requested.

custom hashes may be registered via *register_crypt_handler()*, or they may be registered by this function, which will delay actually importing and loading the handler until a call to *get_crypt_handler()* is made for the specified name.

Parameters

- **name** – name of handler
- **path** – module import path

the specified module path should contain a password hash handler called *name*, or the path may contain a colon, specifying the module and module attribute to use. for example, the following would cause `get_handler("myhash")` to look for a class named `myhash` within the `myapp.helpers` module:

```
>>> from passlib.registry import registry_crypt_handler_path
>>> registry_crypt_handler_path("myhash", "myapp.helpers")
```

...while this form would cause `get_handler("myhash")` to look for a class name `MyHash` within the `myapp.helpers` module:

```
>>> from passlib.registry import registry_crypt_handler_path
>>> registry_crypt_handler_path("myhash", "myapp.helpers:MyHash")
```

passlib.registry.**register_crypt_handler**(*handler*, *force=False*)

register password hash handler.

this method immediately registers a handler with the internal passlib registry, so that it will be returned by *get_crypt_handler()* when requested.

Parameters

- **handler** – the password hash handler to register
- **force** – force override of existing handler (defaults to False)
- **_attr** – [internal kwd] if specified, ensures `handler.name` matches this value, or raises `ValueError`.

Raises

- **TypeError** – if the specified object does not appear to be a valid handler.
- **ValueError** – if the specified object's name (or other required attributes) contain invalid values.
- **KeyError** – if a (different) handler was already registered with the same name, and `force=True` was not specified.

Note: All password hashes registered with passlib can be imported by name from the *passlib.hash* module. This is true not just of the built-in hashes, but for any hash registered with the registration functions in this module.

3.11.2 Usage

Example showing how to use `registry_crypt_handler_path()`:

```
>>> # register the location of a handler without loading it
>>> from passlib.registry import register_crypt_handler_path
>>> register_crypt_handler_path("myhash", "myapp.support.hashes")

>>> # even before being loaded, its name will show up as available
>>> from passlib.registry import list_crypt_handlers
>>> 'myhash' in list_crypt_handlers()
True
>>> 'myhash' in list_crypt_handlers(loaded_only=True)
False

>>> # when the name "myhash" is next referenced,
>>> # the class "myhash" will be imported from the module "myapp.support.hashes"
>>> from passlib.context import CryptContext
>>> cc = CryptContext(schemes=["myhash"]) #<-- this will cause autoimport
```

Example showing how to load a hash by name:

```
>>> from passlib.registry import get_crypt_handler
>>> get_crypt_handler("sha512_crypt")
<class 'passlib.handlers.sha2_crypt.sha512_crypt'>

>>> get_crypt_handler("missing_hash")
KeyError: "no crypt handler found for algorithm: 'missing_hash'"

>>> get_crypt_handler("missing_hash", None)
None
```

3.12 passlib.totp – TOTP / Two Factor Authentication

New in version 1.7.

3.12.1 Overview

The passlib.totp module provides a number of classes for implementing two-factor authentication (2FA) using the TOTP[1] specification. This page provides a reference to all the classes and methods in this module.

Passlib's TOTP support is centered around the *TOTP* class. There are also some additional helpers, including the *AppWallet* class, which helps to securely encrypt TOTP keys for storage.

See also:

- *TOTP Tutorial* – Overview of this module and walkthrough of how to use it.

3.12.2 TOTP Class

class passlib.totp.**TOTP**(*key=None*, *format="base32"*, ***, *new=False*, ***kwds*)

Helper for generating and verifying TOTP codes.

Given a secret key and set of configuration options, this object offers methods for token generation, token validation, and serialization. It can also be used to track important persistent TOTP state, such as the last counter used.

This class accepts the following options (only **key** and **format** may be specified as positional arguments).

Parameters

- **key** (*str*) – The secret key to use. By default, should be encoded as a base32 string (see **format** for other encodings).

  Exactly one of **key** or new=True must be specified.

- **format** (*str*) – The encoding used by the **key** parameter. May be one of: "base32" (base32-encoded string), "hex" (hexadecimal string), or "raw" (raw bytes). Defaults to "base32".

- **new** (*bool*) – If True, a new key will be generated using random.SystemRandom.

  Exactly one new=True or **key** must be specified.

[1] TOTP Specification - RFC 6238

- **label** (*str*) – Label to associate with this token when generating a URI. Displayed to user by most OTP client applications (e.g. Google Authenticator), and typically has format such as `"John Smith"` or `"jsmith@webservice.example.org"`. Defaults to `None`. See *to_uri()* for details.
- **issuer** (*str*) – String identifying the token issuer (e.g. the domain name of your service). Used internally by some OTP client applications (e.g. Google Authenticator) to distinguish entries which otherwise have the same label. Optional but strongly recommended if you're rendering to a URI. Defaults to `None`. See *to_uri()* for details.
- **size** (*int*) – Number of bytes when generating new keys. Defaults to size of hash algorithm (e.g. 20 for SHA1).

  Warning: Overriding the default values for `digits`, `period`, or `alg` may cause problems with some OTP client programs (such as Google Authenticator), which may have these defaults hardcoded.

- **digits** (*int*) – The number of digits in the generated / accepted tokens. Defaults to `6`. Must be in range [6 .. 10].

  Caution: Due to a limitation of the HOTP algorithm, the 10th digit can only take on values 0 .. 2, and thus offers very little extra security.

- **alg** (*str*) – Name of hash algorithm to use. Defaults to `"sha1"`. `"sha256"` and `"sha512"` are also accepted, per **RFC 6238**.
- **period** (*int*) – The time-step period to use, in integer seconds. Defaults to `30`.

See below for all the `TOTP` methods & attributes...

3.12.3 Alternate Constructors

There are a few alternate class constructors offered. These range from simple convenience wrappers such as *TOTP.new()*, to deserialization methods such as *TOTP.from_source()*.

classmethod `TOTP.new()`
convenience alias for creating new TOTP key, same as `TOTP(new=True)`

classmethod `TOTP.from_source`(*source*)
Load / create a TOTP object from a serialized source. This acts as a wrapper for the various deserialization methods:

- TOTP URIs are handed off to *from_uri()*
- Any other strings are handed off to *from_json()*
- Dicts are handed off to *from_dict()*

Parameters **source** – Serialized TOTP object.

Raises **ValueError** – If the key has been encrypted, but the application secret isn't available; or if the string cannot be recognized, parsed, or decoded.

See *TOTP.using()* for how to configure application secrets.

Returns a *TOTP* instance.

classmethod TOTP.**from_uri**(*uri*)
create an OTP instance from a URI (such as returned by *to_uri()*).

Returns *TOTP* instance.

Raises **ValueError** – if the uri cannot be parsed or contains errors.

See also:

Configuring Clients tutorial for a usage example

classmethod TOTP.**from_json**(*source*)
Load / create an OTP object from a serialized json string (as generated by *to_json()*).

Parameters **json** – Serialized output from *to_json()*, as unicode or ascii bytes.

Raises **ValueError** – If the key has been encrypted, but the application secret isn't available; or if the string cannot be recognized, parsed, or decoded.

See *TOTP.using()* for how to configure application secrets.

Returns a *TOTP* instance.

See also:

Storing TOTP instances tutorial for a usage example

classmethod TOTP.**from_dict**(*source*)
Load / create a TOTP object from a dictionary (as generated by *to_dict()*)

Parameters **source** – dict containing serialized TOTP key & configuration.

Raises **ValueError** – If the key has been encrypted, but the application secret isn't available; or if the dict cannot be recognized, parsed, or decoded.

See *TOTP.using()* for how to configure application secrets.

Returns A *TOTP* instance.

See also:

Storing TOTP instances tutorial for a usage example

3.12.4 Factory Creation

One powerful method offered by the TOTP class is *TOTP.using()*. This method allows you to quickly create TOTP subclasses with preconfigured defaults, for configuration application secrets and setting default TOTP behavior for your application:

classmethod TOTP.**using**(*digits=None*, *alg=None*, *period=None*, *issuer=None*, *wallet=None*, *now=None*, ***kwds*)
Dynamically create subtype of TOTP class which has the specified defaults set.

Parameters **digits, alg, period, issuer**:

All these options are the same as in the *TOTP* constructor, and the resulting class will use any values you specify here as the default for all TOTP instances it creates.

Parameters

- **wallet** – Optional *AppWallet* that will be used for encrypting/decrypting keys.
- **secrets_path, encrypt_cost** (*secrets,*) – If specified, these options will be passed to the *AppWallet* constructor, allowing you to directly specify the secret keys that should be used to encrypt & decrypt stored keys.

Returns subclass of `TOTP`.

This method is useful for creating a TOTP class configured to use your application's secrets for encrypting & decrypting keys, as well as create new keys using it's desired configuration defaults.

As an example:

```
>>> # your application can create a custom class when it initializes
>>> from passlib.totp import TOTP, generate_secret
>>> TotpFactory = TOTP.using(secrets={"1": generate_secret()})

>>> # subsequent TOTP objects created from this factory
>>> # will use the specified secrets to encrypt their keys...
>>> totp = TotpFactory.new()
>>> totp.to_dict()
{'enckey': {'c': 14,
  'k': 'H77SYXWORDPGVOQTFRR2HFUB3C45XXI7',
  's': 'G5DOQPIHIBUM2OOHHADQ',
  't': '1',
  'v': 1},
 'type': 'totp',
 'v': 1}
```

See also:

Creating TOTP Instances and *Storing TOTP instances* tutorials for a usage example

3.12.5 Basic Attributes

All the TOTP objects offer the following attributes, which correspond to the constructor options above. Most of this information will be serialized by *TOTP.to_uri()* and *TOTP.to_json()*:

`TOTP.`**`key`**
secret key as raw bytes

`TOTP.`**`hex_key`**
secret key encoded as hexadecimal string

`TOTP.`**`base32_key`**
secret key encoded as base32 string

`TOTP.`**`label = None`**
default label for *to_uri()*

`TOTP.`**`issuer = None`**
default issuer for *to_uri()*

`TOTP.`**`digits = 6`**
number of digits in the generated tokens.

`TOTP.`**`alg = 'sha1'`**
name of hash algorithm in use (e.g. `"sha1"`)

`TOTP.`**`period = 30`**
number of seconds per counter step. *(TOTP uses an internal time-derived counter which increments by 1 every* `period` *seconds).*

`TOTP.`**`changed = False`**
Flag set by deserialization methods to indicate the object needs to be re-serialized. This can be for a number of reasons – encoded using deprecated format, or encrypted using a deprecated key or too few rounds.

3.12.6 Token Generation

Token generation is generally useful client-side, and for generating values to test your server implementation. There is one main generation method:

`TOTP.`**`generate`**(*time=None*)

Generate token for specified time (uses current time if none specified).

Parameters **`time`** – Can be `None`, a `datetime`, or class:*!float* / `int` unix epoch timestamp. If `None` (the default), uses current system time. Naive datetimes are treated as UTC.

Returns A *`TotpToken`* instance, which can be treated as a sequence of `(token, expire_time)` – see that class for more details.

Usage example:

```
>>> # generate a new token, wrapped in a TotpToken instance...
>>> otp = TOTP('s3jdvb7qd2r7jpxx')
>>> otp.generate(1419622739)
<TotpToken token='897212' expire_time=1419622740>

>>> # when you just need the token...
>>> otp.generate(1419622739).token
'897212'
```

Warning: Tokens should be displayed as strings, as they may contain leading zeros which will get stripped if they are first converted to an `int`.

3.12.6.1 TotpToken

The `TOTP.generate()` method returns instances of the following class, which offers up detailed information about the generated token:

`class` `passlib.totp.`**`TotpToken`**

Object returned by *`TOTP.generate()`*. It can be treated as a sequence of `(token, expire_time)`, or accessed via the following attributes:

`token = None`

Token as decimal-encoded ascii string.

`expire_time`

Timestamp marking end of period when token is valid

`counter = None`

HOTP counter value used to generate token (derived from time)

`remaining`

number of (float) seconds before token expires

`valid`

whether token is still valid

3.12.7 Token Matching / Verification

Matching user-provided tokens is the main operation when implementing server-side TOTP support. Passlib offers one main method: `TOTP.match()`, as well as a convenience wrapper `TOTP.verify()`:

TOTP.**match**(*token*, *time=None*, *window=30*, *skew=0*, *last_counter=None*)

Match TOTP token against specified timestamp. Searches within a window before & after the provided time, in order to account for transmission delay and small amounts of skew in the client's clock.

Parameters

- **token** – Token to validate. may be integer or string (whitespace and hyphens are ignored).
- **time** – Unix epoch timestamp, can be any of `float`, `int`, or `datetime`. if `None` (the default), uses current system time. *this should correspond to the time the token was received from the client*.
- **window** (*int*) – How far backward and forward in time to search for a match. Measured in seconds. Defaults to `30`. Typically only useful if set to multiples of *period*.
- **skew** (*int*) – Adjust timestamp by specified value, to account for excessive client clock skew. Measured in seconds. Defaults to `0`.

  Negative skew (the common case) indicates transmission delay, and/or that the client clock is running behind the server.

  Positive skew indicates the client clock is running ahead of the server (and by enough that it cancels out any negative skew added by the transmission delay).

  You should ensure the server clock uses a reliable time source such as NTP, so that only the client clock's inaccuracy needs to be accounted for.

  This is an advanced parameter that should usually be left at `0`; The **window** parameter is usually enough to account for any observed transmission delay.
- **last_counter** – Optional value of last counter value that was successfully used. If specified, verify will never search earlier counters, no matter how large the window is.

  Useful when client has previously authenticated, and thus should never provide a token older than previously verified value.

Raises *TokenError* – If the token is malformed, fails to match, or has already been used.

Returns TotpMatch Returns a *TotpMatch* instance on successful match. Can be treated as tuple of `(counter, time)`. Raises error if token is malformed / can't be verified.

Usage example:

```
>>> totp = TOTP('s3jdvb7qd2r7jpxx')

>>> # valid token for this time period
>>> totp.match('897212', 1419622729)
<TotpMatch counter=47320757 time=1419622729 cache_seconds=60>

>>> # token from counter step 30 sec ago (within allowed window)
>>> totp.match('000492', 1419622729)
<TotpMatch counter=47320756 time=1419622729 cache_seconds=60>

>>> # invalid token -- token from 60 sec ago (outside of window)
>>> totp.match('760389', 1419622729)
Traceback:
    ...
InvalidTokenError: Token did not match
```

classmethod TOTP.**verify**(*token*, *source*, ***kwds*)

Convenience wrapper around *TOTP.from_source()* and *TOTP.match()*.

This parses a TOTP key & configuration from the specified source, and tries and match the token. It's designed to parallel the *passlib.ifc.PasswordHash.verify()* method.

Parameters

- **token** – Token string to match.
- **source** – Serialized TOTP key. Can be anything accepted by *TOTP.from_source()*.
- ****kwds** – All additional keywords passed to *TOTP.match()*.

Returns A *TotpMatch* instance, or raises a `TokenError`.

See also:

Verifying Tokens tutorial for a usage example

3.12.7.1 TotpMatch

If successful, the `TOTP.verify()` method returns instances of the following class, which offers up detailed information about the matched token:

class passlib.totp.**TotpMatch**

Object returned by *TOTP.match()* and *TOTP.verify()* on a successful match.

It can be treated as a sequence of `(counter, time)`, or accessed via the following attributes:

counter = 0

TOTP counter value which matched token. (Best practice is to subsequently ignore tokens matching this counter or earlier)

time = 0

Timestamp when verification was performed.

expected_counter = 0

Counter value expected for timestamp.

skipped = 0

How many steps were skipped between expected and actual matched counter value (may be positive, zero, or negative).

expire_time = 0

Timestamp marking end of period when token is valid

cache_seconds = 60

Number of seconds counter should be cached before it's guaranteed to have passed outside of verification window.

cache_time = 0

Timestamp marking when counter has passed outside of verification window.

This object will always have a `True` boolean value.

3.12.8 Client Configuration Methods

Once a server has generated a new TOTP key & configuration, it needs to be communicated to the user in order for them to store it in a suitable TOTP client.

This can be done by displaying the key & configuration for the user to hand-enter into their client, or by encoding TOTP object into a URI[3]. These configuration URIs can subsequently be displayed as a QR code, for easy transfer to many smartphone-based TOTP clients (such as Authy or Google Authenticator).

TOTP.**to_uri**(*label=None*, *issuer=None*)

Serialize key and configuration into a URI, per Google Auth's KeyUriFormat.

Parameters

- **label** (*str*) – Label to associate with this token when generating a URI. Displayed to user by most OTP client applications (e.g. Google Authenticator), and typically has format such as `"John Smith"` or `"jsmith@webservice.example.org"`.

  Defaults to **label** constructor argument. Must be provided in one or the other location. May not contain `:`.

- **issuer** (*str*) – String identifying the token issuer (e.g. the domain or canonical name of your service). Optional but strongly recommended if you're rendering to a URI. Used internally by some OTP client applications (e.g. Google Authenticator) to distinguish entries which otherwise have the same label.

  Defaults to **issuer** constructor argument, or `None`. May not contain `:`.

Raises **ValueError** –

- if a label was not provided either as an argument, or in the constructor.
- if the label or issuer contains invalid characters.

Returns all the configuration information for this OTP token generator, encoded into a URI.

These URIs are frequently converted to a QRCode for transferring to a TOTP client application such as Google Auth. Usage example:

```
>>> from passlib.totp import TOTP
>>> tp = TOTP('s3jdvb7qd2r7jpxx')
>>> uri = tp.to_uri("user@example.org", "myservice.another-example.org")
>>> uri
'otpauth://totp/user@example.org?secret=S3JDVB7QD2R7JPXX&issuer=myservice.another-
↪example.org'
```

Changed in version 1.7.2: This method now prepends the issuer URI label. This is recommended by the KeyURI specification, for compatibility with older clients.

TOTP.**pretty_key**(*format='base32'*, *sep='-'*)

pretty-print the secret key.

This is mainly useful for situations where the user cannot get the qrcode to work, and must enter the key manually into their TOTP client. It tries to format the key in a manner that is easier for humans to read.

Parameters

- **format** – format to output secret key. `"hex"` and `"base32"` are both accepted.
- **sep** – separator to insert to break up key visually. can be any of `"-"` (the default), `" "`, or `False` (no separator).

Returns key as native string.

Usage example:

[3] Google's OTPAuth URI format - https://github.com/google/google-authenticator/wiki/Key-Uri-Format

```
>>> t = TOTP('s3jdvb7qd2r7jpxx')
>>> t.pretty_key()
'S3JD-VB7Q-D2R7-JPXX'
```

See also:

- The *TOTP.from_source()* and *TOTP.from_uri()* constructors for decoding URIs.
- The *Configuring Clients* tutorial for details about these methods, and how to render URIs to a QR Code.

3.12.9 Serialization Methods

The *TOTP.to_uri()* method is useful, but limited, because it requires additional information (label & issuer), and lacks the ability to encrypt the key. The *TOTP* provides the following methods for serializing TOTP objects to internal storage. When application secrets are configured via *TOTP.using()*, these methods will automatically encrypt the resulting keys.

TOTP.**to_json**(*encrypt=None*)
Serialize configuration & internal state to a json string, mainly useful for persisting client-specific state in a database. All keywords passed to *to_dict()*.

Returns json string containing serializes configuration & state.

TOTP.**to_dict**(*encrypt=None*)
Serialize configuration & internal state to a dict, mainly useful for persisting client-specific state in a database.

Parameters **encrypt** – Whether to output should be encrypted.

- None (the default) – uses encrypted key if application secrets are available, otherwise uses plaintext key.
- True – uses encrypted key, or raises TypeError if application secret wasn't provided to OTP constructor.
- False – uses raw key.

Returns dictionary, containing basic (json serializable) datatypes.

See also:

- The *TOTP.from_source()* and *TOTP.from_json()* constructors for decoding the results of these methods.
- The *Storing TOTP instances* tutorial for more details.

3.12.10 Helper Methods

While *TOTP.generate()*, *TOTP.match()*, and *TOTP.verify()* automatically handle normalizing tokens & time values, the following methods are exposed in case they are useful in other contexts:

TOTP.**normalize_token**(*token*)
Normalize OTP token representation: strips whitespace, converts integers to a zero-padded string, validates token content & number of digits.

This is a hybrid method – it can be called at the class level, as TOTP.normalize_token(), or the instance level as TOTP().normalize_token(). It will normalize to the instance-specific number of *digits*, or use the class default.

Parameters **token** – token as ascii bytes, unicode, or an integer.

Raises **ValueError** – if token has wrong number of digits, or contains non-numeric characters.

Returns token as `unicode` string, containing only digits 0-9.

classmethod `TOTP.`**normalize_time**(*time*)

Normalize time value to unix epoch seconds.

Parameters **time** – Can be `None`, `datetime`, or unix epoch timestamp as `float` or `int`. If `None`, uses current system time. Naive datetimes are treated as UTC.

Returns unix epoch timestamp as `int`.

3.12.11 AppWallet

The `AppWallet` class is used internally by the *TOTP.using()* method to store the application secrets provided for handling encrypted keys. If needed, they can also be created and passed in directly.

class `passlib.totp.`**AppWallet**(*secrets=None*, *default_tag=None*, *encrypt_cost=None*, *secrets_path=None*)

This class stores application-wide secrets that can be used to encrypt & decrypt TOTP keys for storage. It's mostly an internal detail, applications usually just need to pass `secrets` or `secrets_path` to *TOTP.using()*.

See also:

Storing TOTP instances for more details on this workflow.

3.12.11.1 Arguments

Parameters

- **secrets** – Dict of application secrets to use when encrypting/decrypting stored TOTP keys. This should include a secret to use when encrypting new keys, but may contain additional older secrets to decrypt existing stored keys.

  The dict should map tags -> secrets, so that each secret is identified by a unique tag. This tag will be stored along with the encrypted key in order to determine which secret should be used for decryption. Tag should be string that starts with regex range `[a-z0-9]`, and the remaining characters must be in `[a-z0-9_.-]`.

  It is recommended to use something like a incremental counter ("1", "2", ...), an ISO date ("2016-01-01", "2016-05-16", ...), or a timestamp ("19803495", "19813495", ...) when assigning tags.

  This mapping be provided in three formats:

  – A python dict mapping tag -> secret

  – A JSON-formatted string containing the dict

  – A multiline string with the format `"tag: value\ntag: value\n..."`

  (This last format is mainly useful when loading from a text file via **secrets_path**)

  See also:

  generate_secret() to create a secret with sufficient entropy

- **secrets_path** – Alternately, callers can specify a separate file where the application-wide secrets are stored, using either of the string formats described in **secrets**.

- **default_tag** – Specifies which tag in **secrets** should be used as the default for encrypting new keys. If omitted, the tags will be sorted, and the largest tag used as the default.

  if all tags are numeric, they will be sorted numerically; otherwise they will be sorted alphabetically. this permits tags to be assigned numerically, or e.g. using `YYYY-MM-DD` dates.

- **encrypt_cost** – Optional time-cost factor for key encryption. This value corresponds to log2() of the number of PBKDF2 rounds used.

Warning: The application secret(s) should be stored in a secure location by your application, and each secret should contain a large amount of entropy (to prevent brute-force attacks if the encrypted keys are leaked).

generate_secret() is provided as a convenience helper to generate a new application secret of suitable size.

Best practice is to load these values from a file via **secrets_path**, and then have your application give up permission to read this file once it's running.

3.12.11.2 Public Methods

has_secrets
whether at least one application secret is present

default_tag = None
tag for default secret

3.12.11.3 Semi-Private Methods

The following methods are used internally by the *TOTP* class in order to encrypt & decrypt keys using the provided application secrets. They will generally not be publically useful, and may have their API changed periodically.

get_secret(*tag*)
resolve a secret tag to the secret (as bytes). throws a KeyError if not found.

encrypt_key(*key*)
Helper used to encrypt TOTP keys for storage.

Parameters **key** – TOTP key to encrypt, as raw bytes.

Returns dict containing encrypted TOTP key & configuration parameters. this format should be treated as opaque, and potentially subject to change, though it is designed to be easily serialized/deserialized (e.g. via JSON).

Note: This function requires installation of the external cryptography package.

To give some algorithm details: This function uses AES-256-CTR to encrypt the provided data. It takes the application secret and randomly generated salt, and uses PBKDF2-HMAC-SHA256 to combine them and generate the AES key & IV.

decrypt_key(*enckey*)
Helper used to decrypt TOTP keys from storage format. Consults configured secrets to decrypt key.

Parameters **source** – source object, as returned by *encrypt_key()*.

Returns

`(key, needs_recrypt)` –

key will be the decrypted key, as bytes.

needs_recrypt will be a boolean flag indicating whether encryption cost or default tag is too old, and henace that key needs re-encrypting before storing.

Note: This function requires installation of the external cryptography package.

3.12.12 Support Functions

`passlib.totp.`**`generate_secret`**`(`*entropy=256*`)`

generate a random string suitable for use as an *AppWallet* application secret.

Parameters **`entropy`** – number of bits of entropy (controls size/complexity of password).

3.12.13 Deviations

- The TOTP Spec[1] includes an param (`T0`) providing an optional offset from the base time. Passlib omits this parameter (fixing it at `0`), but so do pretty much all other TOTP implementations.

3.13 `passlib.utils` - Helper Functions

Warning: This module is primarily used as an internal support module. Its interface has not been finalized yet, and may be changed somewhat between major releases of Passlib, as the internal code is cleaned up and simplified.

This module primarily contains utility functions used internally by Passlib. However, end-user applications may find some of the functions useful, in particular:

- *consteq()*
- *saslprep()*
- *generate_password()*

3.13.1 Constants

`passlib.utils.`**`unix_crypt_schemes`**

List of the names of all the hashes in *passlib.hash* which are natively supported by `crypt()` on at least one operating system.

For all hashes in this list, the expression `passlib.hash.`*`alg`*`.has_backend("os_crypt")` will return `True` if the host OS natively supports the hash. This list is used by *host_context* and *ldap_context* to determine which hashes are supported by the host.

See also:

Identifiers & Platform Support for a table of which OSes are known to support which hashes.

3.13.2 Unicode Helpers

passlib.utils.**consteq**(*left*, *right*)
Check two strings/bytes for equality.

This is functionally equivalent to `left == right`, but attempts to take constant time relative to the size of the righthand input.

The purpose of this function is to help prevent timing attacks during digest comparisons: the standard == operator aborts after the first mismatched character, causing its runtime to be proportional to the longest prefix shared by the two inputs. If an attacker is able to predict and control one of the two inputs, repeated queries can be leveraged to reveal information about the content of the second argument. To minimize this risk, `consteq()` is designed to take `THETA(len(right))` time, regardless of the contents of the two strings. It is recommended that the attacker-controlled input be passed in as the left-hand value.

Warning: This function is *not* perfect. Various VM-dependant issues (e.g. the VM's integer object instantiation algorithm, internal unicode representation, etc), may still cause the function's run time to be affected by the inputs, though in a less predictable manner. *To minimize such risks, this function should not be passed* `unicode` *inputs that might contain non-* `ASCII` *characters.*

New in version 1.6.

Changed in version 1.7: This is an alias for stdlib's `hmac.compare_digest()` under Python 3.3 and up.

passlib.utils.**saslprep**(*source*, *param='value'*)
Normalizes unicode strings using SASLPrep stringprep profile.

The SASLPrep profile is defined in **RFC 4013**. It provides a uniform scheme for normalizing unicode usernames and passwords before performing byte-value sensitive operations such as hashing. Among other things, it normalizes diacritic representations, removes non-printing characters, and forbids invalid characters such as `\n`. Properly internationalized applications should run user passwords through this function before hashing.

Parameters

- **source** – unicode string to normalize & validate
- **param** – Optional noun identifying source parameter in error messages (Defaults to the string `"value"`). This is mainly useful to make the caller's error messages make more sense contextually.

Raises

- **ValueError** – if any characters forbidden by the SASLPrep profile are encountered.
- **TypeError** – if input is not `unicode`

Returns normalized unicode string

Note: This function is not available under Jython, as the Jython stdlib is missing the `stringprep` module (Jython issue 1758320).

New in version 1.6.

3.13.3 Bytes Helpers

passlib.utils.**xor_bytes**(*left*, *right*)
Perform bitwise-xor of two byte strings (must be same size)

passlib.utils.**render_bytes**(*source*, **args*)
Peform % formating using bytes in a uniform manner across Python 2/3.

This function is motivated by the fact that bytes instances do not support % or {} formatting under Python 3. This function is an attempt to provide a replacement: it converts everything to unicode (decoding bytes instances as latin-1), performs the required formatting, then encodes the result to latin-1.

Calling render_bytes(source, *args) should function roughly the same as source % args under Python 2.

Todo: python >= 3.5 added back limited support for bytes %, can revisit when 3.3/3.4 is dropped.

---

passlib.utils.**int_to_bytes**(*value*, *count*)
encode integer as single big-endian byte string

passlib.utils.**bytes_to_int**(*value*)
decode byte string as single big-endian integer

3.13.4 Encoding Helpers

passlib.utils.**is_same_codec**(*left*, *right*)
Check if two codec names are aliases for same codec

passlib.utils.**is_ascii_codec**(*codec*)
Test if codec is compatible with 7-bit ascii (e.g. latin-1, utf-8; but not utf-16)

passlib.utils.**is_ascii_safe**(*source*)
Check if string (bytes or unicode) contains only 7-bit ascii

passlib.utils.**to_bytes**(*source*, *encoding='utf-8'*, *param='value'*, *source_encoding=None*)
Helper to normalize input to bytes.

Parameters

- **source** – Source bytes/unicode to process.
- **encoding** – Target encoding (defaults to "utf-8").
- **param** – Optional name of variable/noun to reference when raising errors
- **source_encoding** – If this is specified, and the source is bytes, the source will be transcoded from *source_encoding* to *encoding* (via unicode).

Raises **TypeError** – if source is not unicode or bytes.

Returns

- unicode strings will be encoded using *encoding*, and returned.
- if *source_encoding* is not specified, byte strings will be returned unchanged.
- if *source_encoding* is specified, byte strings will be transcoded to *encoding*.

passlib.utils.**to_unicode**(*source*, *encoding='utf-8'*, *param='value'*)
Helper to normalize input to unicode.

Parameters

- **source** – source bytes/unicode to process.
- **encoding** – encoding to use when decoding bytes instances.
- **param** – optional name of variable/noun to reference when raising errors.

Raises `TypeError` – if source is not unicode or bytes.

Returns

- returns unicode strings unchanged.
- returns bytes strings decoded using *encoding*

passlib.utils.**to_native_str**(*source*, *encoding='utf-8'*, *param='value'*)
Take in unicode or bytes, return native string.

Python 2: encodes unicode using specified encoding, leaves bytes alone. Python 3: leaves unicode alone, decodes bytes using specified encoding.

Raises `TypeError` – if source is not unicode or bytes.

Parameters

- **source** – source unicode or bytes string.
- **encoding** – encoding to use when encoding unicode or decoding bytes. this defaults to `"utf-8"`.
- **param** – optional name of variable/noun to reference when raising errors.

Returns `str` instance

3.13.5 Randomness

passlib.utils.**rng**
The random number generator used by Passlib to generate salt strings and other things which don't require a cryptographically strong source of randomness.

If `os.urandom()` support is available, this will be an instance of `random.SystemRandom`, otherwise it will use the default python PRNG class, seeded from various sources at startup.

passlib.utils.**getrandbytes**(*rng*, *count*)
return byte-string containing *count* number of randomly generated bytes, using specified rng

passlib.utils.**getrandstr**(*rng*, *charset*, *count*)
return string containing *count* number of chars/bytes, whose elements are drawn from specified charset, using specified rng

passlib.utils.**generate_password**(*size=10*, *charset=<default charset>*)
generate random password using given length & charset

param size size of password.

param charset optional string specified set of characters to draw from.

the default charset contains all normal alphanumeric characters, except for the characters `1IiLl0OoS5`, which were omitted due to their visual similarity.

returns `str` containing randomly generated password.

Note: Using the default character set, on a OS with `SystemRandom` support, this function should generate passwords with 5.7 bits of entropy per character.

Deprecated since version 1.7: and will be removed in version 2.0, use passlib.pwd.genword() / passlib.pwd.genphrase() instead.

3.13.6 Interface Tests

passlib.utils.**is_crypt_handler**(*obj*)
check if object follows the *PasswordHash API*

passlib.utils.**is_crypt_context**(*obj*)
check if object appears to be a *CryptContext* instance

passlib.utils.**has_rounds_info**(*handler*)
check if handler provides the optional *rounds information* attributes

passlib.utils.**has_salt_info**(*handler*)
check if handler provides the optional *salt information* attributes

3.13.7 Submodules

There are also a few sub modules which provide additional utility functions:

3.13.7.1 passlib.utils.handlers - Framework for writing password hashes

Warning: This module is primarily used as an internal support module. Its interface has not been finalized yet, and may be changed somewhat between major releases of Passlib, as the internal code is cleaned up and simplified.

Todo: This module, and the instructions on how to write a custom handler, definitely need to be rewritten for clarity. They are not yet organized, and may leave out some important details.

Implementing Custom Handlers

All that is required in order to write a custom handler that will work with Passlib is to create an object (be it module, class, or object) that exposes the functions and attributes required by the *PasswordHash API*. For classes, Passlib does not make any requirements about what a class instance should look like (if the implementation even uses them).

That said, most of the handlers built into Passlib are based around the *GenericHandler* class, and its associated mixin classes. While deriving from this class is not required, doing so will greatly reduce the amount of additional code that is needed for all but the most convoluted password hash schemes.

Once a handler has been written, it may be used explicitly, passed into a CryptContext constructor, or registered globally with Passlib via the *passlib.registry* module.

See also:

Testing Hash Handlers for details about how to test custom handlers against Passlib's unittest suite.

The GenericHandler Class

Design

Most of the handlers built into Passlib are based around the *GenericHandler* class. This class is designed under the assumption that the common workflow for hashes is some combination of the following:

1. parse hash into constituent parts - performed by *from_string()*.
2. validate constituent parts - performed by GenericHandler's constructor, and the normalization functions such as *_norm_checksum()* and *_norm_salt()* which are provided by its related mixin classes.
3. calculate the raw checksum for a specific password - performed by *_calc_checksum()*.
4. assemble hash, including new checksum, into a new string - performed by *to_string()*.

With this in mind, GenericHandler provides implementations of most of the *PasswordHash API* methods, eliminating the need for almost all the boilerplate associated with writing a password hash.

In order to minimize the amount of unneeded features that must be loaded in, the GenericHandler class itself contains only the parts which are needed by almost all handlers: parsing, rendering, and checksum validation. Validation of all other parameters (such as salt, rounds, etc) is split out into separate *mixin classes* which enhance GenericHandler with additional features.

Usage

In order to use GenericHandler, just subclass it, and then do the following:

- fill out the name attribute with the name of your hash.
- fill out the setting_kwds attribute with a tuple listing all the settings your hash accepts.
- provide an implementation of the from_string() classmethod.

  this method should take in a potential hash string, parse it into components, and return an instance of the class which contains the parsed components. It should throw a ValueError if no hash, or an invalid hash, is provided.
- provide an implementation of the to_string() instance method.

  this method should render an instance of your handler class (such as returned by from_string()), returning a hash string.
- provide an implementation of the _calc_checksum() instance method.

  this is the heart of the hash; this method should take in the password as the first argument, then generate and return the digest portion of the hash, according to the settings (such as salt, etc) stored in the parsed instance this method was called from.

  note that it should not return the full hash with identifiers, etc; that job should be performed by to_string().

Some additional notes:

- In addition to simply subclassing GenericHandler, most handlers will also benefit from adding in some of the mixin classes that are designed to add features to GenericHandler. See *GenericHandler Mixins* for more details.
- Most implementations will want to alter/override the default *identify()* method. By default, it returns True for all hashes that *from_string()* can parse without raising a ValueError; which is reliable, but somewhat slow. For faster identification purposes, subclasses may fill in the *ident* attribute with the hash's identifying prefix, which *identify()* will then test for instead of calling *from_string()*. For more complex situations, a custom implementation should be used; the *HasManyIdents* mixin may also be helpful.
- This class does not support context kwds of any type, since that is a rare enough requirement inside passlib.

Interface

class passlib.utils.handlers.**GenericHandler**(*checksum=None*, *use_defaults=False*, ***kwds*)

helper class for implementing hash handlers.

GenericHandler-derived classes will have (at least) the following constructor options, though others may be added by mixins and by the class itself:

Parameters

- **checksum** – this should contain the digest portion of a parsed hash (mainly provided when the constructor is called by *from_string()*). defaults to None.
- **use_defaults** – If False (the default), a TypeError should be thrown if any settings required by the handler were not explicitly provided.

  If True, the handler should attempt to provide a default for any missing values. This means generate missing salts, fill in default cost parameters, etc.

  This is typically only set to True when the constructor is called by *hash()*, allowing user-provided values to be handled in a more permissive manner.
- **relaxed** – If False (the default), a ValueError should be thrown if any settings are out of bounds or otherwise invalid.

  If True, they should be corrected if possible, and a warning issue. If not possible, only then should an error be raised. (e.g. under relaxed=True, rounds values will be clamped to min/max rounds).

  This is mainly used when parsing the config strings of certain hashes, whose specifications implementations to be tolerant of incorrect values in salt strings.

Class Attributes

ident
[optional] If this attribute is filled in, the default *identify()* method will use it as a identifying prefix that can be used to recognize instances of this handler's hash. Filling this out is recommended for speed.

This should be a unicode str.

_hash_regex
[optional] If this attribute is filled in, the default *identify()* method will use it to recognize instances of the hash. If *ident* is specified, this will be ignored.

This should be a unique regex object.

checksum_size
[optional] Specifies the number of characters that should be expected in the checksum string. If omitted, no check will be performed.

checksum_chars
[optional] A string listing all the characters allowed in the checksum string. If omitted, no check will be performed.

This should be a unicode str.

_stub_checksum
Placeholder checksum that will be used by genconfig() in lieu of actually generating a hash for the empty string. This should be a string of the same datatype as *checksum*.

Instance Attributes

checksum
The checksum string provided to the constructor (after passing it through *_norm_checksum()*).

Required Subclass Methods

The following methods must be provided by handler subclass:

classmethod from_string(*hash*, ***context*)
return parsed instance from hash/configuration string

Parameters ****context** – context keywords to pass to constructor (if applicable).

Raises **ValueError** – if hash is incorrectly formatted

Returns hash parsed into components, for formatting / calculating checksum.

to_string()
render instance to hash or configuration string

Returns

hash string with salt & digest included.

should return native string type (ascii-bytes under python 2, unicode under python 3)

_calc_checksum(*secret*)
given secret; calcuate and return encoded checksum portion of hash string, taking config from object state

calc checksum implementations may assume secret is always either unicode or bytes, checks are performed by verify/etc.

Default Methods

The following methods have default implementations that should work for most cases, though they may be overridden if the hash subclass needs to:

_norm_checksum(*checksum*, *relaxed=False*)
validates checksum keyword against class requirements, returns normalized version of checksum.

classmethod genconfig(***kwds*)
compile settings into a configuration string for genhash()

Deprecated since version 1.7: As of 1.7, this method is deprecated, and slated for complete removal in Passlib 2.0.

For all known real-world uses, hashing a constant string should provide equivalent functionality.

This deprecation may be reversed if a use-case presents itself in the mean time.

classmethod genhash(*secret*, *config*, ***context*)
generated hash for secret, using settings from config/hash string

Deprecated since version 1.7: As of 1.7, this method is deprecated, and slated for complete removal in Passlib 2.0.

This deprecation may be reversed if a use-case presents itself in the mean time.

classmethod identify(*hash*)
check if hash belongs to this scheme, returns True/False

classmethod hash(*secret*, ***kwds*)
Hash secret, returning result. Should handle generating salt, etc, and should return string containing identifier, salt & other configuration, as well as digest.

Parameters

- ****settings_kwds** – Pass in settings to customize configuration of resulting hash.

  Deprecated since version 1.7: Starting with Passlib 1.7, callers should no longer pass settings keywords (e.g. `rounds` or `salt` directly to `hash()`); should use `.using(**settings).hash(secret)` construction instead.

  Support will be removed in Passlib 2.0.

- ****context_kwds** – Specific algorithms may require context-specific information (such as the user login).

classmethod verify(*secret*, *hash*, ***context*)
verify secret against hash, returns True/False

GenericHandler Mixins

class passlib.utils.handlers.**HasSalt**(*salt=None*, ***kwds*)
mixin for validating salts.

This *GenericHandler* mixin adds a `salt` keyword to the class constuctor; any value provided is passed through the *_norm_salt()* method, which takes care of validating salt length and content, as well as generating new salts if one it not provided.

Parameters

- **salt** – optional salt string
- **salt_size** – optional size of salt (only used if no salt provided); defaults to *default_salt_size*.

Class Attributes

In order for `_norm_salt()` to do its job, the following attributes should be provided by the handler subclass:

min_salt_size
The minimum number of characters allowed in a salt string. An `ValueError` will be throw if the provided salt is too small. Defaults to `0`.

max_salt_size
The maximum number of characters allowed in a salt string. By default an `ValueError` will be throw if the provided salt is too large; but if `relaxed=True`, it will be clipped and a warning issued instead. Defaults to `None`, for no maximum.

default_salt_size
[required] If no salt is provided, this should specify the size of the salt that will be generated by *_generate_salt()*. By default this will fall back to *max_salt_size*.

salt_chars
A string containing all the characters which are allowed in the salt string. An `ValueError` will be throw if any other characters are encountered. May be set to `None` to skip this check (but see in *default_salt_chars*).

default_salt_chars
[required] This attribute controls the set of characters use to generate *new* salt strings. By default, it mirrors *salt_chars*. If salt_chars is None, this attribute must be specified in order to generate new salts. Aside from that purpose, the main use of this attribute is for hashes which wish to generate salts from a restricted subset of salt_chars; such as accepting all characters, but only using a-z.

Instance Attributes

salt
This instance attribute will be filled in with the salt provided to the constructor (as adapted by *_norm_salt()*)

Subclassable Methods

classmethod _norm_salt(*salt*, *relaxed=False*)
helper to normalize & validate user-provided salt string

Parameters **salt** – salt string

Raises

- **TypeError** – If salt not correct type.
- **ValueError** –
  - if salt contains chars that aren't in *salt_chars*.
  - if salt contains less than *min_salt_size* characters.
  - if relaxed=False and salt has more than *max_salt_size* characters (if relaxed=True, the salt is truncated and a warning is issued instead).

Returns normalized salt

classmethod _generate_salt()
helper method for _init_salt(); generates a new random salt string.

class passlib.utils.handlers.**HasRounds**(*rounds=None*, ***kwds*)
mixin for validating rounds parameter

This *GenericHandler* mixin adds a rounds keyword to the class constuctor; any value provided is passed through the *_norm_rounds()* method, which takes care of validating the number of rounds.

Parameters **rounds** – optional number of rounds hash should use

Class Attributes

In order for _norm_rounds() to do its job, the following attributes must be provided by the handler subclass:

min_rounds
The minimum number of rounds allowed. A ValueError will be thrown if the rounds value is too small. Defaults to 0.

max_rounds
The maximum number of rounds allowed. A ValueError will be thrown if the rounds value is larger than this. Defaults to None which indicates no limit to the rounds value.

default_rounds
If no rounds value is provided to constructor, this value will be used. If this is not specified, a rounds value *must* be specified by the application.

rounds_cost
[required] The `rounds` parameter typically encodes a cpu-time cost for calculating a hash. This should be set to `"linear"` (the default) or `"log2"`, depending on how the rounds value relates to the actual amount of time that will be required.

Class Methods

Todo: document using() and needs_update() options

Instance Attributes

rounds
This instance attribute will be filled in with the rounds value provided to the constructor (as adapted by *_norm_rounds()*)

Subclassable Methods

classmethod _norm_rounds(*rounds*, *relaxed=False*, *param='rounds'*)
helper for normalizing rounds value.

Parameters

- **rounds** – an integer cost parameter.
- **relaxed** – if `True` (the default), issues PasslibHashWarning is rounds are outside allowed range. if `False`, raises a ValueError instead.
- **param** – optional name of parameter to insert into error/warning messages.

Raises

- **TypeError** –
  - if `use_defaults=False` and no rounds is specified
  - if rounds is not an integer.
- **ValueError** –
  - if rounds is `None` and class does not specify a value for *default_rounds*.
  - if `relaxed=False` and rounds is outside bounds of *min_rounds* and *max_rounds* (if `relaxed=True`, the rounds value will be clamped, and a warning issued).

Returns normalized rounds value

class passlib.utils.handlers.**HasManyIdents**(*ident=None*, ***kwds*)
mixin for hashes which use multiple prefix identifiers

For the hashes which may use multiple identifier prefixes, this mixin adds an `ident` keyword to constructor. Any value provided is passed through the `norm_idents()` method, which takes care of validating the identifier, as well as allowing aliases for easier specification of the identifiers by the user.

Todo: document this class's usage

Class Methods

Todo: document using() and needs_update() options

class passlib.utils.handlers.**HasManyBackends**(*checksum=None*, *use_defaults=False*, ***kwds*)

GenericHandler mixin which provides selecting from multiple backends.

Todo: finish documenting this class's usage

For hashes which need to select from multiple backends, depending on the host environment, this class offers a way to specify alternate `_calc_checksum()` methods, and will dynamically chose the best one at runtime.

Changed in version 1.7: This class now derives from `BackendMixin`, which abstracts out a more generic framework for supporting multiple backends. The public api (`get_backend()`, `has_backend()`, `set_backend()`) is roughly the same.

Private API (Subclass Hooks)

As of version 1.7, classes should implement `_load_backend_{name}()`, per `BackendMixin`. This hook should invoke `_set_calc_checksum_backcend()` to install it's backend method.

Deprecated since version 1.7: The following api is deprecated, and will be removed in Passlib 2.0:

_has_backend_{name}
: private class attribute checked by `has_backend()` to see if a specific backend is available, it should be either `True` or `False`. One of these should be provided by the subclass for each backend listed in `backends`.

_calc_checksum_{name}
: private class method that should implement `_calc_checksum()` for a given backend. it will only be called if the backend has been selected by `set_backend()`. One of these should be provided by the subclass for each backend listed in `backends`.

class passlib.utils.handlers.**HasRawSalt**(*salt=None*, ***kwds*)

mixin for classes which use decoded salt parameter

A variant of `HasSalt` which takes in decoded bytes instead of an encoded string.

Todo: document this class's usage

class passlib.utils.handlers.**HasRawChecksum**(*checksum=None*, *use_defaults=False*, ***kwds*)

mixin for classes which work with decoded checksum bytes

Todo: document this class's usage

Examples

Todo: Show some walk-through examples of how to use GenericHandler and its mixins

The StaticHandler class

class passlib.utils.handlers.**StaticHandler**(*checksum=None*, *use_defaults=False*, ***kwds*)

GenericHandler mixin for classes which have no settings.

This mixin assumes the entirety of the hash ise stored in the `checksum` attribute; that the hash has no rounds, salt, etc. This class provides the following:

- a default `genconfig()` that always returns None.
- a default `from_string()` and `to_string()` that store the entire hash within `checksum`, after optionally stripping a constant prefix.

All that is required by subclasses is an implementation of the `_calc_checksum()` method.

Todo: Show some examples of how to use StaticHandler

Other Constructors

class passlib.utils.handlers.**PrefixWrapper**(*name*, *wrapped*, *prefix=''*, *orig_prefix=''*, *lazy=False*, *doc=None*, *ident=None*)

wraps another handler, adding a constant prefix.

instances of this class wrap another password hash handler, altering the constant prefix that's prepended to the wrapped handlers' hashes.

this is used mainly by the *ldap crypt* handlers; such as *ldap_md5_crypt* which wraps *md5_crypt* and adds a `{CRYPT}` prefix.

usage:

```
myhandler = PrefixWrapper("myhandler", "md5_crypt", prefix="$mh$", orig_prefix="$1
↪$")
```

Parameters

- **name** – name to assign to handler
- **wrapped** – handler object or name of registered handler
- **prefix** – identifying prefix to prepend to all hashes
- **orig_prefix** – prefix to strip (defaults to '').
- **lazy** – if True and wrapped handler is specified by name, don't look it up until needed.

Testing Hash Handlers

Within its unittests, Passlib provides the *HandlerCase* class, which can be subclassed to provide a unittest-compatible test class capable of checking if a handler adheres to the *PasswordHash API*.

Usage

As an example of how to use HandlerCase, the following is an annotated version of the unittest for *passlib.hash.des_crypt*:

```
from passlib.hash import des_crypt
from passlib.tests.utils import HandlerCase

# create a subclass for the handler...
class DesCryptTest(HandlerCase):
    "test des-crypt algorithm"

    # [required] - store the handler object itself in the handler attribute
    handler = des_crypt

    # [required] - this should be a list of (password, hash) pairs,
    #              which should all verify correctly using your handler.
    #              it is recommend include pairs which test all of the following:
    #
    #              * empty string & short strings for passwords
    #              * passwords with 2 byte unicode characters
    #              * hashes with varying salts, rounds, and other options
    known_correct_hashes = (
        # format: (password, hash)
        ('', 'OgAwTx2l6NADI'),
        (' ', '/Hk.VPuwQTXbc'),
        ('test', 'N1tQbOFcM5fpg'),
        ('Compl3X AlphaNu3meric', 'um.Wguz3eVCx2'),
        ('4lpHa N|_|M3r1K W/ Cur5Es: #$%(*)(*%#', 'sNYqfOyauIyic'),
        ('AlOtBsOl', 'cEpWz5IUCShqM'),
        (u'hell\u00D6', 'saykDgk3BPZ9E'),
        )

    # [optional] - if there are hashes which are similar in format
    #              to your handler, and you want to make sure :meth:`identify`
    #              does not return ``True`` for such hashes,
    #              list them here. otherwise this can be omitted.
    #
    known_unidentified_hashes = [
        # bad char in otherwise correctly formatted hash
        '!gAwTx2l6NADI',
        ]
```

Interface

class passlib.tests.utils.**HandlerCase**

base class for testing password hash handlers (esp passlib.utils.handlers subclasses)

In order to use this to test a handler, create a subclass will all the appropriate attributes filled as listed in the example below, and run the subclass via unittest.

Todo: Document all of the options HandlerCase offers.

Note: This is subclass of `unittest.TestCase` (or `unittest2.TestCase` if available).

3.13.7.2 `passlib.utils.binary` - Binary Helper Functions

Warning: This module is primarily used as an internal support module. Its interface has not been finalized yet, and may be changed somewhat between major releases of Passlib, as the internal code is cleaned up and simplified.

Constants

`passlib.utils.binary.`**`BASE64_CHARS`**
Character map used by standard MIME-compatible Base64 encoding scheme.

`passlib.utils.binary.`**`HASH64_CHARS`**
Base64 character map used by a number of hash formats; the ordering is wildly different from the standard base64 character map.

This encoding system appears to have originated with *des_crypt*, but is used by *md5_crypt*, *sha256_crypt*, and others. Within Passlib, this encoding is referred as the "hash64" encoding, to distinguish it from normal base64 and others.

`passlib.utils.binary.`**`BCRYPT_CHARS`**
Base64 character map used by *bcrypt*. The ordering is wildly different from both the standard base64 character map, and the common hash64 character map.

Base64 Encoding

Base64Engine Class

Passlib has to deal with a number of different Base64 encodings, with varying endianness, as well as wildly different character <-> value mappings. This is all encapsulated in the *Base64Engine* class, which provides common encoding actions for an arbitrary base64-style encoding scheme. There are also a couple of predefined instances which are commonly used by the hashes in Passlib.

class `passlib.utils.binary.`**`Base64Engine`**(*charmap*, *big=False*)
Provides routines for encoding/decoding base64 data using arbitrary character mappings, selectable endianness, etc.

Parameters

- **charmap** – A string of 64 unique characters, which will be used to encode successive 6-bit chunks of data. A character's position within the string should correspond to its 6-bit value.
- **big** – Whether the encoding should be big-endian (default False).

Note: This class does not currently handle base64's padding characters in any way what so ever.

Raw Bytes <-> Encoded Bytes

The following methods convert between raw bytes, and strings encoded using the engine's specific base64 variant:

encode_bytes(*source*)
encode bytes to base64 string.

Parameters **source** – byte string to encode.

Returns byte string containing encoded data.

decode_bytes(*source*)
decode bytes from base64 string.

Parameters **source** – byte string to decode.

Returns byte string containing decoded data.

encode_transposed_bytes(*source*, *offsets*)
encode byte string, first transposing source using offset list

decode_transposed_bytes(*source*, *offsets*)
decode byte string, then reverse transposition described by offset list

Integers <-> Encoded Bytes

The following methods allow encoding and decoding unsigned integers to and from the engine's specific base64 variant. Endianess is determined by the engine's `big` constructor keyword.

encode_int6(*value*)
encodes 6-bit integer -> single hash64 character

decode_int6(*source*)
decode single character -> 6 bit integer

encode_int12(*value*)
encodes 12-bit integer -> 2 char string

decode_int12(*source*)
decodes 2 char string -> 12-bit integer

encode_int24(*value*)
encodes 24-bit integer -> 4 char string

decode_int24(*source*)
decodes 4 char string -> 24-bit integer

encode_int64(*value*)
encode 64-bit integer -> 11 char hash64 string

this format is used primarily by des-crypt & variants to encode the DES output value used as a checksum.

decode_int64(*source*)
decode 11 char base64 string -> 64-bit integer

this format is used primarily by des-crypt & variants to encode the DES output value used as a checksum.

Informational Attributes

charmap
unicode string containing list of characters used in encoding; position in string matches 6bit value of character.

bytemap
bytes version of *charmap*

big
boolean flag indicating this using big-endian encoding.

Predefined Instances

passlib.utils.binary.**h64**
Predefined instance of *Base64Engine* which uses the HASH64_CHARS character map and little-endian encoding. (see *HASH64_CHARS* for more details).

passlib.utils.binary.**h64big**
Predefined variant of *h64* which uses big-endian encoding. This is mainly used by *des_crypt*.

Changed in version 1.6: Previous versions of Passlib contained a module named passlib.utils.h64; As of Passlib 1.6 this was replaced by the the h64 and h64big instances of the *Base64Engine* class; the interface remains mostly unchanged.

Other

passlib.utils.binary.**ab64_encode**(*data*)
encode using shortened base64 format which omits padding & whitespace. uses custom ./ altchars.

it is primarily used by Passlib's custom pbkdf2 hashes.

passlib.utils.binary.**ab64_decode**(*data*)
decode from shortened base64 format which omits padding & whitespace. uses custom ./ altchars, but supports decoding normal +/ altchars as well.

it is primarily used by Passlib's custom pbkdf2 hashes.

passlib.utils.binary.**b64s_encode**(*data*)
encode using shortened base64 format which omits padding & whitespace. uses default +/ altchars.

passlib.utils.binary.**b64s_decode**(*data*)
decode from shortened base64 format which omits padding & whitespace. uses default +/ altchars.

passlib.utils.binary.**b32encode**(*source*)
wrapper around base64.b32encode() which strips padding, and returns a native string.

passlib.utils.binary.**b32decode**(*source*)
wrapper around base64.b32decode() which handles common mistyped chars. padding optional, ignored if present.

3.13.7.3 passlib.utils.des - DES routines [deprecated]

Warning: This module is deprecated as of Passlib 1.7: It has been relocated to *passlib.crypto.des*; and the aliases here will be removed in Passlib 2.0.

This module contains routines for encrypting blocks of data using the DES algorithm. Note that these functions do not support multi-block operation or decryption, since they are designed primarily for use in password hash algorithms (such as *des_crypt* and *bsdi_crypt*).

passlib.utils.des.**expand_des_key**(*key*)
convert DES from 7 bytes to 8 bytes (by inserting empty parity bits)

Deprecated since version 1.7: and will be removed in version 1.8, use passlib.crypto.des.expand_des_key instead.

passlib.utils.des.**des_encrypt_block**(*key*, *input*, *salt=0*, *rounds=1*)
encrypt single block of data using DES, operates on 8-byte strings.

arg key DES key as 7 byte string, or 8 byte string with parity bits (parity bit values are ignored).

arg input plaintext block to encrypt, as 8 byte string.

arg salt Optional 24-bit integer used to mutate the base DES algorithm in a manner specific to *des_crypt* and its variants. The default value `0` provides the normal (unsalted) DES behavior. The salt functions as follows: if the `i`'th bit of `salt` is set, bits `i` and `i+24` are swapped in the DES E-box output.

arg rounds Optional number of rounds of to apply the DES key schedule. the default (`rounds=1`) provides the normal DES behavior, but *des_crypt* and its variants use alternate rounds values.

raises TypeError if any of the provided args are of the wrong type.

raises ValueError if any of the input blocks are the wrong size, or the salt/rounds values are out of range.

returns resulting 8-byte ciphertext block.

Deprecated since version 1.7: and will be removed in version 1.8, use passlib.crypto.des.des_encrypt_block instead.

passlib.utils.des.**des_encrypt_int_block**(*key*, *input*, *salt=0*, *rounds=1*)
encrypt single block of data using DES, operates on 64-bit integers.

this function is essentially the same as *des_encrypt_block()*, except that it operates on integers, and will NOT automatically expand 56-bit keys if provided (since there's no way to detect them).

arg key DES key as 64-bit integer (the parity bits are ignored).

arg input input block as 64-bit integer

arg salt optional 24-bit integer used to mutate the base DES algorithm. defaults to `0` (no mutation applied).

arg rounds optional number of rounds of to apply the DES key schedule. defaults to `1`.

raises TypeError if any of the provided args are of the wrong type.

raises ValueError if any of the input blocks are the wrong size, or the salt/rounds values are out of range.

returns resulting ciphertext as 64-bit integer.

Deprecated since version 1.7: and will be removed in version 1.8, use passlib.crypto.des.des_encrypt_int_block instead.

3.13.7.4 passlib.utils.pbkdf2 - PBKDF2 key derivation algorithm [deprecated]

Warning: This module has been deprecated as of Passlib 1.7, and will be removed in Passlib 2.0. The functions in this module have been replaced by equivalent (but not identical) functions in the *passlib.crypto* module.

This module provides a couple of key derivation functions, as well as supporting utilities. Primarily, it offers *pbkdf2()*, which provides the ability to generate an arbitrary length key using the PBKDF2 key derivation algorithm, as specified in rfc 2898. This function can be helpful in creating password hashes using schemes which have been based around the pbkdf2 algorithm.

PKCS#5 Key Derivation Functions

passlib.utils.pbkdf2.**pbkdf1**(*secret*, *salt*, *rounds*, *keylen=None*, *hash='sha1'*)

pkcs#5 password-based key derivation v1.5

Parameters

- **secret** – passphrase to use to generate key
- **salt** – salt string to use when generating key
- **rounds** – number of rounds to use to generate key
- **keylen** – number of bytes to generate (if None, uses digest's native size)
- **hash** – hash function to use. must be name of a hash recognized by hashlib.

Returns raw bytes of generated key

Note: This algorithm has been deprecated, new code should use PBKDF2. Among other limitations, keylen cannot be larger than the digest size of the specified hash.

Deprecated since version 1.7: This has been relocated to *passlib.crypto.digest.pbkdf1()*, and this version will be removed in Passlib 2.0. *Note the call signature has changed.*

passlib.utils.pbkdf2.**pbkdf2**(*secret*, *salt*, *rounds*, *keylen=None*, *prf='hmac-sha1'*)

pkcs#5 password-based key derivation v2.0

Parameters

- **secret** – passphrase to use to generate key
- **salt** – salt string to use when generating key
- **rounds** – number of rounds to use to generate key
- **keylen** – number of bytes to generate. if set to None, will use digest size of selected prf.
- **prf** – psuedo-random family to use for key strengthening. this must be a string starting with "hmac-", followed by the name of a known digest. this defaults to "hmac-sha1" (the only prf explicitly listed in the PBKDF2 specification)

Returns raw bytes of generated key

Deprecated since version 1.7: This has been deprecated in favor of *passlib.crypto.digest.pbkdf2_hmac()*, and will be removed in Passlib 2.0. *Note the call signature has changed.*

Note: The details of PBKDF1 and PBKDF2 are specified in **RFC 2898**.

Helper Functions

passlib.utils.pbkdf2.**norm_hash_name**(*name*, *format='hashlib'*)

Normalize hash function name (convenience wrapper for `lookup_hash()`).

arg name Original hash function name.

This name can be a Python `hashlib` digest name, a SCRAM mechanism name, IANA assigned hash name, etc. Case is ignored, and underscores are converted to hyphens.

param format Naming convention to normalize to. Possible values are:

- `"hashlib"` (the default) - normalizes name to be compatible with Python's `hashlib`.
- `"iana"` - normalizes name to IANA-assigned hash function name. For hashes which IANA hasn't assigned a name for, this issues a warning, and then uses a heuristic to return a "best guess" name.

returns Hash name, returned as native `str`.

Deprecated since version 1.7: and will be removed in version 1.8, use passlib.crypto.digest.norm_hash_name instead.

passlib.utils.pbkdf2.**get_prf**(*name*)

Lookup pseudo-random family (PRF) by name.

Parameters **name** – This must be the name of a recognized prf. Currently this only recognizes names with the format `hmac-`*`digest`*, where *`digest`* is the name of a hash function such as `md5`, `sha256`, etc.

todo: restore text about callables.

Raises

- **ValueError** – if the name is not known
- **TypeError** – if the name is not a callable or string

Returns

a tuple of (*`prf_func`*, *`digest_size`*), where:

- *`prf_func`* is a function implementing the specified PRF, and has the signature `prf_func(secret, message) -> digest`.
- *`digest_size`* is an integer indicating the number of bytes the function returns.

Usage example:

```
>>> from passlib.utils.pbkdf2 import get_prf
>>> hmac_sha256, dsize = get_prf("hmac-sha256")
>>> hmac_sha256
<function hmac_sha256 at 0x1e37c80>
>>> dsize
32
>>> digest = hmac_sha256('password', 'message')
```

Deprecated since version 1.7: This function is deprecated, and will be removed in Passlib 2.0. This only related replacement is `passlib.crypto.digest.compile_hmac()`.

CHAPTER 4

Other Documentation

Additional pages and documentation which don't fit anywhere else:

4.1 Frequently Asked Questions

This sections documents some frequently asked questions about Passlib, and password hashing in general. But it also includes some common misconceptions, as well as some esoteric and infrequently asked questions.

- **Calling `PasswordHash.hash()` multiple times for the same input generates a different result! What's going on?**

  For all the hashes which include a salt, `PasswordHash.hash()` will automatically generate a new one each time it's invoked. Thus the salt & digest portions of the resulting hash string will be different every time.

- **Do I need to provide a salt each time `PasswordHash.hash()` is called, and store the salt separately in my database?**

  No. Nearly all of the hash classes *passlib.hash* which use a salt will automatically generate a salt, and include it as part of the hash that's returned.

  There are just a few hashes which require an external salt (like a username), or don't contain a salt at all. These generally aren't secure, and shouldn't be used in unless you already know *why* you need to use them.

- **How do I decrypt the hashes generated by Passlib?**

  Short answer: You can't.

  Long answer:

  The hash algorithms in Passlib were explicitly designed so they are as hard to reverse as possible: you can hash a password, you can check if a password matches an existing hash, and that's it. Unless it's an ancient algorithm whose security has been fundamentally undermined, the only way to reverse a hash is to use brute force: hash all potential passwords until one matches. To fight this, one of the main goals of password hashing is to make this search take as long as possible.

However, if you really need it, there are programs dedicated to this task, two prominent ones include John the Ripper and HashCat.

There is one single decryptable hash in Passlib: *cisco_type7*, which was deliberately designed this way; and Passlib's implementation offers a convenient `decrypt()` method.

- **Why use *`PasswordHash.verify()`* instead of hashing user input and using == to compare it with the stored hash?**

  There are two reasons for this: One, `PasswordHash.verify()` uses a "constant time" equality check internally, which mitigates a class of timing attacks that == is potentially vulnerable to. These attacks are mostly theoretical for modern password hashes with a sufficient sized-salt, but it's better to be safe than sorry.

  Two, many hash string formats encode a number of configuration parameters, some unhelpfully allow multiple encodings of the *same* parameters. Thus, to make sure passwords are hashed correctly for comparison, you'll have to parse the hash string and pass the configuration parameters in yourself. *`PasswordHash.verify()`* takes care of this transparently.

- **Is SHA256-Crypt the same as SHA256? Is MD5-Crypt the same as MD5?**

  No. MD5 and SHA256 are cryptographic hash functions, which whereas *md5_crypt* and *sha256_crypt* are complex password hash algorithms, containing a randomly generated salt, variable rounds, etc. They derive their names from the fact that they use the respective hash functions internally.

4.2 Modular Crypt Format

A explanation about a standard that isn't

See also:

Deprecated (as of 2016) in favor of the PHC String Format

In the opinion of the main Passlib author, the modular crypt format (described below) should be considered deprecated when creating new hashes. The PHC String Format is an attempt to specify a common hash string format that's a restricted & well defined subset of the Modular Crypt Format. New hashes are strongly encouraged to adhere to the PHC specification, rather than the much looser Modular Crypt Format.

4.2.1 Overview

A number of the hashes in Passlib are described as adhering to the "Modular Crypt Format". This page is an attempt to document what that means.

In short, the modular crypt format (MCF) is a standard for encoding password hash strings, which requires hashes have the format `$`*`identifier`*`$`*`content`*; where *`identifier`* is an short alphanumeric string uniquely identifying a particular scheme, and *`content`* is the contents of the scheme, using only the characters in the regexp range `[a-zA-Z0-9./]`.

However, there's no official specification document describing this format. Nor is there a central registry of identifiers, or actual rules. The modular crypt format is more of an ad-hoc idea rather than a true standard.

The rest of this page is an attempt to describe what is known, at least as far as the hashes supported by Passlib.

4.2.2 History

Historically, most unix systems supported only *`des_crypt`*. Around the same time, many incompatible variations were also developed, but their hashes were not easily distinguishable from each other (see *Archaic Unix Hashes*); making it impossible to use multiple hashes on one system, or progressively migrate to a newer scheme.

This was solved with the advent of the MCF, which was introduced around the time that *md5_crypt* was developed. This format allows hashes from multiple schemes to exist within the same database, by requiring that all hash strings begin with a unique prefix using the format $*identifier*$.

4.2.3 Requirements

Unfortunately, there is no specification document for this format. Instead, it exists in *de facto* form only; the following is an attempt to roughly identify the conventions followed by the modular crypt format hashes found in Passlib:

1. Hash strings should use only 7-bit ascii characters.

   No known OS or application generates hashes which violate this rule. However, some systems (e.g. Linux) will happily accept hashes which contain 8-bit characters in their salt, This is probably a case of "permissive in what you accept, strict in what you generate".

2. Hash strings should start with the prefix $*identifier*$, where *identifier* is a short string uniquely identifying hashes generated by that algorithm, using only lower case ascii letters, numbers, and hyphens (c.f. the list of *known identifiers* below).

   When MCF was first introduced, most schemes choose a single digit as their identifier (e.g. 1 for *md5_crypt*). Because of this, some older systems only look at the first character when attempting to distinguish hashes. However, as Unix variants have branched off, new schemes were developed which used larger identifying strings (e.g. $sha1$ for *sha1_crypt*).

   At this point, any new hash schemes should probably use a 6-8 character descriptive identifier, to avoid potential namespace clashes.

3. Hashes should only contain the ascii letters a-z and A-Z, ascii numbers 0-9, and the characters ./; though additionally they may use the $ character as an internal field separator.

   This is the least adhered-to of any modular crypt format convention. Other characters (such as +=,-) are used by various formats.

   The only hard and fast stricture is that :;!* and all non-printable or 8-bit characters be avoided, since this would interfere with parsing of the Unix shadow password file, where these hashes are typically stored.

   Pretty much all older modular-crypt-format hashes use ascii letters, numbers, ., and / to provide base64 encoding of their raw data, though the exact character value assignments vary between hashes (see passlib.utils.h64). Many newer hashes use + instead of ., to adhere closer to the base64 standard.

4. Hash schemes should put their "digest" portion at the end of the hash, preferably separated by a $.

   This allows password hashes to be easily truncated to a "configuration string" containing just the identifying prefix, rounds, salt, etc.

   This configuration string then encodes all the information generated needed to generate a new hash in order to verify a password, without having to perform excessive parsing.

   Most modular crypt format hashes follow this convention, though some (like *bcrypt*) omit the $ separator between the configuration and the digest.

   Furthermore, there is no set standard about whether configuration strings should or should not include a trailing $ at the end, though the general rule is that hashing should behave the same in either case (*sun_md5_crypt* behaves particularly poorly regarding this last point).

Note: All of the above is guesswork based on examination of existing hashes and OS implementations; and was written merely to clarify the issue of what the "modular crypt format" is. It is drawn from no authoritative sources.

4.2.4 Identifiers & Platform Support

4.2.4.1 OS Defined Hashes

The following table lists of all the major MCF hashes supported by Passlib, and indicates which operating systems offer native support:

Scheme	Prefix	Linux	FreeBSD	NetBSD	OpenBSD	Solaris
des_crypt		y	y	y	y	y
bsdi_crypt	_		y	y	y	
md5_crypt	1	y	y	y	y	y
bcrypt	2, $2a$, $2x$, $2y$ $2b$		y	y	y	y
bsd_nthash	3		y			
sha256_crypt	5	y	8.3+			y
sha512_crypt	6	y	8.3+			y
sun_md5_crypt	$md5$, $md5,					y
sha1_crypt	$sha1$			y		

Note: Linux systems using libxcrypt instead of `libcrypt` will have native support for additional formats, including nearly all those listed above.

4.2.4.2 Additional Platforms

The modular crypt format is also supported to some degree by the following operating systems and platforms:

MacOS X	Darwin's native `crypt()` provides limited functionality, supporting only *des_crypt* and *bsdi_crypt*. OS X uses a separate system for its own password hashes.
Google App Engine	As of 2011-08-19, Google App Engine's `crypt()` implementation appears to match that of a typical Linux system (as listed in the previous table).

4.2.4.3 Application-Defined Hashes

The following table lists the other MCF hashes supported by Passlib. These hashes can be found in various libraries and applications (and are not natively supported by any known OS):

Scheme	Prefix	Primary Use (if known)
apr_md5_crypt	`$apr1$`	Apache htdigest files
argon2	`$argon2i$`, `$argon2d$`	
bcrypt_sha256	`$bcrypt-sha256$`	Passlib-specific
phpass	`$P$`, `$H$`	PHPass-based applications
pbkdf2_sha1	`$pbkdf2$`	Passlib-specific
pbkdf2_sha256	`$pbkdf2-sha256$`	Passlib-specific
pbkdf2_sha512	`$pbkdf2-sha512$`	Passlib-specific
scram	`$scram$`	Passlib-specific
cta_pbkdf2_sha1	`$p5k2$`[1]	
dlitz_pbkdf2_sha1	`$p5k2$`[1]	
scrypt	`$scrypt$`	Passlib-specific

4.3 Release History

See also:

For the latest release: see *What's New* in Passlib 1.7

Warning: Passlib 1.8 will drop support for Python 2.x, 3.3, and 3.4; and will require Python >= 3.5. The 1.7 series will be the last to support Python 2.7. (See issue 119 for rationale).

4.3.1 Passlib 1.7

Warning: Passlib 1.8 will drop support for Python 2.x, 3.3, and 3.4; and will require Python >= 3.5. The 1.7 series will be the last to support Python 2.7. (See issue 119 for rationale).

4.3.1.1 1.7.4 (2020-10-08)

Small followup to 1.7.3 release.

Bugfixes

- Fixed some Python 2.6 errors from last release (issue 128)

Other Changes

- *passlib.ext.django* – updated tests to pass for Django 1.8 - 3.1 (issue 98); along with some internal refactoring of the test classes.
- *CryptContext* will now throw *UnknownHashError* when it can't identify a hash provided to methods such as `CryptContext.verify()`. Previously it would throw a generic `ValueError`.

[1] `cta_pbkdf2_sha1` and `dlitz_pbkdf2_sha1` both use the same identifier. While there are other internal differences, the two can be quickly distinguished by the fact that cta hashes always end in =, while dlitz hashes contain no = at all.

Deprecations

- *passlib.ext.django*: This extension will require Django 2.2 or newer as of Passlib 1.8.

4.3.1.2 1.7.3 (2020-10-06)

This release rolls up assorted bug & compatibility fixes since 1.7.2.

Administrative Changes

Note: Passlib has moved to Heptapod!

Due to BitBucket deprecating Mercurial support, Passlib's public repository and issue tracker has been relocated. It's now located at https://foss.heptapod.net/python-libs/passlib, and is powered by Heptapod.

Hosting for this and other open-source projects graciously provided by the people at Octobus and CleverCloud!

The mailing list and documentation urls remain the same.

New Features

- *ldap_salted_sha512*: LDAP "salted hash" support added for SHA-256 and SHA-512 (issue 124).

Bugfixes

- *bcrypt*: Under python 3, OS native backend wasn't being detected on BSD platforms. This was due to a few internal issues in feature-detection code, which have been fixed.
- `passlib.utils.safe_crypt()`: Support `crypt.crypt()` unexpectedly returning bytes under Python 3 (issue 113).
- `passlib.utils.safe_crypt()`: Support `crypt.crypt()` throwing `OSError`, which can happen as of Python 3.9 (issue 115).
- *passlib.ext.django*: fixed lru_cache import (django 3 compatibility)
- `passlib.tests`: fixed bug where `HandlerCase.test_82_crypt_support()` wasn't being run on systems lacking support for the hasher being tested. This test now runs regardless of system support.

Other Changes

- *bcrypt_sha256*: Internal algorithm has been changed to use HMAC-SHA256 instead of plain SHA256. This should strengthen the hash against brute-force attempts which bypass the intermediary hash by using known-sha256-digest lookup tables (issue 114).
- *bcrypt*: OS native backend ("os_crypt") now raises the new *PasswordValueError* if password is provided as non-UTF8 bytes under python 3 (These can't be passed through, due to limitation in stdlib's `crypt.crypt()`). Prior to this release, it confusingly raised *MissingBackendError* instead.

  Also improved legacy bcrypt format workarounds, to support a few more UTF8 edge cases than before.

- Modified some internals to help run on FIPS systems (issue 116):

  In particular, when MD5 hash is not available, *hex_md5* will now return a dummy hasher which throws an error if used; rather than throwing an uncaught `ValueError` when an application attempts to import it. (Similar behavior added for the other unsalted digest hashes).

  Also, *lookup_hash()*'s `required=False` kwd was modified to report unsupported hashes via the *HashInfo.supported* attribute; rather than letting ValueErrors through uncaught.

  This should allow CryptContext instances to be created on FIPS systems without having a load-time error (though they will still receive an error if an attempt is made to actually *use* a FIPS-disabled hash).

- Internal errors calling stdlib's `crypt.crypt()`, or third party libraries, will now raise the new *InternalBackendError* (a RuntimeError); where previously it would raise an `AssertionError`.
- Various Python 3.9 compatibility fixes (including `NotImplemented`-related warning, issue 125)

4.3.1.3 1.7.2 (2019-11-22)

This release rolls up assorted bug & compatibility fixes since 1.7.1.

New Features

- *argon2*: Now supports Argon2 "ID" and "D" hashes (assuming new enough backend library). Now defaults to "ID" hashes instead of "I" hashes, but this can be overridden via `type` keyword. (issue 101)
- *scrypt*: Now uses python 3.6 stdlib's `hashlib.scrypt()` as backend, if present (issue 86).

Bugfixes

- Python 3.8 compatibility fixes
- *passlib.apache.HtpasswdFile*: Now generates bcrypt hashes using the `"$2y$"` prefix, which should work properly with Apache 2.4's `htpasswd` tool. Previous releases used the functionally equivalent `"$2b$"` prefix, which `htpasswd` was unable to read (issue 95).
- *passlib.totp*: The *TOTP.to_uri()* method now prepends the issuer to URI label, (per the KeyURI spec). This should fix some compatibility issues with older TOTP clients (issue 92)
- Fixed error in `argon2.parsehash()` (issue 97)
- **unittests**: `crypt()` unittests now account for linux systems running libxcrypt (such as recent Fedora releases)

Deprecations

Warning: Due to lack of `pip` and `venv` support, Passlib is no longer fully tested on Python 2.6 & 3.3. There are no known issues, and bugfixes against these versions will still be accepted for the Passlib 1.7.x series. However, **Passlib 1.8 will drop support for Python 2.x & 3.3,** and require Python >= 3.4.

- Support for Python 2.x & 3.3 is deprecated; and will be dropped in Passlib 1.8. *(2020-05-10: Updated to include all of Python 2.x; when 1.7.2 was released, only Python 2.6 / 3.3 support was deprecated)*
- *bcrypt*: `py-bcrypt` and `bcryptor` backends are deprecated, and support will be removed in Passlib 1.8. Please switch to the `bcrypt` backend.

Other Changes

- **setup.py**: now honors `$SOURCE_DATE_EPOCH` to help with reproducible builds
- *argon2*: Now throws helpful error if "argon2" package is actually an incompatible or supported version of argon2_cffi (issue 99).
- **documentation**: Various updates & corrections. building the documentation now requires Sphinx 1.6 or newer.

4.3.1.4 1.7.1 (2017-1-30)

This release rolls up assorted bug & compatibility fixes since 1.7.0.

Bugfixes

- *cisco_asa* and *cisco_pix*: Fixed a number of issues which under *certain conditions* caused prior releases to generate hashes that were unverifiable on Cisco systems.
- *PasswordHash.hash()* will now warn if passed any settings keywords. This usage was deprecated in 1.7.0, but warning wasn't properly enabled. See *Customizing the Configuration* for the preferred way to pass settings.
- **setup.py**: Don't append timestamp when run from an sdist. This should fix some downstream build issues.
- `passlib.tests.test_totp`: Test suite now traps additional errors that `datetime.utcfromtimestamp()` may throw under python 3, which should fix some test failures on architectures with rarer ILP sizes. It also works around Python 3.6 bug 29100.

Deprecations

- *CryptContext*: The `harden_verify` flag has been turned into a NOOP and deprecated. It will be removed in passlib 1.8 along with the already-deprecated `min_verify_time` (issue 83).

Other Changes

- `passlib.tests.utils`: General truncation policy details were hammered out, and additional hasher tests were added to enforce them.
- **documentation**: Various updates & corrections.

4.3.1.5 1.7.0 (2016-11-22)

Overview

Welcome to Passlib 1.7!

This release includes a number of new features, cleans up some long-standing design issues, and contains a number of internal improvements; all part of the roadmap towards a leaner and simpler Passlib 2.0.

Highlights include:

- Support for *argon2* and *scrypt* hashes.
- TOTP Two-Factor Authentications helpers in the *passlib.totp* module.

- The misnamed `PasswordHash.encrypt()` method has been renamed to `PasswordHash.hash()` (and the old alias deprecated). This is part of a much larger project to clean up passlib's password hashing API, see the *PasswordHash Tutorial* for a walkthrough.
- Large speedup of the internal PBKDF2 routines.
- Updated documentation

Requirements

- **Passlib now requires Python 2.6, 2.7, or >= 3.3**. Support for Python versions 2.5 and 3.0 through 3.2 have been dropped. Support for PyPy 1.x has also been dropped.
- The `passlib.ext.django` extension now requires Django 1.8 or better. Django 1.7 and earlier are no longer supported.

New Features

New Hashes

- *passlib.hash.argon2* – Support for the Argon2 password hash (issue 69).
- *passlib.hash.scrypt* – New password hash format which uses the SCrypt KDF (issue 8).
- *passlib.hash.cisco_asa* – Support for Cisco ASA 7.0 and newer hashes (issue 51). *Note: this should be considered experimental, and needs verification of it's test vectors.*

New Modules

- New `passlib.totp` module provides full support for TOTP tokens on both client and server side. This module contains both low-level primitives, and high-level helpers for persisting and tracking client state.
- New `passlib.pwd` module added to aid in password generation. Features support for alphanumeric passwords, or generation of phrases using the EFF's password generation wordlist.

CryptContext Features

- The `CryptContext` object now has helper methods for dealing with hashes representing *disabled accounts* (issue 45).
- All hashers which truncate passwords (e.g. `bcrypt` and `des_crypt`) can now be configured to raise a `PasswordTruncateError` when a overly-large password is provided. This configurable via (for example) `bcrypt.using(truncate_error=True).hash(secret)`, or globally as an option to CryptContext (issue 59).

Cryptographic Backends

- The `pbkdf2_hmac()` function and all PBKDF2-based hashes have been sped up by ~20% compared to Passlib 1.6. For an even greater speedup, it will now take advantage of the external fastpbk2 library, or stdlib's `hashlib.pbkdf2_hmac()` (when available).

Other Changes

Other changes of note in Passlib 1.7:

- New workflows have been for configuring the hashers through `PasswordHash.using()`, and testing hashes through `PasswordHash.needs_update()`. See the *PasswordHash Tutorial* for a walkthrough.
- `bcrypt` and `bcrypt_sha256` now default to the "2b" format.

- Added support for Django's Argon2 wrapper (*django_argon2*)
- *passlib.apache.HtpasswdFile* has been updated to support all of Apache 2.4's hash schemes, as well as all host OS crypt formats; allowing for much more secure hashes in htpasswd files.

  You can now specify if the default hash should be compatible with apache 2.2 or 2.4, and host-specific or portable. See the `default_schemes` keyword for details.
- Large parts of the documentation have been rewritten, to separate tutorial & api reference content, and provide more detail on various features.
- Official documentation is now at https://passlib.readthedocs.io

Internal Changes

- The majority of CryptContext's internal rounds handling & migration code has been moved to the password hashes themselves, taking advantage of the new *PasswordHash.using()* and *PasswordHash.needs_update()* methods.

  This allows much more flexibility when configuring a hasher directly, as well making it easier for CryptContext to support hash-specific parameters.
- The shared `PasswordHash` unittests now check all hash handlers for basic thread-safety (motivated by the pybcrypt 0.2 concurrency bug).
- *consteq()* is now wraps stdlib's `hmac.compare_digest()` when available (python 2.7.11, python 3.3 and up).

Bugfixes

- *bcrypt*: Passlib will now detect and work around a fatal concurrency bug in py-bcrypt 0.2 and earlier (a *PasslibSecurityWarning* will also be issued). Nevertheless, users are *strongly* encouraged to upgrade to py-bcrypt 0.3 or another bcrypt library if you are using the *bcrypt* hash.
- `CryptContext` instances now pass contextual keywords (such as *"user"*) to the hashes that support them, but ignore them for hashes that don't (issue 63).
- The *passlib.apache* htpasswd helpers now preserve blank lines and comments, rather than throwing a parse error (issue 73).
- *passlib.ext.django* and unittests: compatibility fixes for Django 1.9 / 1.10, and some internal refactoring (issue 68).
- The *django_disabled* hash now appends a 40-char alphanumeric string, to match Django's behavior.

Deprecations

As part of a long-range plan to restructure and simplify both the API and the internals of Passlib, a number of methods have been deprecated & replaced. The eventually goal is a large cleanup and overhaul as part of Passlib 2.0. There will be at least one more 1.x version before Passlib 2.0, to provide a final transitional release (see the Project Roadmap).

Password Hash API Deprecations

As part of this cleanup, the *PasswordHash* API (used by all hashes in passlib), has had a number of changes:

See also:

PasswordHash Tutorial, which walks through using the new hasher interface.

- **[major]** The `PasswordHash.encrypt()` method has been renamed to *PasswordHash.hash()*, to clarify that it's performing one-way hashing rather than reversiable encryption. A compatibility alias will remain in place until Passlib 2.0. This should fix the longstanding issue 21.
- **[major]** Passing explicit configuration options to the `PasswordHash.encrypt()` method (now called *PasswordHash.hash()*) is deprecated. To provide settings such as `rounds` and `salt_size`, callers should use the new *PasswordHash.using()* method, which generates a new hasher with a customized configuration. For example, instead of:

```
>>> sha256_crypt.encrypt("secret", rounds=12345)
```

  ... applications should now use:

```
>>> sha256_crypt.using(rounds=12345).hash("secret")
```

  Support for the old syntax will be removed in Passlib 2.0.

  Note: This doesn't apply to contextual options such as *cisco_pix*'s `user` keyword, which should still be passed into the `hash()` method.

- **[minor]** The little-used *PasswordHash.genhash()* and *PasswordHash.genconfig()* methods have been deprecated. Compatibility aliases will remain in place until Passlib 2.0, at which point they will be removed entirely.

Crypt Context API Deprecations

Applications which use passlib's *CryptContext* should not be greatly affected by this release; only one major deprecation was made:

- **[major]** To match the `PasswordHash` API changes above, the `CryptContext.encrypt()` method was renamed to *CryptContext.hash()*. A compatibility alias will remain until Passlib 2.0.

A fewer internal options and infrequently used features have been deprecated:

- **[minor]** *CryptContext.hash()*, *verify()*, *verify_and_update()*, and *needs_update()*: The `scheme` keyword is now deprecated; support will be removed in Passlib 2.0.
- **[minor]** *CryptContext.hash()*: Passing settings keywords to `hash()` such as `rounds` and `salt` is deprecated. Code should now get ahold of the default hasher, and invoke it explicitly:

```
>>> # for example, calls that did this:
>>> context.hash(secret, rounds=1234)

>>> # should use this instead:
>>> context.handler().using(rounds=1234).hash(secret)
```

- **[minor]** The `vary_rounds` option has been deprecated, and will be removed in Passlib 2.0. It provided very little security benefit, and was judged not worth the additional code complexity it requires.
- **[minor]** The special wildcard `all` scheme name has been deprecated, and will be removed in Passlib 2.0. The only legitimate use was to support `vary_rounds`, which itself will be removed in 2.0.

Other Deprecations

A few other assorted deprecations have been made:

- The `passlib.utils.generate_secret()` function has been deprecated in favor of the new *passlib.pwd* module, and the old function will be removed in Passlib 2.0.
- Most of passlib's internal cryptography helpers have been moved from *passlib.utils* to *passlib.crypto*, and the APIs refactored. This allowed unification of various hash management routines, some speed ups to the HMAC and PBKDF2 primitives, and opens up the architecture to support more optional backend libraries.

  Compatibility wrappers will be kept in place at the old location until Passlib 2.0.
- Some deprecations and internal changes have been made to the *passlib.utils.handlers* module, which provides the common framework Passlib uses to implement hashers.

Caution: More backwards-incompatible relocations are planned for the internal `passlib.utils` module in the Passlib 1.8 / 1.9 releases.

Backwards Incompatibilities

Changes in existing behavior:

- **[minor]** M2Crypto no longer used to accelerate pbkdf2-hmac-sha1; applications relying on this to speed up *pbkdf2_sha1* should install fastpbkdf2.

Scheduled removal of features:

- **[minor]** *passlib.context*: The *min_verify_time* keyword that was deprecated in release 1.6, is now completely ignored. Support will be removed entirely in release 1.8.
- **[trivial]** *passlib.hash*: The internal `PasswordHash.parse_rounds()` method, deprecated in 1.6, has been removed.

Minor incompatibilities:

- **[minor]** *passlib.hash*: The little-used method *genconfig()* will now always return a valid hash, rather than a truncated configuration string or `None`.
- **[minor]** *passlib.hash*: The little-used method *genhash()* no longer accepts `None` as a config argument.
- **[trivial]** *passlib.utils.pbkdf2.pbkdf2()* no longer supports custom PRF callables. this was an unused feature, and prevented some useful optimizations.

4.3.2 Passlib 1.6

4.3.2.1 1.6.5 (2015-08-04)

Fixed some minor bugs in the test suite which were causing erroneous test failures (issue 57 and issue 58). The passlib library itself is unchanged.

4.3.2.2 1.6.4 (2015-07-25)

This release rolls up assorted bug & compatibility fixes since 1.6.2.

Bugfixes

- Correctly detect bcrypt 2.0. Previous releases were incorrectly detecting it as py-bcrypt, causing spurious errors (issue 56).
- CryptContext now accepts scheme names as unicode (issue 54).
- *passlib.ext.django* now works correctly with Django 1.7-1.8. Previous releases had various test failures (issue 52).
- *passlib.apache.HtpasswdFile* now recognizes bcrypt, sha256_crypt, sha512_crypt hashes (issue 55).

BCrypt Changes

A few changes have been made to the *bcrypt* hash:

- It now supports the `$2b$` hash format.
- It will now issue a *PasslibSecurityWarning* if the active backend is vulnerable to the *wraparound bug*, and automatically enable a workaround (py-bcrypt is known to be vulnerable as of v0.4).
- It will throw a *PasslibSecurityError* if the active backend is vulnerable to the *8-bit bug* (none of Passlib's backends are known to be vulnerable as of 2015-07).
- Updated documentation to indicate the cffi-based bcrypt library is now the recommended bcrypt backend.
- Backend capability detection code refactored to rely on runtime detection rather than hardcoded information.

Other Changes

- Source repo's `tox.ini` updated. Now assumes python3 by default, and refactored test environments to more cleanly delineate the different setups being tested.
- Passlib releases are now published as wheels instead of eggs.

4.3.2.3 1.6.3 (2015-07-25)

This was relabeled as **1.6.4** due to PyPI upload issues.

4.3.2.4 1.6.2 (2013-12-26)

Minor changes & compatibility fixes

- Re-tuned the *default_rounds* values for all of the hashes.
- Added the new *bcrypt_sha256* hash, which wraps BCrypt using SHA256 in order to work around BCrypt's password size limitations (issue 43).
- *passlib.hash.bcrypt*: Added support for the bcrypt library as one of the possible bcrypt backends that will be used if available. (issue 49)
- *passlib.ext.django*: Passlib's Django extension (and it's related hashes and unittests) have been updated to handle some minor API changes in Django 1.5-1.6. They should now be compatible with Django 1.2 and up. (issue 50)

4.3.2.5 1.6.1 (2012-08-02)

Minor bugfix release

- *bugfix*: Various *CryptContext* methods would incorrectly raise TypeError if passed a unicode user category under Python 2. For consistency, unicode user category values are now encoded to utf-8 bytes under Python 2.
- *bugfix*: Reworked internals of the CryptContext config compiler to fix a couple of border cases (issue 39):
  - It will now throw a ValueError if the *default* scheme is marked as *deprecated*.
  - If no default scheme is specified, it will use the first *non-deprecated* scheme.
  - Finally, it will now throw a ValueError if all schemes are marked as deprecated.
- *bugfix*: FreeBSD 8.3 added native support for *sha256_crypt* – updated Passlib's unittests and documentation accordingly (issue 35).
- *bugfix:* Fixed bug which caused some passlib.apache unittests to fail if mtime resolution >= 1 second (issue 35).
- *bugfix:* Fixed minor bug in passlib.registry, should now work correctly under Python 3.3.
- Various documentation updates and corrections.

4.3.2.6 1.6.0 (2012-05-01)

Overview

Welcome to Passlib 1.6.

The main goal of this release was to clean up the codebase, tighten input validation, and simplify the publically exposed interfaces. This release also brings a number of other improvements: 10 or so new hash algorithms, additional security precautions for the existing algorithms, a number of speed improvements, and updated documentation.

Deprecated APIs

In order to improve the publically exposed interface, some of the more cumbersome and less-used functions in Passlib have been deprecated / renamed. This should not affect 99% of applications. That said, all the deprecated interfaces are still present, and will continue to be supported for at least one more major release. To help with migration, all deprecated functions should issue an informative DeprecationWarning when they are invoked, detailing their suggested replacement. The following interfaces have changed:

- The semi-internal CryptPolicy class has been deprecated in its entirety. All functionality has been rolled into the parent CryptContext class (see *below* for more).
- The interface of the *passlib.apache* classes has been improved: some confusing methods and options have been renamed, some new constructors and other functions have been added.
- The (undocumented) passlib.win32 module has been deprecated, all of its functionality is now offered through the *lmhash* and *nthash* algorithms.

New Hashes

The release adds support for a number of hash algorithms:

cisco_pix*, *cisco_type7 Two hash formats frequently found on various Cisco devices *(for Cisco Type 5 hashes, see md5_crypt).*

django_pbkdf2_sha256*, *django_pbkdf2_sha1*, *django_bcrypt All three of the new hash schemes introduced in Django 1.4.

lmhash*, *nthash Microsoft's legacy "Lan Manager" hash, and the replacement NT password hash. *(the old* `nthash` *algorithm in Passlib 1.5 has been renamed to* `bsd_nthash` *, to reflect its lineage).*

msdcc*, *msdcc2 Microsoft Windows' Domain Cached Credentials, versions 1 and 2. These algorithms also go by the names "DCC", "MSCache", and "MSCash".

mssql2000*, *mssql2005 Hash algorithms used by MS SQL Server 2000 and later.

scram A hash format added specifically for storing the complex digest information needed to authenticate a user via the SCRAM protocol (**RFC 5802**). It can also be used in the same way as any other password hash in Passlib.

Existing Hashes

Additionally, the following new features have been added to the existing hashes:

Password Size Limit All hashes in Passlib will now throw `PasswordSizeError` if handed a password that's larger than 4096 characters.

This limit should be larger than any reasonable password size, and prevents various things including DOS abuses, and exploitation of OSes with a buggy `crypt()` implementation. See `PasswordSizeError` for how to change this limit.

Constant Time Comparison All hash comparisons in Passlib now use the "constant time"[1] comparison function `consteq()`, instead of ==.

This change is motivated a well-known hmac timing attack which exploits short-circuit string comparisons. While this attack is not currently feasible against most password hashes, some of the weaker unsalted hashes supported by Passlib may be vulnerable; and this change has been made preventatively to all of them.

Strict Parameters Previous releases of Passlib would silently correct any invalid values (such as `rounds` parameters that were out of range). This is was deemed undesirable, as it leaves developers unaware they are requesting an incorrect (and potentially insecure) value.

Starting with this release, providing invalid values to `PasswordHash.encrypt` will result in a `ValueError`. However, most hashes now accept an optional `relaxed=True` keyword, which causes Passlib to try and correct invalid values, and if successful, issue a `PasslibHashWarning` instead. These warnings can then be filtered if desired.

bcrypt The BCrypt hash now supports the crypt_blowfish project's `$2y$` hash prefix.

On an unrelated note, Passlib now offers an (experimental) pure-python implementation of BCrypt. Unfortunately, it's still *WAY* too slow to be suitable for production use; and is disabled by default. If you really need it, see the BCrypt *documentation* for how to enable it.

bsdi_crypt BSDi-Crypt will now issue a `PasslibSecurityWarning` if an application requests an even number of rounds, due to a known weakness in DES. Existing hashes with an even number of rounds will now be flagged by `CryptContext.needs_update()`.

[1] "constant time" is a misnomer, it actually takes `THETA(len(righthand_value))` time.

ldap_salted_{digest} The LDAP salted digests now support salts of any size from 4-16 bytes, though they still default to 4 (issue 30).

md5_crypt, *sha256_crypt*, *sha512_crypt* The builtin implementation of these hashes has been sped up by about 25%, using an additional pre-computation step.

unix_disabled The `unix_fallback` handler has been deprecated, and will be removed in Passlib 1.8. Applications should use the stricter-but-equivalent `unix_disabled` handler instead.

This most likely only affects internal Passlib code.

CryptContext

The *CryptContext* class has had a thorough internal overhaul. While the primary interface has not changed at all, the internals are much stricter about input validation, common methods have shorter code-paths, and the construction and introspection of `CryptContext` objects has been greatly simplified. Changes include:

- All new (and hopefully clearer) *tutorial* and *reference* documentation.
- The *`CryptPolicy`* class and the `CryptContext.policy` attribute have been deprecated.

  This was a semi-internal class, which most applications were not involved with at all, but to be conservative about breaking things, the existing CryptPolicy interface will remain in-place and supported until Passlib 1.8.

  All of the functionality of this class has been rolled into `CryptContext` itself, so there's one less class to remember. Many of the methods provided by `CryptPolicy` are now `CryptContext` methods, most with the same name and call syntax. Information on migrating existing code can be found in the deprecation warnings issued by the class itself, and in the *`CryptPolicy`* documentation.
- Two new class constructors have been added (*`CryptContext.from_path()`* and *`CryptContext.from_string()`*) to aid in loading CryptContext objects directly from a configuration file.
- The *deprecated* keyword can now be set to the special string `"auto"`; which will automatically deprecate all schemes except for the default one.
- The *min_verify_time* keyword has been deprecated, will be ignored in release 1.7, and will be removed in release 1.8. It was never very useful, and now complicates the internal code needlessly.
- All string parsing now uses stdlib's `SafeConfigParser`.

  Previous releases used the original `ConfigParser` interpolation; which was deprecated in Passlib 1.5, and has now been removed. This should only affect strings which contained raw `%` characters, they will now need to be escaped via `%%`.

Other Modules

- The api for the *`passlib.apache`* module has been updated to add more flexibility, and to fix some ambiguous method and keyword names. The old interface is still supported, but deprecated, and will be removed in Passlib 1.8.
- Added the *`django14_context`* preset to the the `passlib.apps` module. this preconfigured CryptContext object should support all the hashes found in a typical Django 1.4 deployment.

- **new**: Added *passlib.ext.django*, a Django plugin which can be used to override Django's password hashing framework with a custom Passlib policy (an undocumented beta version of this was present in the 1.5 release).
- **new**: The *passlib.utils.saslprep()* function may be useful for applications which need to normalize the unicode representation of passwords before they are hashed.

Bugfixes

- Handle platform-specific error strings that may be returned by the `crypt()` methods of some OSes.
- Fixed rare `'NoneType' object has no attribute 'decode'` error that sometimes occurred on platforms with a deviant implementation of `crypt()`.

Internal Changes

The following changes should not affect most end users, and have been documented just to keep track of them:

- Passlib is now source-compatible with Python 2.5+ and Python 3.x. It no longer requires the use of the **2to3** command to translate it for Python 3.
- The unittest suite has been rewritten. It handles a number of additional border cases, enforcing uniform behavior across all hashes, and even features the addition of some simplistic fuzz testing. It will take a bit longer to run though. While not perfect, statement coverage is at about 95%. Additionally, the hash test suite has been enhanced with many more test vectors across the board, including 8-bit test vectors.
- The internal framework used to construct the hash classes (*passlib.utils.handlers*) was rewritten drastically. The new version provides stricter input checking, reduction in boilerplate code. *These changes should not affect any publically exposed routines*.
  - *GenericHandler*'s `strict` keyword was removed, `strict=True` is now the class's default behavior: all values must be specified, and be within the correct bounds. The new keywords `use_defaults` and `relaxed` can be used to disable these two requirements.
  - Most of the private methods of *GenericHandler* were renamed to begin with an underscore, to clarify their status; and turned into instance methods, to simplify the internals. (for example, `norm_salt` was renamed to `_norm_salt`).
  - *StaticHandler* now derives from `GenericHandler`, and requires `_calc_checksum()` be implemented instead of `encrypt()`. The old style is supported but deprecated, and support will be removed in Passlib 1.8.
  - Calls to `HasManyBackends.set_backend()` should now use the string `"any"` instead of the value `None`. `None` was deprecated in release 1.5, and is no longer supported.
- `passlib.utils.h64` has been replaced by an instance of the new *Base64Engine* class. This instance is imported under the same name, and has (mostly) the same interface; but should be faster, more flexible, and better unit-tested.
- deprecated some unused support functions within `passlib.utils`, they will be removed in release 1.7.

4.3.3 Passlib 1.5

4.3.3.1 1.5.3 (2011-10-08)

Bugfix release – fixes BCrypt padding/verification issue (issue 25)

This release fixes a single issue with Passlib's BCrypt support: Many BCrypt hashes generated by Passlib (<= 1.5.2) will not successfully verify under some of the other BCrypt implementations, such as OpenBSD's `/etc/master.passwd`.

In detail:

> BCrypt hashes contain 4 "padding" bits in the encoded salt, and Passlib (<= 1.5.2) generated salts in a manner which frequently set some of the padding bits to 1. While Passlib ignores these bits, many BCrypt implementations perform password verification in a way which rejects *all* passwords if any of the padding bits are set. Thus Passlib's BCrypt salt generation needed to be fixed to ensure compatibility, and a route provided to correct existing hashes already out in the wild issue 25.

Changes in this release:

- BCrypt hashes generated by Passlib now have all padding bits cleared.
- Passlib will continue to accept BCrypt hashes that have padding bits set, but when it encounters them, it will issue a `UserWarning` recommending that the hash should be fixed (see below).
- Applications which use *`CryptContext.verify_and_update()`* will have any such hashes automatically re-encoded the next time the user logs in.

To fix existing hashes:

> If you have BCrypt hashes which might have their padding bits set, you can import `passlib.hash.bcrypt`, and call `clean_hash = bcrypt.normhash(hash)`. This function will clear the padding bits of any BCrypt hashes, and should leave all other strings alone.

4.3.3.2 1.5.2 (2011-09-19)

Minor bugfix release – mainly Django-related fixes

Hashes

- *bugfix:* *`django_des_crypt`* now accepts all *`hash64`* characters in its salts; previously it accepted only lower-case hexadecimal characters (issue 22).
- Additional unittests added for all standard *Django hashes*.
- *`django_des_crypt`* now rejects hashes where salt and checksum containing mismatched salt characters.

CryptContext

- *bugfix:* fixed exception in *`CryptPolicy.iter_config()`* that occurred when iterating over deprecation options.
- Added documentation for the (mistakenly undocumented) *`CryptContext.verify_and_update()`* method.

4.3.3.3 1.5.1 (2011-08-17)

Minor bugfix release – now compatible with Google App Engine.

- *bugfix:* make `passlib.hash.__loader__` attribute writable - needed by Google App Engine (GAE) issue 19.
- *bugfix:* provide fallback for loading `passlib/default.cfg` if `pkg_resources` is not present, such as for GAE issue 19.
- *bugfix:* fixed error thrown by CryptContext.verify when issuing min_verify_time warning issue 17.
- removed min_verify_time setting from custom_app_context, min_verify_time is too host & load dependant to be hardcoded issue 17.
- under GAE, disable all unittests which require writing to filesystem.
- more unittest coverage for *passlib.apps* and *passlib.hosts*.
- improved version datestamps in build script.

4.3.3.4 1.5.0 (2011-07-11)

"20% more unicode than the leading breakfast cereal"

The main new feature in this release is that Passlib now supports Python 3 (via the 2to3 tool). Everything has been recoded to have better separation between unicode and bytes, and to use unicode internally where possible. When run under Python 2, Passlib 1.5 attempts to provide the same behavior as Passlib 1.4; but when run under Python 3, most functions will return unicode instead of ascii bytes.

Besides this major change, there have been some other additions:

Hashes

- added support for Cryptacular's PBKDF2 format.
- added support for the FSHP family of hashes.
- added support for using BCryptor as BCrypt backend.
- added support for all of Django's hash formats.

CryptContext

- interpolation deprecation:

  CryptPolicy.from_path() and *CryptPolicy.from_string()* now use `SafeConfigParser` instead of `ConfigParser`. This may cause some existing config files containing unescaped `%` to result in errors; Passlib 1.5 will demote these to warnings, but any extant config files should be updated, as the errors will be fatal in Passlib 1.6.
- added encoding keyword to `CryptPolicy`'s `.from_path()`, `.from_string()`, and `.to_string()` methods.
- both classes in *passlib.apache* now support specifying an encoding for the username/realm.

Documentation

- Password Hash API expanded to include explicit *unicode vs bytes policy*.
- Added quickstart guide to documentation.

- Various minor improvements.

Internal Changes

- Added more handler utility functions to reduce code duplication.
- Expanded kdf helpers in `passlib.utils.pbkdf2`.
- Removed deprecated parts of *`passlib.utils.handlers`*.
- Various minor changes to *`passlib.utils.handlers.HasManyBackends`*; main change is that multi-backend handlers now raise *`MissingBackendError`* if no backends are available.
- Builtin tests now use `unittest2` if available.
- Setup script no longer requires distribute or setuptools.
- added (undocumented, experimental) Django app for overriding Django's default hash format, see `docs/lib/passlib.ext.django.rst` for more.

4.3.4 Passlib 1.4 & Earlier

4.3.4.1 1.4 (2011-05-04)

This release contains a large number of changes, both large and small. It adds a number of PBKDF2-based schemes, better support for LDAP-format hashes, improved documentation, and faster load times. In detail...

Hashes

- added LDAP `{CRYPT}` support for all hashes known to be supported by OS crypt()
- added 3 custom PBKDF2 schemes for general use, as well as 3 LDAP-compatible versions.
- added support for Dwayne Litzenberger's PBKDF2 scheme.
- added support for Grub2's PBKDF2 hash scheme.
- added support for Atlassian's PBKDF2 password hash
- added support for all hashes used by the Roundup Issue Tracker
- bsdi_crypt, sha1_crypt now check for OS crypt() support
- `salt_size` keyword added to encrypt() method of all the hashes which support variable-length salts.
- security fix: disabled unix_fallback's "wildcard password" support unless explicitly enabled by user.

CryptContext

- host_context now dynamically detects which formats OS crypt() supports, instead of guessing based on sys.platform.
- added predefined context for Roundup Issue Tracker database.
- added CryptContext.verify_and_update() convenience method, to make it easier to perform both operations at once.
- *bugfix:* fixed NameError in category+min_verify_time border case

- apps & hosts modules now use new `LazyCryptContext` wrapper class - this should speed up initial import, and reduce memory by not loading unneeded hashes.

Documentation

- greatly expanded documentation on how to use CryptContexts.
- roughly documented framework for writing & testing custom password handlers.
- various minor improvements.

Internals

- added generate_password() convenience method
- refactored framework for building hash handlers, using new mixin-based system.
- deprecated old handler framework - will remove in 1.5
- deprecated list_to_bytes & bytes_to_list - not used, will remove in 1.5

Other

- password hash api - as part of cleaning up optional attributes specification, renamed a number of them to reduce ambiguity:
  - renamed *{xxx}_salt_chars* attributes -> *xxx_salt_size*
  - renamed *salt_charset* -> *salt_chars*
  - old attributes still present, but deprecated - will remove in 1.5
- password hash api - tightened specifications for salt & rounds parameters, added support for hashes w/ no max salt size.
- improved password hash api conformance tests
- PyPy compatibility

4.3.4.2 1.3.1 (2011-03-28)

Minor bugfix release.

- bugfix: replaced "sys.maxsize" reference that was failing under py25
- bugfix: fixed default_rounds>max_rounds border case that could cause ValueError during CryptContext.encrypt()
- minor documentation changes
- added instructions for building html documentation from source

4.3.4.3 1.3 (2011-03-25)

First public release.

- documentation completed
- 99% unittest coverage
- some refactoring and lots of bugfixes
- added support for a number of additional password schemes: bigcrypt, crypt16, sun md5 crypt, nthash, lmhash, oracle10 & 11, phpass, sha1, generic hex digests, ldap digests.

4.3.4.4 1.2 (2011-01-06)

Note: For this and all previous versions, Passlib did not exist independently, but as a subpackage of *BPS*, a private & unreleased toolkit library.

- many bugfixes
- global registry added
- transitional release for applications using BPS library.
- first truly functional release since splitting from BPS library (see below).

4.3.4.5 1.0 (2009-12-11)

- CryptContext & CryptHandler framework
- added support for: des-crypt, bcrypt (via py-bcrypt), postgres, mysql
- added unit tests

4.3.4.6 0.5 (2008-05-10)

- initial production version
- consolidated from code scattered across multiple applications
- MD5-Crypt, SHA256-Crypt, SHA512-Crypt support

See also:

See the Project Roadmap for a list of future changes that may impact applications.

4.4 Copyrights & Licenses

4.4.1 Credits

Passlib is primarily developed by Eli Collins.

Special thanks to Darin Gordon for testing and feedback on the `passlib.totp` module.

4.4.2 License for Passlib

Passlib is (c) Assurance Technologies, and is released under the BSD license:

```
Passlib
Copyright (c) 2008-2020 Assurance Technologies, LLC.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

* Redistributions of source code must retain the above copyright
  notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.

* Neither the name of Assurance Technologies, nor the names of the
  contributors may be used to endorse or promote products derived
  from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

4.4.3 Licenses for incorporated software

Passlib contains some code derived from the following sources:

4.4.3.1 MD5-Crypt

The source file `passlib/handlers/md5_crypt.py` contains code derived from the original FreeBSD md5-crypt implementation, which is available under the following license:

```
"THE BEER-WARE LICENSE" (Revision 42):
<phk@login.dknet.dk> wrote this file.  As long as you retain this notice you
can do whatever you want with this stuff. If we meet some day, and you think
this stuff is worth it, you can buy me a beer in return.   Poul-Henning Kamp

converted to python May 2008
by Eli Collins
```

4.4.3.2 DES

The source file `passlib/crypto/des.py` contains code derived from UnixCrypt.java, a pure-java implementation of the historic unix-crypt password hash algorithm. It is available under the following license:

```
UnixCrypt.java 0.9 96/11/25
Copyright (c) 1996 Aki Yoshida. All rights reserved.
Permission to use, copy, modify and distribute this software
for non-commercial or commercial purposes and without fee is
hereby granted provided that this copyright notice appears in
all copies.

modified April 2001
by Iris Van den Broeke, Daniel Deville

modified Aug 2005
by Greg Wilkins (gregw)

converted to python Jun 2009
by Eli Collins
```

4.4.3.3 jBCrypt

The source file `passlib/crypto/_blowfish/base.py` contains code derived from jBcrypt 0.2, a Java implementation of the BCrypt password hash algorithm. It is available under a BSD/ISC license:

```
Copyright (c) 2006 Damien Miller <djm@mindrot.org>

Permission to use, copy, modify, and distribute this software for any
purpose with or without fee is hereby granted, provided that the above
copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTUOUS ACTION, ARISING OUT OF
OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

4.4.3.4 Wordsets

The EFF wordsets in `passlib/_data/wordsets` are (c) 2016 the Electronic Freedom Foundation. They were downloaded from https://www.eff.org/deeplinks/2016/07/new-wordlists-random-passphrases, and are released under the Creative Commons License.

- General Index
- Module List

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X