zope.password Documentation

Release 4.1

Zope Foundation and Contributors

Oct 18, 2018

Contents

1 Using `zope.password` 3
- 1.1 Password Manager Interfaces . . . 4
- 1.2 Looking Up Password Managers via a Vocabulary . . . 4
- 1.3 Encrypting Passwords with `zpasswd` . . . 5

2 `zope.password` API 7
- 2.1 Interfaces . . . 7
- 2.2 Password Manager Implementations . . . 7
- 2.3 Vocabulary . . . 18

3 Indices and tables 19

Python Module Index 21

Contents:

CHAPTER 1

Using `zope.password`

This package provides a password manager mechanism. Password manager is an utility object that can encode and check encoded passwords. Beyond the generic interface, this package also provides eight implementations:

zope.password.password.PlainTextPasswordManager

The most simple and the less secure one. It does not do any password encoding and simply checks password by string equality. It's useful in tests or as a base class for more secure implementations.

zope.password.password.MD5PasswordManager

A password manager that uses MD5 algorithm to encode passwords. It's generally weak against dictionary attacks due to a lack of a salt.

zope.password.password.SMD5PasswordManager

A password manager that uses MD5 algorithm, together with a salt to encode passwords. It's better protected against against dictionary attacks, but the MD5 hashing algorithm is not as strong as the SHA1 algorithm.

zope.password.password.SHA1PasswordManager

A password manager that uses SHA1 algorithm to encode passwords. It has the same weakness as the MD5PasswordManager.

zope.password.password.SSHAPasswordManager

A password manager that is strong against dictionary attacks. It's basically SHA1-encoding password manager which also incorporates a salt into the password when encoding it.

`zope.password.password.CryptPasswordManager`

A manager implementing the crypt(3) hashing scheme. Only available if the python crypt module is installed. This is a legacy manager, only present to ensure that zope.password can be used for all schemes defined in RFC 2307 (LDAP).

`zope.password.password.MySQLPasswordManager`

A manager implementing the digest scheme as implemented in the MySQL PASSWORD function in MySQL versions before 4.1. Note that this method results in a very weak 16-byte hash.

zope.password.password.BCRYPTPasswordManager

A manager implementing the bcrypt hashing scheme. Only available if the bcrypt module is installed. This manager is considered one of the most secure.

zope.password.password.BCRYPTKDFPasswordManager

A manager implementing the bcrypt_kdf hashing scheme. Only available if the bcrypt module is installed. This manager is considered one of the most secure.

The `Crypt`, `MD5`, `SMD5`, `SHA` and `SSHA` password managers are all compatible with RFC 2307 LDAP implementations of the same password encoding schemes.

Note: It is strongly recommended to use the BCRYPTPasswordManager or BCRYPTKDFPasswordManager, as they are the most secure.

The package also provides a script, **zpasswd**, to generate principal entries in typical `site.zcml` files.

1.1 Password Manager Interfaces

The *zope.password.interfaces.IPasswordManager* interface defines only two methods:

```
    def encodePassword(password):
        """Return encoded data for the given password

        Return encoded bytes.
        """
```

```
    def checkPassword(encoded_password, password):
        """Does the encoded password match the given password?

        Return True if they match, else False.
        """
```

An extended interface, *zope.password.interfaces.IMatchingPasswordManager*, adds one additional method:

```
    def match(encoded_password):
        """Was the given data was encoded with this manager's scheme?

        Return True when the given data was encoded with the scheme
        implemented by this password manager.
        """
```

1.2 Looking Up Password Managers via a Vocabulary

The *zope.password.vocabulary* module provides a vocabulary of registered password manager utility names. It is typically registered as an `zope.schema.interfaces.IVocabularyFactory` utility named "Password Manager Names".

It's intended to be used with `zope.component` and `zope.schema`, so you need to have them installed and the utility registrations needs to be done properly. The `configure.zcml` file contained in `zope.password` does the registrations, as well as in `zope.password.testing.setUpPasswordManagers()`.

1.3 Encrypting Passwords with `zpasswd`

zpasswd is a script to generate principal entries in typical `site.zcml` files.

You can create a **zpasswd** script in your buildout by adding a section like this to your `buildout.cfg`:

```
[zpasswd]
recipe = z3c.recipe.dev:script
eggs = zope.password
module = zope.password.zpasswd
method = main
```

This will generate a script **zpasswd** next time you run **buildout**.

When run, the script will ask you for all parameters needed to create a typical principal entry, including the encrypted password.

Use:

```
$ bin/zpasswd --help
```

to get a list of options.

Using

```
$ bin/zpasswd -c some/site.zcml
```

the script will try to lookup any password manager you defined and registered in your environment. This is lookup is not necessary if you go with the standard password managers defined in `zope.password`.

A typical **zpasswd** session might look like:

```
$ ./bin/zpasswd

Please choose an id for the principal.

Id: foo


Please choose a title for the principal.

Title: The Foo


Please choose a login for the principal.

Login: foo

Password manager:

 1. Plain Text
 2. MD5
 3. SMD5
 4. SHA1
 5. SSHA
 6. BCRYPT

Password Manager Number [6]:
BCRYPT password manager selected
```

(continues on next page)

(continued from previous page)

```
Please provide a password for the principal.

Password:
Verify password:

Please provide an optional description for the principal.

Description: The main foo

================================================
Principal information for inclusion in ZCML:

  <principal
    id="foo"
    title="The Foo"
    login="foo"
    password="{BCRYPT}$2b$12$ez4eHl6W1PfAWix5bPIbe.drdnyqjpuT1Cp0N.xcdxkAEbA7K6AHK"
    description="The main foo"
    password_manager="BCRYPT"
    />
```

CHAPTER 2

zope.password API

2.1 Interfaces

Password manager interface

interface zope.password.interfaces.**IPasswordManager**
Password manager utility interface.

encodePassword(*password*)
Return encoded data for the given password

Return encoded bytes.

checkPassword(*encoded_password*, *password*)
Does the encoded password match the given password?

Return True if they match, else False.

interface zope.password.interfaces.**IMatchingPasswordManager**
Extends: *zope.password.interfaces.IPasswordManager*

Password manager with hash matching support

match(*encoded_password*)
Was the given data was encoded with this manager's scheme?

Return True when the given data was encoded with the scheme implemented by this password manager.

2.2 Password Manager Implementations

Password managers

class zope.password.password.**PlainTextPasswordManager**
Bases: object

Plain text password manager.

```
>>> from zope.interface.verify import verifyObject
>>> from zope.password.interfaces import IMatchingPasswordManager
>>> from zope.password.password import PlainTextPasswordManager
```

```
>>> manager = PlainTextPasswordManager()
>>> verifyObject(IMatchingPasswordManager, manager)
True
```

```
>>> password = u"right \N{CYRILLIC CAPITAL LETTER A}"
>>> encoded = manager.encodePassword(password)
>>> encoded == password.encode('utf-8')
True
>>> manager.checkPassword(encoded, password)
True
>>> manager.checkPassword(encoded, password + u"wrong")
False
```

The plain text password manager *never* claims to implement the scheme, because this would open a security hole, where a hash from a different scheme could be used as-is as a plain-text password. Authentication code that needs to support plain-text passwords need to explicitly check for plain-text password matches after all other options have been tested for:

```
>>> manager.match(encoded)
False
```

class zope.password.password.**SSHAPasswordManager**

Bases: zope.password.password._PrefixedPasswordManager

SSHA password manager.

SSHA is basically SHA1-encoding which also incorporates a salt into the encoded string. This way, stored passwords are more robust against dictionary attacks of attackers that could get access to lists of encoded passwords.

SSHA is regularly used in LDAP databases and we should be compatible with passwords used there.

```
>>> from zope.interface.verify import verifyObject
>>> from zope.password.interfaces import IMatchingPasswordManager
>>> from zope.password.password import SSHAPasswordManager
```

```
>>> manager = SSHAPasswordManager()
>>> verifyObject(IMatchingPasswordManager, manager)
True
```

```
>>> password = u"right \N{CYRILLIC CAPITAL LETTER A}"
>>> encoded = manager.encodePassword(password, salt="")
>>> isinstance(encoded, bytes)
True
>>> print(encoded.decode())
{SSHA}BLTuxxVMXzouxtKVb7gLgNxzdAI=
```

```
>>> manager.match(encoded)
True
>>> manager.match(encoded.decode())
True
>>> manager.checkPassword(encoded, password)
True
```

(continues on next page)

(continued from previous page)

```
>>> manager.checkPassword(encoded, password + u"wrong")
False
```

Using the *slappasswd* utility to encode `secret`, we get `{SSHA}x3HIoiF9y6YRi/I4W1fkptbzTDiNr+9l` as seeded hash.

Our password manager generates the same value when seeded with the same salt, so we can be sure, our output is compatible with standard LDAP tools that also use SSHA:

```
>>> from base64 import standard_b64decode
>>> salt = standard_b64decode('ja/vZQ==')
>>> password = 'secret'
>>> encoded = manager.encodePassword(password, salt)
>>> isinstance(encoded, bytes)
True
>>> print(encoded.decode())
{SSHA}x3HIoiF9y6YRi/I4W1fkptbzTDiNr+9l
```

```
>>> manager.checkPassword(encoded, password)
True
>>> manager.checkPassword(encoded, password + u"wrong")
False
```

We can also pass a salt that is a text string:

```
>>> salt = u'salt'
>>> password = 'secret'
>>> encoded = manager.encodePassword(password, salt)
>>> isinstance(encoded, bytes)
True
>>> print(encoded.decode())
{SSHA}gVK8WC9YyFT1gMsQHTGCgT3sSv5zYWx0
```

Because a random salt is generated, the output of encodePassword is different every time you call it.

```
>>> manager.encodePassword(password) != manager.encodePassword(password)
True
```

The password manager should be able to cope with unicode strings for input:

```
>>> passwd = u'foobar\u2211' # sigma-sign.
>>> manager.checkPassword(manager.encodePassword(passwd), passwd)
True
>>> manager.checkPassword(manager.encodePassword(passwd).decode(), passwd)
True
```

The manager only claims to implement SSHA encodings, anything not starting with the string {SSHA} returns False:

```
>>> manager.match('{MD5}someotherhash')
False
```

An older version of this manager used the urlsafe variant of the base64 encoding (replacing / and + characters with _ and - respectively). Hashes encoded with the old manager are still supported:

```
>>> encoded = '{SSHA}x3HIoiF9y6YRi_I4W1fkptbzTDiNr-9l'
>>> manager.checkPassword(encoded, 'secret')
True
```

class zope.password.password.**SMD5PasswordManager**

Bases: zope.password.password._PrefixedPasswordManager

SMD5 password manager.

SMD5 is basically SMD5-encoding which also incorporates a salt into the encoded string. This way, stored passwords are more robust against dictionary attacks of attackers that could get access to lists of encoded passwords:

```
>>> from zope.interface.verify import verifyObject
>>> from zope.password.interfaces import IMatchingPasswordManager
>>> from zope.password.password import SMD5PasswordManager
```

```
>>> manager = SMD5PasswordManager()
>>> verifyObject(IMatchingPasswordManager, manager)
True
```

```
>>> password = u"right \N{CYRILLIC CAPITAL LETTER A}"
>>> encoded = manager.encodePassword(password, salt="")
>>> isinstance(encoded, bytes)
True
>>> print(encoded.decode())
{SMD5}ht3czsRdtFmfGsAAGOVBOQ==
```

```
>>> manager.match(encoded)
True
>>> manager.match(encoded.decode())
True
>>> manager.checkPassword(encoded, password)
True
>>> manager.checkPassword(encoded, password + u"wrong")
False
```

Using the slappasswd utility to encode secret, we get {SMD5}zChC6x0tl2zr9fjvjZzKePV5KWA= as seeded hash.

Our password manager generates the same value when seeded with the same salt, so we can be sure, our output is compatible with standard LDAP tools that also use SMD5:

```
>>> from base64 import standard_b64decode
>>> salt = standard_b64decode('9XkpYA==')
>>> password = 'secret'
>>> encoded = manager.encodePassword(password, salt)
>>> isinstance(encoded, bytes)
True
>>> print(encoded.decode())
{SMD5}zChC6x0tl2zr9fjvjZzKePV5KWA=
```

```
>>> manager.checkPassword(encoded, password)
True
>>> manager.checkPassword(encoded, password + u"wrong")
False
```

We can also pass a salt that is a text string:

```
>>> salt = u'salt'
>>> password = 'secret'
>>> encoded = manager.encodePassword(password, salt)
>>> isinstance(encoded, bytes)
True
>>> print(encoded.decode())
{SMD5}mc0uWpXVVe5747A4pKhGJXNhbHQ=
```

Because a random salt is generated, the output of encodePassword is different every time you call it.

```
>>> manager.encodePassword(password) != manager.encodePassword(password)
True
```

The password manager should be able to cope with unicode strings for input:

```
>>> passwd = u'foobar\u2211' # sigma-sign.
>>> manager.checkPassword(manager.encodePassword(passwd), passwd)
True
>>> manager.checkPassword(manager.encodePassword(passwd).decode(), passwd)
True
```

The manager only claims to implement SMD5 encodings, anything not starting with the string {SMD5} returns False:

```
>>> manager.match('{MD5}someotherhash')
False
```

class zope.password.password.**MD5PasswordManager**

Bases: zope.password.password._PrefixedPasswordManager

MD5 password manager.

```
>>> from zope.interface.verify import verifyObject
>>> from zope.password.interfaces import IMatchingPasswordManager
>>> from zope.password.password import MD5PasswordManager
```

```
>>> manager = MD5PasswordManager()
>>> verifyObject(IMatchingPasswordManager, manager)
True
```

```
>>> password = u"right \N{CYRILLIC CAPITAL LETTER A}"
>>> encoded = manager.encodePassword(password)
>>> isinstance(encoded, bytes)
True
>>> print(encoded.decode())
{MD5}ht3czsRdtFmfGsAAGOVBOQ==
>>> manager.match(encoded)
True
>>> manager.match(encoded.decode())
True
>>> manager.checkPassword(encoded, password)
True
>>> manager.checkPassword(encoded, password + u"wrong")
False
```

This password manager is compatible with other RFC 2307 MD5 implementations. For example the output of the slappasswd command for a MD5 hashing of `secret` is `{MD5}Xr4ilOzQ4PCOq3aQ0qbuaQ==`, and our implementation returns the same hash:

```
>>> print(manager.encodePassword('secret').decode())
{MD5}Xr4ilOzQ4PCOq3aQ0qbuaQ==
```

The password manager should be able to cope with unicode strings for input:

```
>>> passwd = u'foobar\u2211' # sigma-sign.
>>> manager.checkPassword(manager.encodePassword(passwd), passwd)
True
>>> manager.checkPassword(manager.encodePassword(passwd).decode(), passwd)
True
```

A previous version of this manager also created a cosmetic salt, added to the start of the hash, but otherwise not used in creating the hash itself. Moreover, it generated the MD5 hash as a hex digest, not a base64 encoded value and did not include the {MD5} prefix. Such hashed values are still supported too:

```
>>> encoded = 'salt86dddccec45db4599f1ac00018e54139'
>>> manager.checkPassword(encoded, password)
True
```

However, because the prefix is missing, the password manager cannot claim to implement the scheme:

```
>>> manager.match(encoded)
False
```

class zope.password.password.**SHA1PasswordManager**

Bases: `zope.password.password._PrefixedPasswordManager`

SHA1 password manager.

```
>>> from zope.interface.verify import verifyObject
>>> from zope.password.interfaces import IMatchingPasswordManager
>>> from zope.password.password import SHA1PasswordManager
```

```
>>> manager = SHA1PasswordManager()
>>> verifyObject(IMatchingPasswordManager, manager)
True
```

```
>>> password = u"right \N{CYRILLIC CAPITAL LETTER A}"
>>> encoded = manager.encodePassword(password)
>>> isinstance(encoded, bytes)
True
>>> print(encoded.decode())
{SHA}BLTuxxVMXzouxtKVb7gLgNxzdAI=
>>> manager.match(encoded)
True
>>> manager.match(encoded.decode())
True
>>> manager.checkPassword(encoded, password)
True
>>> manager.checkPassword(encoded, password + u"wrong")
False
```

This password manager is compatible with other RFC 2307 SHA implementations. For example the output of the slappasswd command for a SHA hashing of `secret` is `{SHA}5en6G6MezRroT3XKqkdPOmY/BfQ=`,

and our implementation returns the same hash:

```
>>> print(manager.encodePassword('secret').decode())
{SHA}5en6G6MezRroT3XKqkdPOmY/BfQ=
```

The password manager should be able to cope with unicode strings for input:

```
>>> passwd = u'foobar\u2211' # sigma-sign.
>>> manager.checkPassword(manager.encodePassword(passwd), passwd)
True
>>> manager.checkPassword(manager.encodePassword(passwd).decode(), passwd)
True
```

A previous version of this manager also created a cosmetic salt, added to the start of the hash, but otherwise not used in creating the hash itself. Moreover, it generated the SHA hash as a hex digest, not a base64 encoded value and did not include the {SHA} prefix. Such hashed values are still supported too:

```
>>> encoded = 'salt04b4eec7154c5f3a2ec6d2956fb80b80dc737402'
>>> manager.checkPassword(encoded, password)
True
```

However, because the prefix is missing, the password manager cannot claim to implement the scheme:

```
>>> manager.match(encoded)
False
```

Previously, this password manager used {SHA1} as a prefix, but this was changed to be compatible with LDAP (RFC 2307). The old prefix is still supported (note the hexdigest encoding as well):

```
>>> password = u"right \N{CYRILLIC CAPITAL LETTER A}"
>>> encoded = '{SHA1}04b4eec7154c5f3a2ec6d2956fb80b80dc737402'
>>> manager.match(encoded)
True
>>> manager.checkPassword(encoded, password)
True
>>> manager.checkPassword(encoded, password + u"wrong")
False
```

class zope.password.password.**BCRYPTPasswordManager**

Bases: zope.password.password._PrefixedPasswordManager

BCRYPT password manager.

In addition to the passwords encoded by this class, this class can also recognize passwords encoded by z3c.bcrypt and properly match and check them.

Note: This uses the bcrypt library in its implementation, which only uses the first 72 characters of the password when computing the hash.

---

checkPassword(*hashed_password*, *clear_password*)

Check a *hashed_password* against a *clear_password*.

```
>>> from zope.password.password import BCRYPTPasswordManager
>>> manager = BCRYPTPasswordManager()
>>> manager.checkPassword(b'not from here', None)
False
```

Parameters

- **hashed_password** (*bytes*) – The encoded password.
- **clear_password** (*unicode*) – The password to check.

Returns True iif hashed passwords are equal.

Return type bool

encodePassword(*password*, *salt=None*)
Encode a *password*, with an optional *salt*.

If *salt* is not provided, a unique hash will be generated for each invokation.

Parameters

- **password** (*unicode*) – The clear-text password.
- **salt** – The salt to be used to hash the password.

Return type str

Returns The encoded password as a byte-siring.

match(*hashed_password*)
Was the password hashed with this password manager?

Parameters **hashed_password** (*bytes*) – The encoded password.

Return type bool

Returns True iif the password was hashed with this manager.

class zope.password.password.**BCRYPTKDFPasswordManager**
Bases: zope.password.password._PrefixedPasswordManager

BCRYPT KDF password manager.

This manager converts a plain text password into a byte array. The password and salt values (randomly generated when the password is encoded) are combined and repeatedly hashed *rounds* times. The repeated hashing is designed to thwart discovery of the key via password guessing attacks. The higher the number of rounds, the slower each attempt will be.

Compared to the *BCRYPTPasswordManager*, this has the advantage of allowing tunable rounds, so as computing devices get more powerful making brute force attacks faster, the difficulty level can be raised (for newly encoded passwords).

```
>>> from zope.password.password import BCRYPTKDFPasswordManager
>>> manager = BCRYPTKDFPasswordManager()
>>> manager.checkPassword(b'not from here', None)
False
```

Let's encode a password. We'll use the minimum acceptable number of rounds so that the tests run fast:

```
>>> manager.rounds = 51
>>> password = u"right \N{CYRILLIC CAPITAL LETTER A}"
>>> encoded = manager.encodePassword(password)
>>> print(encoded.decode())
{BCRYPTKDF}33...
```

It checks out:

```
>>> manager.checkPassword(encoded, password)
True
```

We can change the number of rounds for future encodings:

```
>>> manager.rounds = 100
>>> encoded2 = manager.encodePassword(password)
>>> print(encoded2.decode())
{BCRYPTKDF}64...
>>> manager.checkPassword(encoded2, password)
True
```

And the old password still checks out:

```
>>> manager.checkPassword(encoded, password)
True
```

rounds = 1024
The number of rounds of hashing that should be applied. The higher the number, the slower it is. It should be at least 50.

keylen = 32
The number of bytes long the encoded password will be. It must be at least 1 and no more than 512.

2.2.1 Deprecated Implementations

Warning: The following password managers are deprecated, because they produce unacceptably-weak password hashes. They are only included to allow apps which previously used them to migrate smoothly to a supported implementation.

Legacy password managers, using now-outdated, insecure methods for hashing

class zope.password.legacy.**CryptPasswordManager**
Bases: object

Crypt password manager.

Implements a UNIX crypt(3) hashing scheme. Note that crypt is considered far inferior to more modern schemes such as SSHA hashing, and only uses the first 8 characters of a password.

```
>>> from zope.interface.verify import verifyObject
>>> from zope.password.interfaces import IMatchingPasswordManager
>>> from zope.password.legacy import CryptPasswordManager
```

```
>>> manager = CryptPasswordManager()
>>> verifyObject(IMatchingPasswordManager, manager)
True
```

```
>>> password = u"right \N{CYRILLIC CAPITAL LETTER A}"
>>> encoded = manager.encodePassword(password, salt="..")
>>> encoded
'{CRYPT}..I1I8wps4Na2'
>>> manager.match(encoded)
True
```

(continues on next page)

(continued from previous page)

```
>>> manager.checkPassword(encoded, password)
True
```

Note that this object fails to return bytes from the `encodePassword` function on Python 3:

```
>>> isinstance(encoded, str)
True
```

Unfortunately, crypt only looks at the first 8 characters, so matching against an 8 character password plus suffix always matches. Our test password (including utf-8 encoding) is exactly 8 characters long, and thus affixing 'wrong' to it tests as a correct password:

```
>>> manager.checkPassword(encoded, password + u"wrong")
True
```

Using a completely different password is rejected as expected:

```
>>> manager.checkPassword(encoded, 'completely wrong')
False
```

Using the *openssl passwd* command-line utility to encode `secret`, we get `erz50QD3gv4Dw` as seeded hash.

Our password manager generates the same value when seeded with the same salt, so we can be sure, our output is compatible with standard LDAP tools that also use crypt:

```
>>> salt = 'er'
>>> password = 'secret'
>>> encoded = manager.encodePassword(password, salt)
>>> encoded
'{CRYPT}erz50QD3gv4Dw'
```

```
>>> manager.checkPassword(encoded, password)
True
>>> manager.checkPassword(encoded, password + u"wrong")
False
```

```
>>> manager.encodePassword(password) != manager.encodePassword(password)
True
```

The manager only claims to implement CRYPT encodings, anything not starting with the string {CRYPT} returns False:

```
>>> manager.match('{MD5}someotherhash')
False
```

class zope.password.legacy.**MySQLPasswordManager**

Bases: `object`

A MySQL digest manager.

This Password Manager implements the digest scheme as implemented in the MySQL PASSWORD function in MySQL versions before 4.1. Note that this method results in a very weak 16-byte hash.

```
>>> from zope.interface.verify import verifyObject
>>> from zope.password.interfaces import IMatchingPasswordManager
>>> from zope.password.legacy import MySQLPasswordManager
```

```
>>> manager = MySQLPasswordManager()
>>> verifyObject(IMatchingPasswordManager, manager)
True
```

```
>>> password = u"right \N{CYRILLIC CAPITAL LETTER A}"
>>> encoded = manager.encodePassword(password)
>>> isinstance(encoded, bytes)
True
>>> print(encoded.decode())
{MYSQL}0ecd752c5097d395
>>> manager.match(encoded)
True
>>> manager.match(encoded.decode())
True
>>> manager.checkPassword(encoded.decode(), password)
True
>>> manager.checkPassword(encoded, password)
True
>>> manager.checkPassword(encoded, password + u"wrong")
False
```

Using the password 'PHP & Information Security' should result in the hash `379693e271cd3bd6`, according to http://phpsec.org/articles/2005/password-hashing.html

Our password manager generates the same value when seeded with the same seed, so we can be sure, our output is compatible with MySQL versions before 4.1:

```
>>> password = 'PHP & Information Security'
>>> encoded = manager.encodePassword(password)
>>> isinstance(encoded, bytes)
True
>>> print(encoded.decode())
{MYSQL}379693e271cd3bd6
```

```
>>> manager.checkPassword(encoded, password)
True
>>> manager.checkPassword(encoded, password + u"wrong")
False
```

The manager only claims to implement MYSQL encodings, anything not starting with the string {MYSQL} returns False:

```
>>> manager.match('{MD5}someotherhash')
False
```

Spaces and tabs are ignored:

```
>>> encoded = manager.encodePassword('      ign or ed')
>>> print(encoded.decode())
{MYSQL}75818366052c6a78
>>> encoded = manager.encodePassword('ignored')
>>> print(encoded.decode())
{MYSQL}75818366052c6a78
```

2.3 Vocabulary

Vocabulary of password manager utility names

For use with zope.component and zope.schema.

zope.password.vocabulary.**PasswordManagerNamesVocabulary**(*context=None*)
Return a vocabulary listing password manager implementations by name.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

z

zope.password.interfaces, 7
zope.password.legacy, 15
zope.password.password, 7
zope.password.vocabulary, 18

Index

B

BCRYPTKDFPasswordManager (class in zope.password.password), 14
BCRYPTPasswordManager (class in zope.password.password), 13

C

checkPassword() (zope.password.interfaces.IPasswordManager method), 7
checkPassword() (zope.password.password.BCRYPTPasswordManager method), 13
CryptPasswordManager (class in zope.password.legacy), 15

E

encodePassword() (zope.password.interfaces.IPasswordManager method), 7
encodePassword() (zope.password.password.BCRYPTPasswordManager method), 14

I

IMatchingPasswordManager (interface in zope.password.interfaces), 7
IPasswordManager (interface in zope.password.interfaces), 7

K

keylen (zope.password.password.BCRYPTKDFPasswordManager attribute), 15

M

match() (zope.password.interfaces.IMatchingPasswordManager method), 7
match() (zope.password.password.BCRYPTPasswordManager method), 14
MD5PasswordManager (class in zope.password.password), 11
MySQLPasswordManager (class in zope.password.legacy), 16

P

PasswordManagerNamesVocabulary() (in module zope.password.vocabulary), 18
PlainTextPasswordManager (class in zope.password.password), 7

R

rounds (zope.password.password.BCRYPTKDFPasswordManager attribute), 15

S

SHA1PasswordManager (class in zope.password.password), 12
SMD5PasswordManager (class in zope.password.password), 10
SSHAPasswordManager (class in zope.password.password), 8

Z

zope.password.interfaces (module), 7
zope.password.legacy (module), 15
zope.password.password (module), 7
zope.password.vocabulary (module), 18