`hmac` — Keyed-Hashing for Message Authentication

Source code: Lib/hmac.py

This module implements the HMAC algorithm as described by **RFC 2104**.

`hmac.`**`new`**(*`key`*, *`msg=None`*, *`digestmod=''`*)

Return a new hmac object. *key* is a bytes or bytearray object giving the secret key. If *msg* is present, the method call `update(msg)` is made. *digestmod* is the digest name, digest constructor or module for the HMAC object to use. It may be any name suitable to `hashlib.new()`. Despite its argument position, it is required.

Changed in version 3.4: Parameter *key* can be a bytes or bytearray object. Parameter *msg* can be of any type supported by `hashlib`. Parameter *digestmod* can be the name of a hash algorithm.

Deprecated since version 3.4, removed in version 3.8: MD5 as implicit default digest for *digestmod* is deprecated. The digestmod parameter is now required. Pass it as a keyword argument to avoid awkwardness when you do not have an initial msg.

`hmac.`**`digest`**(*`key`*, *`msg`*, *`digest`*)

Return digest of *msg* for given secret *key* and *digest*. The function is equivalent to `HMAC(key, msg, digest).digest()`, but uses an optimized C or inline implementation, which is faster for messages that fit into memory. The parameters *key*, *msg*, and *digest* have the same meaning as in `new()`.

CPython implementation detail, the optimized C implementation is only used when *digest* is a string and name of a digest algorithm, which is supported by OpenSSL.

New in version 3.7.

An HMAC object has the following methods:

`HMAC.`**`update`**(*`msg`*)

Update the hmac object with *msg*. Repeated calls are equivalent to a single call with the concatenation of all the arguments: `m.update(a); m.update(b)` is equivalent to `m.update(a + b)`.

Changed in version 3.4: Parameter *msg* can be of any type supported by `hashlib`.

`HMAC.`**`digest`**()

Return the digest of the bytes passed to the `update()` method so far. This bytes object will be the same length as the *digest_size* of the digest given to the constructor. It may contain non-ASCII bytes, including NUL bytes.

Warning: When comparing the output of `digest()` to an externally-supplied digest during a verification routine, it is recommended to use the `compare_digest()` function instead of the `==` operator to reduce the vulnerability to timing attacks.

HMAC.**hexdigest**()

Like `digest()` except the digest is returned as a string twice the length containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

Warning: When comparing the output of `hexdigest()` to an externally-supplied digest during a verification routine, it is recommended to use the `compare_digest()` function instead of the `==` operator to reduce the vulnerability to timing attacks.

HMAC.**copy**()

Return a copy (“clone”) of the hmac object. This can be used to efficiently compute the digests of strings that share a common initial substring.

A hash object has the following attributes:

HMAC.**digest_size**

The size of the resulting HMAC digest in bytes.

HMAC.**block_size**

The internal block size of the hash algorithm in bytes.

New in version 3.4.

HMAC.**name**

The canonical name of this HMAC, always lowercase, e.g. `hmac-md5`.

New in version 3.4.

Deprecated since version 3.9: The undocumented attributes `HMAC.digest_cons`, `HMAC.inner`, and `HMAC.outer` are internal implementation details and will be removed in Python 3.10.

This module also provides the following helper function:

hmac.**compare_digest**(*a*, *b*)

Return `a == b`. This function uses an approach designed to prevent timing analysis by avoiding content-based short circuiting behaviour, making it appropriate for cryptography. *a* and *b* must both be of the same type: either `str` (ASCII only, as e.g. returned by `HMAC.hexdigest()`), or a bytes-like object.

Note: If *a* and *b* are of different lengths, or if an error occurs, a timing attack could theoretically reveal information about the types and lengths of *a* and *b*—but not their values.

New in version 3.3.

Changed in version 3.9: The function uses OpenSSL's `CRYPTO_memcmp()` internally when available.

See also:

Module `hashlib`
The Python module providing secure hash functions.