hmac — Cryptographic Message Signing and Verification

Purpose: The hmac module implements keyed-hashing for message authentication, as described in RFC 2104.

The HMAC algorithm can be used to verify the integrity of information passed between applications or stored in a potentially vulnerable location. The basic idea is to generate a cryptographic hash of the actual data combined with a shared secret key. The resulting hash can then be used to check the transmitted or stored message to determine a level of trust, without transmitting the secret key.

Warning

Disclaimer: I am not a security expert. For the full details on HMAC, check out **RFC 2104**.

Signing Messages

The `new()` function creates a new object for calculating a message signature. This example uses the default MD5 hash algorithm.

```
# hmac_simple.py

import hmac

digest_maker = hmac.new(b'secret-shared-key-goes-here')

with open('lorem.txt', 'rb') as f:
    while True:
        block = f.read(1024)
        if not block:
            break
        digest_maker.update(block)

digest = digest_maker.hexdigest()
print(digest)
```

When run, the code reads a data file and computes an HMAC signature for it.

```
$ python3 hmac_simple.py

4bcb287e284f8c21e87e14ba2dc40b16
```

Alternate Digest Types

Although the default cryptographic algorithm for hmac is MD5, that is not the most secure method to use. MD5 hashes have some weaknesses, such as collisions (where two different messages produce the same hash). The SHA-1 algorithm is considered to be stronger, and should be used instead.

```
# hmac_sha.py

import hmac
import hashlib

digest_maker = hmac.new(
    b'secret-shared-key-goes-here',
    b'',
    hashlib.sha1,
)

with open('hmac_sha.py', 'rb') as f:
    while True:
        block = f.read(1024)
        if not block:
            break
        digest_maker.update(block)

digest = digest_maker.hexdigest()
print(digest)
```

The `new()` function takes three arguments. The first is the secret key, which should be shared between the two endpoints that are communicating so both ends can use the same value. The second value is an initial message. If the message content that needs to be authenticated is small, such as a timestamp or HTTP POST, the entire body of the message can be passed to `new()` instead of using the `update()` method. The last argument is the digest module to be used. The default is `hashlib.md5`. This example passes `'sha1'`, causing hmac to use `hashlib.sha1`

```
$ python3 hmac_sha.py

dcee20eeee9ef8a453453f510d9b6765921cf099
```

Binary Digests

The previous examples used the `hexdigest()` method to produce printable digests. The hexdigest is a different representation of the value calculated by the `digest()` method, which is a binary value that may include unprintable characters, including `NUL`. Some web services (Google checkout, Amazon S3) use the base64 encoded version of the binary digest instead of the hexdigest.

```
# hmac_base64.py

import base64
import hmac
import hashlib

with open('lorem.txt', 'rb') as f:
    body = f.read()

hash = hmac.new(
    b'secret-shared-key-goes-here',
    body,
    hashlib.sha1,
)

digest = hash.digest()
print(base64.encodebytes(digest))
```

The base64 encoded string ends in a newline, which frequently needs to be stripped off when embedding the string in http headers or other formatting-sensitive contexts.

```
$ python3 hmac_base64.py

b'olW2DoXHGJEKGU0aE9fOwSVE/o4=\n'
```

Applications of Message Signatures

HMAC authentication should be used for any public network service, and any time data is stored where security is important. For example, when sending data through a pipe or socket, that data should be signed and then the signature should be tested before the data is used. The extended example given here is available in the file `hmac_pickle.py`.

The first step is to establish a function to calculate a digest for a string, and a simple class to be instantiated and passed through a communication channel.

```
# hmac_pickle.py

import hashlib
import hmac
import io
import pickle
import pprint
```

```
def make_digest(message):
    "Return a digest for the message."
    hash = hmac.new(
        b'secret-shared-key-goes-here',
        message,
        hashlib.sha1,
    )
    return hash.hexdigest().encode('utf-8')


class SimpleObject:
    """Demonstrate checking digests before unpickling.
    """

    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name
```

Next, create a BytesIO buffer to represent the socket or pipe. The example uses a naive, but easy to parse, format for the data stream. The digest and length of the data are written, followed by a new line. The serialized representation of the object, generated by pickle, follows. A real system would not want to depend on a length value, since if the digest is wrong the length is probably wrong as well. Some sort of terminator sequence not likely to appear in the real data would be more appropriate.

The example program then writes two objects to the stream. the first is written using the correct digest value.

```
# Simulate a writable socket or pipe with a buffer
out_s = io.BytesIO()

# Write a valid object to the stream:
#  digest\nlength\npickle
o = SimpleObject('digest matches')
pickled_data = pickle.dumps(o)
digest = make_digest(pickled_data)
header = b'%s %d\n' % (digest, len(pickled_data))
print('WRITING: {}'.format(header))
out_s.write(header)
out_s.write(pickled_data)
```

The second object is written to the stream with an invalid digest, produced by calculating the digest for some other data instead of the pickle.

```
# Write an invalid object to the stream
o = SimpleObject('digest does not match')
pickled_data = pickle.dumps(o)
digest = make_digest(b'not the pickled data at all')
header = b'%s %d\n' % (digest, len(pickled_data))
print('\nWRITING: {}'.format(header))
out_s.write(header)
out_s.write(pickled_data)

out_s.flush()
```

Now that the data is in the BytesIO buffer, it can be read back out again. Start by reading the line of data with the digest and data length. Then read the remaining data, using the length value. pickle.load() could read directly from the stream, but that assumes a trusted data stream and this data is not yet trusted enough to unpickle it. Reading the pickle as a string from the stream, without actually unpickling the object, is safer.

```
# Simulate a readable socket or pipe with a buffer
in_s = io.BytesIO(out_s.getvalue())

# Read the data
while True:
    first_line = in_s.readline()
    if not first_line:
        break
    incoming_digest, incoming_length = first_line.split(b' ')
    incoming_length = int(incoming_length.decode('utf-8'))
    print('\nREAD:', incoming_digest, incoming_length)
```

Once the pickled data is in memory, the digest value can be recalculated and compared against the data read using `compare_digest()`. If the digests match, it is safe to trust the data and unpickle it.

```
    incoming_pickled_data = in_s.read(incoming_length)

    actual_digest = make_digest(incoming_pickled_data)
    print('ACTUAL:', actual_digest)

    if hmac.compare_digest(actual_digest, incoming_digest):
        obj = pickle.loads(incoming_pickled_data)
        print('OK:', obj)
    else:
        print('WARNING: Data corruption')
```

The output shows that the first object is verified and the second is deemed “corrupted”, as expected.

```
$ python3 hmac_pickle.py

WRITING: b'f49cd2bf7922911129e8df37f76f95485a0b52ca 69\n'

WRITING: b'b01b209e28d7e053408ebe23b90fe5c33bc6a0ec 76\n'

READ: b'f49cd2bf7922911129e8df37f76f95485a0b52ca' 69
ACTUAL: b'f49cd2bf7922911129e8df37f76f95485a0b52ca'
OK: digest matches

READ: b'b01b209e28d7e053408ebe23b90fe5c33bc6a0ec' 76
ACTUAL: b'2ab061f9a9f749b8dd6f175bf57292e02e95c119'
WARNING: Data corruption
```

Comparing two digests with a simple string or bytes comparison can be used in a timing attack to expose part or all of the secret key by passing digests of different lengths. `compare_digest()` implements a fast but constant-time comparison function to protect against timing attacks.

See also

- Standard library documentation for hmac
- **RFC 2104** – HMAC: Keyed-Hashing for Message Authentication
- `hashlib` – The `hashlib` module provides MD5 and SHA1 hash generators.
- `pickle` – Serialization library.
- WikiPedia: MD5 – Description of the MD5 hashing algorithm.
- Signing and Authenticating REST Requests (Amazon AWS) – Instructions for authenticating to S3 using HMAC-SHA1 signed credentials.

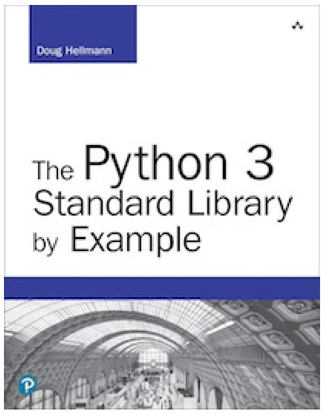
hashlib — Cryptographic Hashing

Concurrency with Processes, Threads, and Coroutines

Quick Links

This page was last updated 2017-07-30.

Navigation



Get the book

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

Other Writing

Blog
The Python Standard Library By Example