Mohawk Documentation

Release 0.1

Kumar McMillan, Austin King

Jan 09, 2019

Contents

Mohawk is an alternate Python implementation of the Hawk HTTP authorization scheme. Hawk lets two parties securely communicate with each other using messages signed by a shared key. It is based on HTTP MAC access authentication (which was based on parts of OAuth 1.0).

The Mohawk API is a little different from that of the Node library (i.e. the living Hawk spec). It was redesigned to be more intuitive to developers, less prone to security problems, and more Pythonic.

CHAPTER 1

Installation

Requirements:

- Python 2.6+ or 3.4+
- six

Using pip:

```
pip install mohawk
```

If you want to install from source, visit https://github.com/kumar303/mohawk

CHAPTER 2

Bugs

You can submit bugs / patches on Github: https://github.com/kumar303/mohawk

Important: If you think you found a security vulnerability please try emailing kumar.mcmillan@gmail.com before submitting a public issue.

CHAPTER 3

Topics

Using Mohawk

There are two parties involved in Hawk communication: a *sender* and a *receiver*. They use a shared secret to sign and verify each other's messages.

Sender A client who wants to access a Hawk-protected resource. The client will sign their request and upon receiving a response will also verify the response signature.

Receiver A server that uses Hawk to protect its resources. The server will check the signature of an incoming request before accepting it. It also signs its response using the same shared secret.

What are some good use cases for Hawk? This library was built for the case of securing API connections between two back-end servers. Hawk is a good fit for this because you can keep the shared secret safe on each machine. Hawk may not be a good fit for scenarios where you can't protect the shared secret.

After getting familiar with usage, you may want to consult the *Security Considerations* section.

Sending a request

Let's say you want to make an HTTP request like this:

```
>>> url = 'https://some-service.net/system'
>>> method = 'POST'
>>> content = 'one=1&two=2'
>>> content_type = 'application/x-www-form-urlencoded'
```

Set up your Hawk request by creating a *mohawk.Sender* object with all the elements of the request that you need to sign:

```
>>> from mohawk import Sender
>>> sender = Sender({'id': 'some-sender',
...                  'key': 'a long, complicated secret',
...                  'algorithm': 'sha256'},
...                 url,
...                 method,
...                 content=content,
...                 content_type=content_type)
```

This provides you with a Hawk `Authorization` header to send along with your request:

```
>>> sender.request_header
u'Hawk mac="...", hash="...", id="some-sender", ts="...", nonce="..."'
```

Using the requests library just as an example, you would send your POST like this:

```
>>> requests.post(url, data=content,
...               headers={'Authorization': sender.request_header,
...                        'Content-Type': content_type})
```

Notice how both the content and content-type values were signed by the Sender. In the case of a GET request you'll probably need to sign empty strings like `Sender(...,'GET',content='',content_type='')`, that is, if your request library doesn't automatically set a content-type for GET requests.

If you only intend to work with *mohawk.Sender*, skip down to *Verifying a response*.

Receiving a request

On the receiving end, such as a web server, you'll need to set up a *mohawk.Receiver* object to accept and respond to *mohawk.Sender* requests.

First, you need to give the receiver a callable that it can use to look up sender credentials:

```
>>> def lookup_credentials(sender_id):
...     if sender_id in allowed_senders:
...         # Return a credentials dictionary formatted like the sender example.
...         return allowed_senders[sender_id]
...     else:
...         raise LookupError('unknown sender')
```

An incoming request will probably arrive in an object like this, depending on your web server framework:

```
>>> request = {'headers': {'Authorization': sender.request_header,
...                        'Content-Type': content_type},
...            'url': url,
...            'method': method,
...            'content': content}
```

Create a *mohawk.Receiver* using values from the incoming request:

```
>>> from mohawk import Receiver
>>> receiver = Receiver(lookup_credentials,
...                     request['headers']['Authorization'],
...                     request['url'],
...                     request['method'],
...                     content=request['content'],
...                     content_type=request['headers']['Content-Type'])
```

If this constructor does not raise any *Exceptions* then the signature of the request is correct and you can proceed.

Important: The server running *mohawk.Receiver* code should synchronize its clock with something like TLS-date to make sure it compares timestamps correctly.

Responding to a request

It's optional per the Hawk spec but a *mohawk.Receiver* should sign its response back to the client to prevent certain attacks.

The receiver starts by building a message it wants to respond with:

```
>>> response_content = '{"msg": "Hello, dear friend"}'
>>> response_content_type = 'application/json'
>>> header = receiver.respond(content=response_content,
...                           content_type=response_content_type)
```

This provides you with a similar Hawk header to use in the response:

```
>>> receiver.response_header
u'Hawk mac="...", hash="...="'
```

Using your web server's framework, respond with a `Server-Authorization` header. For example:

```
>>> response = {
...     'headers': {'Server-Authorization': receiver.response_header,
...                 'Content-Type': response_content_type},
...     'content': response_content
... }
```

Verifying a response

When the *mohawk.Sender* receives a response it should verify the signature to make sure nothing has been tampered with:

```
>>> sender.accept_response(response['headers']['Server-Authorization'],
...                        content=response['content'],
...                        content_type=response['headers']['Content-Type'])
```

If this method does not raise any *Exceptions* then the signature of the response is correct and you can proceed.

Allowing senders to adjust their timestamps

The easiest way to avoid timestamp problems is to synchronize your server clock using something like TLSdate.

If a sender's clock is out of sync with the receiver, its message might expire prematurely. In this case the receiver should respond with a header the sender can use to adjust its timestamp.

When receiving a request you might get a *mohawk.exc.TokenExpired* exception. You can access the `www_authenticate` property on the exception object to respond correctly like this:

```
>>> from mohawk.exc import TokenExpired
>>> try:
...     receiver = Receiver(lookup_credentials,
...                         request['headers']['Authorization'],
...                         request['url'],
...                         request['method'],
...                         content=request['content'],
...                         content_type=request['headers']['Content-Type'])
... except TokenExpired as expiry:
...     pass
```

```
>>> expiry.www_authenticate
'Hawk ts="...", tsm="...", error="token with UTC timestamp...has expired..."'
>>> response['headers']['WWW-Authenticate'] = expiry.www_authenticate
```

A compliant client can look for this response header and parse the `ts` property (the server's "now" timestamp) and the `tsm` property (a MAC calculation of `ts`). It can then recalculate the MAC using its own credentials and if the MACs both match it can trust that this is the real server's timestamp. This allows the sender to retry the request with an adjusted timestamp.

Using a nonce to prevent replay attacks

A replay attack is when someone copies a Hawk authorized message and re-sends the message without altering it. Because the Hawk signature would still be valid, the receiver may accept the message. This could have unintended side effects such as increasing the quantity of an item just purchased if it were a commerce API that had an `increment-item` service.

Hawk protects against replay attacks in a couple ways. First, a receiver checks the timestamp of the message which may result in a *mohawk.exc.TokenExpired* exception. Second, every message includes a cryptographic nonce which is a unique identifier. In combination with the sender's id and the request's timestamp, a receiver can use the nonce to know if it has *already* received the request. If so, the *mohawk.exc.AlreadyProcessed* exception is raised.

By default, Mohawk doesn't know how to check nonce values; this is something your application needs to do.

Important: If you don't configure nonce checking, your application could be susceptible to replay attacks.

Make a callable that returns True if a sender's nonce plus its timestamp has been seen already. Here is an example using something like memcache:

```
>>> def seen_nonce(sender_id, nonce, timestamp):
...     key = '{id}:{nonce}:{ts}'.format(id=sender_id, nonce=nonce,
...                                      ts=timestamp)
...     if memcache.get(key):
...         # We have already processed this nonce + timestamp.
...         return True
...     else:
...         # Save this nonce + timestamp for later.
...         memcache.set(key, True)
...         return False
```

Because messages will expire after a short time you don't need to store nonces for much longer than that timeout. See *mohawk.Receiver* for the default timeout.

Pass your callable as a `seen_nonce` argument to *mohawk.Receiver*:

```
>>> receiver = Receiver(lookup_credentials,
...                     request['headers']['Authorization'],
...                     request['url'],
...                     request['method'],
...                     content=request['content'],
...                     content_type=request['headers']['Content-Type'],
...                     seen_nonce=seen_nonce)
```

If `seen_nonce()` returns True, *mohawk.exc.AlreadyProcessed* will be raised.

When a *sender* calls *mohawk.Sender.accept_response()*, it will receive a Hawk message but the nonce will be that of the original request. In other words, the nonce received is the same nonce that the sender generated and signed when initiating the request. This generally means you don't have to worry about *response* replay attacks. However, if you expose your *mohawk.Sender.accept_response()* call somewhere publicly over HTTP then you may need to protect against response replay attacks. You can do so by constructing a *mohawk.Sender* with the same `seen_nonce` keyword:

```
>>> sender = Sender({'id': 'some-sender',
...                  'key': 'a long, complicated secret',
...                  'algorithm': 'sha256'},
...                 url,
...                 method,
...                 content=content,
...                 content_type=content_type,
...                 seen_nonce=seen_nonce)
```

Skipping content checks

In some cases you may not be able to hash request/response content. For example, the content could be too large. If you run into this, Hawk might not be the best fit for you but Hawk does allow you to accept content without a declared hash if you wish.

Important: By allowing content without a declared hash, both the sender and receiver are susceptible to content tampering.

You can send a request without signing the content by passing this keyword argument to a *mohawk.Sender*:

```
>>> sender = Sender(credentials, url, method, always_hash_content=False)
```

This says to skip hashing of the `content` and `content_type` values if they are both *mohawk.base.EmptyValue*.

Now you'll get an `Authorization` header without a `hash` attribute:

```
>>> sender.request_header
u'Hawk mac="...", id="some-sender", ts="...", nonce="..."'
```

The *mohawk.Receiver* must also be constructed to accept content without a declared hash using `accept_untrusted_content=True`:

```
>>> receiver = Receiver(lookup_credentials,
...                     sender.request_header,
...                     request['url'],
...                     request['method'],
...                     content=request['content'],
...                     content_type=request['headers']['Content-Type'],
...                     accept_untrusted_content=True)
```

This will skip checking the hash of `content` and `content_type` only if the `Authorization` header omits the `hash` attribute. If the `hash` attribute is present, it will be checked as normal.

Empty requests

For requests whose `content` (and by extension `content_type`) is `None` or an empty string, it is acceptable for the sender to omit the declared hash, regardless of the `accept_untrusted_content` value provided to the *`mohawk.Receiver`*. For example, a `GET` request typically has empty content and some libraries may or may not hash the content.

If the `hash` attribute *is* present, a `None` value for either `content` or `content_type` will be coerced to an empty string prior to hashing.

Generating protected URLs

Hawk lets you protect a URL with a token derived from a secret key. After a period of time, access to the URL will expire. As an example, you could use this to deliver a URL for purchased media, such a zip file of MP3s. The user could access the URL for a short period of time but after that, the same URL would not be accessible.

In the Hawk spec, this is referred to as Single URI Authorization, or bewit.

Here's an example of protecting access to this URL with Mohawk:

```
>>> url = 'https://site.org/purchases/music-album.zip'
```

Let's say you want to allow access for 5 minutes:

```
>>> from mohawk.util import utc_now
>>> url_expires_at = utc_now() + (60 * 5)
```

Set up Hawk credentials like in previous examples:

```
>>> credentials = {
...     'id': 'some-recipient',
...     'key': 'a long, complicated secret',
...     'algorithm': 'sha256'
... }
```

Define the resource that you want to protect:

```
>>> from mohawk.base import Resource
>>> resource = Resource(
...     credentials=credentials,
...     url=url,
...     method='GET',
...     nonce='',
...     timestamp=url_expires_at,
... )
```

Generate a bewit token:

```
>>> from mohawk.bewit import get_bewit
>>> bewit = get_bewit(resource)
```

Add that token as a `bewit` query string parameter back to the same URL:

```
>>> protected_url = '{url}?bewit={bewit}'.format(url=url, bewit=bewit)
>>> protected_url
'https://site.org/purchases/music-album.zip?bewit=...'
```

Now you can deliver this bewit protected URL to the recipient.

Serving protected URLs

When handling a request for a bewit protected URL on the server, you can begin by checking the bewit to make sure it's valid. If `True`, the server can respond with access to the resource. The `check_bewit` function returns `True` or `False` and will also raise an exception for invalid `bewit` values.

```
>>> allowed_recipients = {}
>>> allowed_recipients['some-recipient'] = credentials
>>> def lookup_credentials(recipient_id):
...     if recipient_id in allowed_recipients:
...         # Return a credentials dictionary
...         return allowed_recipients[recipient_id]
...     else:
...         raise LookupError('unknown recipient_id')
>>> from mohawk.bewit import check_bewit
>>> check_bewit(protected_url, credential_lookup=lookup_credentials)
True
```

Note: Well, that was complicated! At a future time, `get_bewit` and `check_bewit` will be complimented with a higher level function that is easier to work with. See https://github.com/kumar303/mohawk/issues/17

Logging

All internal logging channels stem from `mohawk`. For example, the `mohawk.receiver` channel will just contain receiver messages. These channels correspond to the submodules within mohawk.

To debug *`mohawk.exc.MacMismatch` Exceptions* and other authorization errors, set the `mohawk` channel to `DEBUG`.

Going further

Well, hey, that about summarizes the concepts and basic usage of Mohawk. Check out the *API* for details. Also make sure you are familiar with *Security Considerations*.

Security Considerations

Hawk HTTP authorization uses a message authentication code (MAC) algorithm to provide partial cryptographic verification of HTTP requests/responses.

Important: Take a look at Hawk's own security considerations.

Here are some additional security considerations:

- `mohawk` is intended to be used as a low-level library. You should *never* expose its *Exceptions* publicly, say, in an HTTP response, as they may provide hints to an attacker.
- Using a shared secret for signatures means that if the secret leaks out then messages can be signed all day long. Make sure secrets are stored somewhere safe and never transmitted over an insecure channel. For example, putting a shared secret in memory on a web browser page may or may not be secure enough.

- What does *partial verification* mean? While all major request/response artifacts are signed (URL, protocol, method, content), *only* the `content-type` header is signed. You'll want to make sure your sender and receiver aren't susceptible to header poisoning in case an attacker finds a way to replay a valid Hawk request with additional headers. For example, if an attacker can find a way to replay a request and add the header `x-token: hijacked-token` then the request might still be valid because this random header is not part of the signature.
- Consider *Using a nonce to prevent replay attacks*.
- Hawk lets you verify that you're talking to the person you think you are. In a lot of ways, this is more trustworthy than SSL/TLS but to guard against your own stupidity as well as prevent general eavesdropping, you should probably use both HTTPS and Hawk.
- The Hawk spec says that signing request/response content is *optional* but just for extra paranoia, Mohawk raises an exception if you skip content checks unintentionally. Read *Skipping content checks* for how to intentionally make it optional. This does not apply to *empty requests*.

API

This is a detailed look at the Mohawk API. For general usage patterns see *Using Mohawk*.

Sender

class mohawk.**Sender**(*credentials*, *url*, *method*, *content=EmptyValue*, *content_type=EmptyValue*, *always_hash_content=True*, *nonce=None*, *ext=None*, *app=None*, *dlg=None*, *seen_nonce=None*, *_timestamp=None*)

A Hawk authority that will emit requests and verify responses.

Parameters

- **credentials** (*dict*) – Dict of credentials with keys `id`, `key`, and `algorithm`. See *Using Mohawk* for an example.
- **url** (*str*) – Absolute URL of the request.
- **method** (*str*) – Method of the request. E.G. POST, GET
- **content=EmptyValue** (*str*) – Byte string of request body.
- **content_type=EmptyValue** (*str*) – content-type header value for request.
- **always_hash_content=True** (*bool*) – When True, `content` and `content_type` must be provided. Read *Skipping content checks* to learn more.
- **nonce=None** (*str*) – A string that when coupled with the timestamp will uniquely identify this request to prevent replays. If None, a nonce will be generated for you.
- **ext=None** (*str*) – An external Hawk string. If not None, this value will be signed so that the receiver can trust it.
- **app=None** (*str*) – A Hawk application string. If not None, this value will be signed so that the receiver can trust it.
- **dlg=None** (*str*) – A Hawk delegation string. If not None, this value will be signed so that the receiver can trust it.
- **seen_nonce=None** (*callable*) – A callable that returns True if a nonce has been seen. See *Using a nonce to prevent replay attacks* for details.

accept_response(*response_header*, *content=EmptyValue*, *content_type=EmptyValue*, *accept_untrusted_content=False*, *localtime_offset_in_seconds=0*, *timestamp_skew_in_seconds=60*, ***auth_kw*)
Accept a response to this request.

Parameters

- **response_header** (*str*) – A Hawk `Server-Authorization` header such as one created by *mohawk.Receiver*.
- **content=EmptyValue** (*str*) – Byte string of the response body received.
- **content_type=EmptyValue** (*str*) – Content-Type header value of the response received.
- **accept_untrusted_content=False** (*bool*) – When True, allow responses that do not hash their content. Read *Skipping content checks* to learn more.
- **localtime_offset_in_seconds=0** (*float*) – Seconds to add to local time in case it's out of sync.
- **timestamp_skew_in_seconds=60** (*float*) – Max seconds until a message expires. Upon expiry, *mohawk.exc.TokenExpired* is raised.

request_header = None
Value suitable for an `Authorization` header.

Receiver

class mohawk.**Receiver**(*credentials_map*, *request_header*, *url*, *method*, *content=EmptyValue*, *content_type=EmptyValue*, *seen_nonce=None*, *localtime_offset_in_seconds=0*, *accept_untrusted_content=False*, *timestamp_skew_in_seconds=60*, ***auth_kw*)
A Hawk authority that will receive and respond to requests.

Parameters

- **credentials_map** (*callable*) – Callable to look up the credentials dict by sender ID. The credentials dict must have the keys: `id`, `key`, and `algorithm`. See *Receiving a request* for an example.
- **request_header** (*str*) – A Hawk `Authorization` header such as one created by *mohawk.Sender*.
- **url** (*str*) – Absolute URL of the request.
- **method** (*str*) – Method of the request. E.G. POST, GET
- **content=EmptyValue** (*str*) – Byte string of request body.
- **content_type=EmptyValue** (*str*) – content-type header value for request.
- **accept_untrusted_content=False** (*bool*) – When True, allow requests that do not hash their content. Read *Skipping content checks* to learn more.
- **localtime_offset_in_seconds=0** (*float*) – Seconds to add to local time in case it's out of sync.
- **timestamp_skew_in_seconds=60** (*float*) – Max seconds until a message expires. Upon expiry, *mohawk.exc.TokenExpired* is raised.

respond(*content=EmptyValue*, *content_type=EmptyValue*, *always_hash_content=True*, *ext=None*)
Respond to the request.

This generates the *mohawk.Receiver.response_header* attribute.

Parameters

- **content=EmptyValue** (*str*) – Byte string of response body that will be sent.
- **content_type=EmptyValue** (*str*) – content-type header value for response.
- **always_hash_content=True** (*bool*) – When True, `content` and `content_type` must be provided. Read *Skipping content checks* to learn more.
- **ext=None** (*str*) – An external Hawk string. If not None, this value will be signed so that the sender can trust it.

response_header = None
Value suitable for a `Server-Authorization` header.

Exceptions

If you want to catch any exception that might be raised, catch *mohawk.exc.HawkFail*.

Important: Never expose an exception message publicly, say, in an HTTP response, as it may provide hints to an attacker.

---

exception mohawk.exc.**AlreadyProcessed**
The message has already been processed and cannot be re-processed.

See *Using a nonce to prevent replay attacks* for details.

exception mohawk.exc.**BadHeaderValue**
There was an error with an attribute or value when parsing or creating a Hawk header.

exception mohawk.exc.**CredentialsLookupError**
A *mohawk.Receiver* could not look up the credentials for an incoming request.

exception mohawk.exc.**HawkFail**
All Mohawk exceptions derive from this base.

exception mohawk.exc.**InvalidBewit**
The bewit is invalid; e.g. it doesn't contain the right number of parameters.

exception mohawk.exc.**InvalidCredentials**
The specified Hawk credentials are invalid.

For example, the dict could be formatted incorrectly.

exception mohawk.exc.**MacMismatch**
The locally calculated MAC did not match the MAC that was sent.

exception mohawk.exc.**MisComputedContentHash**
The signature of the content did not match the actual content.

exception mohawk.exc.**MissingAuthorization**
No authorization header was sent by the client.

exception mohawk.exc.**MissingContent**
A payload's *content* or *content_type* were not provided.

See *Skipping content checks* for details.

exception mohawk.exc.**TokenExpired**(**args*, ***kw*)
The timestamp on a message received has expired.

You may also receive this message if your server clock is out of sync. Consider synchronizing it with something like TLSdate.

If you are unable to synchronize your clock universally, The Hawk spec mentions how you can adjust your sender's time to match that of the receiver in the case of unexpected expiration.

The www_authenticate attribute of this exception is a header that can be returned to the client. If the value is not None, it will include a timestamp HMAC'd with the sender's credentials. This will allow the client to verify the value and safely apply an offset.

localtime_in_seconds = None
Current local time in seconds that was used to compare timestamps.

Base

class mohawk.base.**Resource**(***kw*)
Normalized request / response resource.

Parameters

- **credentials** – A dict of credentials; it must have the keys: id, key, and algorithm. See *Sending a request* for an example.
- **url** (*str*) – Absolute URL of the request / response.
- **method** (*str*) – Method of the request / response. E.G. POST, GET
- **content=EmptyValue** (*str*) – Byte string of request / response body.
- **content_type=EmptyValue** (*str*) – content-type header value for request / response.
- **always_hash_content=True** (*bool*) – When True, content and content_type must be provided. Read *Skipping content checks* to learn more.
- **ext=None** (*str*) – An external Hawk string. If not None, this value will be signed so that the sender can trust it.
- **app=None** (*str*) – A Hawk string identifying an external application.
- **dlg=None** (*str*) – A Hawk string identifying a "delegated by" value.
- **timestamp=utc_now()** – A unix timestamp integer, in UTC
- **nonce=None** (*str*) – A string that when coupled with the timestamp will uniquely identify this request / response.
- **seen_nonce=None** (*callable*) – A callable that returns True if a nonce has been seen. See *Using a nonce to prevent replay attacks* for details.

mohawk.base.**EmptyValue = EmptyValue**
This represents an empty value but not None.

This is typically used as a placeholder of a default value so that internal code can differentiate it from None.

Developers

Grab the source from Github: https://github.com/kumar303/mohawk

Run the tests

You can run the full test suite with the tox command:

```
tox
```

To just run Python 2.7 unit tests type:

```
tox -e py27
```

To just run doctests type:

```
tox -e docs
```

Set up an environment

Using a virtualenv you can set yourself up for development like this:

```
pip install -r requirements/dev.txt
python setup.py develop
```

Build the docs

In your virtualenv, you can build the docs like this:

```
make -C docs/ html doctest
open docs/_build/html/index.html
```

Publish a release

Do this first to prepare for a release:

- make sure the changelog is up to date
- make sure you bumped the module version in `setup.py`
- commit, tag (like `git tag 0.3.1`), and push upstream (like `git push --tags upstream`).

Run this from the repository root to publish a new release to PyPI as both a source distribution and wheel:

```
rm -rf dist/*
python setup.py sdist bdist_wheel
twine upload dist/*
```

Why Mohawk?

- I started using PyHawk because it was written by Austin King and he's awesome.
- PyHawk is a direct port from Node but this did not seem to fit right with Python, especially in how Node's style is to attempt internal error recovery and Python's style is to raise exceptions that calling code can recover from.
- I was paranoid about how PyHawk (and maybe the Node lib too) makes it easy to ignore content hashing. If programmers accidentally disregard hash checks then that would be bad.

- I started patching PyHawk but became confused about the lifecycle of the request/response.
- PyHawk didn't have a lot of tests for edge cases (like content tampering) so it was hard to patch.
- I started on some Django middleware using PyHawk and found myself creating a lot of adapters for undocumented internal dictionary structures which felt wrong.
- The PyHawk/Node API relies on pre-generated header artifacts but this feels clunky to me. I wanted that to be an implementation detail.
- The required order in which you need to pre-generate artifacts is not implicitly enforced by the PyHawk/Node API which can lead to mistakes if programmers re-use objects across requests.
- I re-wrote the class/function interface into something that I thought made sense then I re-wrote it three more times until it started to actually make sense.
- I developed test first with a comprehensive suite focusing on the threat model that Hawk is designed to protect you from. This helped me arrive at an API that should help developers write secure applications by default.
- I re-used a lot of PyHawk code :)

CHAPTER 4

Framework integration

Mohawk is a low level library that focuses on Hawk communication. The following higher-level libraries integrate Mohawk into specific web frameworks:

- Hawkrest: adds Hawk to Django Rest Framework
- Did we miss one? Send a pull request so we can link to it.

CHAPTER 5

TODO

- Support NTP-like (but secure) synchronization for local server time. See TLSdate.
- Support auto-retrying a *mohawk.Sender* request with an offset if there is timestamp skew.

Changelog

- **UNRELEASED**
  - (Unreleased features should be listed here.)
- **1.0.0** (2019-01-09)
  - **Security related**: Bewit MACs were not compared in constant time and were thus possibly circumventable by an attacker.
  - **Breaking change**: Escape characters in header values (such as a back slash) are no longer allowed, potentially breaking clients that depended on this behavior. See https://github.com/kumar303/mohawk/issues/34
  - A sender is allowed to omit the content hash as long as their request has no content. The *mohawk.Receiver* will skip the content hash check in this situation, regardless of the value of `accept_untrusted_content`. See *Empty requests* for more details.
  - Introduced max limit of 4096 characters in the Authorization header
  - Changed default values of `content` and `content_type` arguments to *mohawk.base.EmptyValue* in order to differentiate between misconfiguration and cases where these arguments are explicitly given as `None` (as with some web frameworks). See *Skipping content checks* for more details.
  - Failing to pass `content` and `content_type` arguments to *mohawk.Receiver* or *mohawk.Sender.accept_response()* without specifying `accept_untrusted_content=True` will now raise *mohawk.exc.MissingContent* instead of `ValueError`.
- **0.3.4** (2017-01-07)
  - Fixed `AttributeError` exception (it now raises *mohawk.exc.MissingAuthorization*) for cases when the client sends a None type authorization header. See issue 23.
  - Fixed Python 3.6 compatibility problem (a regex pattern was using the deprecated `LOCALE` flag). See issue 32.
- **0.3.3** (2016-07-12)

  - Fixed some cases where *mohawk.exc.MacMismatch* was raised instead of *mohawk.exc.MisComputedContentHash*. This follows the Hawk HTTP authorization scheme implementation more closely. See issue 15.
  - Published as a Python wheel

- **0.3.2.1** (2016-02-25)
  - Re-did the `0.3.2` release; the tag was missing some commits. D'oh.
- **0.3.2** (2016-02-24)
  - Improved Python 3 support.
  - Fixed bug in handling `ext` values that have more than one equal sign.
  - Configuration objects no longer need to be strictly dicts.
- **0.3.1** (2016-01-07)
  - Initial bewit support (undocumented). Complete support with documentation is still forthcoming.
- **0.3.0** (2015-06-22)
  - **Breaking change:** The `seen_nonce()` callback signature has changed. You must update your callback from `seen_nonce(nonce,timestamp)` to `seen_nonce(sender_id,nonce,timestamp)` to avoid unnecessary collisions. See *Using a nonce to prevent replay attacks* for details.
- **0.2.2** (2015-01-05)
  - Receiver can now respond with a `WWW-Authenticate` header so that senders can adjust their timestamps. Thanks to jcwilson for the patch.
- **0.2.1** (2014-03-03)
  - Fixed Python 2 bug in how unicode was converted to bytes when calculating a payload hash.
- **0.2.0** (2014-03-03)
  - Added support for Python 3.3 or greater.
  - Added support for Python 2.6 (this was just a test suite fix).
  - Added six as dependency.
  - *mohawk.Sender.request_header* and *mohawk.Receiver.response_header* are now Unicode objects. They will never contain non-ascii characters though.
- **0.1.0** (2014-02-19)
  - Implemented optional content hashing per spec but in a less error prone way
  - Added complete documentation
- **0.0.4** (2014-02-11)
  - Bug fix: response processing now re-uses sender's nonce and timestamp per the Node Hawk lib
  - No longer assume content-type: text/plain if content type is not specificed
- **0.0.3** (2014-02-07)
  - Bug fix: Macs were made using URL safe base64 encoding which differs from the Node Hawk lib (it just uses regular base64)
  - exposed `localtime_in_seconds` on `TokenExpired` exception per Hawk spec
  - better localtime offset and skew handling

- **0.0.2** (2014-02-06)
  - Responding with a custom ext now works
  - Protected app and dlg according to spec when accepting responses
- **0.0.1** (2014-02-05)
  - initial release of partial implementation

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

m