HTTPolice Documentation

Release 0.5.2

Vasiliy Faronov

Jul 27, 2017

Contents

HTTPolice is a lint for HTTP requests and responses. It checks them for conformance to standards and best practices.

This manual explains all features of HTTPolice in detail. If you want a brief introduction, see the *Quickstart*.

There is also a list of all notices that HTTPolice can output.

For recent changes in HTTPolice, see the changelog.

CHAPTER 1

Quickstart

Installation

HTTPolice is a Python package that can be installed with pip (on Python 2.7 or 3.4+):

```
$ pip install HTTPolice
```

If you're not familiar with pip, check the manual's *Installation* section.

Using HAR files

Let's start with something easy.

If you're running Google Chrome, Firefox, or Microsoft Edge, you can use their developer tools to export HTTP requests and responses as a HAR file, which can then be analyzed by HTTPolice.

For example, in Firefox, press F12 to open the toolbox, and switch to its Network pane. Then, open a simple Web site—I'm going to use h2o.example.net here. All HTTP exchanges made by the browser appear in the Network pane. Right-click inside that pane and select "Save All As HAR".

Then feed this HAR file to HTTPolice:

```
$ httpolice -i har /path/to/file.har
------------ request: GET /analytics.js
------------ response: 200 OK
C 1035 Deprecated media type text/javascript
D 1168 Age header implies response from cache
C 1258 HTTP/2 should use ALTSVC frame instead of Alt-Svc header
------------ request: GET /r/collect?v=1&_v=j49&a=2057119860&t=pageview&_s=1...
------------ response: 200 OK
E 1108 Wrong day of week in Expires
C 1162 Pragma: no-cache is for requests
C 1258 HTTP/2 should use ALTSVC frame instead of Alt-Svc header
```

```
------------ request: GET /repos/h2o/h2o?callback=callback
------------ response: 200 OK
C 1277 Obsolete 'X-' prefix in X-RateLimit-Limit
C 1277 Obsolete 'X-' prefix in X-RateLimit-Remaining
C 1277 Obsolete 'X-' prefix in X-RateLimit-Reset
C 1277 Obsolete 'X-' prefix in X-Served-By
```

Better reports

By default, HTTPolice prints a simple text report which may be hard to understand. Use the `-o html` option to make a detailed HTML report instead. You will also need to redirect it to a file:

```
$ httpolice -i har -o html /path/to/file.har >report.html
```

Open `report.html` in your Web browser and enjoy.

Using mitmproxy

What if you have an HTTP API that is accessed by special clients? Let's say curl is special enough:

```
$ curl -ksiX POST https://eve-demo.herokuapp.com/people \
>   -H 'Content-Type: application/json' \
>   -d '{"firstname":"John", "lastname":"Smith"}'
HTTP/1.1 201 CREATED
Connection: keep-alive
Content-Type: application/json
Content-Length: 279
Server: Eve/0.6.1 Werkzeug/0.10.4 Python/2.7.4
Date: Mon, 25 Apr 2016 09:21:32 GMT
Via: 1.1 vegur

{"_links": {"self": {"href": "people/571de19c4fd7bd0003356826", "title": "person"}},
↪"_etag": "3b1f9c356f87a615645e2e51f8d3e05e0e462c03", "_id":
↪"571de19c4fd7bd0003356826", "_created": "Mon, 25 Apr 2016 09:21:32 GMT", "_updated
↪": "Mon, 25 Apr 2016 09:21:32 GMT", "_status": "OK"}
```

How do you get this into HTTPolice?

One way is to use mitmproxy, an advanced tool for intercepting HTTP traffic. Install it in a Python 3.5+ environment with HTTPolice integration:

```
$ pip3 install mitmproxy-HTTPolice
```

(see also the instructions for installing mitmproxy from source).

We're going to use mitmproxy's command-line tool—mitmdump. The following command will start mitmdump as a reverse proxy in front of your API on port 8080, with HTTPolice integration:

```
$ mitmdump --reverse https://eve-demo.herokuapp.com \
>   -s "`python3 -m mitmproxy_httpolice` -o html -w report.html --tail 5"
```

Now tell your client to talk to port 8080 instead of directly to the API:

```
$ curl -ksiX POST https://localhost:8080/people \
>   -H 'Content-Type: application/json' \
>   -d '{"firstname":"Mark", "lastname":"Williams"}'
```

In the output of mitmdump, you will see that it has intercepted the exchange. Now you can open `report.html` and see what HTTPolice thinks of it. With the `--tail 5` option, `report.html` will always contain the **last 5 exchanges** seen by mitmproxy (the latest one is at the bottom).

Note: If you prefer Fiddler over mitmproxy, you can use Fiddler's HAR 1.2 export to get the data into HTTPolice.

Django integration

Suppose you're building a Web application with Django (1.8+). You probably have a test suite that makes requests to your app and checks responses. You can easily instrument this test suite with HTTPolice and get instant feedback when you break the protocol.

```
$ pip install Django-HTTPolice
```

Add the HTTPolice middleware to the top of your middleware list:

```
MIDDLEWARE = [
    'django_httpolice.HTTPoliceMiddleware',
    'django.middleware.common.CommonMiddleware',
    # ...
]
```

Add a couple settings:

```
HTTPOLICE_ENABLE = True
HTTPOLICE_RAISE = 'error'
```

Now let's run the tests and see what's broken:

```
$ python manage.py test
...E
======================================================================
ERROR: test_query_plain (example_app.test.ExampleTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  [...]
  File "[...]/django_httpolice/middleware.py", line 92, in process_response
    raise ProtocolError(exchange)
django_httpolice.common.ProtocolError: HTTPolice found problems in this response:
------------ request: GET /api/v1/words/?query=er
C 1070 No User-Agent header
------------ response: 200 OK
E 1038 Bad JSON body


----------------------------------------------------------------------
Ran 4 tests in 0.380s

FAILED (errors=1)
```

In this example, the app sent a wrong `Content-Type` header and HTTPolice caught it.

More options

There are other ways to get your data into HTTPolice. Check the *full manual*.

CHAPTER 2

Installation

HTTPolice is a Python package that requires Python 2.7 or 3.4+. It can be installed like all other Python packages: with pip from PyPI.

If you're not familiar with pip, you may need to install it manually or from your OS distribution. You may also need development files and tools to compile dependencies.

PyPy (the 2.7 variant) is also supported, but you may experience problems with older PyPy versions (5.3.1 should be OK).

On Debian/Ubuntu

```
$ sudo apt-get install python-pip python-dev libxml2-dev libxslt1-dev zlib1g-dev␣
↪libffi-dev
```

Then, to install the HTTPolice command-line tool into `~/.local/bin`:

```
$ pip install --user HTTPolice
```

Or, to install it system-wide:

```
$ sudo pip install HTTPolice
```

Check that the installation was successful:

```
$ httpolice --version
HTTPolice 0.4.0
```

On Fedora

Same as above, but use the following command to install dependencies:

```
$ sudo dnf install python-pip gcc gcc-c++ redhat-rpm-config python-devel libxml2-
↪devel libxslt-devel libffi-devel
```

On Windows

HTTPolice uses libraries (lxml and brotlipy) that include binary CPython extensions. You probably want precompiled versions of these extensions, and to get them, you may need specific versions of Python, lxml and brotlipy.

For example, at the time of writing, you can install Python 3.5 (**not** 3.6) and then simply do:

```
C:\Users\Vasiliy\...\Python35>Scripts\pip install HTTPolice
```

Check that the installation was successful:

```
C:\Users\Vasiliy\...\Python35>Scripts\httpolice --version
HTTPolice 0.4.0
```

However, it's possible that new versions of lxml and brotlipy might not have precompiled binaries for your version of Python, and then you will have to check the PyPI pages of these libraries to find a version that has suitable binaries (look for `*-win32.whl`), and install those specific versions **before** installing HTTPolice. For example:

```
C:\Users\Vasiliy\...\Python35>Scripts\pip install lxml==3.7.2
C:\Users\Vasiliy\...\Python35>Scripts\pip install brotlipy==0.6.0
```

CHAPTER 3

General concepts

Exchanges

HTTPolice takes HTTP *exchanges* (also known as *transactions*) as input. Every exchange can consist of 1 request and 1+ responses. Usually there is just 1 response, but sometimes there are interim (1xx) responses before the main one.

If you only want to check the request, you can omit responses from the exchange.

On the other hand, if you only want to check the *responses*, you should still provide the request (if possible), because responses cannot be properly analyzed without it. If you really have no access to the request, you can omit it, but **many checks will be disabled**.

Reports

The output of HTTPolice is a *report* containing *notices*.

Every notice has an ID (such as "1061") that can be used to *silence* it, and one of three *severities*:

error Something is clearly wrong. For example, a "MUST" requirement of a standard is clearly violated.

Please note that **not all errors may be actual problems**. Sometimes there is a good reason to violate a standard. Sometimes you just don't care. You decide which errors to fix and which to ignore. If you don't want to see an error, you can *silence* it.

comment Something is *possibly* wrong or sub-optimal, but HTTPolice isn't sure. For example, a "SHOULD" requirement of a standard is clearly violated.

debug This just explains why HTTPolice did (or did not do) something. For example, when HTTPolice thinks that a response was served from cache, it will report a debug notice to explain why it thinks so. This may help you understand further cache-related notices for that response.

Silencing unwanted notices

You can *silence* notices that you don't want to see. They will disappear from reports and from the *Python API*.

Please note that some notice IDs can stand for a range of problems. For example, most errors in header syntax are reported as notice 1000, so if you silence it, you **lose a big chunk** of HTTPolice's functionality.

Silencing globally

When using the `httpolice` command-line tool, you can use the `-s` option to specify notice IDs to silence:

```
$ httpolice -s 1089 -s 1194 ...
```

Integration methods have similar mechanisms. For example, mitmproxy integration understands the same `-s` option.

Silencing locally

You can also silence notices on individual messages by adding the special `HTTPolice-Silence` header to them. Its value is a comma-separated list of notice IDs. For example:

```
HTTP/1.1 405 Method Not Allowed
Content-Length: 0
HTTPolice-Silence: 1089, 1110
```

Requests can also silence notices on responses (but not vice-versa) by adding a `resp` keyword after an ID:

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/5.0
HTTPolice-Silence: 1033 resp, 1031
```

CHAPTER 4

Analyzing raw TCP streams

An obvious way to capture HTTP requests and responses is to dump them with a network sniffer. This only works for cleartext connections (without TLS encryption), but on the other hand, you don't need to change your clients or servers.

HTTPolice can parse HTTP/1.x streams from the ground up. Parsing HTTP/2 is not yet supported.

Using tcpflow

You may be familiar with tcpdump, but it won't work: HTTPolice needs the reassembled TCP streams, not individual packets. You can get these streams with a tool called tcpflow:

```
$ mkdir dump

$ cd dump/

$ sudo tcpflow -T'%t-%A-%a-%B-%b-%#' port 80
tcpflow: listening on wlp4s0
```

(Note the `-T` option—it is necessary to get the right output.)

tcpflow starts capturing all connections to or from TCP port 80. For example, you can launch a Web browser and go to an 'http:' site. Once you are done, exit the browser, then stop tcpflow with Ctrl+C. (It is important that connections are closed before tcpflow shuts down, otherwise they may be incomplete.)

Now you have one or more pairs of stream files:

```
$ ls
1469847441-054.175.219.008-00080-172.016.000.100-38656-0  report.xml
1469847441-172.016.000.100-38656-054.175.219.008-00080-0
```

Tell HTTPolice to read this directory with the `tcpflow` input format:

```
$ httpolice -i tcpflow .
```

HTTPolice will combine the files into pairs based on their filenames. Due to a limitation in tcpflow, this only works if every combination of source+destination address+port is unique. If there are duplicates, you will get an error.

It's OK if you capture some streams that are not HTTP/1.x. HTTPolice will just complain with notices such as 1279. This means you can run tcpflow without a filter, capturing *all* TCP traffic on a given network interface, and then let HTTPolice sort it out while *silencing* those notices:

```
$ sudo tcpflow -T'%t-%A-%a-%B-%b-%#'

$ httpolice -i tcpflow -o html -s 1279 . >../report.html
```

Using tcpick

tcpick is another tool for reassembling TCP streams. It doesn't have the "unique port" limitation of tcpflow, but it has a different problem: sometimes it produces files that are clearly invalid HTTP streams (HTTPolice will fail to parse them with notices like 1009).

Anyway, using it is very similar to using tcpflow:

```
$ mkdir dump

$ cd dump/

$ sudo tcpick -wR -F2 'port 80'
Starting tcpick 0.2.1 at 2016-07-30 06:14 MSK
Timeout for connections is 600
tcpick: listening on wlp4s0
setting filter: "port 80"
[...]
^C
3837 packets captured
30 tcp sessions detected

$ httpolice -i tcpick .
```

(Note the `-wR -F2` options.)

Other sniffers

If you use some other tool to capture the TCP streams, use the `streams` input format to pass pairs of files:

```
$ httpolice -i streams requests1.dat responses1.dat requests2.dat ...
```

Or `req-stream` if you only have request streams:

```
$ httpolice -i req-stream requests1.dat requests2.dat ...
```

Or `resp-stream` if you only have response streams (*not recommended*):

```
$ httpolice -i resp-stream responses1.dat responses2.dat ...
```

Note that `resp-stream` may not work at all if any of the requests are HEAD, because responses to HEAD are parsed differently.

Combined format

Sometimes you want to compose an HTTP exchange by hand, to test something. To make this easier, there's a special input format that combines the request and response streams into one file:

```
The lines at the beginning are ignored.
You can use them for comments.

======== BEGIN INBOUND STREAM ========
GET / HTTP/1.1
Host: example.com
User-Agent: demo

======== BEGIN OUTBOUND STREAM ========
HTTP/1.1 200 OK
Date: Thu, 31 Dec 2015 18:26:56 GMT
Content-Type: text/plain
Connection: close

Hello world!
```

It must be saved with **CRLF (Windows)** line endings.

Also, for this format, the filename suffix (extension) is important. If it is `.https`, the request URI is assumed to have an `https:` scheme. If it is `.noscheme`, the scheme is unknown. Otherwise, the `http:` scheme is assumed.

Now, tell HTTPolice to use the `combined` format:

```
$ httpolice -i combined exchange1.txt
```

More examples can be found in HTTPolice's test suite.

CHAPTER 5

Analyzing HAR files

HAR is a quasi-standardized JSON format for saving HTTP traffic. It is supported by many HTTP-related tools, including developer consoles of some Web browsers.

HTTPolice can analyze HAR files with the `-i har` option:

```
$ httpolice -i har myfile.har
```

However, please note that HAR support in major Web browsers is **erratic**. HTTPolice tries to do a reasonable job on files exported from Chrome, Firefox, and Edge, but some information is simply lost.

If HTTPolice fails on your HAR files, feel free to submit an issue (don't forget to attach the files), and I'll see what can be done about it.

CHAPTER 6

Viewing reports

Text reports

By default, HTTPolice produces simple plain text reports like this:

```
------------ request: PUT /articles/109226/
E 1000 Malformed If-Match header
C 1093 User-Agent contains no actual product
------------ response: 204 No Content
C 1110 204 response with no Date header
E 1221 Strict-Transport-Security without TLS
------------ request: POST /articles/109226/comments/
...
```

They are intended to be suitable for grep and other Unix-like tools.

HTML reports

Use the `-o html` option to enable much more detailed HTML reports. These include explanations for every notice, cross-referenced with the standards, as well as previews of the actual requests and responses.

Please note that these previews **do not represent exactly** what was sent on the wire. For example, in an HTTP/1.x request, a header may have been split into two physical lines, but will be rendered as one line in the report.

Options

In the top right hand corner of an HTML report, there's an *options* menu.

The first three options allow you to filter the report on the fly. This is independent from *silencing*: you cannot undo silencing with these options.

Hide boring exchanges Check this to hide all exchanges where no problems were found (only debug notices or none at all).

Boring notices Additional notice IDs or severities that should not be considered problems. For example: `1089 1135 C` (C for "comment").

Hide boring notices Check this if you don't want to see those boring notices at all. They will be hidden even from exchanges that have other problems.

Other options:

Show remarks Check this to show remarks before requests and responses, if any. The nature of these remarks depends on where the data comes from. If you use the `httpolice` command-line tool, remarks will contain the names of the input files and (for *streams input* only) the byte offsets within those files. This can help with debugging.

HTML in text

What if you want full details like in HTML reports, but on a textual display? Perhaps you're running HTTPolice on a remote machine via ssh.

You can simply use a text-mode Web browser like w3m:

```
$ httpolice -o html ... | w3m -M -T text/html
```

Exit status

When using the `httpolice` command-line tool, there's another channel of information besides the report itself: the command's exit status. If you pass the `--fail-on` option, the exit status will be non-zero if any notices with the given severity (or higher) have been reported. For example:

```
$ httpolice -i combined --fail-on=comment test/combined_data/1125_1
------------ request: GET /
------------ response: 304 Not Modified
E 1125 Probably wrong use of status code 304

$ echo $?
1
```

This can be used to take automated action (like failing tests) without parsing the report itself.

CHAPTER 7

Python API

HTTPolice can be used as a Python library: for example, to analyze requests or responses as part of a test suite. It is **not intended** to be used inside live production processes.

Example

```
import io
import httpolice

exchanges = [
    httpolice.Exchange(
        httpolice.Request(u'https',
                          u'GET', u'/index.html', u'HTTP/1.1',
                          [(u'Host', b'example.com')],
                          b''),
        [
            httpolice.Response(u'HTTP/1.1', 401, u'Unauthorized',
                               [(u'Content-Type', b'text/plain')],
                               b'No way!'),
        ]
    )
]

bad_exchanges = []

for exch in exchanges:
    exch.silence([1089, 1227])      # Errors we don't care about
    httpolice.check_exchange(exch)
    if any(notice.severity > httpolice.Severity.comment
           for resp in exch.responses       # We only care about responses
           for notice in resp.notices):
        bad_exchanges.append(exch)
```

```
if bad_exchanges:
    with io.open('report.html', 'wb') as f:
        httpolice.html_report(bad_exchanges, f)
    print('%d exchanges had problems; report written to file' %
          len(bad_exchanges))
```

API reference

class `httpolice.`**`Request`**(*scheme*, *method*, *target*, *version*, *header_entries*, *body*, *trailer_entries=None*, *remark=None*)

Parameters

- **`scheme`** – The scheme of the request URI, as a Unicode string (usually `u'http'` or `u'https'`), or `None` if unknown (this disables some checks).
- **`method`** – The request method, as a Unicode string.
- **`target`** – The request target, as a Unicode string. It must be in one of the four forms defined by RFC 7230. (For HTTP/2, it can be reconstructed from pseudo-headers.)
- **`version`** – The request's protocol version, as a Unicode string, or `None` if unknown (this disables some checks).

  For requests sent over HTTP/1.x connections, this should be the HTTP version sent in the request line, such as `u'HTTP/1.0'` or `u'HTTP/1.1'`.

  For requests sent over HTTP/2 connections, this should be `u'HTTP/2'`.
- **`header_entries`** – A list of the request's headers (may be empty). It must **not** include HTTP/2 pseudo-headers.

  Every item of the list must be a `(name, value)` pair.

  `name` must be a Unicode string.

  `value` may be a byte string or a Unicode string. If it is Unicode, HTTPolice will assume that it has been decoded from ISO-8859-1 (the historic encoding of HTTP), and will encode it back into ISO-8859-1 before any processing.
- **`body`** – The request's payload body, as a **byte string**, or `None` if unknown (this disables some checks).

  If the request has no payload (like a GET request), this should be the empty string `b''`.

  This must be the payload body as defined by RFC 7230: **after** removing any `Transfer-Encoding` (like `chunked`), but **before** removing any `Content-Encoding` (like `gzip`).
- **`trailer_entries`** – A list of headers from the request's trailer part (as found in chunked coding or HTTP/2), or `None` if there is no trailer part.

  The format is the same as for `header_entries`.
- **`remark`** – If not `None`, this Unicode string will be shown above the request in HTML reports (when the appropriate option is enabled). For example, it can be used to identify the source of the data: `u'from somefile.dat, offset 1337'`.

`notices`

A list of *`Complaint`* instances reported on this object.

silence(*notice_ids*)
Silence unwanted notices on this object.

Parameters **notice_ids** – An iterable of notice IDs that will be silenced on this object, so they don't appear in `notices` or in reports.

class `httpolice.`**Response**(*version*, *status*, *reason*, *header_entries*, *body*, *trailer_entries=None*, *remark=None*)

Parameters

- **version** – The response's protocol version, as a Unicode string, or `None` if unknown (this disables some checks).

  For responses sent over HTTP/1.x connections, this should be the HTTP version sent in the status line, such as `u'HTTP/1.0'` or `u'HTTP/1.1'`.

  For responses sent over HTTP/2 connections, this should be `u'HTTP/2'`.

- **status** – The response's status code, as an integer.

- **reason** – The response's reason phrase (such as "OK" or "Not Found"), as a Unicode string, or `None` if unknown (as in HTTP/2).

- **header_entries** – A list of the response's headers (may be empty). It must **not** include HTTP/2 pseudo-headers.

  Every item of the list must be a `(name, value)` pair.

  `name` must be a Unicode string.

  `value` may be a byte string or a Unicode string. If it is Unicode, HTTPolice will assume that it has been decoded from ISO-8859-1 (the historic encoding of HTTP), and will encode it back into ISO-8859-1 before any processing.

- **body** – The response's payload body, as a **byte string**, or `None` if unknown (this disables some checks).

  If the response has no payload (like 204 or 304 responses), this should be the empty string `b''`.

  This must be the payload body as defined by RFC 7230: **after** removing any `Transfer-Encoding` (like `chunked`), but **before** removing any `Content-Encoding` (like `gzip`).

- **trailer_entries** – A list of headers from the response's trailer part (as found in chunked coding or HTTP/2), or `None` if there is no trailer part.

  The format is the same as for `header_entries`.

- **remark** – If not `None`, this Unicode string will be shown above this response in HTML reports (when the appropriate option is enabled). For example, it can be used to identify the source of the data: `u'from somefile.dat, offset 1337'`.

notices
A list of *`Complaint`* instances reported on this object.

silence(*notice_ids*)
Silence unwanted notices on this object.

Parameters **notice_ids** – An iterable of notice IDs that will be silenced on this object, so they don't appear in `notices` or in reports.

class `httpolice.`**Exchange**(*req*, *resps*)

Parameters

- **req** – The request, as a *Request* object. If it is not available, you can pass `None`, and the responses will be checked on their own. However, this **disables many checks** which rely on context information from the request.
- **resps** – The responses to `req`, as a list of *Response* objects. Usually this will be a list of 1 element. If you only want to check the request, pass an empty list `[]`.

request
The *Request* object passed to the constructor.

responses
The list of *Response* objects passed to the constructor.

silence(*notice_ids*)
Silence unwanted notices on this object.

Parameters **notice_ids** – An iterable of notice IDs that will be silenced on this object, so they don't appear in `notices` or in reports.

`httpolice.`**check_exchange**(*exch*)
Run all checks on the exchange `exch`, modifying it in place.

class `httpolice.`**Complaint**
A notice as reported in a particular place.

Create new instance of Complaint(notice, context)

id
The notice's ID (an integer).

severity
The notice's severity, as a member of the *Severity* enumeration.

class `httpolice.`**Severity**
A notice's severity.

This is a Python 3.4 style enumeration with the additional feature that its members are ordered:

```
>>> Severity.comment < Severity.error
True
```

The underlying values of this enumeration are **not** part of the API.

comment = 1

debug = 0

error = 2

httpolice.**text_report**(*exchanges*, *buf*)
Generate a plain-text report with check results.

Parameters

- **exchanges** – An iterable of *Exchange* objects. They must be already processed by *check_exchange()*.
- **buf** – The file (or file-like object) to which the report will be written. It must be opened in binary mode (not text).

httpolice.**html_report**(*exchanges*, *buf*)
Generate an HTML report with check results.

Parameters

- **exchanges** – An iterable of *Exchange* objects. They must be already processed by *check_exchange()*.
- **buf** – The file (or file-like object) to which the report will be written. It must be opened in binary mode (not text).

Integration helpers

Functions that may be useful for integrating with HTTPolice.

httpolice.helpers.**headers_from_cgi**(*cgi_dict*)
Convert CGI variables into header entries.

Parameters **cgi_dict** – A mapping of CGI-like meta-variables, as found in (for example) WSGI's `environ` or `django.http.HttpRequest.META`.

Returns A list of header entries, suitable for passing into *httpolice.Request*.

httpolice.helpers.**pop_pseudo_headers**(*entries*)
Remove and return HTTP/2 pseudo-headers from a list of headers.

Parameters **entries** – A list of header name-value pairs, as would be passed to *httpolice.Request* or *httpolice.Response*. It will be modified in-place by removing all names that start with a colon (:).

Returns A dictionary of the removed pseudo-headers.

Integration packages:

- mitmproxy integration
- Django integration
- Chrome extension (third-party)

Python Module Index

h

Index