# Contents

1 Introduction 3
2 Installation 5
3 About Certificates 9
4 How mitmproxy works 13
5 Modes of Operation 19
6 mitmproxy 25
7 mitmdump 35
8 mitmweb 37
9 Configuration 39
10 Anticache 41
11 Filter expressions 43
12 Replacements 45
13 Client-side replay 47
14 Server-side replay 49
15 Set Headers 51
16 Ignore Domains 53
17 Proxy Authentication 55
18 Reverse Proxy 57
19 Response Streaming 59
20 SOCKS Mode 61
**mitmproxy** is an interactive man-in-the-middle proxy for HTTP and HTTPS with a console interface.

**mitmdump** is the command-line version of mitmproxy. Think tcpdump for HTTP.

**mitmweb** is a web-based interface for mitmproxy.

Documentation, tutorials and distribution packages can be found on the mitmproxy website: mitmproxy.org

---

### Features

- Intercept HTTP & HTTPS requests and responses and modify them on the fly
- Save complete HTTP conversations for later replay and analysis
- Replay the client-side of an HTTP conversations
- Replay HTTP responses of a previously recorded server
- Reverse proxy mode to forward traffic to a specified server
- Transparent proxy mode on OSX and Linux
- Make scripted changes to HTTP traffic using Python
- SSL/TLS certificates for interception are generated on the fly
- And much, much more...
CHAPTER 1

Introduction

mitmproxy is an interactive man-in-the-middle proxy for HTTP and HTTPS with a console interface.
mitmdump is the command-line version of mitmproxy. Think tcpdump for HTTP.
mitmweb is a web-based interface for mitmproxy.

Documentation, tutorials and distribution packages can be found on the mitmproxy website: mitmproxy.org

Features

- Intercept HTTP & HTTPS requests and responses and modify them on the fly
- Save complete HTTP conversations for later replay and analysis
- Replay the client-side of an HTTP conversations
- Replay HTTP responses of a previously recorded server
- Reverse proxy mode to forward traffic to a specified server
- Transparent proxy mode on OSX and Linux
- Make scripted changes to HTTP traffic using Python
- SSL/TLS certificates for interception are generated on the fly
- And much, much more...
Please follow the steps for your operating system. Once installation is complete, you can run `mitmproxy`, `mitmdump` or `mitmweb` from a terminal.

### 2.1 Installation on macOS

You can use Homebrew to install everything:

```
brew install mitmproxy
```

Or you can download the pre-built binary packages from our releases.

### 2.2 Installation on Windows

The recommended way to install mitmproxy on Windows is to use the installer provided at mitmproxy.org. After installation, you’ll find shortcuts for `mitmweb` (the web-based interface) and `mitmdump` in the start menu. Both executables are added to your PATH and can be invoked from the command line.

**Note:** Mitmproxy’s console interface is not supported on Windows, but you can use mitmweb (the web-based interface) and mitmdump.

### 2.3 Installation on Linux

The recommended way to run mitmproxy on Linux is to use the pre-built binaries provided at releases.
Our pre-built binaries provide you with the latest version of mitmproxy, a self-contained Python 3.5 environment and a recent version of OpenSSL that supports HTTP/2. Of course, you can also install mitmproxy from source if you prefer that (see Advanced Installation).

2.4 Advanced Installation

2.4.1 Docker Images

You can also use the official mitmproxy images from DockerHub. That being said, our portable binaries are just as easy to install and even easier to use.

2.4.2 Installation on Arch Linux

mitmproxy has been added into the [community] repository. Use pacman to install it:

```bash
>>> sudo pacman -S mitmproxy
```

2.4.3 Installation from Source on Ubuntu

Ubuntu comes with Python but we need to install pip3, python3-dev and several libraries. This was tested on a fully patched installation of Ubuntu 16.04.

```bash
sudo apt-get install python3-dev python3-pip libffi-dev libssl-dev
sudo pip3 install mitmproxy  # or pip3 install --user mitmproxy
```

On older Ubuntu versions, e.g., 12.04 and 14.04, you may need to install a newer version of Python. mitmproxy requires Python 3.5 or higher. Please take a look at pyenv. Make sure to have an up-to-date version of pip by running pip3 install -U pip.

2.4.4 Installation from Source on Fedora

Fedora comes with Python but we need to install pip3, python3-dev and several libraries. This was tested on a fully patched installation of Fedora 24.

```bash
sudo dnf install make gcc redhat-rpm-config python3-devel python3-pip libffi-devel openssl-devel
sudo pip3 install mitmproxy  # or pip3 install --user mitmproxy
```

Make sure to have an up-to-date version of pip by running pip3 install -U pip.

2.4.5 Installation from Source on Windows

Note: Mitmproxy’s console interface is not supported on Windows, but you can use mitmweb (the web-based interface) and mitmdump.

First, install the latest version of Python 3.5 or later from the Python website. During installation, make sure to select Add Python to PATH.
Mitmproxy has no other dependencies on Windows. You can now install mitmproxy by running

```
pip3 install mitmproxy
```

**2.4.6 Latest Development Version**

If you would like to install mitmproxy directly from the master branch on GitHub or would like to get set up to contribute to the project, install the dependencies as you would for a regular installation from source. Then see the project’s README on GitHub. You can check your system information by running: `mitmproxy --version`
CHAPTER 3

About Certificates

3.1 Introduction

Mitmproxy can decrypt encrypted traffic on the fly, as long as the client trusts its built-in certificate authority. Usually this means that the mitmproxy CA certificates have to be installed on the client device.

3.2 Quick Setup

By far the easiest way to install the mitmproxy certificates is to use the built-in certificate installation app. To do this, just start mitmproxy and configure your target device with the correct proxy settings. Now start a browser on the device, and visit the magic domain mitm.it. You should see something like this:

Click to install the mitmproxy certificate:

Click on the relevant icon, follow the setup instructions for the platform you’re on and you are good to go.
## 3.3 Installing the mitmproxy CA certificate manually

Sometimes using the quick install app is not an option - Java or the iOS Simulator spring to mind - or you just need to do it manually for some other reason. Below is a list of pointers to manual certificate installation documentation for some common platforms.

The mitmproxy CA cert is located in `~/.mitmproxy` after it has been generated at the first start of mitmproxy.

### 3.3.1 iOS

See [http://jasdev.me/intercepting-ios-traffic](http://jasdev.me/intercepting-ios-traffic)


### 3.3.2 iOS Simulator

See [https://github.com/ADVTOOLS/ADVTrustStore#how-to-use-advtruststore](https://github.com/ADVTOOLS/ADVTrustStore#how-to-use-advtruststore)

### 3.3.3 Java


### 3.3.4 Android/Android Simulator


### 3.3.5 Windows


### 3.3.6 Windows (automated)

```bash
>>> certutil.exe -importpfx Root mitmproxy-ca-cert.p12
```


### 3.3.7 Mac OS X


### 3.3.8 Ubuntu/Debian

3.3.9 Mozilla Firefox

See https://wiki.mozilla.org/MozillaRootCertificate#Mozilla_Firefox

3.3.10 Chrome on Linux

See https://code.google.com/p/chromium/wiki/LinuxCertManagement

3.4 The mitmproxy certificate authority

The first time mitmproxy or mitmdump is run, the mitmproxy Certificate Authority (CA) is created in the config directory (~/.mitmproxy by default). This CA is used for on-the-fly generation of dummy certificates for each of the SSL sites that your client visits. Since your browser won’t trust the mitmproxy CA out of the box, you will see an SSL certificate warning every time you visit a new SSL domain through mitmproxy. When you are testing a single site through a browser, just accepting the bogus SSL cert manually is not too much trouble, but there are a many circumstances where you will want to configure your testing system or browser to trust the mitmproxy CA as a signing root authority. For security reasons, the mitmproxy CA is generated uniquely on the first start and is not shared between mitmproxy installations on different devices.

3.4.1 Certificate Pinning

Some applications employ Certificate Pinning to prevent man-in-the-middle attacks. This means that mitmproxy and mitmdump’s certificates will not be accepted by these applications without modifying them. It is recommended to use the Ignore Domains feature in order to prevent mitmproxy and mitmdump from intercepting traffic to these specific domains. If you want to intercept the pinned connections, you need to patch the application manually. For Android and (jailbroken) iOS devices, various tools exist to accomplish this.

3.5 CA and cert files

The files created by mitmproxy in the .mitmproxy directory are as follows:

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mitmproxy-ca.pem</td>
<td>The certificate and the private key in PEM format.</td>
</tr>
<tr>
<td>mitmproxy-ca-cert.pem</td>
<td>The certificate in PEM format. Use this to distribute on most non-Windows platforms.</td>
</tr>
<tr>
<td>mitmproxy-ca-cert.p12</td>
<td>The certificate in PKCS12 format. For use on Windows.</td>
</tr>
<tr>
<td>mitmproxy-ca-cert.cer</td>
<td>Same file as .pem, but with an extension expected by some Android devices.</td>
</tr>
</tbody>
</table>

3.6 Using a custom certificate

You can use your own certificate by passing the --cert [domain=]path_to_certificate option to mitmproxy. Mitmproxy then uses the provided certificate for interception of the specified domain instead of generating a certificate signed by its own CA.

The certificate file is expected to be in the PEM format. You can include intermediary certificates right below your leaf certificate, so that your PEM file roughly looks like this:

```
-----BEGIN PRIVATE KEY-----
<private key>
-----END PRIVATE KEY-----
```
For example, you can generate a certificate in this format using these instructions:

```plaintext
>>> openssl genrsa -out cert.key 2048
>>> openssl req -new -x509 -key cert.key -out cert.crt
   (Specify the mitm domain as Common Name, e.g. *.google.com)
>>> cat cert.key cert.crt > cert.pem
```

Now, you can run mitmproxy with the generated certificate:

**For all domain names**

```plaintext
>>> mitmproxy --cert *=cert.pem
```

**For specific domain names**

```plaintext
>>> mitmproxy --cert *.example.com=cert.pem
```

*Note:* `*.example.com` is for all the subdomains. You can also use `www.example.com` for a particular subdomain.

### 3.7 Using a custom certificate authority

By default, mitmproxy will use `~/.mitmproxy/mitmproxy-ca.pem` as the certificate authority to generate certificates for all domains for which no custom certificate is provided (see above). You can use your own certificate authority by passing the `--cadir DIRECTORY` option to mitmproxy. Mitmproxy will then look for `mitmproxy-ca.pem` in the specified directory. If no such file exists, it will be generated automatically.

### 3.8 Using a client side certificate

You can use a client certificate by passing the `--client-certs DIRECTORY|FILE` option to mitmproxy. Using a directory allows certs to be selected based on hostname, while using a filename allows a single specific certificate to be used for all SSL connections. Certificate files must be in the PEM format and should contain both the unencrypted private key and the certificate.

#### 3.8.1 Multiple certs by Hostname

If you’ve specified a directory to `--client-certs`, then the following behavior will be taken:

If you visit example.org, mitmproxy looks for a file named `example.org.pem` in the specified directory and uses this as the client cert.
Mitmproxy is an enormously flexible tool. Knowing exactly how the proxying process works will help you deploy it creatively, and take into account its fundamental assumptions and how to work around them. This document explains mitmproxy’s proxy mechanism in detail, starting with the simplest unencrypted explicit proxying, and working up to the most complicated interaction - transparent proxying of TLS-protected traffic\(^1\) in the presence of Server Name Indication.

### 4.1 Explicit HTTP

Configuring the client to use mitmproxy as an explicit proxy is the simplest and most reliable way to intercept traffic. The proxy protocol is codified in the HTTP RFC, so the behaviour of both the client and the server is well defined, and usually reliable. In the simplest possible interaction with mitmproxy, a client connects directly to the proxy, and makes a request that looks like this:

```
GET http://example.com/index.html HTTP/1.1
```

This is a proxy GET request - an extended form of the vanilla HTTP GET request that includes a schema and host specification, and it includes all the information mitmproxy needs to proceed.

---

\(^1\) The use of “TLS” refers to both SSL (outdated and insecure) and TLS (1.0 and up) in the generic sense, unless otherwise specified.
1. The client connects to the proxy and makes a request.
2. Mitmproxy connects to the upstream server and simply forwards the request on.

### 4.2 Explicit HTTPS

The process for an explicitly proxied HTTPS connection is quite different. The client connects to the proxy and makes a request that looks like this:

```
CONNECT example.com:443 HTTP/1.1
```

A conventional proxy can neither view nor manipulate an TLS-encrypted data stream, so a CONNECT request simply asks the proxy to open a pipe between the client and server. The proxy here is just a facilitator - it blindly forwards data in both directions without knowing anything about the contents. The negotiation of the TLS connection happens over this pipe, and the subsequent flow of requests and responses are completely opaque to the proxy.

#### 4.2.1 The MITM in mitmproxy

This is where mitmproxy’s fundamental trick comes into play. The MITM in its name stands for Man-In-The-Middle - a reference to the process we use to intercept and interfere with these theoretically opaque data streams. The basic idea is to pretend to be the server to the client, and pretend to be the client to the server, while we sit in the middle decoding traffic from both sides. The tricky part is that the Certificate Authority system is designed to prevent exactly this attack, by allowing a trusted third-party to cryptographically sign a server’s certificates to verify that they are legit. If this signature doesn’t match or is from a non-trusted party, a secure client will simply drop the connection and refuse to proceed. Despite the many shortcomings of the CA system as it exists today, this is usually fatal to attempts to MITM an TLS connection for analysis. Our answer to this conundrum is to become a trusted Certificate Authority ourselves. Mitmproxy includes a full CA implementation that generates interception certificates on the fly. To get the client to trust these certificates, we register mitmproxy as a trusted CA with the device manually.
4.2.2 Complication 1: What’s the remote hostname?

To proceed with this plan, we need to know the domain name to use in the interception certificate - the client will verify that the certificate is for the domain it’s connecting to, and abort if this is not the case. At first blush, it seems that the CONNECT request above gives us all we need - in this example, both of these values are “example.com”. But what if the client had initiated the connection as follows:

```
CONNECT 10.1.1.1:443 HTTP/1.1
```

Using the IP address is perfectly legitimate because it gives us enough information to initiate the pipe, even though it doesn’t reveal the remote hostname.

Mitmproxy has a cunning mechanism that smooths this over - *upstream certificate sniffing*. As soon as we see the CONNECT request, we pause the client part of the conversation, and initiate a simultaneous connection to the server. We complete the TLS handshake with the server, and inspect the certificates it used. Now, we use the Common Name in the upstream certificates to generate the dummy certificate for the client. Voi la, we have the correct hostname to present to the client, even if it was never specified.

4.2.3 Complication 2: Subject Alternative Name

Enter the next complication. Sometimes, the certificate Common Name is not, in fact, the hostname that the client is connecting to. This is because of the optional *Subject Alternative Name* field in the certificate that allows an arbitrary number of alternative domains to be specified. If the expected domain matches any of these, the client will proceed, even though the domain doesn’t match the certificate CN. The answer here is simple: when we extract the CN from the upstream cert, we also extract the SANs, and add them to the generated dummy certificate.

4.2.4 Complication 3: Server Name Indication

One of the big limitations of vanilla TLS is that each certificate requires its own IP address. This means that you couldn’t do virtual hosting where multiple domains with independent certificates share the same IP address. In a world with a rapidly shrinking IPv4 address pool this is a problem, and we have a solution in the form of the *Server Name Indication* extension to the TLS protocols. This lets the client specify the remote server name at the start of the TLS handshake, which then lets the server select the right certificate to complete the process.

SNI breaks our upstream certificate sniffing process, because when we connect without using SNI, we get served a default certificate that may have nothing to do with the certificate expected by the client. The solution is another tricky complication to the client connection process. After the client connects, we allow the TLS handshake to continue until just *after* the SNI value has been passed to us. Now we can pause the conversation, and initiate an upstream connection using the correct SNI value, which then serves us the correct upstream certificate, from which we can extract the expected CN and SANs.

4.2.5 Putting it all together

Let’s put all of this together into the complete explicitly proxied HTTPS flow.
1. The client makes a connection to mitmproxy, and issues an HTTP CONNECT request.
2. Mitmproxy responds with a **200 Connection Established**, as if it has set up the CONNECT pipe.
3. The client believes it’s talking to the remote server, and initiates the TLS connection. It uses SNI to indicate the hostname it is connecting to.
4. Mitmproxy connects to the server, and establishes an TLS connection using the SNI hostname indicated by the client.
5. The server responds with the matching certificate, which contains the CN and SAN values needed to generate the interception certificate.
6. Mitmproxy generates the interception cert, and continues the client TLS handshake paused in step 3.
7. The client sends the request over the established TLS connection.
8. Mitmproxy passes the request on to the server over the TLS connection initiated in step 4.

### 4.3 Transparent HTTP

When a transparent proxy is used, the connection is redirected into a proxy at the network layer, without any client configuration being required. This makes transparent proxying ideal for those situations where you can’t change client behaviour - proxy-oblivious Android applications being a common example.

To achieve this, we need to introduce two extra components. The first is a redirection mechanism that transparently reroutes a TCP connection destined for a server on the Internet to a listening proxy server. This usually takes the form of a firewall on the same host as the proxy server - `iptables` on Linux or `pf` on OSX. Once the client has initiated the connection, it makes a vanilla HTTP request, which might look something like this:

```
GET /index.html HTTP/1.1
```

Note that this request differs from the explicit proxy variation, in that it omits the scheme and hostname. How, then, do we know which upstream host to forward the request to? The routing mechanism that has performed the redirection
keeps track of the original destination for us. Each routing mechanism has a different way of exposing this data, so this introduces the second component required for working transparent proxying: a host module that knows how to retrieve the original destination address from the router. In mitmproxy, this takes the form of a built-in set of modules that know how to talk to each platform’s redirection mechanism. Once we have this information, the process is fairly straight-forward.

1. The client makes a connection to the server.
2. The router redirects the connection to mitmproxy, which is typically listening on a local port of the same host. Mitmproxy then consults the routing mechanism to establish what the original destination was.
3. Now, we simply read the client’s request...
4. ... and forward it upstream.

### 4.4 Transparent HTTPS

The first step is to determine whether we should treat an incoming connection as HTTPS. The mechanism for doing this is simple - we use the routing mechanism to find out what the original destination port is. All incoming connections pass through different layers which can determin the actual protocol to use. Automatic TLS detection works for SSLv3, TLS 1.0, TLS 1.1, and TLS 1.2 by looking for a `ClientHello` message at the beginning of each connection. This works independently of the used TCP port.

From here, the process is a merger of the methods we’ve described for transparently proxying HTTP, and explicitly proxying HTTPS. We use the routing mechanism to establish the upstream server address, and then proceed as for explicit HTTPS connections to establish the CN and SANs, and cope with SNI.
1. The client makes a connection to the server.

2. The router redirects the connection to mitmproxy, which is typically listening on a local port of the same host. Mitmproxy then consults the routing mechanism to establish what the original destination was.

3. The client believes it’s talking to the remote server, and initiates the TLS connection. It uses SNI to indicate the hostname it is connecting to.

4. Mitmproxy connects to the server, and establishes an TLS connection using the SNI hostname indicated by the client.

5. The server responds with the matching certificate, which contains the CN and SAN values needed to generate the interception certificate.

6. Mitmproxy generates the interception cert, and continues the client TLS handshake paused in step 3.

7. The client sends the request over the established TLS connection.

8. Mitmproxy passes the request on to the server over the TLS connection initiated in step 4.
Mitmproxy has four modes of operation that allow you to use mitmproxy in a variety of scenarios:

- **Regular** (the default)
- **Transparent**
- **Reverse Proxy**
- **Upstream Proxy**

Now, which one should you pick? Use this flow chart:
5.1 Regular Proxy

Mitmproxy’s regular mode is the simplest and the easiest to set up.

1. Start mitmproxy.
2. Configure your client to use mitmproxy by explicitly setting an HTTP proxy.
3. Quick Check: You should already be able to visit an unencrypted HTTP site through the proxy.
4. Open the magic domain mitm.it and install the certificate for your device.

Note: Unfortunately, some applications bypass the system HTTP proxy settings - Android applications are a common example. In these cases, you need to use mitmproxy’s transparent mode.

If you are proxying an external device, your network will probably look like this:

![Regular Proxy Diagram]

The square brackets signify the source and destination IP addresses. Your client explicitly connects to mitmproxy and mitmproxy explicitly connects to the target server.

5.2 Transparent Proxy

In transparent mode, traffic is directed into a proxy at the network layer, without any client configuration required. This makes transparent proxying ideal for situations where you can’t change client behaviour. In the graphic below, a machine running mitmproxy has been inserted between the router and the internet:

![Transparent Proxy Variant 1: Basic Scenario Diagram]
The square brackets signify the source and destination IP addresses. Round brackets mark the next hop on the Ethernet/data link layer. This distinction is important: when the packet arrives at the mitmproxy machine, it must still be addressed to the target server. This means that Network Address Translation should not be applied before the traffic reaches mitmproxy, since this would remove the target information, leaving mitmproxy unable to determine the real destination.

5.2.1 Common Configurations

There are many ways to configure your network for transparent proxying. We’ll look at two common scenarios:

1. Configuring the client to use a custom gateway/router/"next hop"
2. Implementing custom routing on the router

In most cases, the first option is recommended due to its ease of use.

(a) Custom Gateway

One simple way to get traffic to the mitmproxy machine with the destination IP intact, is to simply configure the client with the mitmproxy box as the default gateway.
In this scenario, we would:

1. Configure the proxy machine for transparent mode. You can find instructions in the *Transparent Proxying* section.
2. Configure the client to use the proxy machine’s IP as the default gateway.
3. Quick Check: At this point, you should already be able to visit an unencrypted HTTP site over the proxy.
4. Open the magic domain *mitm.it* and install the certificate for your device.

Setting the custom gateway on clients can be automated by serving the settings out to clients over DHCP. This lets set up an interception network where all clients are proxied automatically, which can save time and effort.

**Troubleshooting Transparent Mode**

Incorrect transparent mode configurations are a frequent source of error. If it doesn’t work for you, try the following things:

- Open mitmproxy’s event log (press `e`) - do you see clientconnect messages? If not, the packets are not arriving at the proxy. One common cause is the occurrence of ICMP redirects, which means that your machine is telling the client that there’s a faster way to the internet by contacting your router directly (see the *Transparent Proxying* section on how to disable them). If in doubt, Wireshark may help you to see whether something arrives at your machine or not.

- Make sure you have not explicitly configured an HTTP proxy on the client. This is not needed in transparent mode.

- Re-check the instructions in the *Transparent Proxying* section. Anything you missed?

If you encounter any other pitfalls that should be listed here, please let us know!

**Custom Routing**

In some cases, you may need more fine-grained control of which traffic reaches the mitmproxy instance, and which doesn’t. You may, for instance, choose only to divert traffic to some hosts into the transparent proxy. There are a huge number of ways to accomplish this, and much will depend on the router or packet filter you’re using. In most cases, the configuration will look like this:
mitmproxy is usually used with a client that uses the proxy to access the Internet. Using reverse proxy mode, you can use mitmproxy to act like a normal HTTP server:

There are various use-cases:

- Say you have an internal API running at http://example.local/. You could now set up mitmproxy in reverse proxy mode at http://debug.example.local/ and dynamically point clients to this new API endpoint, which provides them with the same data and you with debug information. Similarly, you could move your real server to a different IP/port and set up mitmproxy in the original place to debug and or redirect all sessions.

- Say you’re a web developer working on http://example.com/ (with a development version running on http://localhost:8000/). You can modify your hosts file so that example.com points to 127.0.0.1 and then run mitmproxy in reverse proxy mode on port 80. You can test your app on the example.com domain and get all requests recorded in mitmproxy.
• Say you have some toy project that should get SSL support. Simply set up mitmproxy as a reverse proxy on port 443 and you’re done (`mitmdump -p 443 -R http://localhost:80/`). Mitmproxy auto-detects TLS traffic and intercepts it dynamically. There are better tools for this specific task, but mitmproxy is very quick and simple way to set up an SSL-speaking server.

• Want to add a non-SSL-capable compression proxy in front of your server? You could even spawn a mitmproxy instance that terminates SSL (`-R http://...`), point it to the compression proxy and let the compression proxy point to a SSL-initiating mitmproxy (`-R https://...`), which then points to the real server. As you see, it’s a fairly flexible thing.

Caveat: Interactive Use

Reverse Proxy mode is usually not sufficient to create a copy of an interactive website at different URL. The HTML served to the client remains unchanged - as soon as the user clicks on a non-relative URL (or downloads a non-relative image resource), traffic no longer passes through mitmproxy.

5.4 Upstream Proxy

If you want to chain proxies by adding mitmproxy in front of a different proxy appliance, you can use mitmproxy’s upstream mode. In upstream mode, all requests are unconditionally transferred to an upstream proxy of your choice. mitmproxy supports both explicit HTTP and explicit HTTPS in upstream proxy mode. You could in theory chain multiple mitmproxy instances in a row, but that doesn’t make any sense in practice (i.e. outside of our tests).
mitmproxy is a console tool that allows interactive examination and modification of HTTP traffic. It differs from mitmdump in that all flows are kept in memory, which means that it’s intended for taking and manipulating small-ish samples. Use the ? shortcut key to view, context-sensitive documentation from any mitmproxy screen.

6.1 Flow list

The flow list shows an index of captured flows in chronological order.
• 1: A GET request, returning a 302 Redirect response.
• 2: A GET request, returning 16.75kb of text/html data.
• 3: A replayed request.
• 4: Intercepted flows are indicated with orange text. The user may edit these flows, and then accept them (using the `a` key) to continue. In this case, the request has been intercepted on the way to the server.
• 5: A response intercepted from the server on the way to the client.
• 6: The event log can be toggled on and off using the `e` shortcut key. This pane shows events and errors that may not result in a flow that shows up in the flow pane.
• 7: Flow count.
• 8: Various information on mitmproxy’s state. In this case, we have an interception pattern set to `./*`.
• 9: Bind address indicator - mitmproxy is listening on port 8080 of all interfaces.

### 6.2 Flow view

The **Flow View** lets you inspect and manipulate a single flow:
• 1: Flow summary.

• 2: The Request/Response tabs, showing you which part of the flow you are currently viewing. In the example above, we’re viewing the Response. Hit `tab` to switch between the Response and the Request.

• 3: Headers.

• 4: Body.

• 5: View Mode indicator. In this case, we’re viewing the body in `hex` mode. The other available modes are `pretty`, which uses a number of heuristics to show you a friendly view of various content types, and `raw`, which shows you exactly what’s there without any changes. You can change modes using the `m` key.

### 6.3 Grid Editor

Much of the data that we’d like to interact with in mitmproxy is structured. For instance, headers, queries and form data can all be thought of as a list of key/value pairs. Mitmproxy has a built-in editor that lays this type of data out in a grid for easy manipulation.

At the moment, the Grid Editor is used in four parts of mitmproxy:

- Editing request or response headers (e for edit, then h for headers in flow view)
- Editing a query string (e for edit, then q for query in flow view)
- Editing a URL-encoded form (e for edit, then f for form in flow view)
• Editing replacement patterns (o for options, then R for Replacement Patterns)

If there is no data, an empty editor will be started to let you add some. Here is the editor showing the headers from a request:

![Editing headers](image)

To edit, navigate to the key or value you want to modify using the arrow or vi navigation keys, and press enter. The background color will change to show that you are in edit mode for the specified field:
Modify the field as desired, then press escape to exit edit mode when you’re done. You can also add a row (a key), delete a row (d key), spawn an external editor on a field (e key). Be sure to consult the context-sensitive help (?) key for more.

6.4 Example: Interception

`mitmproxy`’s interception functionality lets you pause an HTTP request or response, inspect and modify it, and then accept it to send it on to the server or client.
6.4.1 1: Set an interception pattern

We press i to set an interception pattern. In this case, the ~q filter pattern tells mitmproxy to intercept all requests. For complete filter syntax, see the Filter expressions section of the documentation, or the built-in help function in mitmproxy.
6.4.2 2: Intercepted connections are indicated with orange text:

![Interception Example](image-url)

6.4. Example: Interception
6.4.3 3: You can now view and modify the request:

In this case, we viewed the request by selecting it, pressed e for “edit” and m for “method” to change the HTTP request method.
6.4.4 4: Accept the intercept to continue:

![Image of terminal with a command to OPTIONS the request]

Finally, we press a to accept the modified request, which is then sent on to the server. In this case, we changed the request from an HTTP GET to OPTIONS, and Google’s server has responded with a 405 “Method not allowed”.
mitmdump is the command-line companion to mitmproxy. It provides tcpdump-like functionality to let you view, record, and programmatically transform HTTP traffic. See the `--help` flag output for complete documentation.

## 7.1 Examples

### 7.1.1 Saving traffic

```bash
>>> mitmdump -w outfile
```

Start up mitmdump in proxy mode, and write all traffic to `outfile`.

### 7.1.2 Filtering saved traffic

```bash
>>> mitmdump -nr infile -w outfile "~m post"
```

Start mitmdump without binding to the proxy port (`-n`), read all flows from `infile`, apply the specified filter expression (only match POSTs), and write to `outfile`.

### 7.1.3 Client replay

```bash
>>> mitmdump -nc outfile
```

Start mitmdump without binding to the proxy port (`-n`), then replay all requests from `outfile (-c filename)`. Flags combine in the obvious way, so you can replay requests from one file, and write the resulting flows to another:

```bash
>>> mitmdump -nc srcfile -w dstfile
```

See the *Client-side replay* section for more information.
7.1.4 Running a script

```
mitmdump -s examples/add_header.py
```

This runs the `add_header.py` example script, which simply adds a new header to all responses.

7.1.5 Scripted data transformation

```
mitmdump -ns examples/add_header.py -r srcfile -w dstfile
```

This command loads flows from `srcfile`, transforms it according to the specified script, then writes it back to `dstfile`. 
mitmweb

mitmweb is mitmproxy’s web-based user interface that allows interactive examination and modification of HTTP traffic. Like mitmproxy, it differs from mitmdump in that all flows are kept in memory, which means that it’s intended for taking and manipulating small-ish samples.

**Warning:** Mitmweb is currently in beta. We consider it stable for all features currently exposed in the UI, but it still misses a lot of mitmproxy’s features.
### Chapter 8. mitmweb

#### mitmproxy - http://localhost:

<table>
<thead>
<tr>
<th>Path</th>
<th>Method</th>
<th>Status</th>
<th>Size</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="https://www.google.com/">https://www.google.com/</a></td>
<td>GET</td>
<td>302</td>
<td>355b</td>
<td>637ms</td>
</tr>
<tr>
<td><a href="http://google.com/">http://google.com/</a></td>
<td>GET</td>
<td>302</td>
<td>258b</td>
<td>33ms</td>
</tr>
<tr>
<td><a href="https://www.google.de">https://www.google.de</a></td>
<td>GET</td>
<td>200</td>
<td>64.3kb</td>
<td>218ms</td>
</tr>
<tr>
<td><a href="https://www.google.de">https://www.google.de</a></td>
<td>GET</td>
<td>200</td>
<td>2.0kb</td>
<td>414ms</td>
</tr>
<tr>
<td><a href="https://www.google.de">https://www.google.de</a></td>
<td>GET</td>
<td>200</td>
<td>136.9kb</td>
<td>242ms</td>
</tr>
<tr>
<td><a href="https://lh3.googleusercontent.com/">https://lh3.googleusercontent.com/</a></td>
<td>GET</td>
<td>200</td>
<td>1.2kb</td>
<td>82ms</td>
</tr>
<tr>
<td><a href="https://clients5.google.com">https://clients5.google.com</a> emailAddress</td>
<td>GET</td>
<td>200</td>
<td>132b</td>
<td>71ms</td>
</tr>
</tbody>
</table>

**HTTP/1.1 204 No Content**

- Content-Type: text/html; charset=utf-8
- Date: Tue, 20 Dec 2016 15:57:48 GMT
- Server: gws
- Content-Length: 0
- X-XSS-Protection: 1; mode=block
- X-Frame-Options: SAMEORIGIN
Mitmproxy is configured with a YAML file, located at `~/.mitmproxy/config.yaml`. We'll have complete documentation for all supported options in the next release in the meantime, please consult the source for a complete list of options and types.
When the `--anticache` option is passed to mitmproxy, it removes headers (`if-none-match` and `if-modified-since`) that might elicit a 304 not modified response from the server. This is useful when you want to make sure you capture an HTTP exchange in its totality. It’s also often used during Client-side replay, when you want to make sure the server responds with complete data.
Many commands in mitmproxy and mitmdump take a filter expression. Filter expressions consist of the following operators:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>~a</td>
<td>Match asset in response: CSS, Javascript, Flash, images.</td>
</tr>
<tr>
<td>~b regex</td>
<td>Body</td>
</tr>
<tr>
<td>~bq regex</td>
<td>Request body</td>
</tr>
<tr>
<td>~bs regex</td>
<td>Response body</td>
</tr>
<tr>
<td>~c int</td>
<td>HTTP response code</td>
</tr>
<tr>
<td>~d regex</td>
<td>Domain</td>
</tr>
<tr>
<td>~dst regex</td>
<td>Match destination address</td>
</tr>
<tr>
<td>~e</td>
<td>Match error</td>
</tr>
<tr>
<td>~h regex</td>
<td>Header</td>
</tr>
<tr>
<td>~hq regex</td>
<td>Request header</td>
</tr>
<tr>
<td>~hs regex</td>
<td>Response header</td>
</tr>
<tr>
<td>~http</td>
<td>Match HTTP flows</td>
</tr>
<tr>
<td>~m regex</td>
<td>Method</td>
</tr>
<tr>
<td>~marked</td>
<td>Match marked flows</td>
</tr>
<tr>
<td>~q</td>
<td>Match request with no response</td>
</tr>
<tr>
<td>~s</td>
<td>Match response</td>
</tr>
<tr>
<td>~src regex</td>
<td>Match source address</td>
</tr>
<tr>
<td>~t regex</td>
<td>Content-type header</td>
</tr>
<tr>
<td>~tcp</td>
<td>Match TCP flows</td>
</tr>
<tr>
<td>~tq regex</td>
<td>Request Content-Type header</td>
</tr>
<tr>
<td>~ts regex</td>
<td>Response Content-Type header</td>
</tr>
<tr>
<td>~u regex</td>
<td>URL</td>
</tr>
<tr>
<td>!</td>
<td>unary not</td>
</tr>
<tr>
<td>&amp;</td>
<td>and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>(...)</td>
<td>grouping</td>
</tr>
</tbody>
</table>

- Regexes are Python-style
• Regexes can be specified as quoted strings
• Header matching (~h, ~hq, ~hs) is against a string of the form “name: value”.
• Strings with no operators are matched against the request URL.
• The default binary operator is &.

11.1 Examples

URL containing “google.com”:

```
google\.com
```

Requests whose body contains the string “test”:

```
~q ~b test
```

Anything but requests with a text/html content type:

```
!(~q & ~t "text/html")
```
Mitmproxy lets you specify an arbitrary number of patterns that define text replacements within flows. Each pattern has 3 components: a filter that defines which flows a replacement applies to, a regular expression that defines what gets replaced, and a target value that defines what is substituted in.

Replace hooks fire when either a client request or a server response is received. Only the matching flow component is affected: so, for example, if a replace hook is triggered on server response, the replacement is only run on the Response object leaving the Request intact. You control whether the hook triggers on the request, response or both using the filter pattern. If you need finer-grained control than this, it’s simple to create a script using the replacement API on Flow components.

Replacement hooks are extremely handy in interactive testing of applications. For instance you can use a replace hook to replace the text “XSS” with a complicated XSS exploit, and then “inject” the exploit simply by interacting with the application through the browser. When used with tools like Firebug and mitmproxy’s own interception abilities, replacement hooks can be an amazingly flexible and powerful feature.

### 12.1 On the command-line

The replacement hook command-line options use a compact syntax to make it easy to specify all three components at once. The general form is as follows:

```
/patt/regex/replacement
```

Here, **patt** is a mitmproxy filter expression, **regex** is a valid Python regular expression, and **replacement** is a string literal. The first character in the expression (/ in this case) defines what the separation character is. Here’s an example of a valid expression that replaces “foo” with “bar” in all requests:

```
::q:foo:bar
```

In practice, it’s pretty common for the replacement literal to be long and complex. For instance, it might be an XSS exploit that weighs in at hundreds or thousands of characters. To cope with this, there’s a variation of the replacement hook specifier that lets you load the replacement text from a file. So, you might start **mitmdump** as follows:
>>> mitmdump --replace-from-file :-q:foo:-/xss-exploit

This will load the replacement text from the file ~/.xss-exploit.

Both the --replace and --replace-from-file flags can be passed multiple times.

## 12.2 Interactively

The R shortcut key in the mitmproxy options menu (o) lets you add and edit replacement hooks using a built-in editor. The context-sensitive help (?) has complete usage information.

<table>
<thead>
<tr>
<th>command-line</th>
<th>--replace, --replace-from-file</th>
</tr>
</thead>
<tbody>
<tr>
<td>mitmproxy shortcut</td>
<td>o then R</td>
</tr>
</tbody>
</table>
Client-side replay does what it says on the tin: you provide a previously saved HTTP conversation, and mitmproxy replays the client requests one by one. Note that mitmproxy serializes the requests, waiting for a response from the server before starting the next request. This might differ from the recorded conversation, where requests may have been made concurrently.

You may want to use client-side replay in conjunction with the Anticache option, to make sure the server responds with complete data.

<table>
<thead>
<tr>
<th>command-line</th>
<th>-c path</th>
</tr>
</thead>
<tbody>
<tr>
<td>mitmproxy shortcut</td>
<td>R then c</td>
</tr>
</tbody>
</table>
14.1 Matching requests with responses

By default, mitmproxy excludes request headers when matching incoming requests with responses from the replay file. This works in most circumstances, and makes it possible to replay server responses in situations where request headers would naturally vary, e.g. using a different user agent. The --rheader headername command-line option allows you to override this behaviour by specifying individual headers that should be included in matching.

14.2 Response refreshing

Simply replaying server responses without modification will often result in unexpected behaviour. For example cookie timeouts that were in the future at the time a conversation was recorded might be in the past at the time it is replayed. By default, mitmproxy refreshes server responses before sending them to the client. The date, expires and last-modified headers are all updated to have the same relative time offset as they had at the time of recording. So, if they were in the past at the time of recording, they will be in the past at the time of replay, and vice versa. Cookie expiry times are updated in a similar way.

You can turn off response refreshing using the --norefresh argument, or using the o options shortcut within mitmproxy.

14.3 Replaying a session recorded in Reverse-proxy Mode

If you have captured the session in reverse proxy mode, in order to replay it you still have to specify the server URL, otherwise you may get the error: ‘HTTP protocol error in client request: Invalid HTTP request form (expected authority or absolute...)’.
During replay, when the client’s requests match previously recorded requests, then the respective recorded responses are simply replayed by mitmproxy. Otherwise, the unmatched requests is forwarded to the upstream server. If forwarding is not desired, you can use the –kill (-k) switch to prevent that.

<table>
<thead>
<tr>
<th>command-line</th>
<th>-S path</th>
</tr>
</thead>
<tbody>
<tr>
<td>mitmproxy shortcut</td>
<td>R then s</td>
</tr>
</tbody>
</table>
This feature lets you specify a set of headers to be added to requests or responses, based on a filter pattern. You can specify these either on the command-line, or through an interactive editor in mitmproxy.

Example: Set the **Host** header to “example.com” for all requests.

```
mitmdump -R http://example.com --setheader :~q:Host:example.com
```

<table>
<thead>
<tr>
<th>command-line</th>
<th>--setheader PATTERN</th>
</tr>
</thead>
<tbody>
<tr>
<td>mitmproxy shortcut</td>
<td>o then H</td>
</tr>
</tbody>
</table>
Ignore Domains

There are two main reasons why you may want to exempt some traffic from mitmproxy’s interception mechanism:

- **Certificate pinning**: Some traffic is protected using Certificate Pinning and mitmproxy’s interception leads to errors. For example, the Twitter app, Windows Update or the Apple App Store fail to work if mitmproxy is active.

- **Convenience**: You really don’t care about some parts of the traffic and just want them to go away. Note that mitmproxy’s “Limit” option is often the better alternative here, as it is not affected by the limitations listed below.

If you want to peek into (SSL-protected) non-HTTP connections, check out the *TCP Proxy* feature. If you want to ignore traffic from mitmproxy’s processing because of large response bodies, take a look at the *Response Streaming* feature.

### 16.1 How it works

mitmproxy allows you to specify a regex which is matched against a `host:port` string (e.g. “example.com:443”) to determine hosts that should be excluded.

### 16.2 Limitations

There are two important quirks to consider:

- **In transparent mode, the ignore pattern is matched against the IP and ClientHello SNI host.** While we usually infer the hostname from the Host header if the `--host` argument is passed to mitmproxy, we do not have access to this information before the SSL handshake. If the client uses SNI however, then we treat the SNI host as an ignore target.
• **In regular mode, explicit HTTP requests are never ignored.**¹ The ignore pattern is applied on CONNECT requests, which initiate HTTPS or clear-text WebSocket connections.

## 16.3 Tutorial

If you just want to ignore one specific domain, there’s usually a bulletproof method to do so:

1. Run mitmproxy or mitmdump in verbose mode (`-v`) and observe the `host:port` information in the server-connect messages. mitmproxy will filter on these.

2. Take the `host:port` string, surround it with `^` and `$`, escape all dots (`. becomes \.`) and use this as your ignore pattern:

```plaintext
>>> mitmdump -v
127.0.0.1:50588: clientconnect
127.0.0.1:50588: request
   → CONNECT example.com:443 HTTP/1.1
127.0.0.1:50588: Set new server address: example.com:443
127.0.0.1:50588: serverconnect
   → example.com:443
^C
```  

```plaintext
>>> mitmproxy --ignore ^example\.com:443$
```

Here are some other examples for ignore patterns:

```plaintext
# Exempt traffic from the iOS App Store (the regex is lax, but usually just works):
--ignore apple.com:443
# "Correct" version without false-positives:
--ignore '^(.+\.)?apple\.com:443$'
# Ignore example.com, but not its subdomains:
--ignore '^example\.com:'
# Ignore everything but example.com and mitmproxy.org:
--ignore '^(!example\.com)(?!mitmproxy\.org)'
# Transparent mode:
--ignore 17\.[178]\.[96]:59:443
# IP address range:
--ignore 17\.[178]\.[d+]:d+:443
```

See also:

- **TCP Proxy**
- **Response Streaming**
- mitmproxy’s “Limit” feature

¹ This stems from an limitation of explicit HTTP proxying: A single connection can be re-used for multiple target domains. A GET http://example.com/ request may be followed by a GET http://evil.com/ request on the same connection. If we start to ignore the connection after the first request, we would miss the relevant second one.
Asks the user for authentication before they are permitted to use the proxy. Authentication headers are stripped from the flows, so they are not passed to upstream servers. For now, only HTTP Basic authentication is supported. The proxy auth options are not compatible with the transparent, socks or reverse proxy mode.

| command-line | --nonanonymouse, --singleuser USER, --htpasswd PATH |
In reverse proxy mode, mitmproxy accepts standard HTTP(S) requests and forwards them to the specified upstream server. This is in contrast to Upstream proxy mode, in which mitmproxy forwards HTTP(S) proxy requests to an upstream proxy server.

```
command-line -R http[s]://hostname[:port]
```

Here, http[s] signifies if the proxy should use TLS to connect to the server. mitmproxy always accepts both encrypted and unencrypted requests and transforms them to what the server expects.

```
>>> mitmdump -R https://httpbin.org -p 80
>>> curl http://localhost/
# requests will be transparently upgraded to TLS by mitmproxy

>>> mitmdump -R https://httpbin.org -p 443
>>> curl https://localhost/
# mitmproxy will use TLS on both ends.
```

### 18.1 Host Header

In reverse proxy mode, mitmproxy automatically rewrites the Host header to match the upstream server. This allows mitmproxy to easily connect to existing endpoints on the open web (e.g. `mitmproxy -R https://example.com`).

However, keep in mind that absolute URLs within the returned document or HTTP redirects will NOT be rewritten by mitmproxy. This means that if you click on a link for “http://example.com” in the returned web page, you will be taken directly to that URL, bypassing mitmproxy.

One possible way to address this is to modify the hosts file of your OS so that “example.com” resolves to your proxy’s IP, and then access the proxy by going directly to example.com. Make sure that your proxy can still resolve the original IP, or specify an IP in mitmproxy.
By using mitmproxy’s streaming feature, response contents can be passed to the client incrementally before they have been fully received by the proxy. This is especially useful for large binary files such as videos, where buffering the whole file slows down the client’s browser.

By default, mitmproxy will read the entire response, perform any indicated manipulations on it and then send the (possibly modified) response to the client. In some cases this is undesirable and you may wish to “stream” the response back to the client. When streaming is enabled, the response is not buffered on the proxy but directly sent back to the client instead.

19.1 On the command-line

Streaming can be enabled on the command line for all response bodies exceeding a certain size. The SIZE argument understands k/m/g suffixes, e.g. 3m for 3 megabytes.

```
command-line --stream SIZE
```

**Warning:** When response streaming is enabled, streamed response contents will not be recorded or preserved in any way.

**Note:** When response streaming is enabled, the response body cannot be modified by the usual means.

19.2 Customizing Response Streaming

You can also use a script to customize exactly which responses are streamed.

Responses that should be tagged for streaming by setting their .stream attribute to True:
Listing 19.1: examples/complex/stream.py

```python
def responseheaders(flow):
    ""
    Enables streaming for all responses.
    This is equivalent to passing `--stream 0` to mitmproxy.
    ""
    flow.response.stream = True
```

19.3 Implementation Details

When response streaming is enabled, portions of the code which would have otherwise performed changes on the response body will see an empty response body. Any modifications will be ignored.

Streamed responses are usually sent in chunks of 4096 bytes. If the response is sent with a `Transfer-Encoding: chunked` header, the response will be streamed one chunk at a time.

19.4 Modifying streamed data

If the `.stream` attribute is callable, `.stream` will wrap the generator that yields all chunks.

Listing 19.2: examples/complex/stream_modify.py

```python
""
This inline script modifies a streamed response.
If you do not need streaming, see the modify_response_body example.
Be aware that content replacement isn't trivial:
   - If the transfer encoding isn't chunked, you cannot simply change the content_length.
   - If you want to replace all occurrences of "foobar", make sure to catch the cases where one chunk ends with [...]foo" and the next starts with "bar [...]."
""

def modify(chunks):
    ""
    chunks is a generator that can be used to iterate over all chunks.
    ""
    for chunk in chunks:
        yield chunk.replace("foo", "bar")

def responseheaders(flow):
    flow.response.stream = modify
```

See also:

- Ignore Domains
In this mode, mitmproxy acts as a SOCKS5 proxy server.

command-line
```
--socks
```
21.1 Sticky cookies

When the sticky cookie option is set, `mitmproxy` will add the cookie most recently set by the server to any cookie-less request. Consider a service that sets a cookie to track the session after authentication. Using sticky cookies, you can fire up `mitmproxy`, and authenticate to a service as you usually would using a browser. After authentication, you can request authenticated resources through `mitmproxy` as if they were unauthenticated, because `mitmproxy` will automatically add the session tracking cookie to requests. Among other things, this lets you script interactions with authenticated resources (using tools like `wget` or `curl`) without having to worry about authentication.

Sticky cookies are especially powerful when used in conjunction with Client-side replay - you can record the authentication process once, and simply replay it on startup every time you need to interact with the secured resources.

| command-line | -t FILTER |
| mitmproxy shortcut | o then t |

21.2 Sticky auth

The sticky auth option is analogous to the sticky cookie option, in that HTTP Authorization headers are simply replayed to the server once they have been seen. This is enough to allow you to access a server resource using HTTP Basic authentication through the proxy. Note that `mitmproxy` doesn’t (yet) support replay of HTTP Digest authentication.

| command-line | -u FILTER |
| mitmproxy shortcut | o then A |
In case mitmproxy does not handle a specific protocol, you can exempt hostnames from processing, so that mitmproxy acts as a generic TCP forwarder. This feature is closely related to the *Ignore Domains* functionality, but differs in two important aspects:

- The raw TCP messages are printed to the event log.
- SSL connections will be intercepted.

Please note that message interception or modification are not possible yet. If you are not interested in the raw TCP messages, you should use the ignore domains feature.

### 22.1 How it works

For a detailed description how the hostname pattern works, please look at the *Ignore Domains* feature.

See also:

- *Ignore Domains*
- *Response Streaming*
In this mode, mitmproxy accepts proxy requests and unconditionally forwards all requests to a specified upstream proxy server. This is in contrast to *Reverse Proxy*, in which mitmproxy forwards ordinary HTTP requests to an upstream server.

```
command-line -U http://hostname[:port]
```
When mitmproxy receives a connection destined for an SSL-protected service, it freezes the connection before reading its request data, and makes a connection to the upstream server to “sniff” the contents of its SSL certificate. The information gained - the Common Name and Subject Alternative Names - is then used to generate the interception certificate, which is sent to the client so the connection can continue.

This rather intricate little dance lets us seamlessly generate correct certificates even if the client has specified only an IP address rather than the hostname. It also means that we don’t need to sniff additional data to generate certs in transparent mode.

Upstream cert sniffing is on by default, and can optionally be turned off.

<table>
<thead>
<tr>
<th>command-line</th>
<th>--no-upstream-cert</th>
</tr>
</thead>
<tbody>
<tr>
<td>mitmproxy shortcut</td>
<td>o then U</td>
</tr>
</tbody>
</table>
CHAPTER 25

Transparent Proxying

When a transparent proxy is used, traffic is redirected into a proxy at the network layer, without any client configuration being required. This makes transparent proxying ideal for those situations where you can’t change client behaviour - proxy-oblivious Android applications being a common example.

To set up transparent proxying, we need two new components. The first is a redirection mechanism that transparently reroutes a TCP connection destined for a server on the Internet to a listening proxy server. This usually takes the form of a firewall on the same host as the proxy server - `iptables` on Linux or `pf` on OSX. When the proxy receives a redirected connection, it sees a vanilla HTTP request, without a host specification. This is where the second new component comes in - a host module that allows us to query the redirector for the original destination of the TCP connection.

At the moment, mitmproxy supports transparent proxying on OSX Lion and above, and all current flavors of Linux.

25.1 Fully transparent mode

By default mitmproxy will use its own local ip address for its server-side connections. In case this isn’t desired, the `--spoof-source-address` argument can be used to use the client’s ip address for server-side connections. The following config is required for this mode to work:

```bash
CLIENT_NET=192.168.1.0/24
TABLE_ID=100
MARK=1
echo "$TABLE_ID mitmproxy" >> /etc/iproute2/rt_tables
iptables -t mangle -A PREROUTING -d $CLIENT_NET -j MARK --set-mark $MARK
iptables -t nat -A PREROUTING -p tcp -s $CLIENT_NET --match multiport --dports 80,443
   -j REDIRECT --to-port 8080
ip rule add fwmark $MARK lookup $TABLE_ID
ip route add local $CLIENT_NET dev lo table $TABLE_ID
```

This mode does require root privileges though. There’s a wrapper in the examples directory called ‘mitmproxy_shim.c’, which will enable you to use this mode with dropped privileges. It can be used as follows:
gcc examples/complex/full_transparency_shim.c -o mitmproxy_shim -lcap
sudo chown root:root mitmproxy_shim
sudo chmod u+s mitmproxy_shim
./mitmproxy_shim $(which mitmproxy) -T --spoof-source-address
CHAPTER 26

Linux

On Linux, mitmproxy integrates with the iptables redirection mechanism to achieve transparent mode.

1. Install the mitmproxy certificate on the test device
2. Enable IP forwarding:
   
   ```bash
   >>> sysctl -w net.ipv4.ip_forward=1
   ```
   
   You may also want to consider enabling this permanently in `/etc/sysctl.conf`.
3. If your target machine is on the same physical network and you configured it to use a custom gateway, disable ICMP redirects:
   
   ```bash
   >>> echo 0 | sudo tee /proc/sys/net/ipv4/conf/*/send_redirects
   ```
   
   You may also want to consider enabling this permanently in `/etc/sysctl.conf` as demonstrated [here](#).
4. Create an iptables ruleset that redirects the desired traffic to the mitmproxy port. Details will differ according to your setup, but the ruleset should look something like this:
   
   ```bash
   iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 80 -j REDIRECT --to-port 8080
   iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 443 -j REDIRECT --to-port 8080
   ```
5. Fire up mitmproxy. You probably want a command like this:
   
   ```bash
   >>> mitmproxy -T --host
   ```
   
   The `--T` flag turns on transparent mode, and the `--host` argument tells mitmproxy to use the value of the Host header for URL display.
6. Finally, configure your test device to use the host on which mitmproxy is running as the default gateway.

For a detailed walkthrough, have a look at the *Transparently proxify virtual machines* tutorial.
OSX Lion integrated the pf packet filter from the OpenBSD project, which mitmproxy uses to implement transparent mode on OSX. Note that this means we don’t support transparent mode for earlier versions of OSX.

1. Install the mitmproxy certificate on the test device

2. Enable IP forwarding:

   ```
   >>> sudo sysctl -w net.inet.ip.forwarding=1
   ```

3. Place the following two lines in a file called, say, `pf.conf`:

   ```
   rdr on en2 inet proto tcp to any port 80 -> 127.0.0.1 port 8080
   rdr on en2 inet proto tcp to any port 443 -> 127.0.0.1 port 8080
   ```

   These rules tell pf to redirect all traffic destined for port 80 or 443 to the local mitmproxy instance running on port 8080. You should replace `en2` with the interface on which your test device will appear.

4. Configure pf with the rules:

   ```
   >>> sudo pfctl -f pf.conf
   ```

5. And now enable it:

   ```
   >>> sudo pfctl -e
   ```

6. Configure sudoers to allow mitmproxy to access pfctl. Edit the file `/etc/sudoers` on your system as root. Add the following line to the end of the file:

   ```
   ALL ALL=NOPASSWD: /sbin/pfctl -s state
   ```

   Note that this allows any user on the system to run the command `/sbin/pfctl -s state` as root without a password. This only allows inspection of the state table, so should not be an undue security risk. If you’re special feel free to tighten the restriction up to the user running mitmproxy.

7. Fire up mitmproxy. You probably want a command like this:
The `-T` flag turns on transparent mode, and the `--host` argument tells mitmproxy to use the value of the Host header for URL display.

8. Finally, configure your test device to use the host on which mitmproxy is running as the default gateway.

Note: Note that the `rdr` rules in the `pf.conf` given above only apply to inbound traffic. This means that they will NOT redirect traffic coming from the box running pf itself. We can’t distinguish between an outbound connection from a non-mitmproxy app, and an outbound connection from mitmproxy itself - if you want to intercept your OSX traffic, you should use an external host to run mitmproxy. Nonetheless, pf is flexible to cater for a range of creative possibilities, like intercepting traffic emanating from VMs. See the `pf.conf` man page for more.
1. Install the mitmproxy certificate on the test device

2. Enable IP forwarding:
   ```
   >>> sudo sysctl -w net.inet.ip.forwarding=1
   ```

3. Place the following two lines in `/etc/pf.conf`:
   ```
   mitm_if = "re2"
   pass in quick proto tcp from $mitm_if to port { 80, 443 } divert-to 127.0.0.1
    → port 8080
   ```
   These rules tell pf to divert all traffic from $mitm_if destined for port 80 or 443 to the local mitmproxy instance running on port 8080. You should replace $mitm_if value with the interface on which your test device will appear.

4. Configure pf with the rules:
   ```
   >>> doas pfctl -f /etc/pf.conf
   ```

5. And now enable it:
   ```
   >>> doas pfctl -e
   ```

6. Fire up mitmproxy. You probably want a command like this:
   ```
   >>> mitmproxy -T --host
   ```
   The `-T` flag turns on transparent mode, and the `--host` argument tells mitmproxy to use the value of the Host header for URL display.

7. Finally, configure your test device to use the host on which mitmproxy is running as the default gateway.

**Note:** Note that the `divert-to` rules in the pf.conf given above only apply to inbound traffic. **This means that they will NOT redirect traffic coming from the box running pf itself.** We can’t distinguish between an outbound
connection from a non-mitmproxy app, and an outbound connection from mitmproxy itself - if you want to intercept your traffic, you should use an external host to run mitmproxy. Nonetheless, pf is flexible to cater for a range of creative possibilities, like intercepting traffic emanating from VMs. See the **pf.conf** man page for more.
Mitmproxy has a powerful scripting API that allows you to control almost any aspect of traffic being proxied. In fact, much of mitmproxy’s own core functionality is implemented using the exact same API exposed to scripters (see mitmproxy/addons).

### 29.1 A simple example

Scripting is event driven, with named handlers on the script object called at appropriate points of mitmproxy’s operation. Here’s a complete mitmproxy script that adds a new header to every HTTP response before it is returned to the client:

```python
def response(flow):
    flow.response.headers["newheader"] = "foo"
```

All events that deal with an HTTP request get an instance of `HTTPFlow`, which we can use to manipulate the response itself. We can now run this script using mitmdump, and the new header will be added to all responses passing through the proxy:

```bash
>>> mitmdump -s add_header.py
```

### 29.2 Using classes

In the example above, the script object is the `add_header` module itself. That is, the handlers are declared at the global level of the script. This is great for quick hacks, but soon becomes limiting as scripts become more sophisticated. When a script first starts up, the `start` event is called before anything else happens. You can replace the current script object by returning it from this handler. Here’s how this looks when applied to the example above:
Listing 29.2: examples/simple/add_header_class.py

class AddHeader:
    def response(self, flow):
        flow.response.headers["newheader"] = "foo"

    def start():
        return AddHeader()

So here, we’re using a module-level script to “boot up” into a class instance. From this point on, the module-level script is removed from the handler chain, and is replaced by the class instance.

29.3 Handling arguments

Scripts can handle their own command-line arguments, just like any other Python program. Let’s build on the example above to do something slightly more sophisticated - replace one value with another in all responses. Mitmproxy’s HTTPRequest and HTTPResponse objects have a handy replace method that takes care of all the details for us.

Listing 29.3: examples/simple/script_arguments.py

```python
import argparse

class Replacer:
    def __init__(self, src, dst):
        self.src, self.dst = src, dst

    def response(self, flow):
        flow.response.replace(self.src, self.dst)

    def start():
        parser = argparse.ArgumentParser()
        parser.add_argument("src", type=str)
        parser.add_argument("dst", type=str)
        args = parser.parse_args()
        return Replacer(args.src, args.dst)
```

We can now call this script on the command-line like this:

```bash
>>> mitmdump -dd -s "./script_arguments.py html faketml"
```

Whenever a handler is called, mitmproxy rewrites the script environment so that it sees its own arguments as if it was invoked from the command-line.

29.4 Logging and the context

Scripts should not output straight to stderr or stdout. Instead, the log object on the ctx context module should be used, so that the mitmproxy host program can handle output appropriately. So, mitmdump can print colorised script output to the terminal, and mitmproxy console can place script output in the event buffer.

Here’s how this looks:
It is recommended to use `ctx.log` for logging within a script. This goes to the event log in mitmproxy and to stdout in mitmdump.

If you want to help us out: https://github.com/mitmproxy/mitmproxy/issues/1530 :-)

```python
from mitmproxy import ctx

def start():
    ctx.log.info("This is some informative text.")
    ctx.log.error("This is an error.")
```

The `ctx` module also exposes the mitmproxy master object at `ctx.master` for advanced usage.

### 29.5 Running scripts on saved flows

When a flow is loaded from disk, the sequence of events that the flow would have gone through on the wire is partially replayed. So, for instance, an HTTP flow loaded from disk will trigger `requestheaders`, `request`, `responseheaders` and `response` in order. We can use this behaviour to transform saved traffic using scripts. For example, we can invoke the replacer script from above on saved traffic as follows:

```
>>> mitmdump -dd -s "./arguments.py html fakehtml" -r saved -w changed
```

This command starts the `arguments` script, reads all the flows from `saved` transforming them in the process, then writes them all to `changed`.

The mitmproxy console tool provides interactive ways to run transforming scripts on flows - for instance, you can run a one-shot script on a single flow through the `|` (pipe) shortcut.

### 29.6 Concurrency

The mitmproxy script mechanism is single threaded, and the proxy blocks while script handlers execute. This hugely simplifies the most common case, where handlers are light-weight and the blocking doesn’t have a performance impact. It’s possible to implement a concurrent mechanism on top of the blocking framework, and mitmproxy includes a handy example of this that is fit for most purposes. You can use it as follows:

```
import time

from mitmproxy.script import concurrent

@concurrent  # Remove this and see what happens
def request(flow):
    # You don’t want to use mitmproxy.ctx from a different thread
    print("handle request: \$s\$s" % (flow.request.host, flow.request.path))
    time.sleep(5)
    print("start request: \$s\$s" % (flow.request.host, flow.request.path))
```
29.7 Testing

Mitmproxy includes a number of helpers for testing addons. The `mitmproxy.test.taddons` module contains a context helper that takes care of setting up and tearing down the addon event context. The `mitmproxy.test.tflow` module contains helpers for quickly creating test flows. Pydoc is the canonical reference for these modules, and mitmproxy’s own test suite is an excellent source of examples of usage. Here, for instance, is the mitmproxy unit tests for the `anticache` option, demonstrating a good cross-section of the test helpers:

Listing 29.6: `test/mitmproxy/addons/test_anticache.py`

```python
from mitmproxy.test import tflow
from mitmproxy.addons import anticache
from mitmproxy.test import taddons

class TestAntiCache:
    def test_simple(self):
        sa = anticache.AntiCache()
        with taddons.context() as tctx:
            f = tflow.tflow(resp=True)
            f.request.headers["if-modified-since"] = "test"
            f.request.headers["if-none-match"] = "test"

            sa.request(f)
            assert "if-modified-since" in f.request.headers
            assert "if-none-match" in f.request.headers

            tctx.configure(sa, anticache = True)
            sa.request(f)
            assert "if-modified-since" not in f.request.headers
            assert "if-none-match" not in f.request.headers
```

29.8 Developing scripts

Mitmproxy monitors scripts for modifications, and reloads them on change. When this happens, the script is shut down (the `done` event is called), and the new instance is started up as if the script had just been loaded (the `start` and `configure` events are called).
### 30.1 General

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
</table>
| **configure** *(options, updated)* | Called once on startup, and whenever options change.  
  *options* An options.Options object with the total current configuration state of mitmproxy.  
  *updated* A set of strings indicating which configuration options have been updated. This contains all options when configure is called on startup, and only changed options subsequently. |
| **done()** | Called once when the script shuts down, either because it’s been unloaded, or because the proxy itself is shutting down. |
| **log**(entry) | Called whenever an event log is added.  
  *entry* An controller.LogEntry object -  
  entry.msg is the log text, and entry.level is the urgency level ("debug", "info", "warn", "error"). |
| **start()** | Called once on startup, before any other events. If you return a value from this event, it will replace the current addon. This allows you to, “boot into” an addon implemented as a class instance from the module level. |
| **tick()** | Called at a regular sub-second interval as long as the addon is executing. |
## 30.2 Connection

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>clientconnect(root_layer)</code></td>
<td>Called when a client initiates a connection to the proxy. Note that a connection can correspond to multiple HTTP requests. root_layer The root layer (see mitmproxy.proxy.protocol for an explanation what the root layer is), provides transparent access to all attributes of the RootContext. For example, root_layer.client_conn.address gives the remote address of the connecting client.</td>
</tr>
<tr>
<td><code>clientdisconnect(root_layer)</code></td>
<td>Called when a client disconnects from the proxy. root_layer The root layer object.</td>
</tr>
<tr>
<td><code>next_layer(layer)</code></td>
<td>Called whenever layers are switched. You may change which layer will be used by returning a new layer object from this event. layer The next layer, as determined by mitmproxy.</td>
</tr>
<tr>
<td><code>serverconnect(server_conn)</code></td>
<td>Called before the proxy initiates a connection to the target server. Note that a connection can correspond to multiple HTTP requests. server_conn A ServerConnection object. It is guaranteed to have a non-None address attribute.</td>
</tr>
<tr>
<td><code>serverdisconnect(server_conn)</code></td>
<td>Called when the proxy has closed the server connection. server_conn A ServerConnection object.</td>
</tr>
</tbody>
</table>
30.3 HTTP Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
</table>
| **http_connect** *(flow)* | Called when we receive an HTTP CONNECT request. Setting a non 2xx response on the flow will return the response to the client abort the connection. CONNECT requests and responses do not generate the usual HTTP handler events. CONNECT requests are only valid in regular and upstream proxy modes.  
  *flow* A `models.HTTPFlow` object. The flow is guaranteed to have non-None request and requestheaders attributes. |
| **request** *(flow)*     | Called when a client request has been received.  
  *flow* A `models.HTTPFlow` object. At this point, the flow is guaranteed to have a non-None request attribute. |
| **requestheaders** *(flow)* | Called when the headers of a client request have been received, but before the request body is read.  
  *flow* A `models.HTTPFlow` object. At this point, the flow is guaranteed to have a non-None request attribute. |
| **responseheaders** *(flow)* | Called when the headers of a server response have been received, but before the response body is read.  
  *flow* A `models.HTTPFlow` object. At this point, the flow is guaranteed to have a non-none request and response attributes, however the response will have no content. |
| **response** *(flow)*     | Called when a server response has been received.  
  *flow* A `models.HTTPFlow` object. At this point, the flow is guaranteed to have a non-none request and response attributes. The raw response body will be in `response.body`, unless response streaming has been enabled. |
| **error** *(flow)*       | Called when a flow error has occurred, e.g. invalid server responses, or interrupted connections. This is distinct from a valid server HTTP error response, which is simply a response with an HTTP error code.  
  *flow* The flow containing the error. It is guaranteed to have non-None error attribute. |

30.4 WebSocket Events

These events are called only after a connection made an HTTP upgrade with “101 Switching Protocols”. No further HTTP-related events after the handshake are issued, only new WebSocket messages are called.
### 30.5 TCP Events

These events are called only if the connection is in *TCP mode*. So, for instance, TCP events are not called for ordinary HTTP/S connections.

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>websocket_handshake</strong></td>
<td>Called when a client wants to establish a WebSocket connection. The WebSocket-specific headers can be manipulated to alter the handshake. The <code>flow</code> object is guaranteed to have a non-None <code>request</code> attribute.</td>
</tr>
<tr>
<td><strong>websocket_start</strong></td>
<td>Called when WebSocket connection is established after a successful handshake. <code>flow</code> A <code>models.WebSocketFlow</code> object.</td>
</tr>
<tr>
<td><strong>websocket_message</strong></td>
<td>Called when a WebSocket message is received from the client or server. The sender and receiver are identifiable. The most recent message will be <code>flow.messages[-1]</code>. The message is user-modifiable. Currently there are two types of messages, corresponding to the BINARY and TEXT frame types. <code>flow</code> A <code>models.WebSocketFlow</code> object.</td>
</tr>
<tr>
<td><strong>websocket_end</strong></td>
<td>Called when WebSocket connection ends. <code>flow</code> A <code>models.WebSocketFlow</code> object.</td>
</tr>
<tr>
<td><strong>websocket_error</strong></td>
<td>Called when a WebSocket error occurs - e.g. the connection closing unexpectedly. <code>flow</code> A <code>models.WebSocketFlow</code> object.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tcp_start</strong></td>
<td>Called when TCP streaming starts. <code>flow</code> A <code>models.TCPFlow</code> object.</td>
</tr>
<tr>
<td><strong>tcp_message</strong></td>
<td>Called when a TCP payload is received from the client or server. The sender and receiver are identifiable. The most recent message will be <code>flow.messages[-1]</code>. The message is user-modifiable. <code>flow</code> A <code>models.TCPFlow</code> object.</td>
</tr>
<tr>
<td><strong>tcp_end</strong></td>
<td>Called when TCP streaming ends. <code>flow</code> A <code>models.TCPFlow</code> object.</td>
</tr>
<tr>
<td><strong>tcp_error</strong></td>
<td>Called when a TCP error occurs - e.g. the connection closing unexpectedly. <code>flow</code> A <code>models.TCPFlow</code> object.</td>
</tr>
</tbody>
</table>
• Errors
  - mitmproxy.flow.Error

• HTTP
  - mitmproxy.http.HTTPRequest
  - mitmproxy.http.HTTPResponse
  - mitmproxy.http.HTTPFlow

• Logging
  - mitmproxy.log.Log
  - mitmproxy.log.LogEntry

31.1 Errors

class mitmproxy.flow.Error (msg: str, timestamp=None) → None
An Error.

This is distinct from a protocol error response (say, a HTTP code 500), which is represented by a normal HTTPResponse object. This class is responsible for indicating errors that fall outside of normal protocol communications, like interrupted connections, timeouts, protocol errors.

Exposes the following attributes:
  msg: Message describing the error
  timestamp: Seconds since the epoch

get_state ()
  Retrieve object state.

set_state (state)
  Load object state from data returned by a get_state call.
31.2 HTTP

class mitmproxy.http.HTTPRequest(first_line_format, method, scheme, host, port, path, http_version, headers, content, timestamp_start=None, timestamp_end=None, is_replay=False)

A mitmproxy HTTP request.

classmethod wrap(request)
    Wraps an existing mitmproxy.net.http.Request.

anticache()
    Modifies this request to remove headers that might produce a cached response. That is, we remove ETags and If-Modified-Since headers.

anticomp()
    Modifies this request to remove headers that will compress the resource’s data.

constrain_encoding()
    Limits the permissible Accept-Encoding values, based on what we can decode appropriately.

content
    The HTTP message body decoded with the content-encoding header (e.g. gzip)
    Raises ValueError, when the content-encoding is invalid and strict is True.
    See also: raw_content, text

cookies
    The request cookies.
    An empty MultiDictView object if the cookie monster ate them all.

decode(strict=True)
    Decodes body based on the current Content-Encoding header, then removes the header. If there is no Content-Encoding header, no action is taken.
    Raises ValueError, when the content-encoding is invalid and strict is True.

encode(e)
    Encodes body with the encoding e, where e is “gzip”, “deflate”, “identity”, or “br”. Any existing content-encodings are overwritten, the content is not decoded beforehand.
    Raises ValueError, when the specified content-encoding is invalid.

first_line_format
    HTTP request form as defined in RFC7230.
    origin-form and asterisk-form are subsumed as “relative”.

get_content(strict: bool = True) → bytes
    The HTTP message body decoded with the content-encoding header (e.g. gzip)
    Raises ValueError, when the content-encoding is invalid and strict is True.
    See also: raw_content, text

get_text(strict: bool = True) → typing.Union[str, NoneType]
    The HTTP message body decoded with both content-encoding header (e.g. gzip) and content-type header charset.
    Raises ValueError, when either content-encoding or charset is invalid and strict is True.
    See also: content, raw_content
headers
   Message headers object

   **Returns** mitmproxy.net.http.Headers

host
   Target host. This may be parsed from the raw request (e.g. from a GET http://example.com/HTTP/1.1 request line) or inferred from the proxy mode (e.g. an IP in transparent mode).
   Setting the host attribute also updates the host header, if present.

host_header
   The request’s host/authority header.
   This property maps to either request.headers["Host"] or request.headers[":authority"], depending on whether it's HTTP/1.x or HTTP/2.0.

http_version
   Version string, e.g. “HTTP/1.1”

method
   HTTP request method, e.g. “GET”.

multipart_form
   The multipart form data as an MultiDictView object. An empty multidict.MultiDictView if the content-type indicates non-form data or the content could not be parsed.
   Key and value are bytes.

path
   HTTP request path, e.g. “/index.html”. Guaranteed to start with a slash, except for OPTIONS requests, which may just be “*”.

path_components
   The URL’s path components as a tuple of strings. Components are unquoted.

port
   Target port

pretty_host
   Similar to host, but using the Host headers as an additional preferred data source. This is useful in transparent mode where host is only an IP address, but may not reflect the actual destination as the Host header could be spoofed.

pretty_url
   Like url, but using pretty_host instead of host.

query
   The request query string as an MultiDictView object.

raw_content
   The raw (encoded) HTTP message body
   See also: content, text

replace(pattern, repl, flags=0, count=0)
   Replaces a regular expression pattern with repl in the headers, the request path and the body of the request. Encoded content will be decoded before replacement, and re-encoded afterwards.
   **Returns** The number of replacements made.

scheme
   HTTP request scheme, which should be “http” or “https”.

31.2. HTTP
text
The HTTP message body decoded with both content-encoding header (e.g. gzip) and content-type header charset.

Raises ValueError, when either content-encoding or charset is invalid and strict is True.

See also: content, raw_content

timestamp_end
Last byte timestamp

timestamp_start
First byte timestamp

url
The URL string, constructed from the request’s URL components

urlencoded_form
The URL-encoded form data as a MultiDictView object. An empty multidict.MultiDictView if the content-type indicates non-form data or the content could not be parsed.

Starting with mitmproxy 1.0, key and value are strings.

class mitmproxy.http.HTTPResponse(http_version, status_code, reason, headers, content, timestamp_start=None, timestamp_end=None, is_replay=False)
A mitmproxy HTTP response.

classmethod wrap(response)

content
The HTTP message body decoded with the content-encoding header (e.g. gzip)

Raises ValueError, when the content-encoding is invalid and strict is True.

See also: raw_content, text

cookies
The response cookies. A possibly empty MultiDictView, where the keys are cookie name strings, and values are (value, attr) tuples. Value is a string, and attr is an MultiDictView containing cookie attributes. Within attrs, unary attributes (e.g. HTTPOnly) are indicated by a Null value.

Caveats: Updating the attr decode(strict=True)
Decodes body based on the current Content-Encoding header, then removes the header. If there is no Content-Encoding header, no action is taken.

Raises ValueError, when the content-encoding is invalid and strict is True.

encode(e)
Encodes body with the encoding e, where e is “gzip”, “deflate”, “identity”, or “br”. Any existing content-encodings are overwritten, the content is not decoded beforehand.

Raises ValueError, when the specified content-encoding is invalid.

get_content(strict: bool = True) → bytes
The HTTP message body decoded with the content-encoding header (e.g. gzip)

Raises ValueError, when the content-encoding is invalid and strict is True.

See also: raw_content, text
get_text (strict: bool = True) → typing.Union[str, NoneType]
The HTTP message body decoded with both content-encoding header (e.g. gzip) and content-type header charset.

Raises ValueError, when either content-encoding or charset is invalid and strict is True.

See also: content, raw_content

headers
Message headers object

Returns mitmproxy.net.http.Headers

http_version
Version string, e.g. “HTTP/1.1”

make (status_code: int = 200, content: AnyStr = b'', headers: typing.Union[typing.Dict[AnyStr, AnyStr],
    typing.Iterable[typing.Tuple[bytes, bytes]]] = ())
Simplified API for creating response objects.

raw_content
The raw (encoded) HTTP message body

See also: content, text

reason
HTTP Reason Phrase, e.g. “Not Found”. This is always None for HTTP2 requests, because HTTP2 responses do not contain a reason phrase.

refresh (now=None)
This fairly complex and heuristic function refreshes a server response for replay.

• It adjusts date, expires and last-modified headers.

• It adjusts cookie expiration.

replace (pattern, repl, flags=0, count=0)
Replaces a regular expression pattern with repl in both the headers and the body of the message. Encoded body will be decoded before replacement, and re-encoded afterwards.

Returns The number of replacements made.

status_code
HTTP Status Code, e.g. 200.

text
The HTTP message body decoded with both content-encoding header (e.g. gzip) and content-type header charset.

Raises ValueError, when either content-encoding or charset is invalid and strict is True.

See also: content, raw_content

timestamp_end
Last byte timestamp

timestamp_start
First byte timestamp

class mitmproxy.http.HTTPFlow (client_conn, server_conn, live=None, mode='regular')
An HTTPFlow is a collection of objects representing a single HTTP transaction.

request = None
HTTPRequest object
response = None
    HTTPResponse object

error = None
    Error object

Note that it’s possible for a Flow to have both a response and an error object. This might happen, for instance, when a response was received from the server, but there was an error sending it back to the client.

server_conn = None
    ServerConnection object

client_conn = None
    ClientConnection object

intercepted = None
    Is this flow currently being intercepted?

mode = None
    What mode was the proxy layer in when receiving this request?

backup (force=False)
    Save a backup of this Flow, which can be reverted to using a call to .revert().

intercept ()
    Intercept this Flow. Processing will stop until resume is called.

kill ()
    Kill this request.

modified ()
    Has this Flow been modified?

resume ()
    Continue with the flow - called after an intercept().

revert ()
    Revert to the last backed up state.

replace (pattern, repl, *args, **kwargs)
    Replaces a regular expression pattern with repl in both request and response of the flow. Encoded content will be decoded before replacement, and re-encoded afterwards.

    Returns the number of replacements made.

31.3 Logging

class mitmproxy.log.Log (master)
    The central logger, exposed to scripts as mitmproxy.ctx.log.

debug (txt)
    Log with level debug.

info (txt)
    Log with level info.

warn (txt)
    Log with level warn.

error (txt)
    Log with level error.
class mitmproxy.log.LogEntry (msg, level)
My local cafe is serviced by a rickety and unreliable wireless network, generously sponsored with ratepayers’ money by our city council. After connecting, you are redirected to an SSL-protected page that prompts you for a username and password. Once you’ve entered your details, you are free to enjoy the intermittent dropouts, treacle-like speeds and incorrectly configured transparent proxy.

I tend to automate this kind of thing at the first opportunity, on the theory that time spent now will be more than made up in the long run. In this case, I might use Firebug to ferret out the form post parameters and target URL, then fire up an editor to write a little script using Python’s urllib to simulate a submission. That’s a lot of futzing about. With mitmproxy we can do the job in literally 30 seconds, without having to worry about any of the details. Here’s how.

### 32.1 1. Run mitmdump to record our HTTP conversation to a file.

```bash
>>> mitmdump -w wireless-login
```

### 32.2 2. Point your browser at the mitmdump instance.

I use a tiny Firefox addon called Toggle Proxy to switch quickly to and from mitmproxy. I’m assuming you’ve already configured your browser with mitmproxy’s SSL certificate authority.

### 32.3 3. Log in as usual.

And that’s it! You now have a serialized version of the login process in the file wireless-login, and you can replay it at any time like this:

```bash
>>> mitmdump -c wireless-login
```
32.4 Embellishments

We’re really done at this point, but there are a couple of embellishments we could make if we wanted. I use wicd to automatically join wireless networks I frequent, and it lets me specify a command to run after connecting. I used the client replay command above and voila! - totally hands-free wireless network startup.

We might also want to prune requests that download CSS, JS, images and so forth. These add only a few moments to the time it takes to replay, but they’re not really needed and I somehow feel compelled to trim them anyway. So, we fire up the mitmproxy console tool on our serialized conversation, like so:

```bash
>>> mitmproxy -r wireless-login
```

We can now go through and manually delete (using the ci keyboard shortcut) everything we want to trim. When we’re done, we use w to save the conversation back to the file.
CHAPTER 33

Setting highscores on Apple’s GameCenter

33.1 The setup

In this tutorial, I’m going to show you how simple it is to creatively interfere with Apple Game Center traffic using mitmproxy. To set things up, install the mitmproxy root certificate. Then start mitmproxy on your desktop, and configure the iPhone to use it as a proxy.

33.2 Taking a look at the Game Center traffic

Let’s take a first look at the Game Center traffic. The game I’ll use in this tutorial is Super Mega Worm - a great little retro-apocalyptic sidescroller for the iPhone:
After finishing a game (take your time), watch the traffic flowing through mitmproxy:

We see a bunch of things we might expect - initialisation, the retrieval of leaderboards and so forth. Then, right at the end, there’s a POST to this tantalising URL:

https://service.gc.apple.com/WebObjects/GKGameStatsService.woa/wa/submitScore

The contents of the submission are particularly interesting:
This is a property list, containing an identifier for the game, a score (55, in this case), and a timestamp. Looks pretty simple to mess with.

### 33.3 Modifying and replaying the score submission

Let's edit the score submission. First, select it in mitmproxy, then press `enter` to view it. Make sure you're viewing the request, not the response - you can use `tab` to flick between the two. Now press `e` for edit. You'll be prompted for the part of the request you want to change - press `r` for raw body. Your preferred editor (taken from the `EDITOR` environment variable) will now fire up. Let's bump the score up to something a bit more ambitious:

```xml
<plist version="1.0">
  <dict>
    <key>scores</key>
    <array>
      <dict>
        <key>category</key>
        <string>SMW_Adv_USA1</string>
        <key>context</key>
        <integer>0</integer>
        <key>score-value</key>
        <integer>2200272667</integer>
        <key>timestamp</key>
        <integer>1363515361321</integer>
      </dict>
    </array>
  </dict>
</plist>
```

Save the file and exit your editor.

The final step is to replay this modified request. Simply press `r` for replay.
33.4 The glorious result and some intrigue
And that’s it - according to the records, I am the greatest Super Mega Worm player of all time.

There’s a curious addendum to this tale. When I first wrote this tutorial, all the top competitors’ scores were the same: 2,147,483,647 (this is no longer the case, because there are now so many fellow cheaters using this tutorial). If you think that number seems familiar, you’re right: it’s $2^{31}-1$, the maximum value you can fit into a signed 32-bit int. Now let me tell you another peculiar thing about Super Mega Worm - at the end of every game, it submits your highest previous score to the Game Center, not your current score. This means that it stores your highscore somewhere, and I’m guessing that it reads that stored score back into a signed integer. So, if you _were_ to cheat by the relatively pedestrian means of modifying the saved score on your jailbroken phone, then $2^{31}-1$ might well be the maximum score you could get. Then again, if the game itself stores its score in a signed 32-bit int, you could get the same score through perfect play, effectively beating the game. So, which is it in this case? I’ll leave that for you to decide.
This walkthrough illustrates how to set up transparent proxying with mitmproxy. We use VirtualBox VMs with an Ubuntu proxy machine in this example, but the general Internet <-> Proxy VM <-> (Virtual) Internal Network setup can be applied to other setups.

### 34.1 1. Configure Proxy VM

On the proxy machine, `eth0` is connected to the internet. `eth1` is connected to the internal network that will be proxified and configured to use a static ip (192.168.3.1).
34.1.1 VirtualBox configuration
34.1.2 VM Network Configuration
### 34.2 2. Configure DHCP and DNS

We use dnsmasq to provide DHCP and DNS in our internal network. Dnsmasq is a lightweight server designed to provide DNS (and optionally DHCP and TFTP) services to a small-scale network.

- Before we get to that, we need to fix some Ubuntu quirks: **Ubuntu >12.04** runs an internal dnsmasq instance (listening on loopback only) by default [1]. For our use case, this needs to be disabled by changing `dns=dnsmasq` to `#dns=dnsmasq` in `/etc/NetworkManager/NetworkManager.conf` and

  if on Ubuntu 16.04 or newer running:

  ```
  >>> sudo systemctl restart NetworkManager
  ```

  if on Ubuntu 12.04 or 14.04 running:

  ```
  >>> sudo restart network-manager
  ```

  afterwards.

- Now, dnsmasq can be be installed and configured:

  ```
  >>> sudo apt-get install dnsmasq
  ```

  Replace `/etc/dnsmasq.conf` with the following configuration:

  ```
  # Listen for DNS requests on the internal network
  interface=eth1
  # Act as a DHCP server, assign IP addresses to clients
  dhcp-range=192.168.3.10,192.168.3.100,96h
  # Broadcast gateway and dns server information
  dhcp-option=option:router,192.168.3.1
  dhcp-option=option:dns-server,192.168.3.1
  ```

  Apply changes:

  if on Ubuntu 16.04 or newer:

  ```
  >>> sudo systemctl restart dnsmasq
  ```

  if on Ubuntu 12.04 or 14.04:

  ```
  >>> sudo service dnsmasq restart
  ```

  Your **proxied machine** in the internal virtual network should now receive an IP address via DHCP:
34.3 3. Redirect traffic to mitmproxy

To redirect traffic to mitmproxy, we need to add two iptables rules:

```
sudo iptables -t nat -A PREROUTING -i eth1 -p tcp --dport 80 -j REDIRECT --to-port 8080
sudo iptables -t nat -A PREROUTING -i eth1 -p tcp --dport 443 -j REDIRECT --to-port 8080
```

34.4 4. Run mitmproxy

Finally, we can run mitmproxy in transparent mode with

```
>>> mitmproxy -T
```

The proxied machine cannot to leak any data outside of HTTP or DNS requests. If required, you can now install the mitmproxy certificates on the proxied machine.
Pathology 101

35.1 pathod

Pathod is a pathological HTTP daemon designed to let you craft almost any conceivable HTTP response, including ones that creatively violate the standards. HTTP responses are specified using a small, terse language which pathod shares with its evil twin pathoc. To start playing with pathod, fire up the daemon:

```
>>> pathod
```

By default, the service listens on port 9999 of localhost, and the default crafting anchor point is the path `/p/`. Anything after this URL prefix is treated as a response specifier. So, hitting the following URL will generate an HTTP 200 response with 100 bytes of random data:

```
http://localhost:9999/p/200:b@100
```

See the language documentation to get (much) fancier. The pathod daemon also takes a range of configuration options. To view those, use the command-line help:

```
>>> pathod --help
```

35.1.1 Mimicing a proxy

Pathod automatically responds to both straight HTTP and proxy requests. For proxy requests, the upstream host is ignored, and the path portion of the URL is used to match anchors. This lets you test software that supports a proxy configuration by spoofing responses from upstream servers.

By default, we treat all proxy CONNECT requests as HTTPS traffic, serving the response using either pathod’s built-in certificates, or the cert/key pair specified by the user. You can over-ride this behaviour if you’re testing a client that makes a non-SSL CONNECT request using the -C command-line option.
35.1.2 Anchors

Anchors provide an alternative to specifying the response in the URL. Instead, you attach a response to a pre-configured anchor point, specified with a regex. When a URL matching the regex is requested, the specified response is served.

```bash
>>> pathod -a "/foo=200"
```

Here, “/foo” is the regex specifying the anchor path, and the part after the “=” is a response specifier.

35.1.3 File Access

There are two operators in the language that load contents from file - the + operator to load an entire request specification from file, and the > value specifier. In pathod, both of these operators are restricted to a directory specified at startup, or disabled if no directory is specified:

```bash
>>> pathod -d ~/staticdir"
```

35.1.4 Internal Error Responses

Pathod uses the non-standard 800 response code to indicate internal errors, to distinguish them from crafted responses. For example, a request to:

```http
http://localhost:9999/p/foo
```

... will return an 800 response because “foo” is not a valid page specifier.

35.2 pathoc

Pathoc is a perverse HTTP daemon designed to let you craft almost any conceivable HTTP request, including ones that creatively violate the standards. HTTP requests are specified using a small, terse language, which pathod shares with its server-side twin pathod. To view pathoc’s complete range of options, use the command-line help:

```bash
>>> pathoc --help
```

35.2.1 Getting Started

The basic pattern for pathoc commands is as follows:

```bash
pathoc hostname request [request ...]
```

That is, we specify the hostname to connect to, followed by one or more requests. Let’s start with a simple example:

```bash
> pathoc google.com get:/
07-06-16 12:13:43: >> 'GET':/
<< 302 Found: 261 bytes
```

Here, we make a GET request to the path / on port 80 of google.com. Pathoc’s output tells us that the server responded with a 302 redirection. We can tell pathoc to connect using SSL, in which case the default port is changed to 443 (you can over-ride the default port with the -p command-line option):
35.2.2 Multiple Requests

There are two ways to tell pathoc to issue multiple requests. The first is to specify them on the command-line, like so:

```bash
$ pathoc google.com get:/ get:/
07-06-16 12:21:04: >> 'GET':/
<< 302 Found: 261 bytes
07-06-16 12:21:04: >> 'GET':/
<< 302 Found: 261 bytes
```

In this case, pathoc issues the specified requests over the same TCP connection - so in the above example only one connection is made to google.com.

The other way to issue multiple requests is to use the `-n` flag:

```bash
$ pathoc -n 2 google.com get:/
07-06-16 12:21:04: >> 'GET':/
<< 302 Found: 261 bytes
07-06-16 12:21:04: >> 'GET':/
<< 302 Found: 261 bytes
```

The output is identical, but two separate TCP connections are made to the upstream server. These two specification styles can be combined:

```
pathoc -n 2 google.com get:/ get:/
```

Here, two distinct TCP connections are made, with two requests issued over each.

35.2.3 Basic Fuzzing

The combination of pathoc’s powerful request specification language and a few of its command-line options makes for quite a powerful basic fuzzer. Here’s an example:

```bash
$ pathoc -e -I 200 -t 2 -n 1000 localhost get:/@10:ir,@1
```

The request specified here is a valid GET with a body consisting of 10 random bytes, but with 1 random byte inserted in a random place. This could be in the headers, in the initial request line, or in the body itself. There are a few things to note here:

- Corrupting the request in this way will often make the server enter a state where it’s awaiting more input from the client. This is where the `-t` option comes in, which sets a timeout that causes pathoc to disconnect after two seconds.
- The `-n` option tells pathoc to repeat the request 1000 times.
- The `-I` option tells pathoc to ignore HTTP 200 response codes. You can use this to fine-tune what pathoc considers to be an exceptional condition, and therefore log-worthy.
- The `-e` option tells pathoc to print an explanation of each logged request, in the form of an expanded pathoc specification with all random portions and automatic header additions resolved. This lets you precisely replay a request that triggered an error.
35.2.4 Interacting with Proxies

Pathoc has a reasonably sophisticated suite of features for interacting with proxies. The proxy request syntax very closely mirrors that of straight HTTP, which means that it is possible to make proxy-style requests using pathoc without any additional syntax, by simply specifying a full URL instead of a simple path:

```bash
>>> pathoc -p 8080 localhost "get:http://google.com"
```

Another common use case is to use an HTTP CONNECT request to probe remote servers via a proxy. This is done with the `-c` command-line option, which allows you to specify a remote host and port pair:

```bash
>>> pathoc -c google.com:80 -p 8080 localhost get:/
```

Note that pathoc does **not** negotiate SSL without being explicitly instructed to do so. If you’re making a CONNECT request to an SSL-protected resource, you must also pass the `-s` flag:

```bash
>>> pathoc -sc google.com:443 -p 8080 localhost get:/
```

35.2.5 Embedded response specification

One interesting feature of the Request specification language is that you can embed a response specification in it, which is then added to the request path. Here’s an example:

```bash
>>> pathoc localhost:9999 "get:/p/:s'401:ir,@1'"
```

This crafts a request that connects to the pathod server, and which then crafts a response that generates a 401, with one random byte embedded at a random point. The response specification is parsed and expanded by pathoc, so you see syntax errors immediately. This really becomes handy when combined with the `-e` flag to show the expanded request:

```
07-06-16 12:32:01: >> 'GET':/p/:s'401:i35,\x27\x1b\x27:h'Host'='localhost'
<< 401 Unauthorized: 0 bytes
```

Note that the embedded response has been resolved **before** being sent to the server, so that “ir,@1” (embed a random byte at a random location) has become “i15,’o’” (embed the character “o” at offset 15). You now have a pathoc request specification that is precisely reproducible, even with random components. This feature comes in terribly handy when testing a proxy, since you can now drive the server response completely from the client, and have a complete log of reproducible requests to analyze afterwards.

35.3 Request Examples

<table>
<thead>
<tr>
<th>Request</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get:/</td>
<td>Get path /</td>
</tr>
<tr>
<td>get:/b@100</td>
<td>100 random bytes as the body</td>
</tr>
<tr>
<td>get:/h&quot;Etag&quot;=&quot;&amp;;drop table browsers;&quot;</td>
<td>Add a header</td>
</tr>
<tr>
<td>get:/u&quot;&amp;;drop table browsers;&quot;</td>
<td>Add a User-Agent header</td>
</tr>
<tr>
<td>get:/b@100:dr</td>
<td>Drop the connection randomly</td>
</tr>
<tr>
<td>get:/b@100,ascii:ir,@1</td>
<td>100 ASCII bytes as the body, and randomly inject a random byte</td>
</tr>
<tr>
<td>ws:/</td>
<td>Initiate a websocket handshake.</td>
</tr>
</tbody>
</table>
## 35.4 Response Examples

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>A basic HTTP 200 response.</td>
</tr>
<tr>
<td>200:r</td>
<td>A basic HTTP 200 response with no Content-Length header. This will hang.</td>
</tr>
<tr>
<td>200:da</td>
<td>Server-side disconnect after all content has been sent.</td>
</tr>
<tr>
<td>200:b@100</td>
<td>100 random bytes as the body. A Content-Length header is added, so the disconnect is no longer needed.</td>
</tr>
<tr>
<td>200:b@100:h&quot;Etag&quot;=&quot;';drop table servers;&quot;</td>
<td>Add a Server header</td>
</tr>
<tr>
<td>200:b@100:dr</td>
<td>Drop the connection randomly</td>
</tr>
<tr>
<td>200:b@100,ascii:ir,@1</td>
<td>100 ASCII bytes as the body, and randomly inject a random byte</td>
</tr>
<tr>
<td>200:b@1k:c&quot;text/json&quot;</td>
<td>1k of random bytes, with a text/json content type</td>
</tr>
<tr>
<td>200:b@1k:p50,120</td>
<td>1k of random bytes, pause for 120 seconds after 50 bytes</td>
</tr>
<tr>
<td>200:b@1k:pr,f</td>
<td>1k of random bytes, but hang forever at a random location</td>
</tr>
<tr>
<td>200:b@100:h@1k,ascii_letters='foo'</td>
<td>100 ASCII bytes as the body, randomly generated 100k header name, with the value ‘foo’.</td>
</tr>
</tbody>
</table>
36.1 HTTP Request

method: path: [colon-separated list of features]
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>A \texttt{VALUE} specifying the HTTP method to use. Standard methods do not need to be enclosed in quotes, while non-standard methods can be specified as quoted strings. The special method \texttt{ws} creates a valid websocket upgrade GET request, and signals to pathoc to switch to websocket receive mode if the server responds correctly. Apart from that, websocket requests are just like any other, and all aspects of the request can be over-ridden.</td>
</tr>
<tr>
<td>h:VALUE=VALUE</td>
<td>Set a header.</td>
</tr>
<tr>
<td>r</td>
<td>Set the raw flag on this response. Pathod will not calculate a Content-Length header if a body is set.</td>
</tr>
<tr>
<td>cVALUE</td>
<td>A shortcut for setting the Content-Type header. Equivalent to \texttt{h&quot;Content-Type&quot;=VALUE}</td>
</tr>
<tr>
<td>uVALUE uSHORTCUT</td>
<td>Set a User-Agent header on this request. You can specify either a complete \texttt{VALUE}, or a User-Agent shortcut: \texttt{android}, \texttt{blackberry}, \texttt{bingbot}, \texttt{chrome}, \texttt{firefox}, \texttt{googlebot}, \texttt{ie9}, \texttt{ipad}, \texttt{iphone}, \texttt{safari}.</td>
</tr>
<tr>
<td>bVALUE</td>
<td>Set the body. The appropriate Content-Length header is added automatically unless the r flag is set.</td>
</tr>
<tr>
<td>sVALUE</td>
<td>An embedded Response specification, appended to the path of the request.</td>
</tr>
<tr>
<td>xINTEGER</td>
<td>Repeat this message \texttt{N} times.</td>
</tr>
<tr>
<td>dOFFSET</td>
<td>Disconnect after \texttt{OFFSET} bytes (HTTP/1 only).</td>
</tr>
<tr>
<td>iOFFSET,VALUE</td>
<td>Inject the specified value at the offset (HTTP/1 only)</td>
</tr>
<tr>
<td>pOFFSET,SECONDS</td>
<td>Pause for \texttt{SECONDS} seconds after \texttt{OFFSET} bytes. SECONDS can be an integer or “f” to pause forever (HTTP/1 only)</td>
</tr>
</tbody>
</table>

### 36.2 HTTP Response

\texttt{code:[:colon-separated list of features]}
code | An integer specifying the HTTP response code. The special method ws creates a valid websocket upgrade response (code 101), and moves pathod to websocket mode. Apart from that, websocket responses are just like any other, and all aspects of the response can be over-ridden.

m.VALUE | HTTP Reason message. Automatically chosen according to the response code if not specified. (HTTP/1 only)

h:VALUE=VALUE | Set a header.

r | Set the raw flag on this response. Pathod will not calculate a Content-Length header if a body is set.

l.VALUE | A shortcut for setting the Location header. Equivalent to h"Location"=VALUE

c.VALUE | A shortcut for setting the Content-Type header. Equivalent to h"Content-Type"=VALUE

b.VALUE | Set the body. The appropriate Content-Length header is added automatically unless the r flag is set.

d.OFFSET | Disconnect after OFFSET bytes (HTTP/1 only).

i.OFFSET,VALUE | Inject the specified value at the offset (HTTP/1 only)

p.OFFSET,SECONDS | Pause for SECONDS seconds after OFFSET bytes. SECONDS can be an integer or "f" to pause forever (HTTP/1 only)

### 36.3 Websocket Frame

wf:[colon-separated list of features]

| b.VALUE | Set the frame payload. If a masking key is present, the value is encoded automatically. |
| c.INTEGER | Set the op code. This can either be an integer from 0-15, or be one of the following opcode names: text (the default), continue, binary, close, ping, pong. |
| d.OFFSET | Disconnect after OFFSET bytes |
| i.OFFSET,VALUE | Inject the specified value at the offset |
| p.OFFSET,SECONDS | Disconnect after OFFSET bytes. SECONDS can be an integer or "f" to pause forever |
| x.INTEGER | Repeat this message N times. |
| [-]fin | Set or un-set the fin bit. |
| k.VALUE | Set the masking key. The resulting value must be exactly 4 bytes long. The special form knone specifies that no key should be set, even if the mask bit is on. |
| i.INTEGER | Set the payload length in the frame header, regardless of the actual body length. |
| [-]mask | Set or un-set the <b>mask</b> bit. |
| r.VALUE | Set the raw frame payload. This disables masking, even if the key is present. |
| [-]rsv1 | Set or un-set the rsv1 bit. |
| [-]rsv2 | Set or un-set the rsv2 bit. |
| [-]rsv3 | Set or un-set the rsv3 bit. |
36.4 Data types

36.4.1 INTEGER

36.4.2 OFFSET

Offsets are calculated relative to the base message, before any injections or other transforms are applied. They have 3 flavors:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>An integer byte offset</td>
</tr>
<tr>
<td>r</td>
<td>A random location</td>
</tr>
<tr>
<td>a</td>
<td>The end of the message</td>
</tr>
</tbody>
</table>

36.4.3 VALUE

Literals

Literal values are specified as a quoted strings:

```
"foo"
```

Either single or double quotes are accepted, and quotes can be escaped with backslashes within the string:

```
'fo\'o'
```

Literal values can contain Python-style backslash escape sequences:

```
'foo\r\nbar'
```

Generated

An @-symbol lead-in specifies that generated data should be used. There are two components to a generator specification - a size, and a data type. By default pathod assumes a data type of “bytes”.

Here’s a value specifier for generating 100 bytes:

```
@100
```

You can use standard suffixes to indicate larger values. Here, for instance, is a specifier for generating 100 megabytes:

```
@100m
```

Data is generated and served efficiently - if you really want to send a terabyte of data to a client, pathod can do it. The supported suffixes are:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>$1024^0$ (bytes)</td>
</tr>
<tr>
<td>k</td>
<td>$1024^1$ (kilobytes)</td>
</tr>
<tr>
<td>m</td>
<td>$1024^2$ (megabytes)</td>
</tr>
<tr>
<td>g</td>
<td>$1024^3$ (gigabytes)</td>
</tr>
<tr>
<td>t</td>
<td>$1024^4$ (terabytes)</td>
</tr>
</tbody>
</table>

Data types are separated from the size specification by a comma. This specification generates 100mb of ASCII:

```
@100m,ascii
```

Supported data types are:

<table>
<thead>
<tr>
<th>data type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascii</td>
<td>All ASCII characters</td>
</tr>
<tr>
<td>ascii_letters</td>
<td>A-Za-z</td>
</tr>
<tr>
<td>ascii_lowercase</td>
<td>a-z</td>
</tr>
<tr>
<td>ascii_uppercase</td>
<td>A-Z</td>
</tr>
<tr>
<td>bytes</td>
<td>All 256 byte values</td>
</tr>
<tr>
<td>digits</td>
<td>0-9</td>
</tr>
<tr>
<td>hexdigits</td>
<td>0-f</td>
</tr>
<tr>
<td>octdigits</td>
<td>0-7</td>
</tr>
<tr>
<td>punctuation</td>
<td>!&quot;#$%&amp;'()*+,-./:;&lt;=?&gt;@[]^_`{</td>
</tr>
<tr>
<td>whitespace</td>
<td>\t \n \x0b \x0c \r and space</td>
</tr>
</tbody>
</table>

Files

You can load a value from a specified file path. To do so, you have to specify a _staticdir_ option to pathod on the command-line, like so:

```bash
>>> pathod -d ~/myassets
```

All paths are relative paths under this directory. File loads are indicated by starting the value specifier with the left angle bracket:

```text
<my/path
```

The path value can also be a quoted string, with the same syntax as literals:

```text
"my/path"
```
Behind the pathod and pathoc command-line tools lurks the **pathod** library, a powerful way to manipulate and serve HTTP requests and responses from code. The canonical documentation for the library is in the code, and can be accessed using pydoc.

Listing 37.1: examples/pathod/libpathod_pathoc.py

```python
#!/usr/bin/env python
from pathod import pathoc

p = pathoc.Pathoc(('google.com', 80))
p.connect()
print(p.request("/get:/"))
print(p.request("/get:/foo"))
```
The `pathod.test` module is a light, flexible testing layer for HTTP clients. It works by firing up a Pathod instance in a separate thread, letting you use Pathod’s full abilities to generate responses, and then query Pathod’s internal logs to establish what happened. All the mechanics of startup, shutdown, finding free ports and so forth are taken care of for you.

The canonical docs can be accessed using `pydoc`:

```bash
>>> pydoc pathod.test
```

The remainder of this page demonstrates some common interaction patterns using `Nose`. These examples are also applicable with only minor modification to most commonly used Python testing engines.

### 38.1 Context Manager

Listing 38.1: examples/pathod/test_context.py

```python
import requests
from pathod import test

def test_simple():
    
    Testing the requests module with
    a pathod context manager.

    # Start pathod in a separate thread
    with test.Daemon() as d:
        # Get a URL for a pathod spec
        url = d.p("200:b@100")
        # ... and request it
        r = requests.put(url)

        # Check the returned data
```
38.2 One instance per test

Listing 38.2: examples/pathod/test_setup.py

```python
import requests
from pathod import test

class Test:
    """
    Testing the requests module with a pathod instance started for each test.
    """
    def setup(self):
        self.d = test.Daemon()

    def teardown(self):
        self.d.shutdown()

def test_simple(self):
    # Get a URL for a pathod spec
    url = self.d.p("200:b@100")
    # ... and request it
    r = requests.put(url)

    # Check the returned data
    assert r.status_code == 200
    assert len(r.content) == 100

    # Check pathod's internal log
    log = self.d.last_log()\["request"]
    assert log["method"] == "PUT"
```

As an open source project, mitmproxy welcomes contributions of all forms.
Please head over to the README to get started!
TLS Master Secrets

The SSL master keys can be logged by mitmproxy so that external programs can decrypt TLS connections both from and to the proxy. Key logging is enabled by setting the environment variable `SSLKEYLOGFILE` so that it points to a writable text file. Recent versions of WireShark can use these log files to decrypt packets. You can specify the key file path in WireShark via `Edit -> Preferences -> Protocols -> SSL -> (Pre)-Master-Secret log filename`.

Note that `SSLKEYLOGFILE` is respected by other programs as well, e.g. Firefox and Chrome. If this creates any issues, you can set `MITMPROXY_SSLKEYLOGFILE` alternatively.
Index

A
anticache() (mitmproxy.http.HTTPRequest method), 88
anticomp() (mitmproxy.http.HTTPRequest method), 88

B
backup() (mitmproxy.http.HTTPFlow method), 92

C
client_conn (mitmproxy.http.HTTPFlow attribute), 92
clientconnect() (built-in function), 84
clientdisconnect() (built-in function), 84
configure() (built-in function), 83
constrain_encoding() (mitmproxy.http.HTTPRequest method), 88
content (mitmproxy.http.HTTPRequest attribute), 88
content (mitmproxy.http.HTTPResponse attribute), 90
cookies (mitmproxy.http.HTTPRequest attribute), 88
cookies (mitmproxy.http.HTTPResponse attribute), 90

D
debug() (mitmproxy.log.Log method), 92
decode() (mitmproxy.http.HTTPRequest method), 88
decode() (mitmproxy.http.HTTPResponse method), 90
done() (built-in function), 83

E
encode() (mitmproxy.http.HTTPRequest method), 88
encode() (mitmproxy.http.HTTPResponse method), 90
environment variable
   MITMPROXY_SSLKEYLOGFILE, 127
   SSLKEYLOGFILE, 127
Error (class in mitmproxy.flow), 87
error (mitmproxy.http.HTTPFlow attribute), 92
error() (built-in function), 85
error() (mitmproxy.log.Log method), 92

F
first_line_format (mitmproxy.http.HTTPRequest attribute), 88

G
get_content() (mitmproxy.http.HTTPRequest method), 88
get_content() (mitmproxy.http.HTTPResponse method), 90
get_state() (mitmproxy.flow.Error method), 87
get_text() (mitmproxy.http.HTTPRequest method), 88
get_text() (mitmproxy.http.HTTPResponse method), 90

H
headers (mitmproxy.http.HTTPRequest attribute), 88
headers (mitmproxy.http.HTTPResponse attribute), 91
host (mitmproxy.http.HTTPRequest attribute), 89
host_header (mitmproxy.http.HTTPRequest attribute), 89
http_connect() (built-in function), 85
http_version (mitmproxy.http.HTTPRequest attribute), 89
http_version (mitmproxy.http.HTTPResponse attribute), 91
HTTPFlow (class in mitmproxy.http), 91
HTTPRequest (class in mitmproxy.http), 88
HTTPResponse (class in mitmproxy.http), 90

I
info() (mitmproxy.log.Log method), 92
intercept() (mitmproxy.http.HTTPFlow method), 92
intercepted (mitmproxy.http.HTTPFlow attribute), 92

K
kill() (mitmproxy.http.HTTPFlow method), 92

L
Log (class in mitmproxy.log), 92
log() (built-in function), 83
LogEntry (class in mitmproxy.log), 92

M
make() (mitmproxy.http.HTTPResponse method), 91
method (mitmproxy.http.HTTPRequest attribute), 89
MITMPROXY_SSLKEYLOGFILE, 127
mode (mitmproxy.http.HTTPFlow attribute), 92
modified() (mitmproxy.http.HTTPFlow method), 92
multipart_form (mitmproxy.http.HTTPRequest attribute), 89
next_layer() (built-in function), 84
path (mitmproxy.http.HTTPRequest attribute), 89
path_components (mitmproxy.http.HTTPRequest attribute), 89
port (mitmproxy.http.HTTPRequest attribute), 89
pretty_host (mitmproxy.http.HTTPRequest attribute), 89
pretty_url (mitmproxy.http.HTTPRequest attribute), 89
query (mitmproxy.http.HTTPRequest attribute), 89
raw_content (mitmproxy.http.HTTPRequest attribute), 89
reason (mitmproxy.http.HTTPResponse attribute), 91
refresh() (mitmproxy.http.HTTPResponse method), 91
replace() (mitmproxy.http.HTTPFlow method), 92
replace() (mitmproxy.http.HTTPRequest method), 89
replace() (mitmproxy.http.HTTPResponse method), 91
request (mitmproxy.http.HTTPFlow attribute), 91
request() (built-in function), 85
requestheaders() (built-in function), 85
response (mitmproxy.http.HTTPFlow attribute), 91
response() (built-in function), 85
responseheaders() (built-in function), 85
resume() (mitmproxy.http.HTTPFlow method), 92
revert() (mitmproxy.http.HTTPFlow method), 92
scheme (mitmproxy.http.HTTPRequest attribute), 89
server_conn (mitmproxy.http.HTTPFlow attribute), 92
serverconnect() (built-in function), 84
serverdisconnect() (built-in function), 84
set_state() (mitmproxy.flow.Error method), 87
SSLKEYLOGFILE, 127
start() (built-in function), 83
status_code (mitmproxy.http.HTTPResponse attribute), 91
text (mitmproxy.http.HTTPRequest attribute), 89
text (mitmproxy.http.HTTPResponse attribute), 91
tick() (built-in function), 83
timestamp_end (mitmproxy.http.HTTPRequest attribute), 90
timestamp_end (mitmproxy.http.HTTPResponse attribute), 91
timestamp_start (mitmproxy.http.HTTPRequest attribute), 90
timestamp_start (mitmproxy.http.HTTPResponse attribute), 91
url (mitmproxy.http.HTTPRequest attribute), 90
urlencoded_form (mitmproxy.http.HTTPRequest attribute), 90
warn() (mitmproxy.log.Log method), 92
websocket_end() (built-in function), 86
websocket_error() (built-in function), 86
websocket_handshake() (built-in function), 86
websocket_message() (built-in function), 86
websocket_start() (built-in function), 86
wrap() (mitmproxy.http.HTTPRequest class method), 88
wrap() (mitmproxy.http.HTTPResponse class method), 90
tcp_end() (built-in function), 86
tcp_error() (built-in function), 86
tcp_message() (built-in function), 86
tcp_start() (built-in function), 86