mitmproxy docs

Release 0.18

July 20, 2016

mitmproxy is an interactive, SSL-capable man-in-the-middle proxy for HTTP with a console interface.

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

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

Features

- Intercept HTTP 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 certificates for interception are generated on the fly.
- And much, much more.

CHAPTER 1

Introduction

mitmproxy is an interactive, SSL-capable man-in-the-middle proxy for HTTP with a console interface.

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

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

Features

- Intercept HTTP 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 certificates for interception are generated on the fly.
- And much, much more.

CHAPTER 2

Installation

2.1 Installation On Ubuntu

Ubuntu comes with Python but we need to install pip, python-dev and several libraries. This was tested on a fully patched installation of Ubuntu 14.04.

```
sudo apt-get install python-pip python-dev libffi-dev libssl-dev libxml2-dev libxslt1-dev libjpeg8-de
sudo pip install mitmproxy  # or pip install --user mitmproxy
```

Once installation is complete you can run *mitmproxy* or *mitmdump* from a terminal.

On **Ubuntu 12.04** (and other systems with an outdated version of pip), you may need to update pip using `pip install -U pip` before installing mitmproxy.

2.1.1 Installation From Source (Ubuntu)

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 mitmproxy installation (see *Installation On Ubuntu*). Then see the Hacking section of the README on GitHub.

2.2 Installation On Fedora

Fedora comes with Python but we need to install pip, python-dev and several libraries. This was tested on a fully patched installation of Fedora 23.

```
sudo dnf install -y python-pip python-devel libffi-devel openssl-devel libxml2-devel libxslt-devel li
sudo pip install mitmproxy  # or pip install --user mitmproxy
```

Once installation is complete you can run *mitmproxy* or *mitmdump* from a terminal.

2.3 Installation On Arch Linux

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

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

2.4 Installation On Mac OS X

The easiest way to get up and running on OSX is to download the pre-built binary packages from mitmproxy.org.

There are a few bits of customization you might want to do to make mitmproxy comfortable to use on OSX. The default color scheme is optimized for a dark background terminal, but you can select a palette for a light terminal background with the `--palette` option. You can use the OSX **open** program to create a simple and effective `~/.mailcap` file to view request and response bodies:

```
application/*; /usr/bin/open -Wn %s
audio/*; /usr/bin/open -Wn %s
image/*; /usr/bin/open -Wn %s
video/*; /usr/bin/open -Wn %s
```

Once installation is complete you can run *mitmproxy* or *mitmdump* from a terminal.

2.4.1 Installation From Source (Mac OS X)

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, there are a few OS X specific things to keep in mind.

- Make sure that XCode is installed from the App Store, and that the command-line tools have been downloaded (XCode/Preferences/Downloads).
- If you're running a Python interpreter installed with homebrew (or similar), you may have to install some dependencies by hand.

Then see the Hacking section of the README on GitHub.

2.5 Installation On Windows

Note: Please note that mitmdump is the only component of mitmproxy that is supported on Windows at the moment.

There is no interactive user interface on Windows.

First, install the latest version of Python 2.7 from the Python website. If you already have an older version of Python 2.7 installed, make sure to install pip (pip is included in Python 2.7.9+ by default). If pip aborts with an error, make sure you are using the current version of pip.

```
>>> python -m pip install --upgrade pip
```

Next, add Python and the Python Scripts directory to your **PATH** variable. You can do this easily by running the following in powershell:

```
>>> [Environment]::SetEnvironmentVariable("Path", "$env:Path;C:\Python27;C:\Python27\Scripts", "User"
```

Now, you can install mitmproxy by running

```
>>> pip install mitmproxy
```

Once the installation is complete, you can run *mitmdump* from a command prompt.

2.5.1 Installation From Source (Windows)

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 Python as outlined above, then see the Hacking section of the README on GitHub.

CHAPTER 3

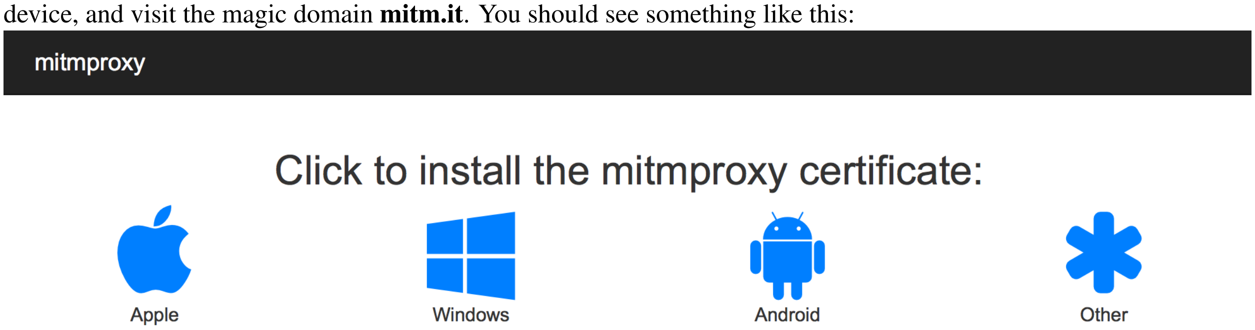
About Certificates

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 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

http://kb.mit.edu/confluence/pages/viewpage.action?pageId=152600377

3.3.2 iOS Simulator

See https://github.com/ADVTOOLS/ADVTrustStore#how-to-use-advtruststore

3.3.3 Java

See http://docs.oracle.com/cd/E19906-01/820-4916/geygn/index.html

3.3.4 Android/Android Simulator

See http://wiki.cacert.org/FAQ/ImportRootCert#Android_Phones_.26_Tablets

3.3.5 Windows

See http://windows.microsoft.com/en-ca/windows/import-export-certificates-private-keys#1TC=windows-7

3.3.6 Windows (automated)

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

See also: https://technet.microsoft.com/en-us/library/cc732443.aspx

3.3.7 Mac OS X

See https://support.apple.com/kb/PH7297?locale=en_US

3.3.8 Ubuntu/Debian

See http://askubuntu.com/questions/73287/how-do-i-install-a-root-certificate/94861#94861

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:

mitmproxy-ca.pem	The certificate **and the private key** in PEM format.
mitmproxy-ca-cert.pem	The certificate in PEM format. Use this to distribute on most non-Windows platforms.
mitmproxy-ca-cert.p12	The certificate in PKCS12 format. For use on Windows.
mitmproxy-ca-cert.cer	Same file as .pem, but with an extension expected by some Android devices.

3.6 Using a custom certificate

You can use your own certificate by passing the `--cert` option to mitmproxy. Mitmproxy then uses the provided certificate for interception of the specified domains 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 you PEM file roughly looks like this:

```
-----BEGIN PRIVATE KEY-----
<private key>
-----END PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
<cert>
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
<intermediary cert (optional)>
-----END CERTIFICATE-----
```

For example, you can generate a certificate in this format using these instructions:

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

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.

CHAPTER 4

How mitmproxy works

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 SSL-protected traffic [1] in the presence of Server Name Indication.

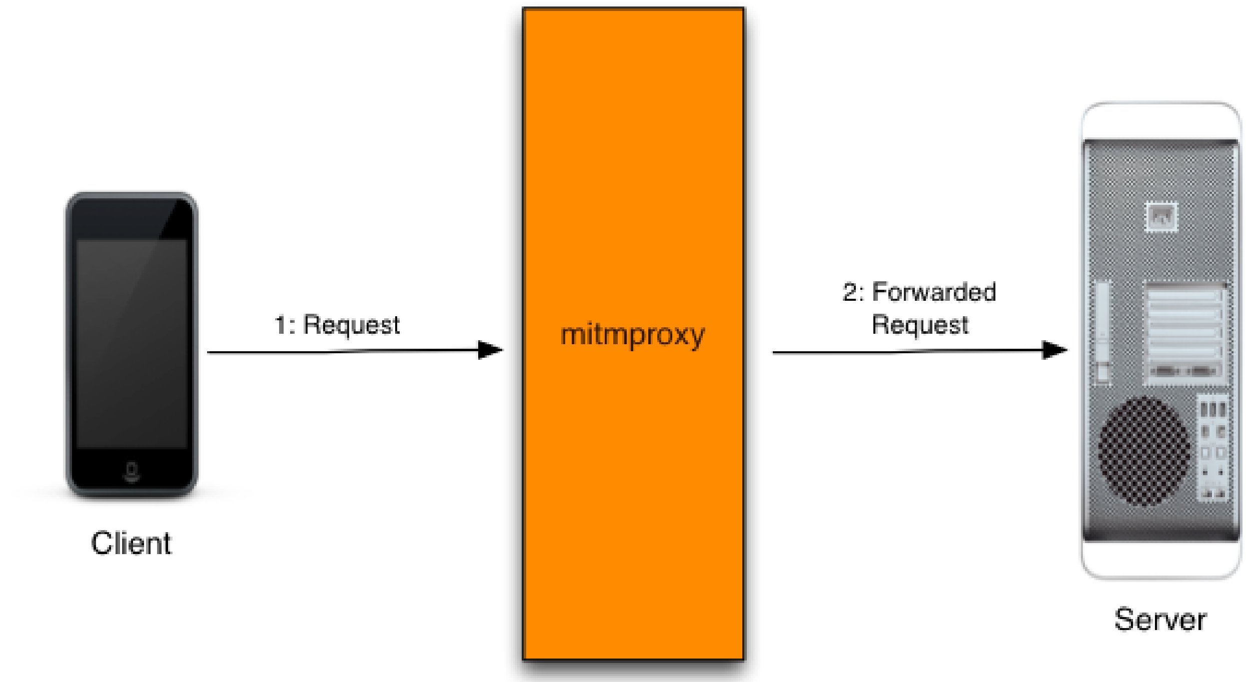
4.1 Explicit HTTP

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] I use "SSL" to refer to both SSL and TLS 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 SSL-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 SSL 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 SSL 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 SSL 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 SSL handshake with the server, and inspect the certificates it used. Now, we use the Common Name in the upstream SSL certificates to generate the dummy certificate for the client. Voila, 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 SSL 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 Common Name. 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.

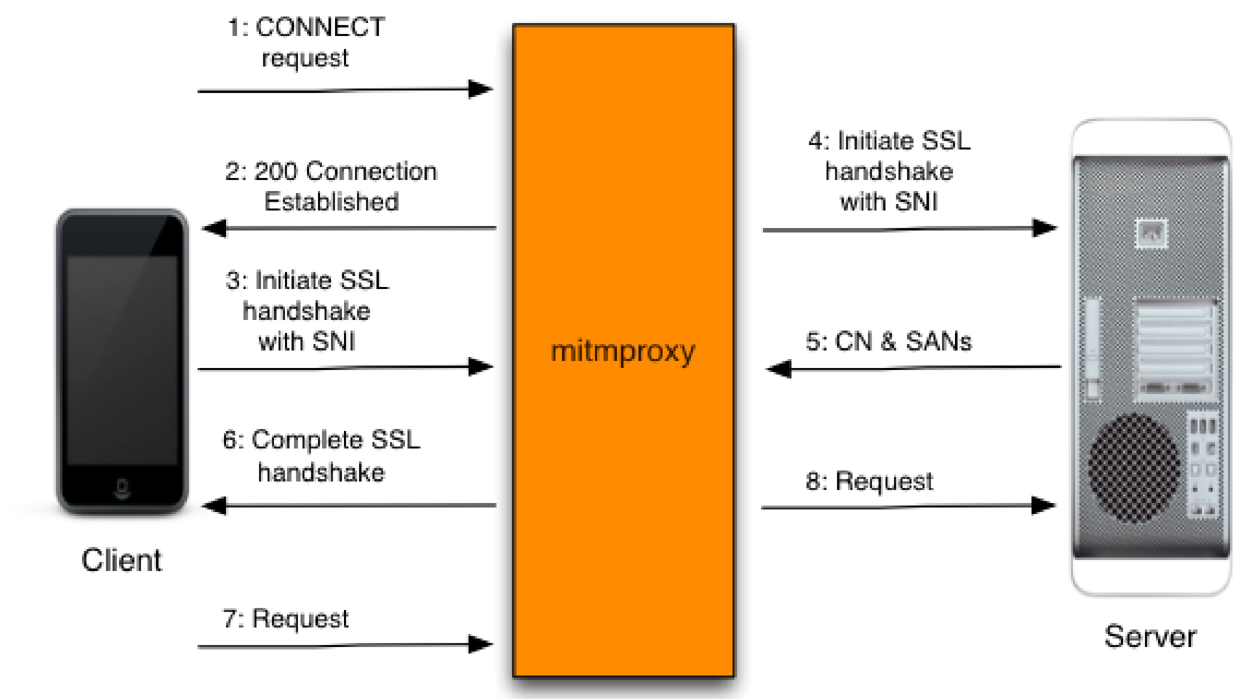
4.2.4 Complication 3: Server Name Indication

One of the big limitations of vanilla SSL 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 SSL and TLS protocols. This lets the client specify the remote server name at the start of the SSL 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 SSL 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

Lets 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 SSL connection. It uses SNI to indicate the hostname it is connecting to.
4. Mitmproxy connects to the server, and establishes an SSL connection using the SNI hostname indicated by the client.
5. The server responds with the matching SSL certificate, which contains the CN and SAN values needed to generate the interception certificate.
6. Mitmproxy generates the interception cert, and continues the client SSL handshake paused in step 3.
7. The client sends the request over the established SSL connection.
8. Mitmproxy passes the request on to the server over the SSL connection initiated in step 4.

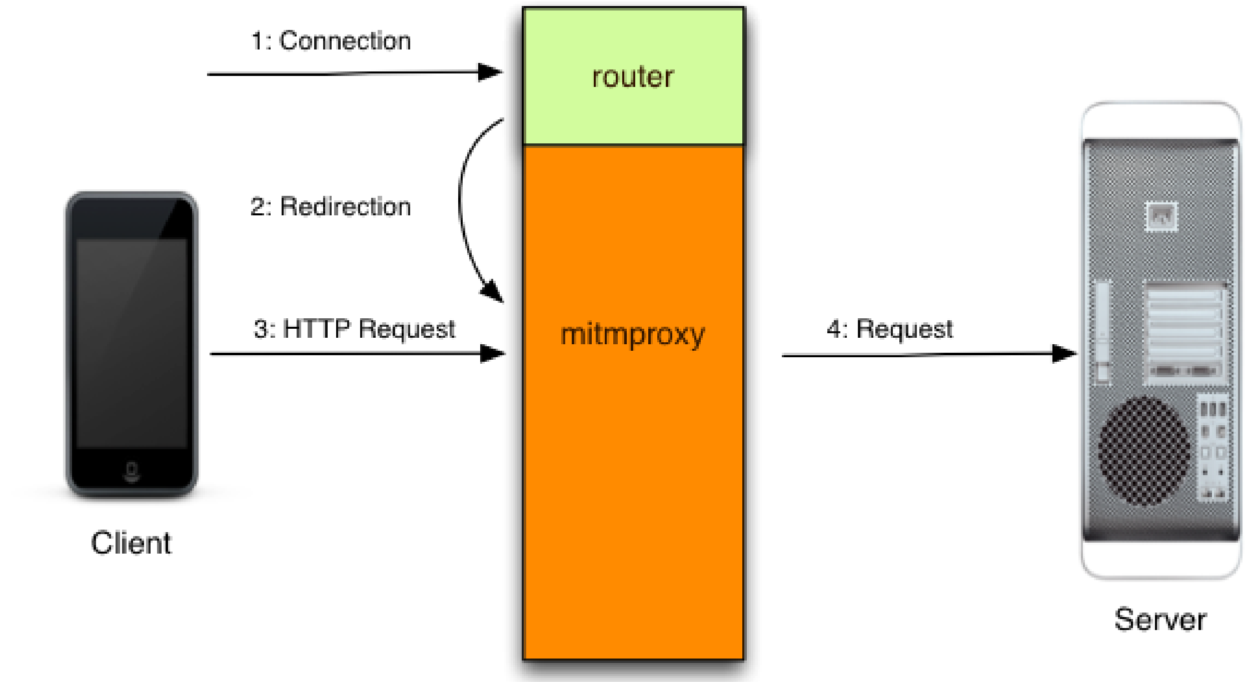
4.3 Transparent HTTP

When a transparent proxy is used, the HTTP/S 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.

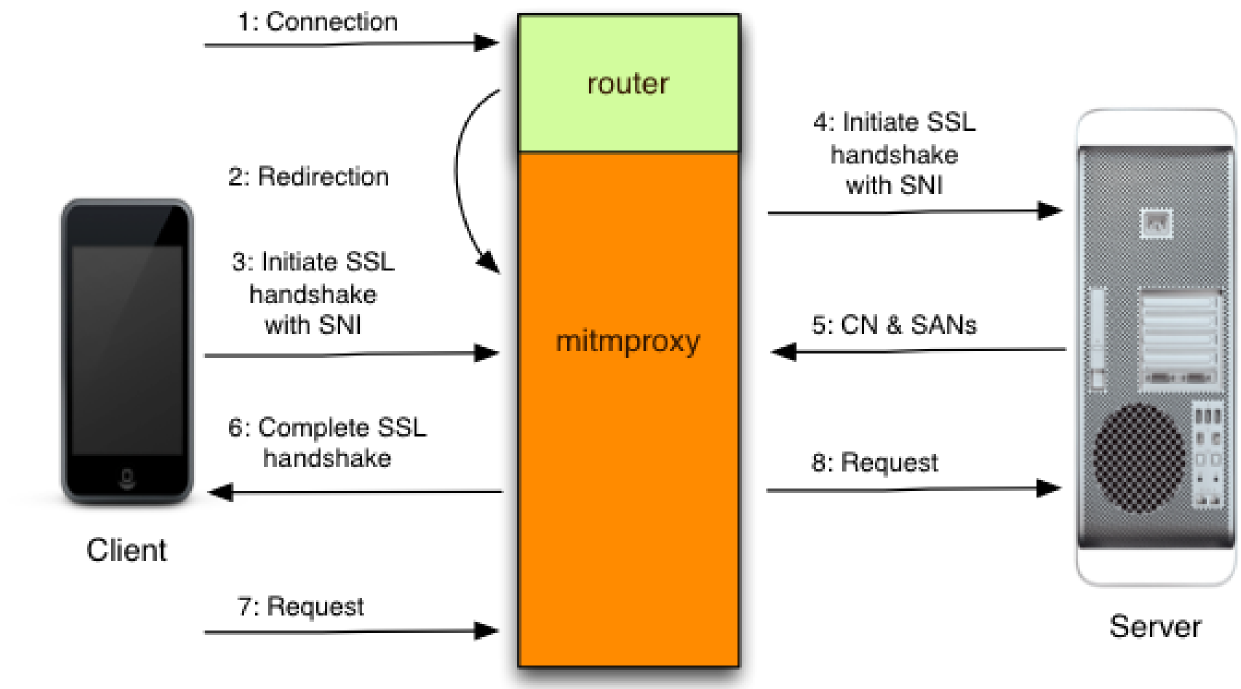


1. The client makes a connection to the server.
2. The router redirects the connection to mitmproxy, which is typically listening on a local port of the same host. Mitmproxy then consults the routing mechanism to establish what the original destination was.
3. Now, we simply read the client's request...
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. By default, we treat all traffic destined for ports 443 and 8443 as SSL.

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 SSL connection. It uses SNI to indicate the hostname it is connecting to.
4. Mitmproxy connects to the server, and establishes an SSL connection using the SNI hostname indicated by the client.
5. The server responds with the matching SSL certificate, which contains the CN and SAN values needed to generate the interception certificate.
6. Mitmproxy generates the interception cert, and continues the client SSL handshake paused in step 3.
7. The client sends the request over the established SSL connection.
8. Mitmproxy passes the request on to the server over the SSL connection initiated in step 4.

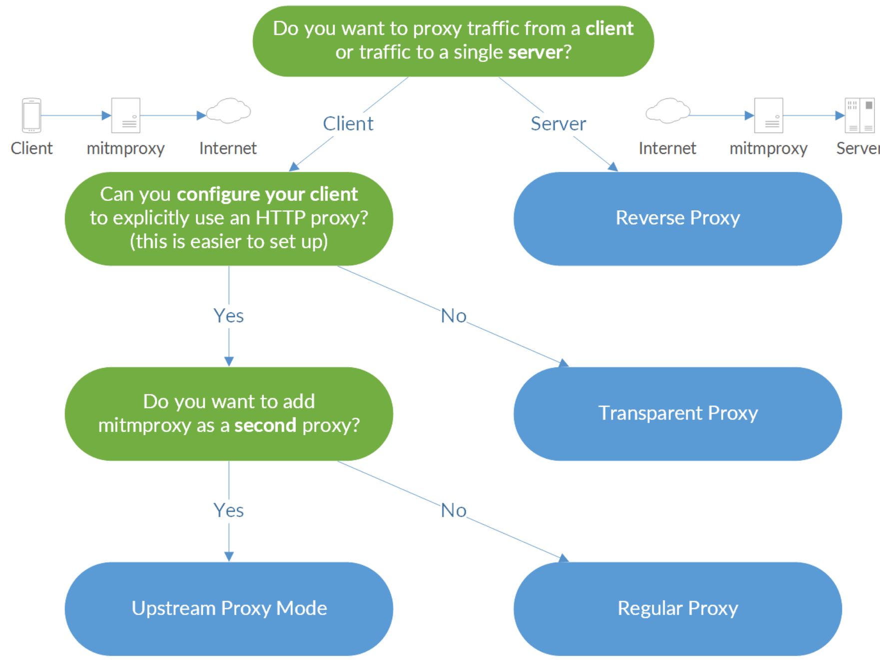
CHAPTER 5

Modes of Operation

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

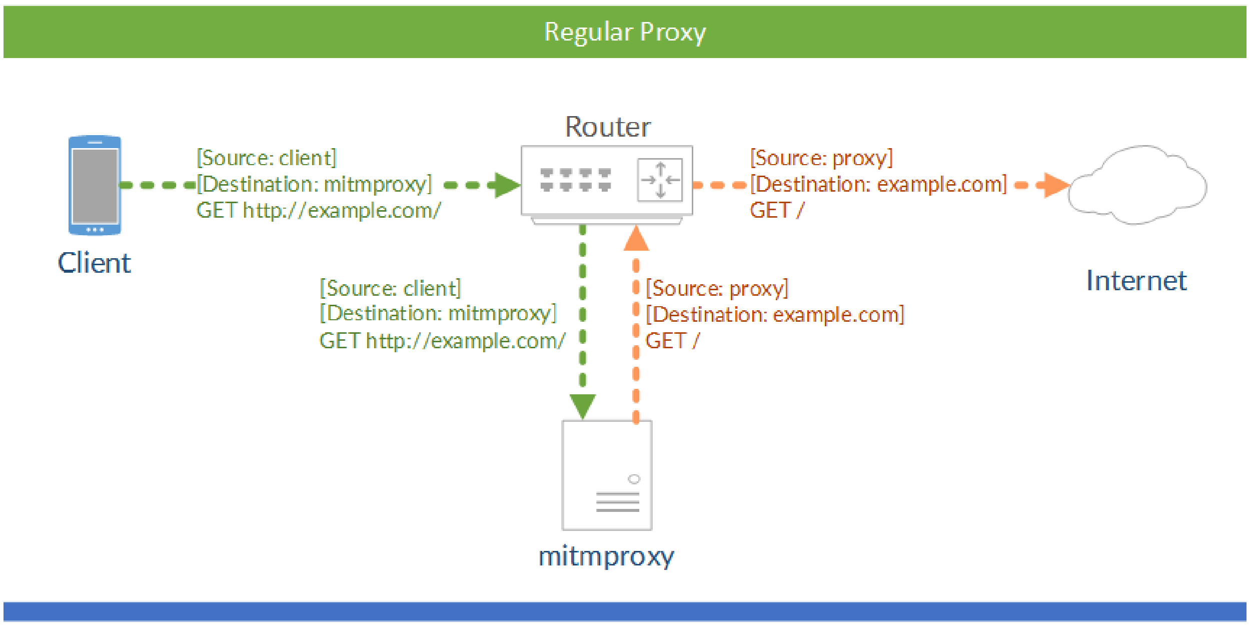
Mitmproxy's regular mode is the simplest and the easiest to set up.

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.

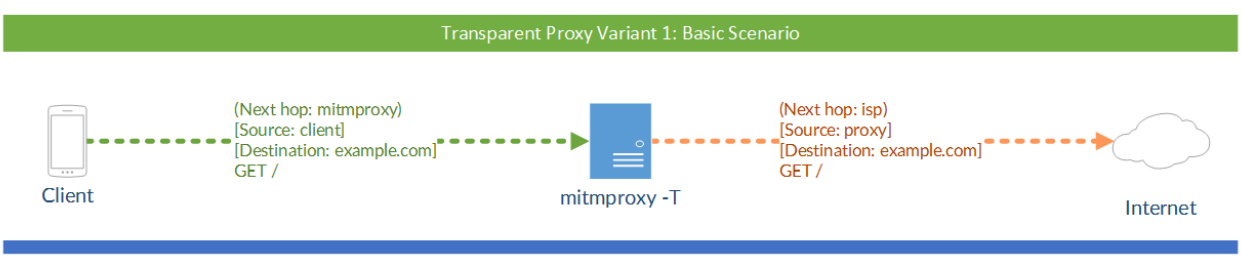
If you are proxying an external device, your network will probably look like this:



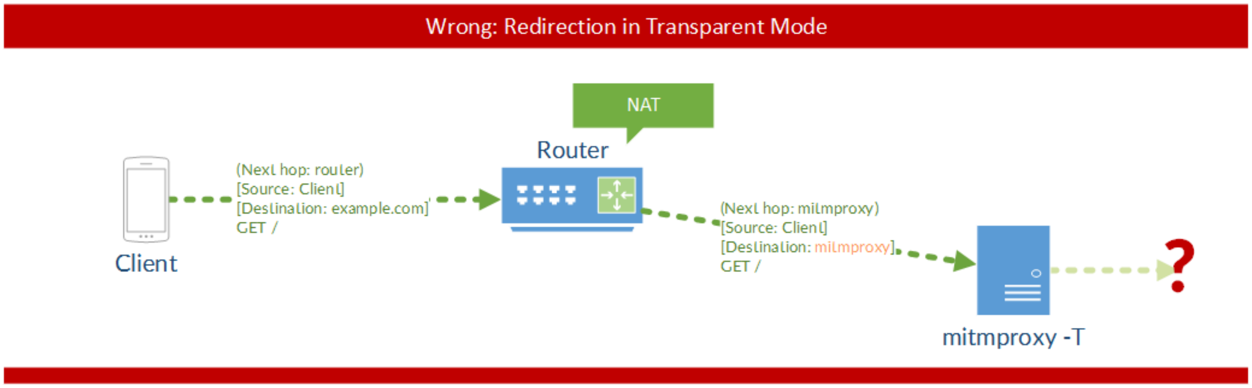
The square brackets signify the source and destination IP addresses. Your client explicitly connects to mitmproxy and mitmproxy explicitly connects to the target server.

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:



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

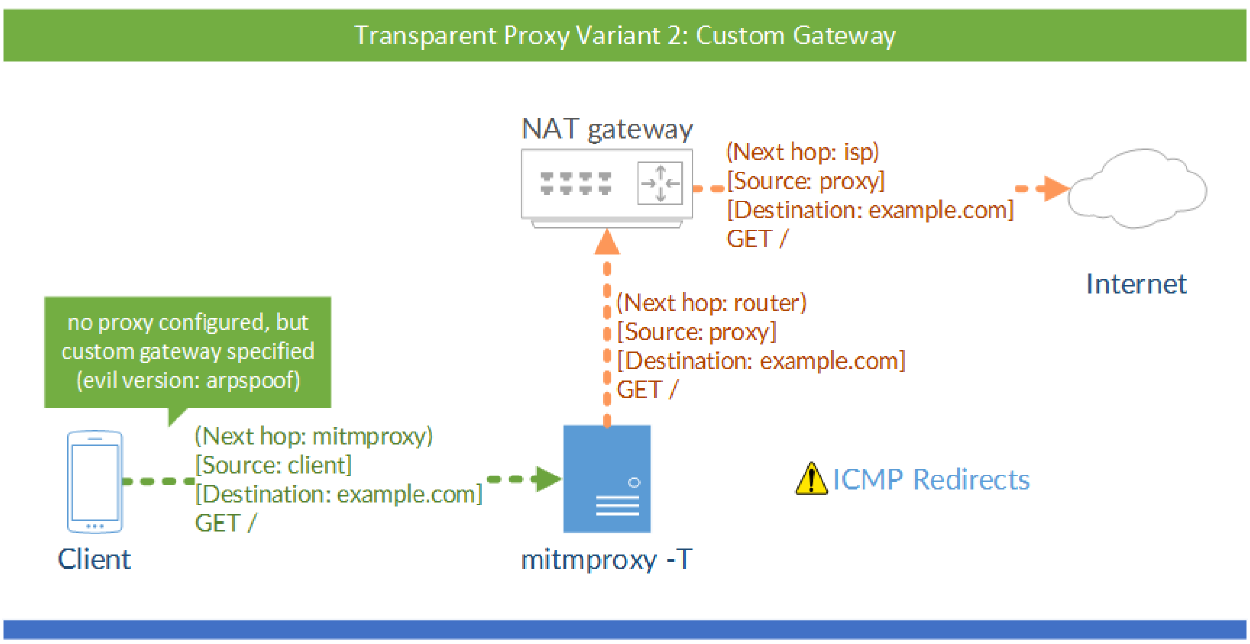
There are many ways to configure your network for transparent proxying. We'll look at two common scenarios:

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

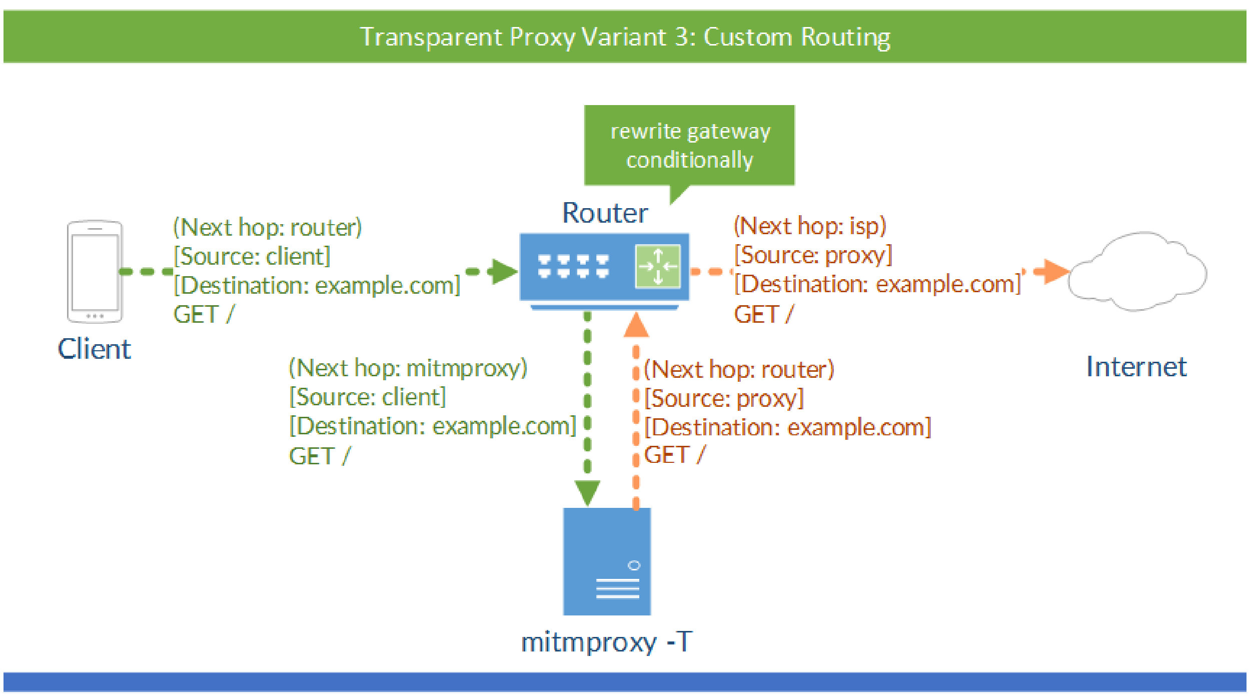
Incorrect transparent mode configurations are a frequent source of error. If it doesn't work for you, try the following things:

- 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!

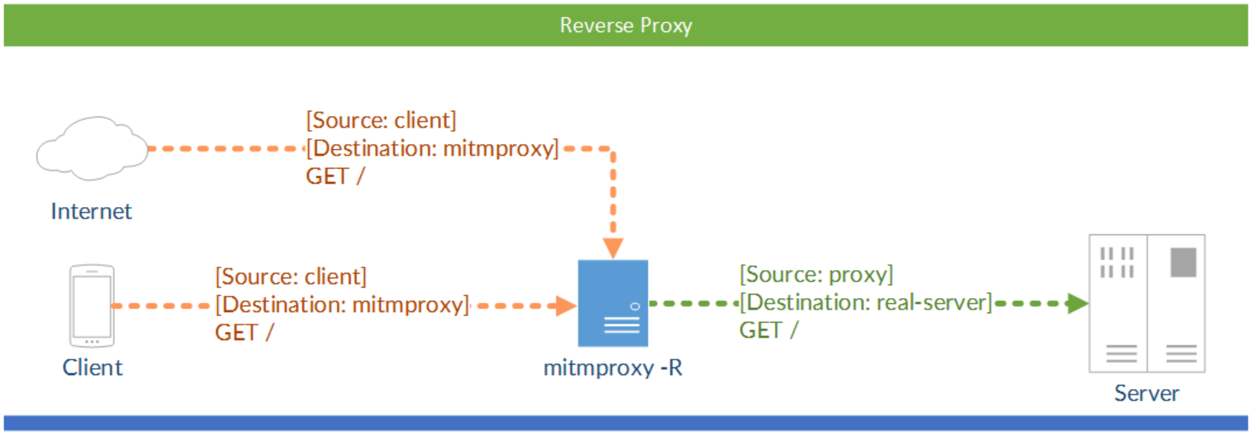
(b) Custom Routing

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:



5.3 Reverse Proxy

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:

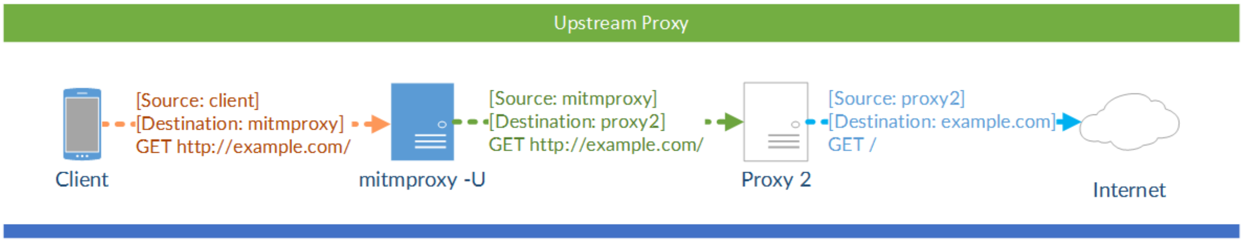
- Say you have an internal API running at http://example.local/. You could now set up mitmproxy in reverse proxy mode at http://debug.example.local/ and dynamically point clients to this new API endpoint, which provides them with the same data and you with debug information. Similarly, you could move 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 an 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).

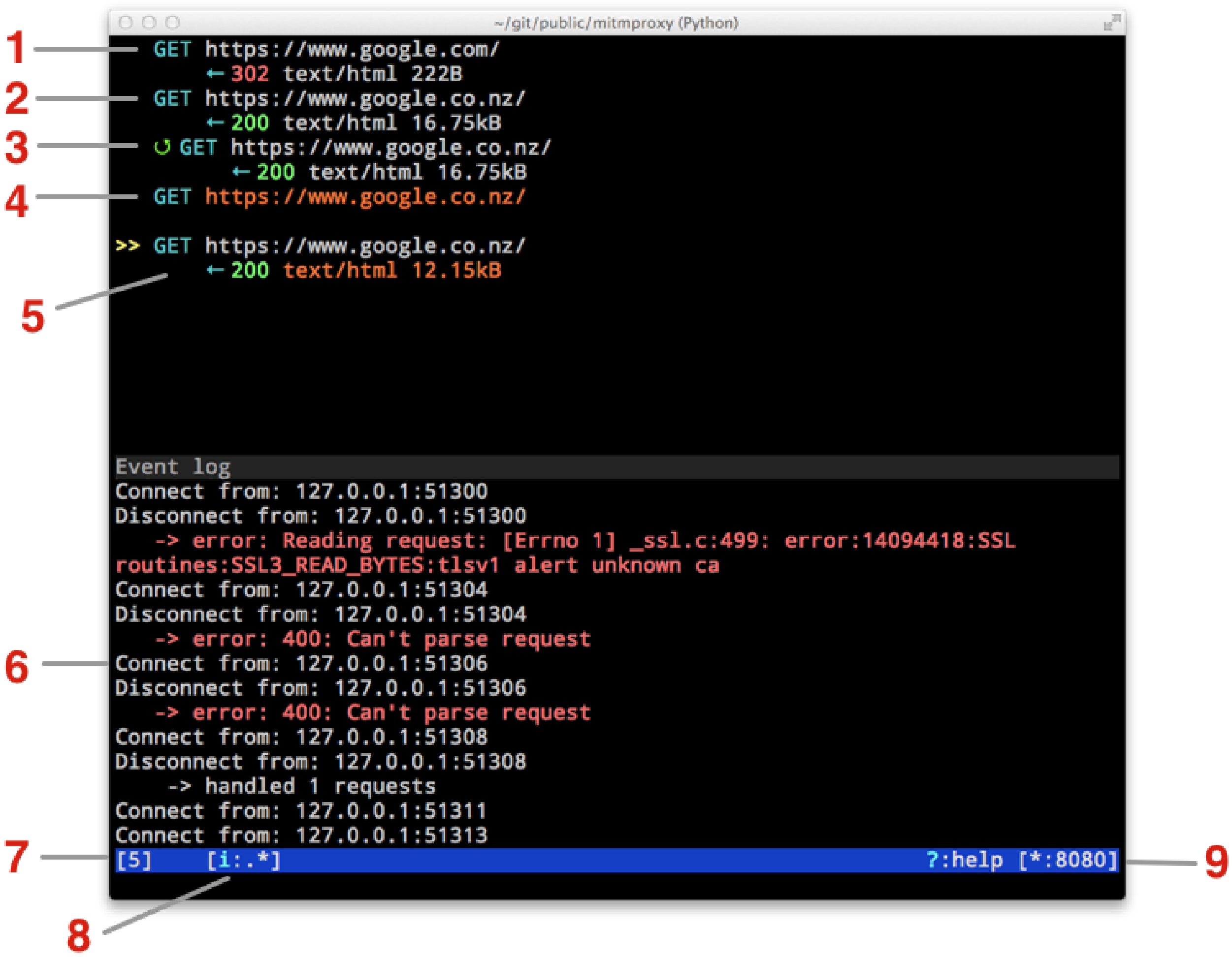
CHAPTER 6

mitmproxy

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

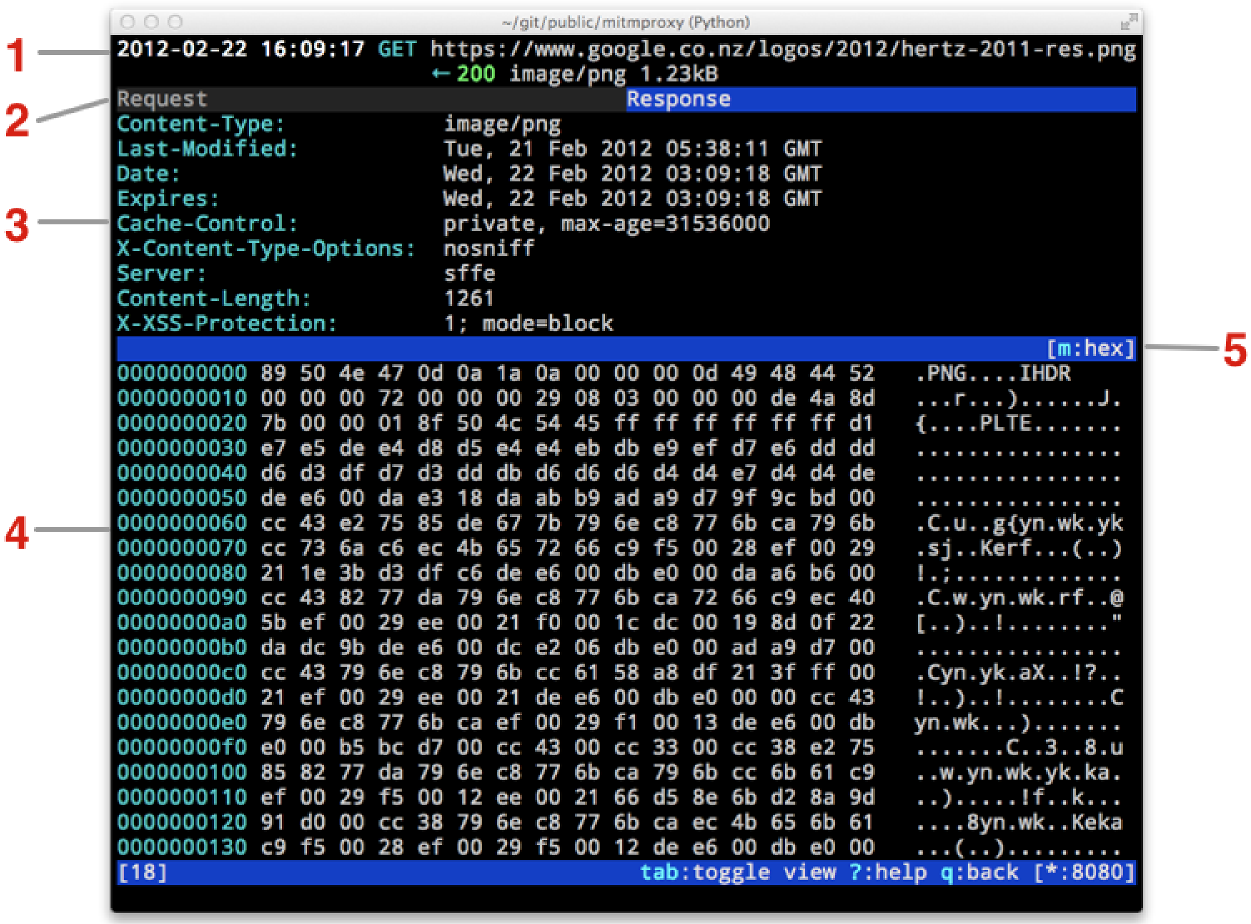
The flow list shows an index of captured flows in chronological order.



- **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.

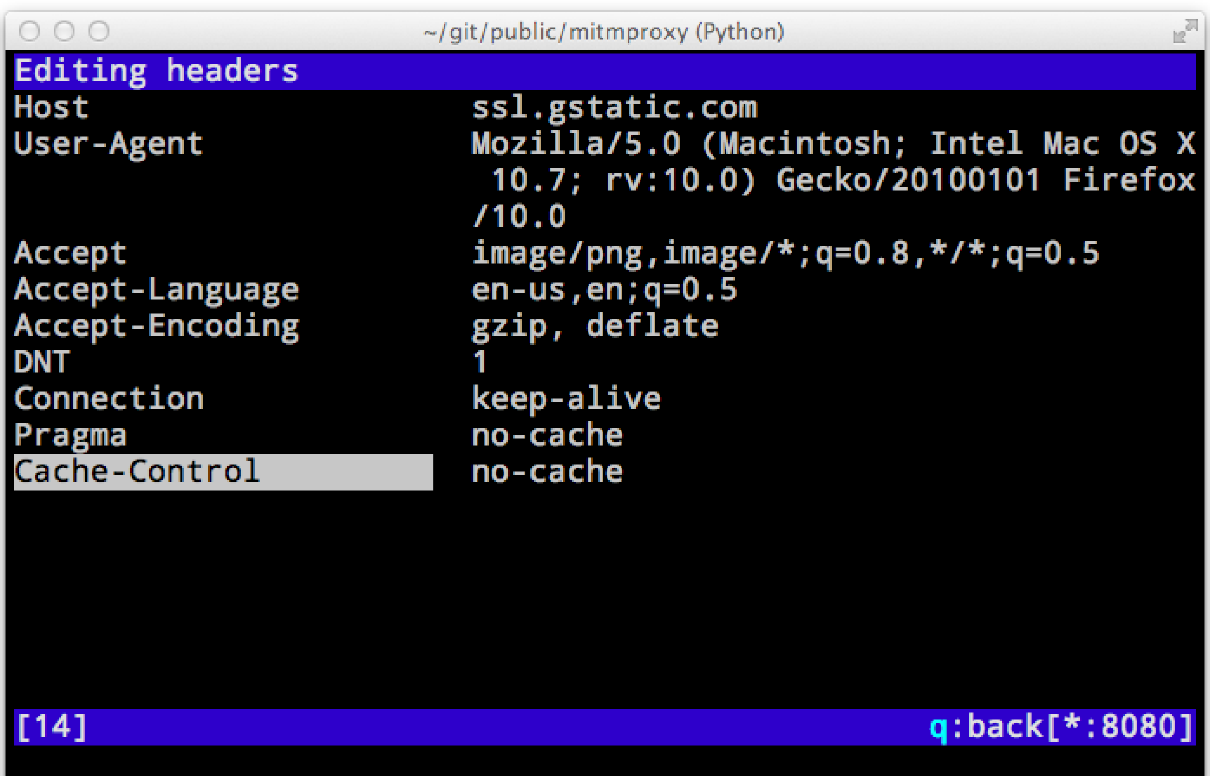
6.3 Grid Editor

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.

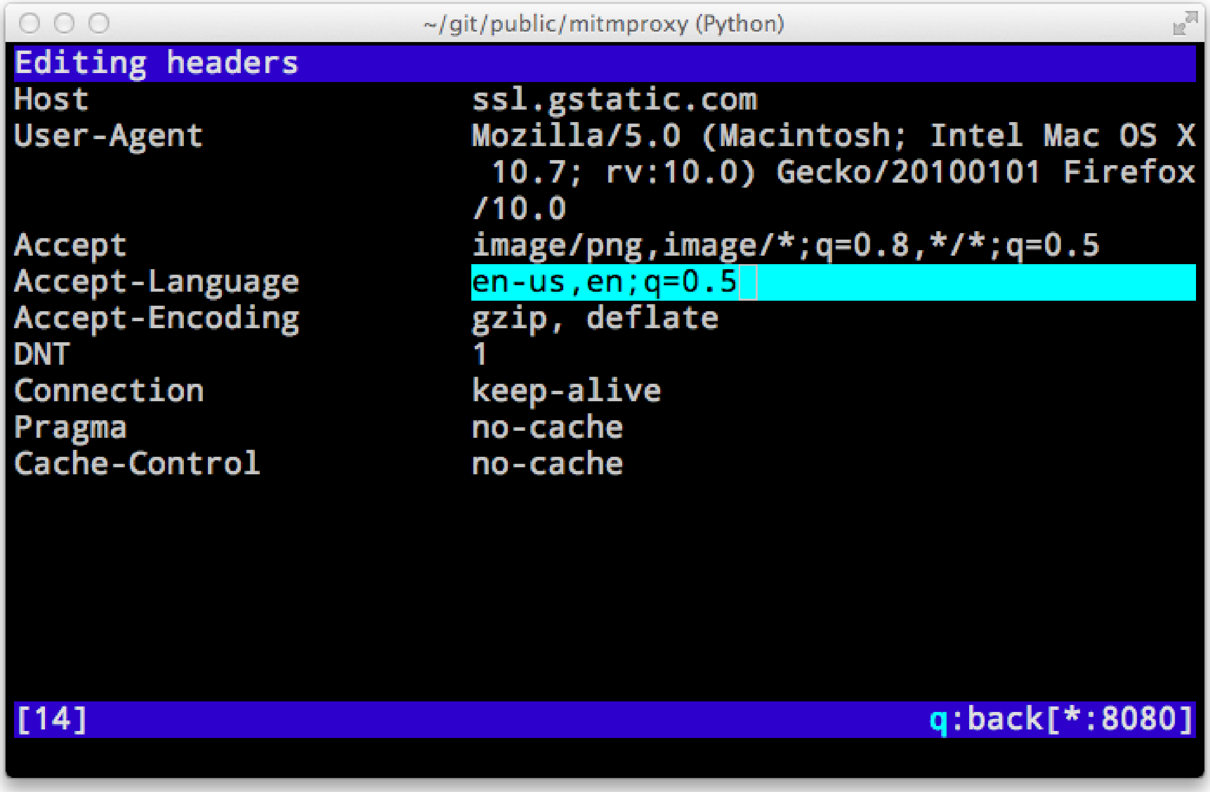
At the moment, the Grid Editor is used in four parts of mitmproxy:

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



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:

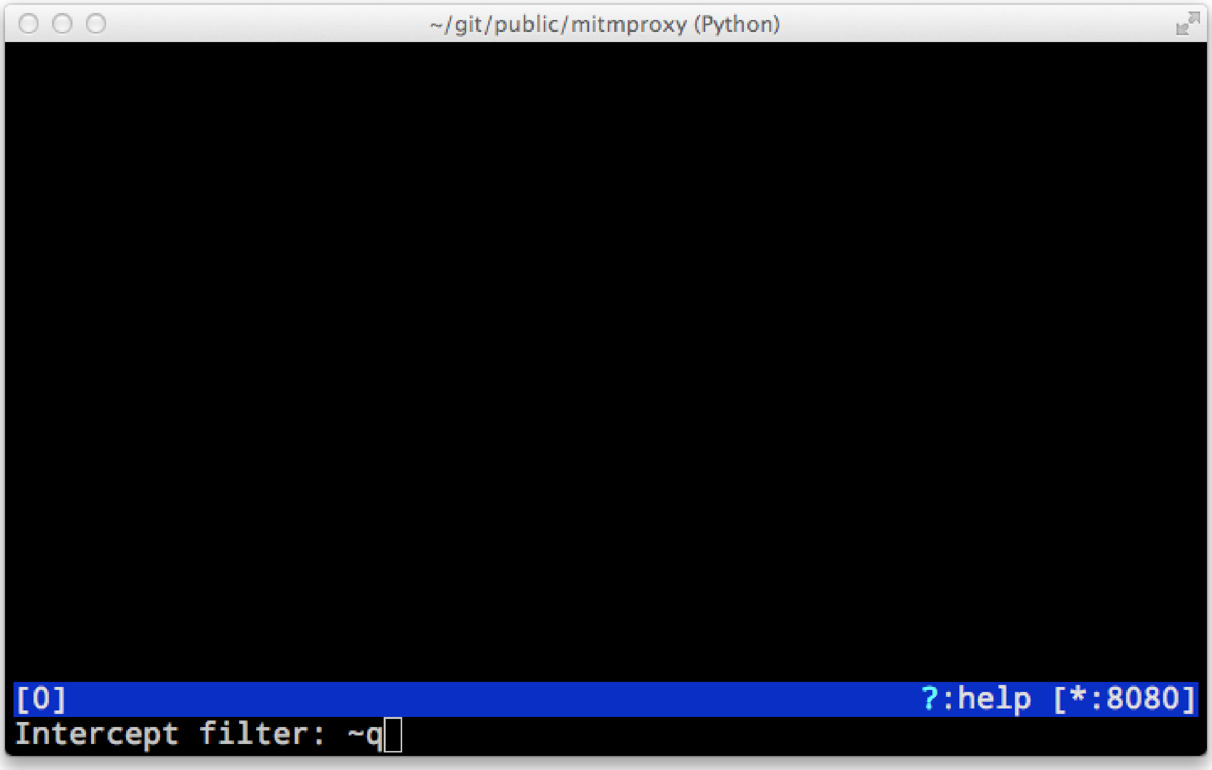


Modify the field as desired, then press escape to exit edit mode when you're done. You can also add a row (`a` key), delete a row (`d` key), spawn an external editor on a field (`e` key). Be sure to consult the context-sensitive help (`?` key) for more.

6.4 Example: Interception

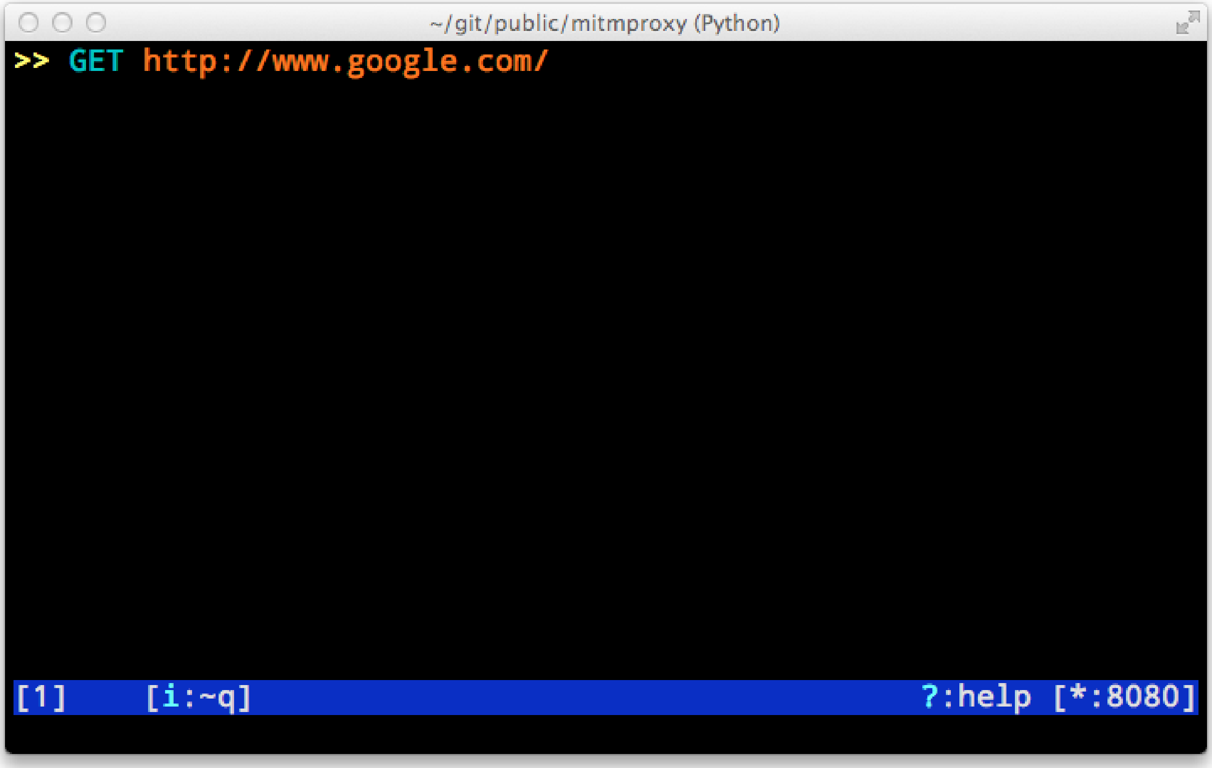
mitmproxy's interception functionality lets you pause an HTTP request or response, inspect and modify it, and then accept it to send it on to the server or client.

6.4.1 1: Set an interception pattern

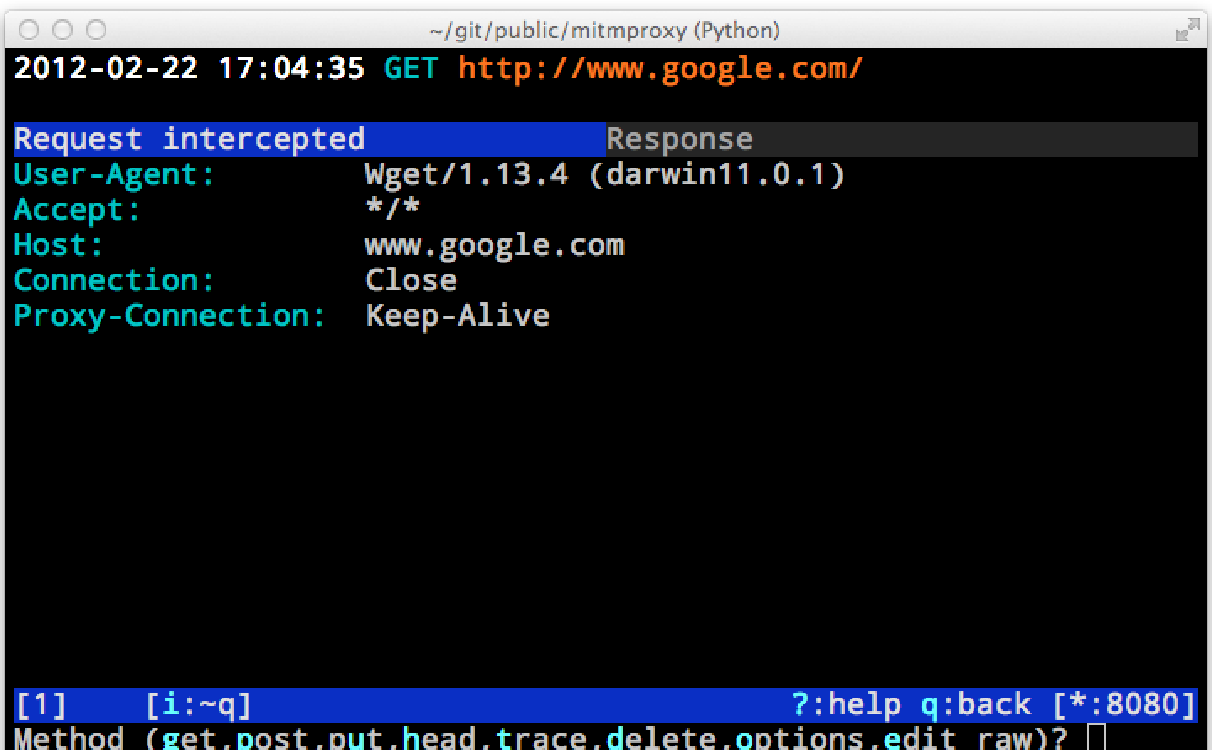


We press `i` to set an interception pattern. In this case, the `~q` filter pattern tells **mitmproxy** to intercept all requests. For complete filter syntax, see the *Filter expressions* section of the documentation, or the built-in help function in **mitmproxy**.

6.4.2 2: Intercepted connections are indicated with orange text:



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:

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".

CHAPTER 7

mitmdump

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

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

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

7.1.2 Filtering saved traffic

```
>>> 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

```
>>> 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:

```
>>> 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**.

CHAPTER 8

Configuration

Mitmproxy is configured through a set of files in the users ~/.mitmproxy directory.

mitmproxy.conf Settings for the **`mitmproxy`**. This file can contain any options supported by mitmproxy.

mitmdump.conf Settings for the **`mitmdump`**. This file can contain any options supported by mitmdump.

common.conf Settings shared between all command-line tools. Settings in this file are over-ridden by those in the tool-specific files. Only options shared by mitmproxy and mitmdump should be used in this file.

8.1 Syntax

8.1.1 Comments

```
# this is a comment
; this is also a comment (.ini style)
--- and this is a comment too (yaml style)
```

8.1.2 Key/Value pairs

- Keys and values are case-sensitive
- Whitespace is ignored
- Lists are comma-delimited, and enclosed in square brackets

```
name = value   # (.ini style)
name: value    # (yaml style)
--name value   # (command-line option style)

fruit = [apple, orange, lemon]
indexes = [1, 12, 35 , 40]
```

8.1.3 Flags

These are boolean options that take no value but true/false.

```
name = true    # (.ini style)
name
--name                 # (command-line option style)
```

8.2 Options

The options available in the config files are precisely those available as command-line flags, with the key being the option's long name. To get a complete list of these, use the `--help` option on each of the tools. Be careful to only specify common options in the **common.conf** file - unsupported options in this file will be detected as an error on startup.

8.3 Examples

8.3.1 common.conf

Note that `--port` is an option supported by all tools.

```
port = 8080
```

8.3.2 mitmproxy.conf

```
palette = light
```

CHAPTER 9

Anticache

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.

command-line	`--anticache`
mitmproxy shortcut	`o` then `a`

CHAPTER 10

Filter expressions

Many commands in **`mitmproxy`** and **`mitmdump`** take a filter expression. Filter expressions consist of the following operators:

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

- 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 &.

10.1 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")
```

CHAPTER 11

Replacements

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.

11.1 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.

11.2 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.

command-line	--replace, --replace-from-file
mitmproxy shortcut	o then R

CHAPTER 12

Client-side replay

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.

command-line	`-c path`
mitmproxy shortcut	`c`

CHAPTER 13

Server-side replay

Server-side replay lets us replay server responses from a saved HTTP conversation.

13.1 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.

13.2 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`**.

command-line	`-S path`
mitmproxy shortcut	`S`

CHAPTER 14

Set Headers

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
```

command-line	`--setheader PATTERN`
mitmproxy shortcut	`o` then `H`

CHAPTER 15

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 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.

15.1 How it works

command-line	`--ignore regex`
mitmproxy shortcut	`o` then `I`

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.

15.2 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.

[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.

15.3 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 serverconnect 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:

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

Here are some other examples for ignore patterns:

```
# 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

CHAPTER 16

Proxy Authentication

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	`--nonanonymous`, `--singleuser USER`, `--htpasswd PATH`

CHAPTER 17

Reverse Proxy

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.
```

17.1 Host Header

In reverse proxy mode, mitmproxy does not rewrite the host header. While often useful, this may lead to issues with public web servers. For example, consider the following scenario:

```
>>> mitmdump -d -R http://example.com/
>>> curl http://localhost:8080/

>> GET https://example.com/
    Host: localhost:8080
    User-Agent: curl/7.35.0
    [...]

<< 404 Not Found 345B
```

Since the Host header doesn't match "example.com", an error is returned. There are two ways to solve this:

1. Modify the hosts file of your OS so that "example.com" resolves to your proxy's IP. Then, access example.com directly. Make sure that your proxy can still resolve the original IP or specify an IP in mitmproxy.
2. Use mitmproxy's *Set Headers* feature to rewrite the host header: `--setheader :~q:Host:example.com`. However, keep in mind that absolute URLs within the returned document or HTTP redirects will cause the client application to bypass the proxy.

Response Streaming

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.

18.1 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.

18.2 Customizing Response Streaming

You can also use an *Inline Scripts* to customize exactly which responses are streamed.

Responses that should be tagged for streaming by setting their `.stream` attribute to `True`:

Listing 18.1: examples/stream.py

```
def responseheaders(flow):
    """
    Enables streaming for all responses.
    """
    flow.response.stream = True
```

18.3 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.

18.4 Modifying streamed data

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

Listing 18.2: examples/stream_modify.py

```
"""
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 occurences 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*

CHAPTER 19

SOCKS Mode

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

command-line	`--socks`

CHAPTER 20

Sticky cookies and auth

20.1 Sticky cookies

When the sticky cookie option is set, __mitmproxy__ will add the cookie most recently set by the server to any cookieless 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`

20.2 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`

TCP Proxy

WebSockets or other non-HTTP protocols are not supported by mitmproxy yet. However, 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.

21.1 How it works

command-line	`--tcp HOST`
mitmproxy shortcut	`o` then `T`

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

See also:

- *Ignore Domains*
- *Response Streaming*

CHAPTER 22

Upstream proxy mode

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]`

Upstream Certificates

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.

command-line	`--no-upstream-cert`
mitmproxy shortcut	`o` then `U`

CHAPTER 24

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.

CHAPTER 25

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:

```
>>> 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:

```
>>> 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:

```
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:

```
>>> 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

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:

```
>>> 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.

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. None the less, 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.

Inline Scripts

mitmproxy has a powerful scripting API that allows you to modify flows on-the-fly or rewrite previously saved flows locally.

The mitmproxy scripting API is event driven - a script is simply a Python module that exposes a set of event methods. Here's a complete mitmproxy script that adds a new header to every HTTP response before it is returned to the client:

Listing 27.1: examples/add_header.py

```
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 or mitmproxy as follows:

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

The new header will be added to all responses passing through the proxy.

27.1 Examples

mitmproxy comes with a variety of example inline scripts, which demonstrate many basic tasks. We encourage you to either browse them locally or on GitHub.

27.2 Events

27.2.1 Script Lifecycle Events

start (*context*)
Called once on startup, before any other events.

Parameters **argv** (*List[str]*) – The inline scripts' arguments. For example, `mitmproxy -s 'example.py --foo 42'` sets argv to `["--foo", "42"]`.

done (*context*)
Called once on script shutdown, after any other events.

27.2.2 Connection Events

clientconnect (*context*, *root_layer*)
Called when a client initiates a connection to the proxy. Note that a connection can correspond to multiple HTTP requests.

Changed in version 0.14.

Parameters root_layer (Layer) – The root layer (see *Protocols* for an explanation what the root layer is), which provides transparent access to all attributes of the `RootContext`. For example, `root_layer.client_conn.address` gives the remote address of the connecting client.

clientdisconnect (*context*, *root_layer*)
Called when a client disconnects from the proxy.

Changed in version 0.14.

Parameters root_layer (Layer) – see `clientconnect()`

serverconnect (*context*, *server_conn*)
Called before the proxy initiates a connection to the target server. Note that a connection can correspond to multiple HTTP requests.

Parameters server_conn (ServerConnection) – The server connection object. It is guaranteed to have a non-None `address` attribute.

serverdisconnect (*context*, *server_conn*)
Called when the proxy has closed the server connection.

New in version 0.14.

Parameters server_conn (ServerConnection) – see `serverconnect()`

27.2.3 HTTP Events

request (*context*, *flow*)
Called when a client request has been received. The `flow` object is guaranteed to have a non-None `request` attribute.

Parameters flow (HTTPFlow) – The flow containing the request which has been received. The object is guaranteed to have a non-None `request` attribute.

responseheaders (*context*, *flow*)
Called when the headers of a server response have been received. This will always be called before the response hook.

Parameters flow (HTTPFlow) – The flow containing the request and response. The object is guaranteed to have non-None `request` and `response` attributes. `response.content` will be `None`, as the response body has not been read yet.

response (*context*, *flow*)
Called when a server response has been received.

Parameters flow (HTTPFlow) – The flow containing the request and response. The object is guaranteed to have non-None `request` and `response` attributes. `response.body` will contain the raw response body, unless response streaming has been enabled.

error (*context*, *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.

Parameters **flow** (HTTPFlow) – The flow containing the error. It is guaranteed to have non-None `error` attribute.

27.2.4 TCP Events

tcp_message (*context*, *tcp_msg*)

> **Warning:** API is subject to change

If the proxy is in *TCP mode*, this event is called when it receives a TCP payload from the client or server.

The sender and receiver are identifiable. The message is user-modifiable.

Parameters **tcp_msg** (*TcpMessage*) – see *examples/tcp_message.py*

27.3 API

The canonical API documentation is the code, which you can browse here, locally or on GitHub. *Use the Source, Luke!*

The main classes you will deal with in writing mitmproxy scripts are:

mitmproxy.flow.FlowMaster

- The "heart" of mitmproxy, usually subclassed as `mitmproxy.dump.DumpMaster` or `mitmproxy.console.ConsoleMaster`.

ClientConnection

- Describes a client connection.

ServerConnection

- Describes a server connection.

HTTPFlow

- A collection of objects representing a single HTTP transaction.

HTTPRequest

- An HTTP request.

HTTPResponse

- An HTTP response.

Error

- A communications error.

netlib.http.Headers

- A dictionary-like object for managing HTTP headers.

netlib.certutils.SSLCert

- Exposes information SSL certificates.

27.4 Running scripts in parallel

We have a single flow primitive, so when a script is blocking, other requests are not processed. While that's usually a very desirable behaviour, blocking scripts can be run threaded by using the `mitmproxy.script.concurrent` decorator. **If your script does not block, you should avoid the overhead of the decorator.**

Listing 27.2: examples/nonblocking.py

```
import time
import mitmproxy
from mitmproxy.script import concurrent


@concurrent  # Remove this and see what happens
def request(flow):
    mitmproxy.ctx.log("handle request: %s%s" % (flow.request.host, flow.request.path))
    time.sleep(5)
    mitmproxy.ctx.log("start  request: %s%s" % (flow.request.host, flow.request.path))
```

27.5 Make scripts configurable with arguments

Sometimes, you want to pass runtime arguments to the inline script. This can be simply done by surrounding the script call with quotes, e.g. `'mitmdump -s 'script.py --foo 42'`. The arguments are then exposed in the start event:

Listing 27.3: examples/modify_response_body.py

```
# Usage: mitmdump -s "modify_response_body.py mitmproxy bananas"
# (this script works best with --anticache)
import sys


state = {}


def start():
    if len(sys.argv) != 3:
        raise ValueError('Usage: -s "modify_response_body.py old new"')
    # You may want to use Python's argparse for more sophisticated argument
    # parsing.
    state["old"], state["new"] = sys.argv[1].encode(), sys.argv[2].encode()


def response(flow):
    flow.response.content = flow.response.content.replace(
        state["old"],
        state["new"]
    )
```

27.6 Running scripts on saved flows

Sometimes, we want to run a script on `Flow` objects that are already complete. This happens when you start a script, and then load a saved set of flows from a file (see the "scripted data transformation" example *here*). It also happens when you run a one-shot script on a single flow through the `|` (pipe) shortcut in mitmproxy.

In this case, there are no client connections, and the events are run in the following order: **start**, **request**, **responseheaders**, **response**, **error**, **done**. If the flow doesn't have a **response** or **error** associated with it, the matching events will be skipped.

27.7 Spaces in the script path

By default, spaces are interpreted as a separator between the inline script and its arguments (e.g. `-s 'foo.py 42'`). Consequently, the script path needs to be wrapped in a separate pair of quotes if it contains spaces: `-s '\'./foo bar/baz.py\' 42'`.

Datastructures

class `mitmproxy.models.`**`HTTPFlow`**(*client_conn*, *server_conn*, *live=None*)
A HTTPFlow is a collection of objects representing a single HTTP transaction.

`request`
`HTTPRequest` object

`response`
`HTTPResponse` object

`error`
`Error` object

`server_conn`
`ServerConnection` object

`client_conn`
`ClientConnection` object

`intercepted`
Is this flow currently being intercepted?

`live`
Does this flow have a live client connection?

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.

`match`(*f*)
Match this flow against a compiled filter expression. Returns True if matched, False if not.

If f is a string, it will be compiled as a filter expression. If the expression is invalid, ValueError is raised.

`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.

class `mitmproxy.models.`**`HTTPRequest`**(*first_line_format*, *method*, *scheme*, *host*, *port*, *path*, *http_version*, *headers*, *content*, *timestamp_start=None*, *timestamp_end=None*, *is_replay=False*, *stickycookie=False*, *stickyauth=False*)
A mitmproxy HTTP request. This is a very thin wrapper on top of *`netlib.http.Request`* and may be removed in the future.

classmethod wrap(*request*)
Wraps an existing *netlib.http.Request*.

class mitmproxy.models.**HTTPResponse**(*http_version*, *status_code*, *reason*, *headers*, *content*, *timestamp_start=None*, *timestamp_end=None*, *is_replay=False*)
A mitmproxy HTTP response. This is a very thin wrapper on top of *netlib.http.Response* and may be removed in the future.

classmethod wrap(*response*)
Wraps an existing *netlib.http.Response*.

class netlib.http.**Request**(**args*, ***kwargs*)
An HTTP request.

Data

first_line_format
HTTP request form as defined in RFC7230.

origin-form and asterisk-form are subsumed as "relative".

method
HTTP request method, e.g. "GET".

scheme
HTTP request scheme, which should be "http" or "https".

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.

port
Target port

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

http_version
Version string, e.g. "HTTP/1.1"

headers
Message headers object

Returns netlib.http.Headers

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*

timestamp_start
First byte timestamp

timestamp_end
Last byte timestamp

Computed Properties and Convenience Methods

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

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

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.

cookies
The request cookies.

An empty *MultiDictView* object if the cookie monster ate them all.

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

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.

urlencoded_form
The URL-encoded 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.

multipart_form
The multipart form data as an *MultiDictView* object. None if the content-type indicates non-form data.

class netlib.http.**Response**(**args*, ***kwargs*)
An HTTP response.

Data

http_version
Version string, e.g. "HTTP/1.1"

status_code
HTTP Status Code, e.g. 200.

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

headers
Message headers object

Returns netlib.http.Headers

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*

timestamp_start
First byte timestamp

timestamp_end
Last byte timestamp

Computed Properties and Convenience Methods

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`

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

class `netlib.http.`**Headers**(*fields=()*, ***headers*)
Header class which allows both convenient access to individual headers as well as direct access to the underlying raw data. Provides a full dictionary interface.

Example:

```
# Create headers with keyword arguments
>>> h = Headers(host="example.com", content_type="application/xml")

# Headers mostly behave like a normal dict.
>>> h["Host"]
"example.com"

# HTTP Headers are case insensitive
>>> h["host"]
"example.com"

# Headers can also be created from a list of raw (header_name, header_value) byte tuples
>>> h = Headers([
    (b"Host",b"example.com"),
    (b"Accept",b"text/html"),
    (b"accept",b"application/xml")
```

```
])

# Multiple headers are folded into a single header as per RFC7230
>>> h["Accept"]
"text/html, application/xml"

# Setting a header removes all existing headers with the same name.
>>> h["Accept"] = "application/text"
>>> h["Accept"]
"application/text"

# bytes(h) returns a HTTP1 header block.
>>> print(bytes(h))
Host: example.com
Accept: application/text

# For full control, the raw header fields can be accessed
>>> h.fields
```

Caveats: For use with the "Set-Cookie" header, see `get_all()`.

`__init__`(*fields=()*, ***headers*)

Parameters

- **`fields`** – (optional) list of `(name, value)` header byte tuples, e.g. `[(b"Host", b"example.com")]`. All names and values must be bytes.
- **`**headers`** – Additional headers to set. Will overwrite existing values from *fields*. For convenience, underscores in header names will be transformed to dashes - this behaviour does not extend to other methods. If `**headers` contains multiple keys that have equal `.lower()` s, the behavior is undefined.

`get_all`(*name*)

Like `get()`, but does not fold multiple headers into a single one. This is useful for Set-Cookie headers, which do not support folding. See also: https://tools.ietf.org/html/rfc7230#section-3.2.2

`set_all`(*name*, *values*)

Explicitly set multiple headers for the given key. See: `get_all()`

`replace`(*pattern*, *repl*, *flags=0*)

Replaces a regular expression pattern with repl in each "name: value" header line.

Returns The number of replacements made.

class `netlib.multidict.`**`MultiDictView`**(*getter*, *setter*)

The MultiDictView provides the MultiDict interface over calculated data. The view itself contains no state - data is retrieved from the parent on request, and stored back to the parent on change.

`get_all`(*key*)

Return the list of all values for a given key. If that key is not in the MultiDict, the return value will be an empty list.

`set_all`(*key*, *values*)

Remove the old values for a key and add new ones.

`add`(*key*, *value*)

Add an additional value for the given key at the bottom.

`insert`(*index*, *key*, *value*)

Insert an additional value for the given key at the specified position.

keys (*multi=False*)
Get all keys.

Parameters **multi** (*bool*) – If True, one key per value will be returned. If False, duplicate keys will only be returned once.

values (*multi=False*)
Get all values.

Parameters **multi** (*bool*) – If True, all values will be returned. If False, only the first value per key will be returned.

items (*multi=False*)
Get all (key, value) tuples.

Parameters **multi** (*bool*) – If True, all (key, value) pairs will be returned If False, only the first (key, value) pair per unique key will be returned.

to_dict ()
Get the MultiDict as a plain Python dict. Keys with multiple values are returned as lists.

Example:

```
# Simple dict with duplicate values.
>>> d = MultiDict([("name", "value"), ("a", False), ("a", 42)])
>>> d.to_dict()
{
    "name": "value",
    "a": [False, 42]
}
```

class `mitmproxy.models.`**Error** (*msg*, *timestamp=None*)
Bases: `mitmproxy.stateobject.StateObject`

An Error.

This is distinct from an 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:

flow: Flow object msg: Message describing the error timestamp: Seconds since the epoch

class `mitmproxy.models.`**ServerConnection** (*address*, *source_address=None*)
Bases: `netlib.tcp.TCPClient`, `mitmproxy.stateobject.StateObject`

A server connection

address
Remote address. Can be both a domain or an IP address.

ip_address
Resolved remote IP address.

source_address
Local IP address

ssl_established
True if TLS is established, False otherwise

cert
The certificate presented by the remote during the TLS handshake

sni
Server Name Indication sent by the proxy during the TLS handshake

via
The underlying server connection (e.g. the connection to the upstream proxy in upstream proxy mode)

timestamp_start
Connection start timestamp

timestamp_tcp_setup
TCP ACK received timestamp

timestamp_ssl_setup
TLS established timestamp

timestamp_end
Connection end timestamp

class `mitmproxy.models.`**`ClientConnection`**(*client_connection*, *address*, *server*)
Bases: `netlib.tcp.BaseHandler`, `mitmproxy.stateobject.StateObject`

A client connection

address
Remote address

ssl_established
True if TLS is established, False otherwise

clientcert
The TLS client certificate

timestamp_start
Connection start timestamp

timestamp_ssl_setup
TLS established timestamp

timestamp_end
Connection end timestamp

CHAPTER 29

FlowMaster

Note:

We strongly encourage you to use *Inline Scripts* rather than subclassing mitmproxy's FlowMaster.

- Inline Scripts are equally powerful and provide an easier syntax.
- Most examples are written as inline scripts.
- Multiple inline scripts can be used together.
- Inline Scripts can either be executed headless with mitmdump or within the mitmproxy UI.

All of mitmproxy's basic functionality is exposed through the **mitmproxy** library. The example below shows a simple implementation of the "sticky cookie" functionality included in the interactive mitmproxy program. Traffic is monitored for `Cookie` and `Set-Cookie` headers, and requests are rewritten to include a previously seen cookie if they don't already have one. In effect, this lets you log in to a site using your browser, and then make subsequent requests using a tool like curl, which will then seem to be part of the authenticated session.

Listing 29.1: examples/stickycookies

```
#!/usr/bin/env python
"""
This example builds on mitmproxy's base proxying infrastructure to
implement functionality similar to the "sticky cookies" option.

Heads Up: In the majority of cases, you want to use inline scripts.
"""
import os
from mitmproxy import controller, proxy
from mitmproxy.proxy.server import ProxyServer


class StickyMaster(controller.Master):
    def __init__(self, server):
        controller.Master.__init__(self, server)
        self.stickyhosts = {}

    def run(self):
        try:
            return controller.Master.run(self)
        except KeyboardInterrupt:
            self.shutdown()
```

```
    @controller.handler
    def request(self, flow):
        hid = (flow.request.host, flow.request.port)
        if "cookie" in flow.request.headers:
            self.stickyhosts[hid] = flow.request.headers.get_all("cookie")
        elif hid in self.stickyhosts:
            flow.request.headers.set_all("cookie", self.stickyhosts[hid])

    @controller.handler
    def response(self, flow):
        hid = (flow.request.host, flow.request.port)
        if "set-cookie" in flow.response.headers:
            self.stickyhosts[hid] = flow.response.headers.get_all("set-cookie")


config = proxy.ProxyConfig(port=8080)
server = ProxyServer(config)
m = StickyMaster(server)
m.run()
```

Client playback: a 30 second example

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.

30.1 1. Run mitmdump to record our HTTP conversation to a file.

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

30.2 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*.

30.3 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:

```
>>> mitmdump -c wireless-login
```

30.4 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:

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

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

Setting highscores on Apple's GameCenter

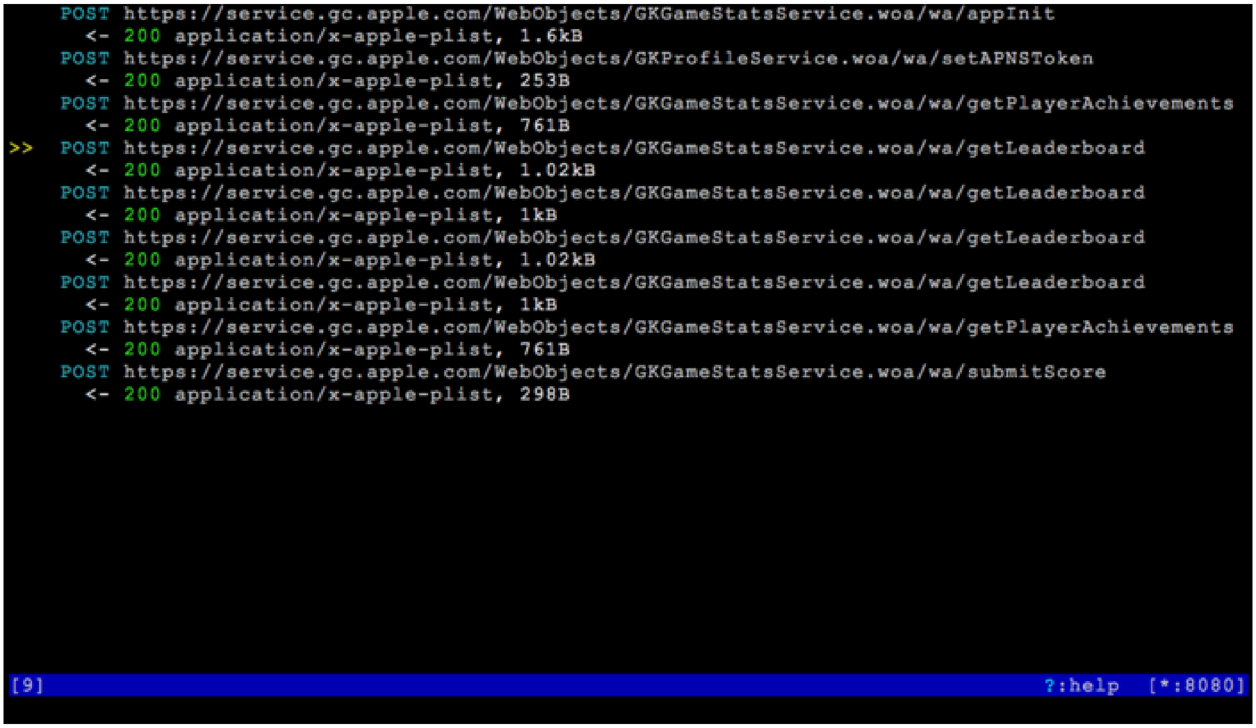
31.1 The setup

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.

31.2 Taking a look at the Game Center traffic

Lets 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:

```
<!--(block|syntax("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>55</integer>
        <key>timestamp</key>
        <integer>1363515361321</integer>
      </dict>
    </array>
  </dict>
</plist>
<!--(end)-->
```

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.

31.3 Modifying and replaying the score submission

Lets 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. Lets bump the score up to something a bit more ambitious:

```
<!--(block|syntax("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>
<!--(end)-->
```

Save the file and exit your editor.

The final step is to replay this modified request. Simply press `r` for replay.

31.4 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.

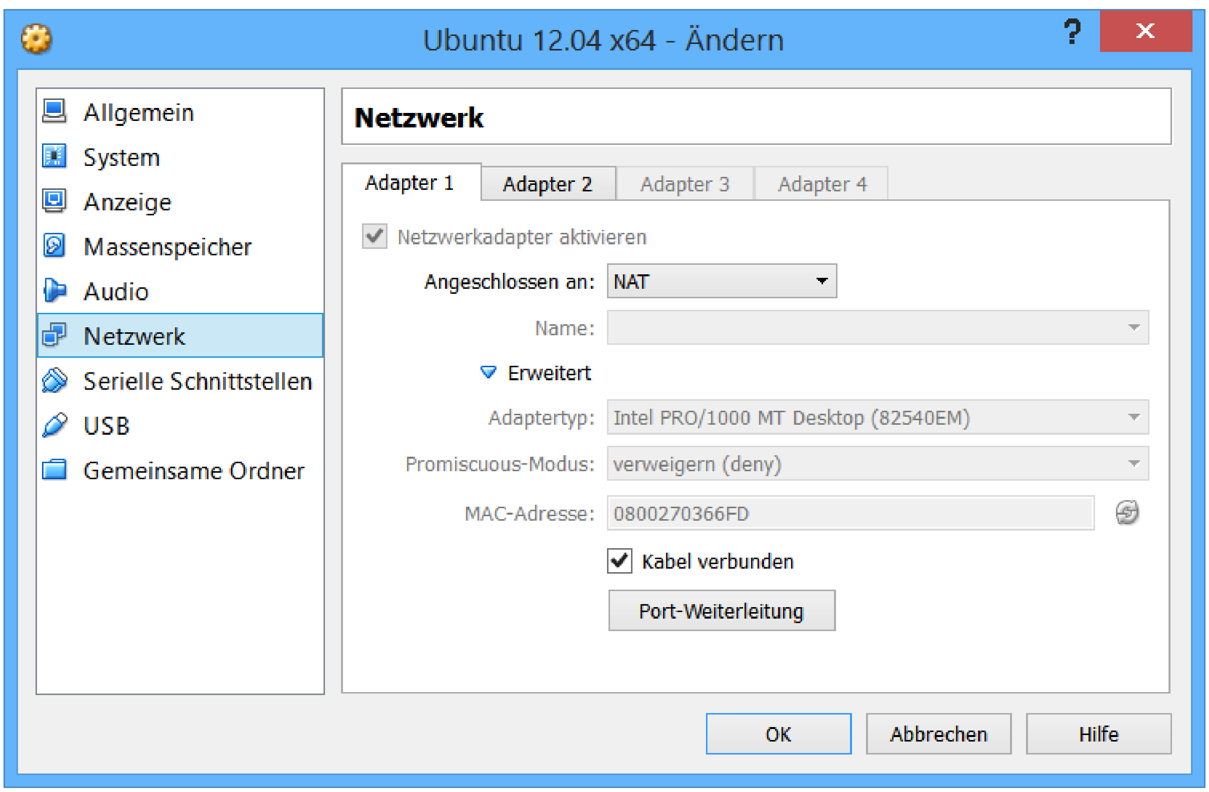
Transparently proxify virtual machines

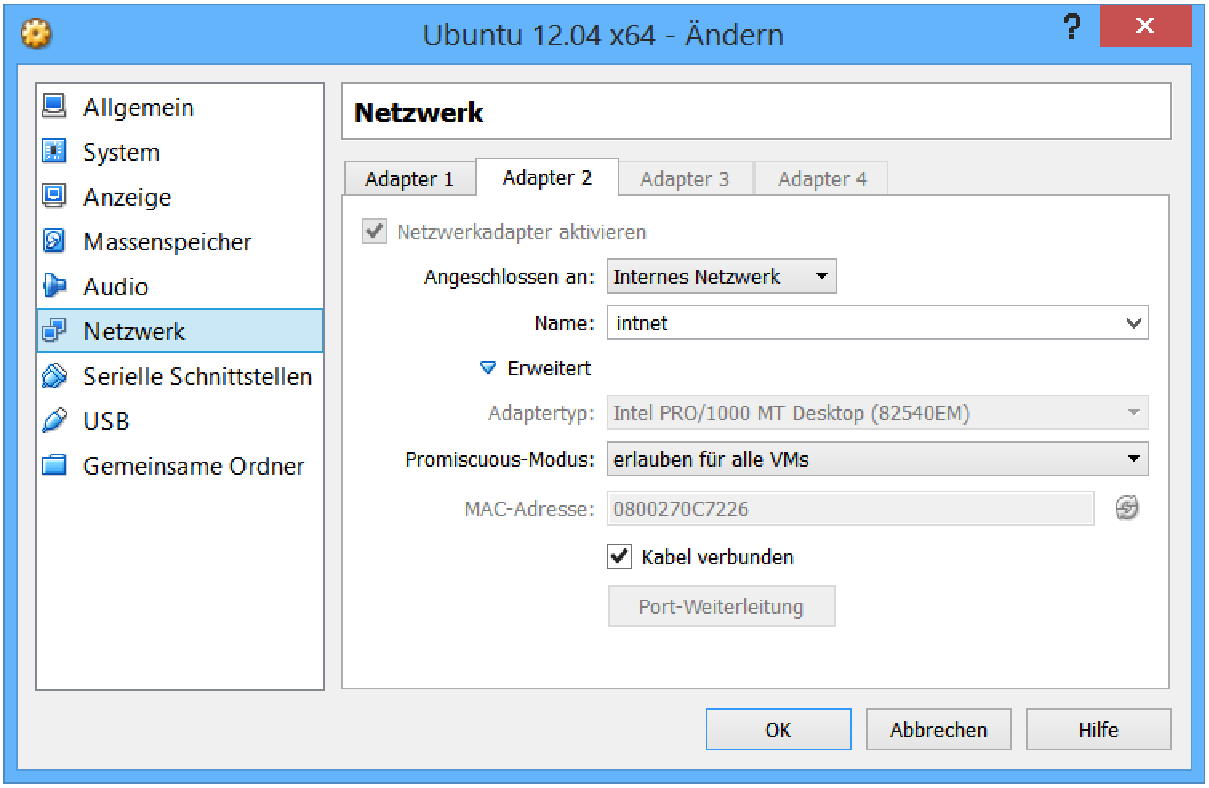
This walkthrough illustrates how to set up transparent proxying with mitmproxy. We use VirtualBox VMs with an Ubuntu proxy machine in this example, but the general *Internet <–> Proxy VM <–> (Virtual) Internal Network* setup can be applied to other setups.

32.1 1. Configure Proxy VM

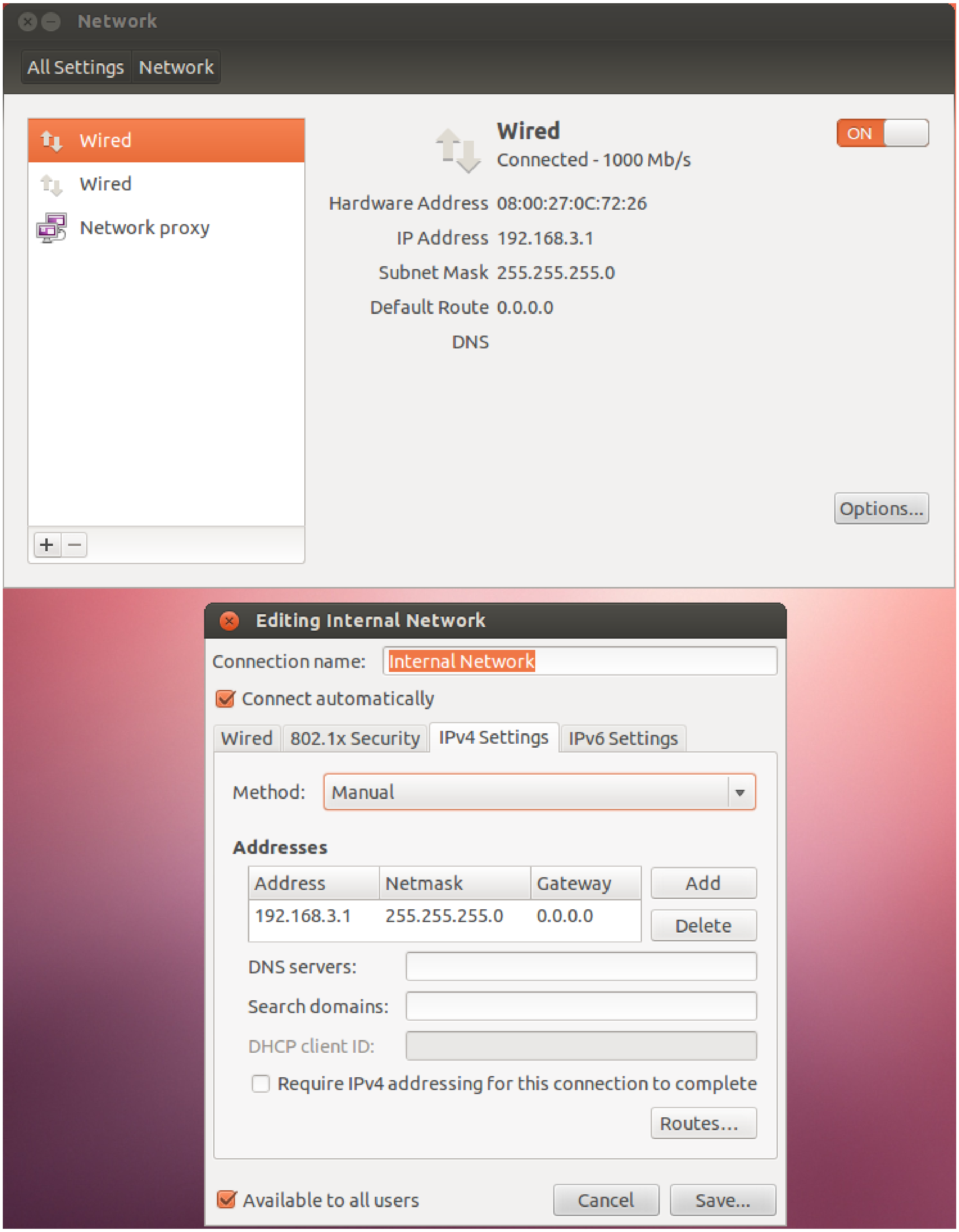
On the proxy machine, **eth0** is connected to the internet. **eth1** is connected to the internal network that will be proxified and configured to use a static ip (192.168.3.1).

32.1.1 VirtualBox configuration




Ubuntu 12.04 x64 - Ändern
Allgemein
System
Anzeige
Massenspeicher
Audio
Netzwerk
Serielle Schnittstellen
USB
Gemeinsame Ordner
Netzwerk
Adapter 1
Adapter 2
Adapter 3
Adapter 4
Netzwerkadapter aktivieren
Angeschlossen an:
Internes Netzwerk
Name:
intnet
Erweitert
Adaptertyp:
Intel PRO/1000 MT Desktop (82540EM)
Promiscuous-Modus:
erlauben für alle VMs
MAC-Adresse:
0800270C7226
Kabel verbunden
Port-Weiterleitung
OK
Abbrechen
Hilfe

32.1.2 VM Network Configuration



32.2 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 running

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

afterwards.

- Now, dnsmasq can be be installed and configured:

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

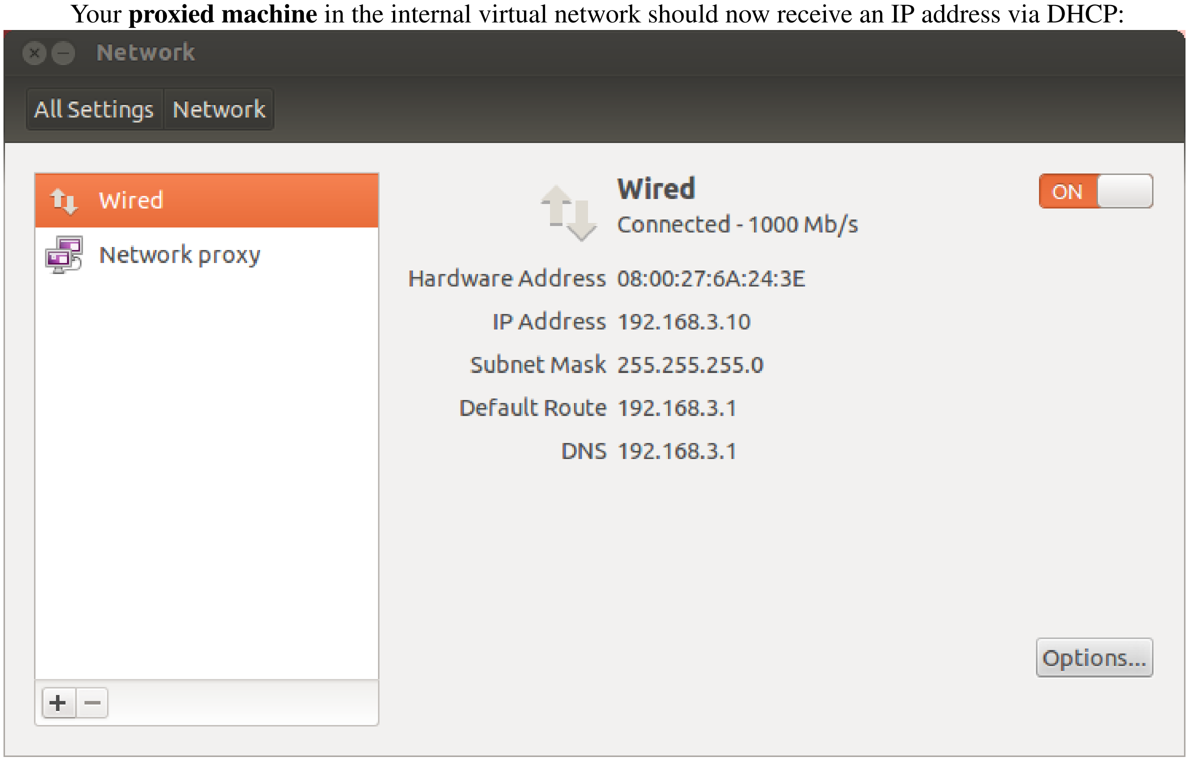
Replace **/etc/dnsmasq.conf** with the following configuration:

```
# 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:

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

Your **proxied machine** in the internal virtual network should now receive an IP address via DHCP:



32.3 3. Redirect traffic to mitmproxy

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

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

32.4 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*.

CHAPTER 33

Pathology 101

33.1 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
```

33.1.1 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.

33.1.2 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.

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

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

33.1.3 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:

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

33.1.4 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://localhost:9999/p/foo

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

33.2 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:

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

33.2.1 Getting Started

The basic pattern for pathoc commands is as follows:

> pathoc hostname request [request ...]

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

```
> 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):

```
> pathoc -s www.google.com get:/
07-06-16 12:14:56: >> 'GET':/
<< 302 Found: 262 bytes
```

33.2.2 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:

```
> 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:

```
> 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.

33.2.3 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:

```
pathoc -e -I 200 -t 2 -n 1000 localhost get:/:b@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.

33.2.4 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:

```
>>> 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:

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

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

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

33.2.5 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:

```
>>> 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\x27Content-Length\x27=\x270\x27:h\x27Conter
<< 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.

33.3 Request Examples

get:/	Get path /
get:/:b@100	100 random bytes as the body
get:/:h"Etag"="&;drop table browsers;"	Add a header
get:/:u"&;drop table browsers;"	Add a User-Agent header
get:/:b@100:dr	Drop the connection randomly
get:/:b@100,ascii:ir,@1	100 ASCII bytes as the body, and randomly inject a random byte
ws:/	Initiate a websocket handshake.

33.4 Response Examples

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

language spec

34.1 HTTP Request

method:path:[colon-separated list of features]

method	A *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 **ws** creates a valid websocket upgrade GET request, and signals to pathoc to switch to websocket recieve 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.
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.
c*VALUE*	A shortcut for setting the Content-Type header. Equivalent to `h"Content-Type"=VALUE`
u*VALUE* uSHORTCUT	Set a User-Agent header on this request. You can specify either a complete *VALUE*, or a User-Agent shortcut: **android**, **blackberry**, **bingbot**, **chrome**, **firefox**, **googlebot**, **ie9**, **ipad**, **iphone**, **safari**.
b*VALUE*	Set the body. The appropriate Content-Length header is added automatically unless the **r** flag is set.
s*VALUE*	An embedded Response specification, appended to the path of the request.
x*INTEGER*	Repeat this message N times.
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)

34.2 HTTP Response

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)

34.3 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	Pause for SECONDS seconds 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.
l*INTEGER*	Set the payload length in the frame header, regardless of the actual body length.
[-]mask	Set or un-set the mask 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.
[-]rsv2	Set or un-set the **rsv2** bit.

34.4 Data types

34.4.1 INTEGER

34.4.2 OFFSET

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

integer	An integer byte offset
r	A random location
a	The end of the message

34.4.3 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:

b	1024**0 (bytes)
k	1024**1 (kilobytes)
m	1024**2 (megabytes)
g	1024**3 (gigabytes)
t	1024**4 (terabytes)

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

```
@100m,ascii
```

Supported data types are:

<table>
<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>!”#$%&’()*+,-./:;<=>?@[\]^_‘{|}~ and space</td></tr>
<tr><td>whitespace</td><td>\t \n \x0b \x0c \r and space</td></tr>
</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:

```
>>> 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:

```
<my/path
```

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

```
<"my/path"
```

CHAPTER 35

pathod library

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 35.1: examples/pathod/libpathod_pathoc.py

```
#!/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"))
```

CHAPTER 36

pathod.test

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:

```
>>> 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.

36.1 Context Manager

Listing 36.1: examples/pathod/test_context.py

```
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
        assert r.status_code == 200
        assert len(r.content) == 100

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

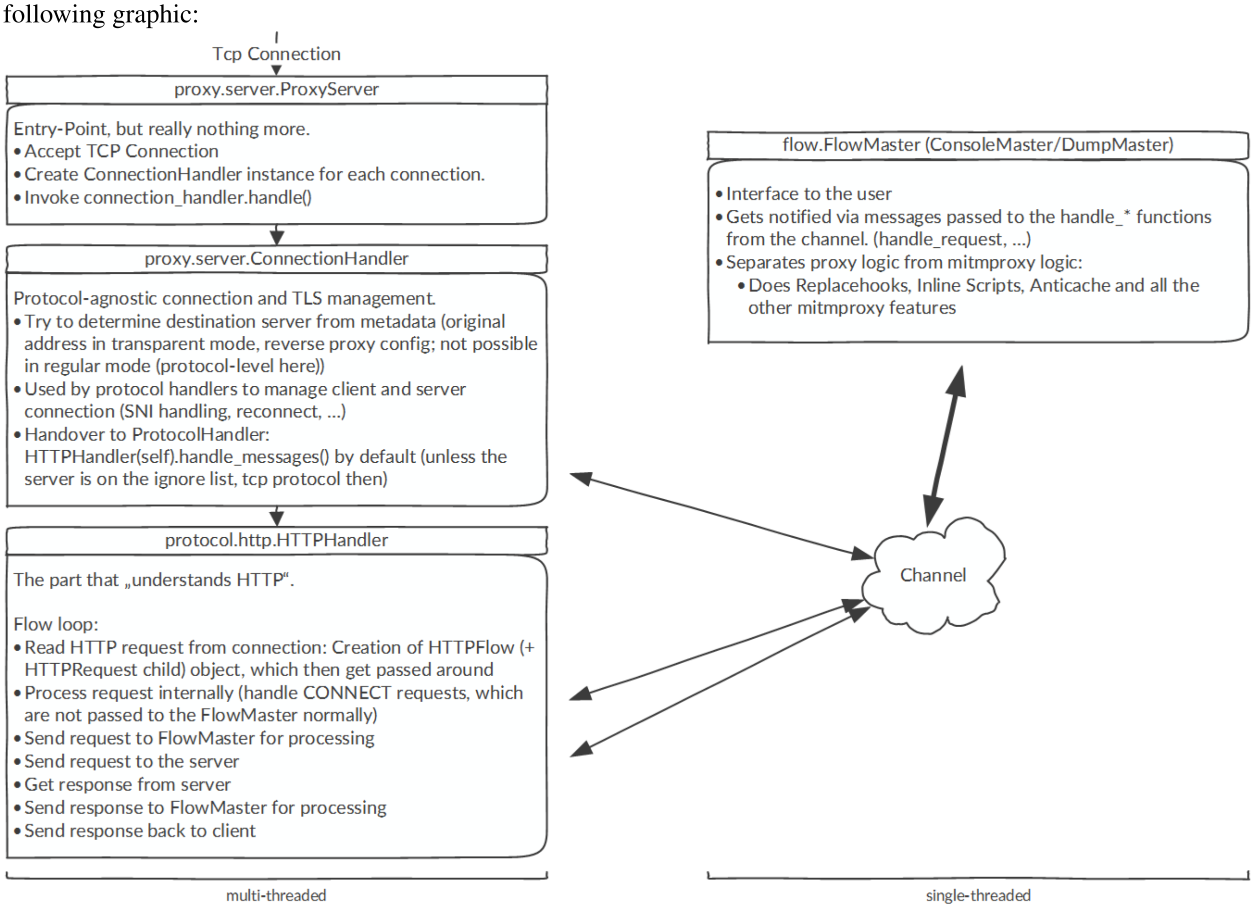
36.2 One instance per test

Listing 36.2: examples/pathod/test_setup.py

```
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"
```

Architecture

To give you a better understanding of how mitmproxy works, mitmproxy's high-level architecture is detailed in the following graphic:



`architecture.pdf`

Please don't refrain from asking any further questions on the mailing list, the Slack channel or the GitHub issue tracker.

Testing

All the mitmproxy projects strive to maintain 100% code coverage. In general, patches and pull requests will be declined unless they're accompanied by a suitable extension to the test suite.

Our tests are written for the py.test or nose test frameworks. At the point where you send your pull request, a command like this:

```
>>> py.test --cov mitmproxy --cov netlib
```

Should give output something like this:

```
> ---------- coverage: platform darwin, python 2.7.2-final-0 --
> Name                     Stmts   Miss  Cover   Missing
> ----------------------------------------------------
> mitmproxy/__init__           0      0   100%
> mitmproxy/app                4      0   100%
> mitmproxy/cmdline          100      0   100%
> mitmproxy/controller        69      0   100%
> mitmproxy/dump             150      0   100%
> mitmproxy/encoding          39      0   100%
> mitmproxy/filt             201      0   100%
> mitmproxy/flow             891      0   100%
> mitmproxy/proxy            427      0   100%
> mitmproxy/script            27      0   100%
> mitmproxy/utils            133      0   100%
> mitmproxy/version            4      0   100%
> ----------------------------------------------------
> TOTAL                     2045      0   100%
> ----------------------------------------------------
> Ran 251 tests in 11.864s
```

There are exceptions to the coverage requirement - for instance, much of the console interface code can't sensibly be unit tested. These portions are excluded from coverage analysis either in the **.coveragerc** file, or using **#pragma no-cover** directives. To keep our coverage analysis relevant, we use these measures as sparingly as possible.

CHAPTER 39

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.

Protocols

In mitmproxy, protocols are implemented as a set of layers, which are composed on top each other. The first layer is usually the proxy mode, e.g. transparent proxy or normal HTTP proxy. Next, various protocol layers are stacked on top of each other - imagine WebSockets on top of an HTTP Upgrade request. An actual mitmproxy connection may look as follows (outermost layer first):

Transparent HTTP proxy, no TLS:

- TransparentProxy
- Http1Layer
- HttpLayer

Regular proxy, CONNECT request with WebSockets over SSL:

- ReverseProxy
- Http1Layer
- HttpLayer
- TLSLayer
- WebsocketLayer (or TCPLayer)

Every layer acts as a read-only context for its inner layers (see `Layer`). To communicate with an outer layer, a layer can use functions provided in the context. The next layer is always determined by a call to `.next_layer()`, which is provided by the root context.

Another subtle design goal of this architecture is that upstream connections should be established as late as possible; this makes server replay without any outgoing connections possible.

class `mitmproxy.protocol.`**`Layer`**(*ctx*, ***mixin_args*)

Base class for all layers. All other protocol layers should inherit from this class.

`__init__`(*ctx*, ***mixin_args*)

Each layer usually passes itself to its child layers as a context. Properties of the context are transparently mapped to the layer, so that the following works:

```
root_layer = Layer(None)
root_layer.client_conn = 42
sub_layer = Layer(root_layer)
print(sub_layer.client_conn) # 42
```

The root layer is passed a `mitmproxy.proxy.RootContext` object, which provides access to `.client_conn`, `.next_layer` and other basic attributes.

Parameters **`ctx`** – The (read-only) parent layer / context.

ctx = None

The parent layer.

Type *Layer*

__call__()

Logic of the layer.

Returns Once the protocol has finished without exceptions.

Raises *ProtocolException* – if an exception occurs. No other exceptions must be raised.

__getattr__(*name*)

Attributes not present on the current layer are looked up on the context.

layers

List of all layers, including the current layer (`[self, self.ctx, self.ctx.ctx, ...]`)

class mitmproxy.protocol.**ServerConnectionMixin**(*server_address=None*)

Mixin that provides a layer with the capabilities to manage a server connection. The server address can be passed in the constructor or set by calling *set_server()*. Subclasses are responsible for calling *disconnect()* before returning.

Recommended Usage:

```
class MyLayer(Layer, ServerConnectionMixin):
    def __call__(self):
        try:
            # Do something.
        finally:
            if self.server_conn:
                self.disconnect()
```

set_server(*address*)

Sets a new server address. If there is an existing connection, it will be closed.

disconnect()

Deletes (and closes) an existing server connection. Must not be called if there is no existing connection.

connect()

Establishes a server connection. Must not be called if there is an existing connection.

Raises *ProtocolException* – if the connection could not be established.

See also:

The *Kill* exception to terminate connections.

Proxy Server

class `mitmproxy.proxy.`**`ProxyServer`**(*config*)

class `mitmproxy.proxy.`**`DummyServer`**(*config*)

class `mitmproxy.proxy.`**`ProxyConfig`**(*options*)

class `mitmproxy.proxy.`**`RootContext`**(*client_conn*, *config*, *channel*)

The outermost context provided to the root layer. As a consequence, every layer has access to methods and attributes defined here.

`client_conn`

The *`client connection`*.

`channel`

A `Channel` to communicate with the FlowMaster. Provides `.ask()` and `.tell()` methods.

`config`

The *`proxy server's configuration`*

`next_layer`(*top_layer*)

This function determines the next layer in the protocol stack.

Parameters **`top_layer`** – the current innermost layer.

Returns The next layer

`log`(*msg*, *level*, *subs=()*)

Send a log message to the master.

Exceptions

We try to be very hygienic regarding the exceptions we throw: Every Exception mitmproxy raises shall be a subclass of ProxyException.

See also: http://lucumr.pocoo.org/2014/10/16/on-error-handling/

exception `mitmproxy.exceptions.`**`ProxyException`**(*message=None*)
  Bases: `exceptions.Exception`

  Base class for all exceptions thrown by mitmproxy.

exception `mitmproxy.exceptions.`**`Kill`**(*message=None*)
  Bases: *`mitmproxy.exceptions.ProxyException`*

  Signal that both client and server connection(s) should be killed immediately.

exception `mitmproxy.exceptions.`**`ProtocolException`**(*message=None*)
  Bases: *`mitmproxy.exceptions.ProxyException`*

exception `mitmproxy.exceptions.`**`TlsProtocolException`**(*message=None*)
  Bases: *`mitmproxy.exceptions.ProtocolException`*

exception `mitmproxy.exceptions.`**`ClientHandshakeException`**(*message*, *server*)
  Bases: *`mitmproxy.exceptions.TlsProtocolException`*

exception `mitmproxy.exceptions.`**`Socks5ProtocolException`**(*message=None*)
  Bases: *`mitmproxy.exceptions.ProtocolException`*

exception `mitmproxy.exceptions.`**`HttpProtocolException`**(*message=None*)
  Bases: *`mitmproxy.exceptions.ProtocolException`*

exception `mitmproxy.exceptions.`**`Http2ProtocolException`**(*message=None*)
  Bases: *`mitmproxy.exceptions.ProtocolException`*

exception `mitmproxy.exceptions.`**`ServerException`**(*message=None*)
  Bases: *`mitmproxy.exceptions.ProxyException`*

exception `mitmproxy.exceptions.`**`ContentViewException`**(*message=None*)
  Bases: *`mitmproxy.exceptions.ProxyException`*

exception `mitmproxy.exceptions.`**`ReplayException`**(*message=None*)
  Bases: *`mitmproxy.exceptions.ProxyException`*

exception `mitmproxy.exceptions.`**`ScriptException`**(*message=None*)
  Bases: *`mitmproxy.exceptions.ProxyException`*

  classmethod **`from_exception_context`**(*cut_tb=1*)
    Must be called while the current stack handles an exception.

Parameters **`cut_tb`** – remove N frames from the stack trace to hide internal calls.

exception `mitmproxy.exceptions.`**`FlowReadException`**(*message=None*)
Bases: *`mitmproxy.exceptions.ProxyException`*

exception `mitmproxy.exceptions.`**`ControlException`**(*message=None*)
Bases: *`mitmproxy.exceptions.ProxyException`*

exception `mitmproxy.exceptions.`**`OptionsError`**
Bases: `exceptions.Exception`

exception `mitmproxy.exceptions.`**`AddonError`**
Bases: `exceptions.Exception`

Python Module Index

e

m

n

p

Symbols

A

C

D

E

F

G

H

I

U

V

W