SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br.

PODRŠKA ZA PROTOKOL HTTP/2 U PROGRAMU MITMPROXY

Emanuel Vukelić

Zagreb, lipanj 2015.

Sadržaj

1. Uvod

Internetska inačica priznatog Oxfordskog rječnika računalni, odnosno *cyber-* napad, definira kao pokušaj oštete ili uništenja računalne mreže, sistema ili web-stranice potajnim mijenjanjem informacije na njima bez dopuštenja. *Man-in-the-middle* napad (često opisan skraćenicama MITM, MitM, MIM, MiM ili MITMA) je svojevrsna personifikacija te opće definicije - naime, to je vrsta računalnog napada pri kojem napadač potajno presreće internetsku vezu između klijenta i poslužitelja, te ima mogućnost promijeniti informacije koje oni razmjenjuju bez njihovog dopuštenja.

Aludirajući na *man-in-the-middle* napad, program **mitmproxy** (engl. *man-in-the-middle proxy*) služi za presretanje, pregledavanje i manipuliranje HTTP (engl. *Hypertext Transfer Protocol*) prometom u svrhu ispitivanja sigurnosti aplikacija i traženja grešaka, te penetracijsko ispitivanje u domeni računalne sigurnosti. Nedavno je organizacija IETF (*Internet Engineering Task Force*) standardizirala novu verziju protokola HTTP s oznakom **HTTP/2**. Tema ovog završnog rada je izrada podrške za verziju HTTP/2 u programu mitmproxy.

Prvo poglavlje će opisati način rada mitmproxyja.
Drugo poglavlje će opisati glavne značajke protokola HTTP, ponuditi usporedbu verzije HTTP/1.1 sa verzijom HTTP/2, te istaknuti neke novine i poboljšanja karakteristična za verziju HTTP/2.
Treće poglavlje će pokazati kako je u programu mitmproxy implementirana podrška za protokol HTTP, te predložiti izmjene potrebne za implementaciju podrške za verziju HTTP/2.
Konačno, zadnje poglavlje će se usredotočiti na dio novoizrađene HTTP/2 podrške koji upravlja HTTP/2 zahtjevima i odgovorima u programu mitmproxy.

2. Program mitmproxy

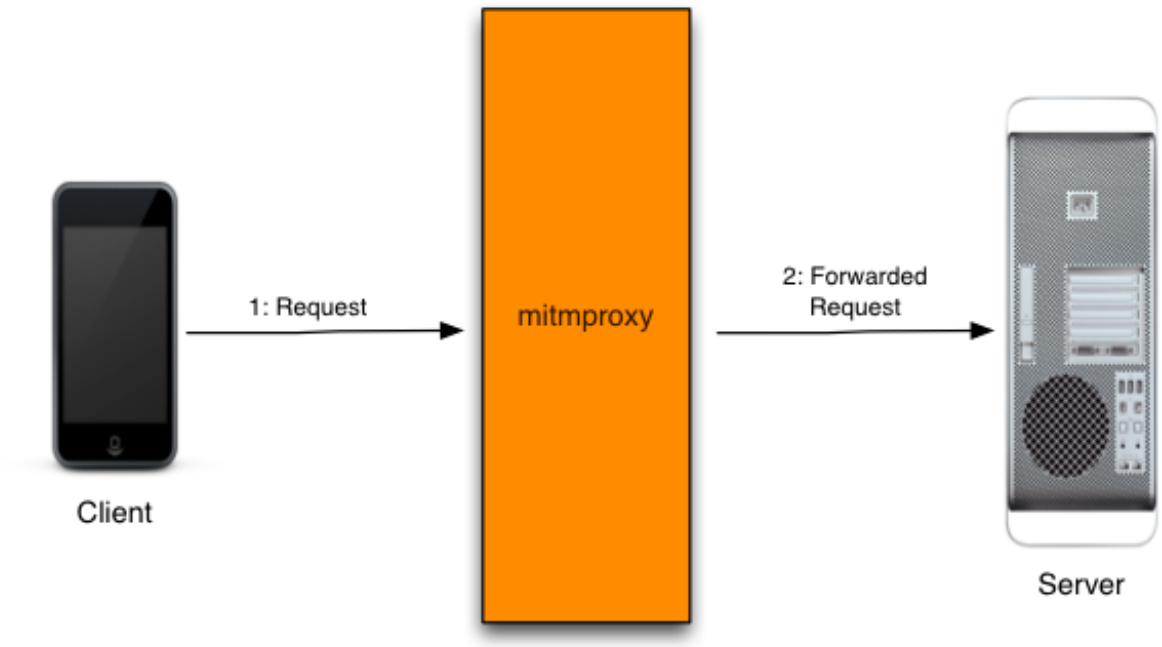
Mitmproxy je, ukratko, konzolni alat koji omogućuje pregled i izmjenu HTTP prometa. Postoji više načina ponašanja mitmproxyja, te će u naredna četiri odjeljka oni biti prikazani kako bi se mogao bolje razumjeti njegov pozadinski rad i upravljanje HTTP vezama.

2.1. Eksplicitni HTTP

Konfiguriranje klijenta za korištenje mitmproxyja kao eksplicitni *proxy* je najjednostavniji i najpouzdaniji način presretanja HTTP prometa. *Proxy* protokol je kodificiran u HTTP RFC-u (engl. *Request for Comments*), tako da je ponašanje kako klijenta tako i servera jasno definirano. Najjednostavniji način interakcije je direktno spajanje klijenta na mitmproxy i slanje zahtjeva nalik sljedećem:

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

Ovo je *proxy* GET zahtjev – prošireni oblik standardnog HTTP GET zahtjeva koji uključuje specifikaciju sheme i *hosta*, te sadrži svu informaciju potrebnu mitmproxyju da nastavi sa daljnjim radom (Slika 1.).



Slika 1. *Princip rada eksplicitnog HTTP-a*

1. Klijent se spaja na *proxy* i šalje zahtjev
2. Mitmproxy se spaja na *upstream* (prvi sljedeći) server i dalje prosljeđuje spomenuti zahtjev

2.2. Eksplicitni HTTPS

Uspostava eksplicitne HTTPS veze preko *proxyja* je podosta drugačija. Klijent se spaja na *proxy* i šalje ovakav zahtjev:

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

Konvencionalni *proxy* ne može niti pregledati niti manipulirati SSL-enkriptirani podatkovni tok (engl. *Secure Sockets Layer*), tako da CONNECT zahtjev zatraži *proxy* da otvori tunel između klijenta i servera. Proxy ovdje služi samo kao opskrbljivač – slijepo prosljeđuje podatke u oba smjera bez znanja o samom sadržaju istih. Pregovaranje SSL veze se događa preko tog tunela, a nadolazeći tok zahtjeva i odgovora je potpuno nesvjestan samog *proxyja*.

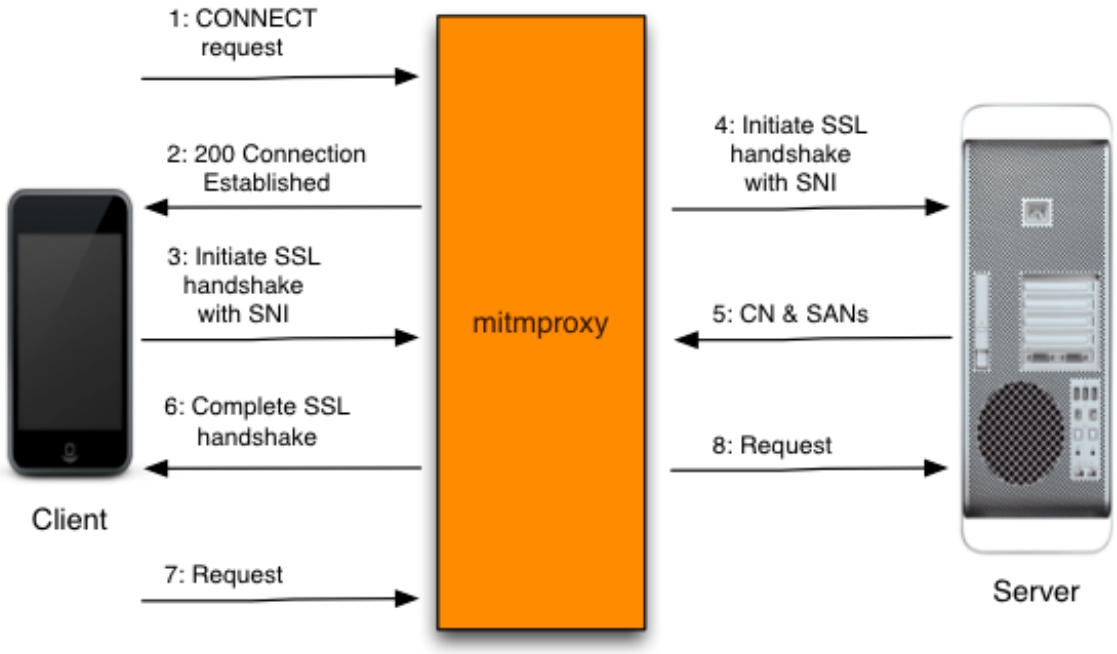
Ideja iza mitmproxyja je da se klijentu ponaša poput servera, te serveru poput klijenta, dok u isto vrijeme presrećemo i dekodiramo promet između te dvije strane. Međutim, CA (engl. *Certificate Authority*) je napravljen sa svrhom da spriječi takvo ponašanje. On to čini preko provjerene treće stranke od povjerenja koja kriptografski potpisuje serverske SSL certifikate kako bi provjerila njihovu dosljednost. Ukoliko provjera ukaže na probleme, veza se naprosto odbacuje. Mitmproxy zaobilazi ovu prepreku tako da on sam postaje provjerena treća stranka od povjerenja, odnosno *Certificate Authority*.

Da bi mitmproxy u tome uspio, potrebno je znati domensko ime koje će se koristiti u certifikatu za presretanje. Klijent će potvrditi da je certifikat uistinu za domenu s kojom ostvaruje vezu, ili ga odbaciti ako to nije slučaj. Međutim, za to se ne može samo koristiti prethodno prikazani CONNECT zahtjev. Klijent bi mogao poslati ovakav zahtjev za uspostavu veze:

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

Kako bi riješio i taj problem, mitmproxy koristi mehanizam zvan *upstream certificate sniffing*. Naime, čim se primijeti CONNECT zahtjev, pauzira se komunikacija s klijentom i stvara se veza sa serverom. Po dovršetku SSL rukovanja sa serverom, mitmproxy provjerava njegove certifikate te, koristeći *Common Name* (potpuno kvalificirano domensko ime povezano sa certifikatom) u *upstream* SSL certifikatima, stvara obmanjujući certifikat za klijenta.

Ovdje mitmproxy nailazi na još jednu u nizu komplikacija – *Common Name* ne mora nužno biti *hostname* na koji se klijent povezuje. Naime, opcionalno *Subject Alternative Name* polje u SSL certifikatu dopušta neodređeni broj alternativnih domena. Rješenje je jednostavno – kada mitmproxy koristi CN iz *upstream* certifikata, također uzima SAN-ove (engl. *Subject Alternative Name*) i pridodaje ih generiranom obmanjujućem certifikatu (Slika 2.).



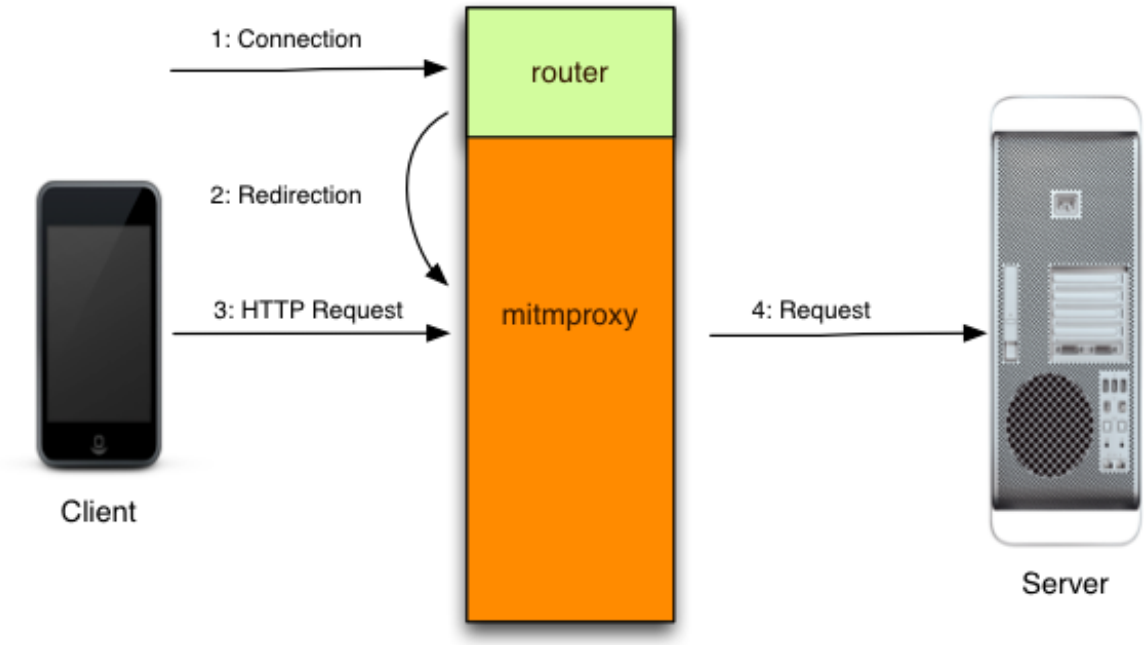
Slika 2. *Princip rada eksplicitnog HTTPS-a*

1. Klijent se spaja na mitmproxy, i šalje HTTP CONNECT zahtjev
2. Mitmproxy odgovara sa *200 Connection Established*, kao da je CONNECT tunel uspostavljen

3. Klijent je uvjeren da komunicira sa udaljenim serverom i pokreće SSL vezu. Koristi SNI (engl. *Server Name* Indication) za indikaciju *hostname*-a na koji se spaja
4. Mitmproxy se spaja na server i uspostavlja SSL vezu koristeći SNI *hostname* koji je dobio od klijenta
5. Server odgovara sa odgovarajućim SSL certifikatom, koji sadrži CN (engl. *Common Name*) i SAN vrijednosti potrebne za stvaranje certifikata za presretanje
6. Mitmproxy generira certifikat za presretanje, te nastavlja SSL-rukovanje s klijentom pauzirano u koraku 3)
7. Klijent šalje zahtjev preko uspostavljene SSL veze
8. Mitmproxy prosljeđuje zahtjev do servera preko SSL veze uspostavljene u koraku 4)

2.3. Transparentni HTTP

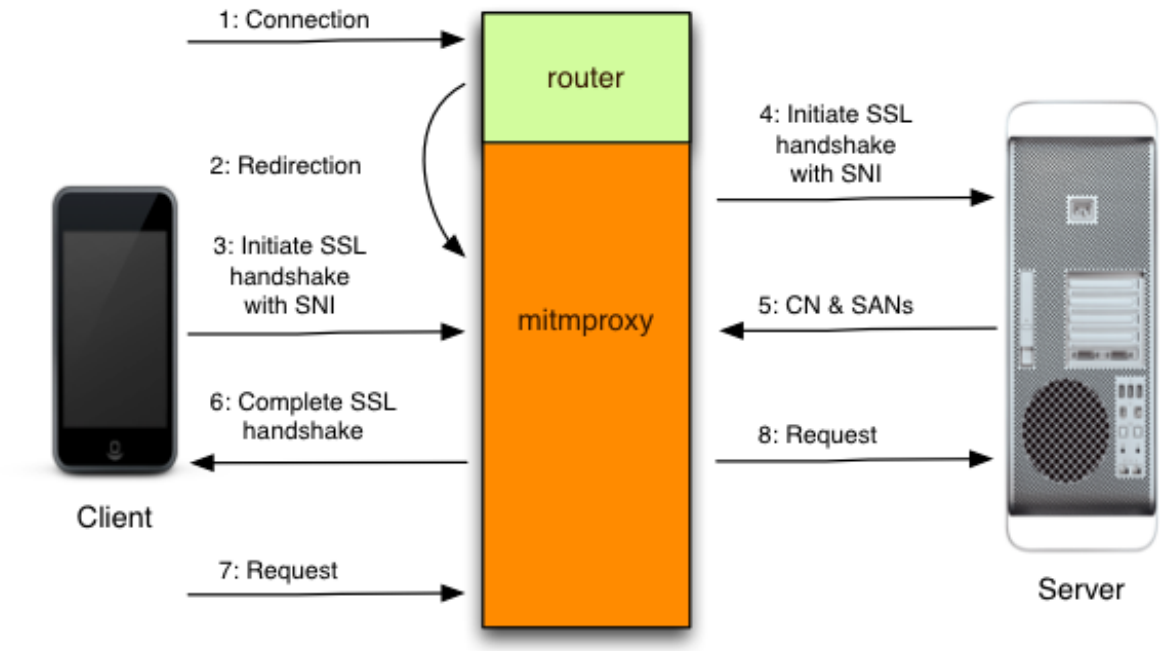
U slučaju transparentnog *proxyja*, HTTP/S veza je proslijeđena na *proxy* na mrežnom sloju, bez potrebe za konfiguracijom klijenta. Ovakav način rada mitmproxyja je poprilično jednostavan, no sama implementacija izlazi van domene ovog završnog rada, te će se situacija samo ilustrirati sljedećim prikazom (Slika 3):



Slika 3. *Princip rada transparentnog HTTP-a*

1. Klijent ostvaruje vezu sa serverom
2. Usmjeritelj preusmjerava vezu na mitmproxy, koji osluškuje na lokalnim vratima istoga *hosta*. Mitmproxy tada pomoću mehanizma preusmjeravanja određuje izvorno odredište
3. Mitmproxy čita klijentov zahtjev
4. Mitmproxy pročitani zahtjev prosljeđuje *upstream* serveru

2.4. Transparentni HTTPS



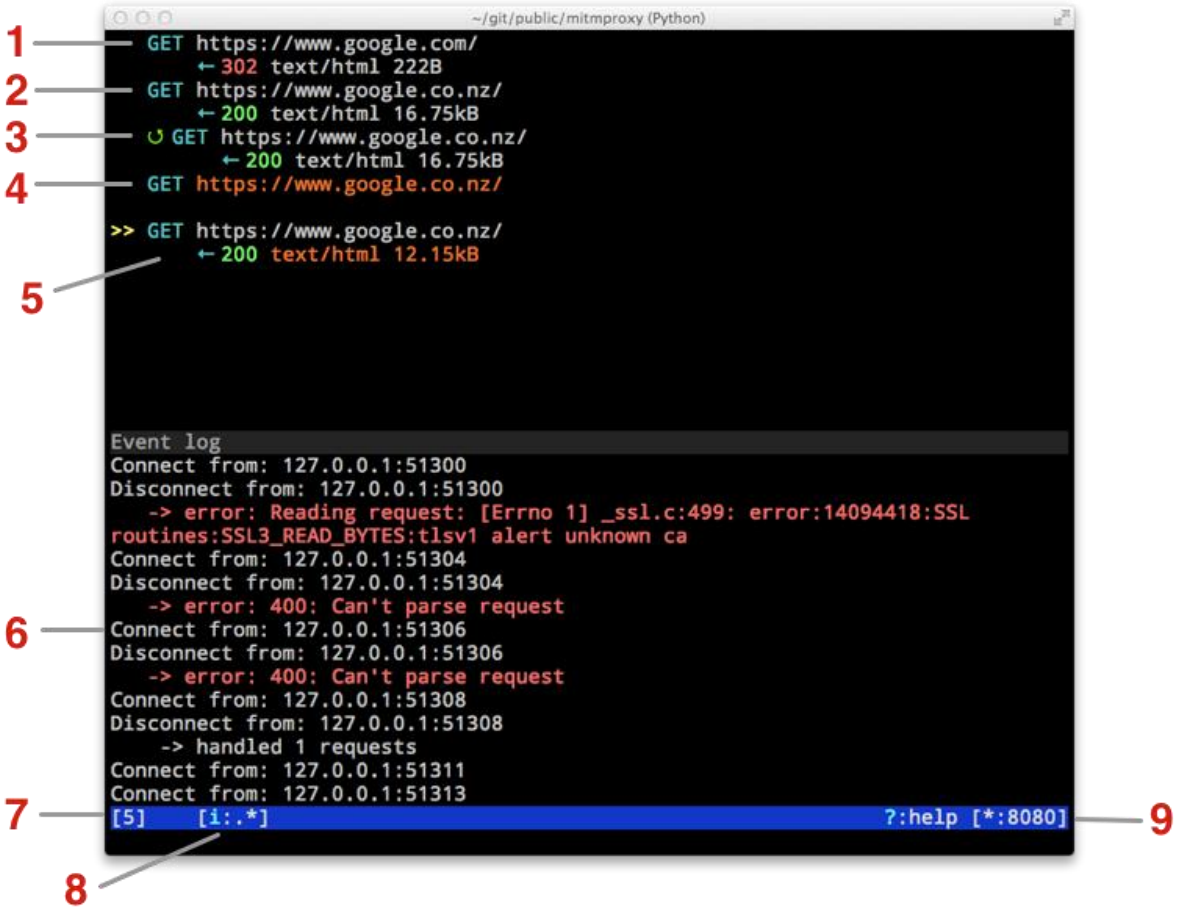
Slika 4. *Princip rada transparentnog HTTPS-a*

1. Klijent ostvaruje vezu sa serverom
2. Usmjeritelj preusmjerava vezu na mitmproxy, koji osluškuje na lokalnim vratima istoga *hosta*. Mitmproxy tada pomoću mehanizma preusmjeravanja određuje izvorno odredište
3. Klijent smatra kako komunicira sa udaljenim serverom, te pokreće SSL vezu. Pri tome koristi SNI kako bi dao do znanja na koji se *hostname* povezuje
4. Mitmproxy se povezuje sa serverom, uspostavlja SSL vezu koristeći SNI *hostname* koji je pokazao klijent
5. Server odgovara sa odgovarajućim SSL certifikatom, koji sadrži CN i SAN vrijednosti potrebne za generiranje certifikata za presretanje

6. Mitmproxy generira certifikat za presretanje, te nastavlja sa SSL-rukovanjem koje je pauzirano u koraku 3)
7. Klijent šalje zahtjev preko uspostavljene SSL veze
8. Mitmproxy prosljeđuje zahtjev serveru preko SSL veze uspostavljene u koraku 4)

2.5. Sučelja mitmproxyja

2.5.1. Lista tokova

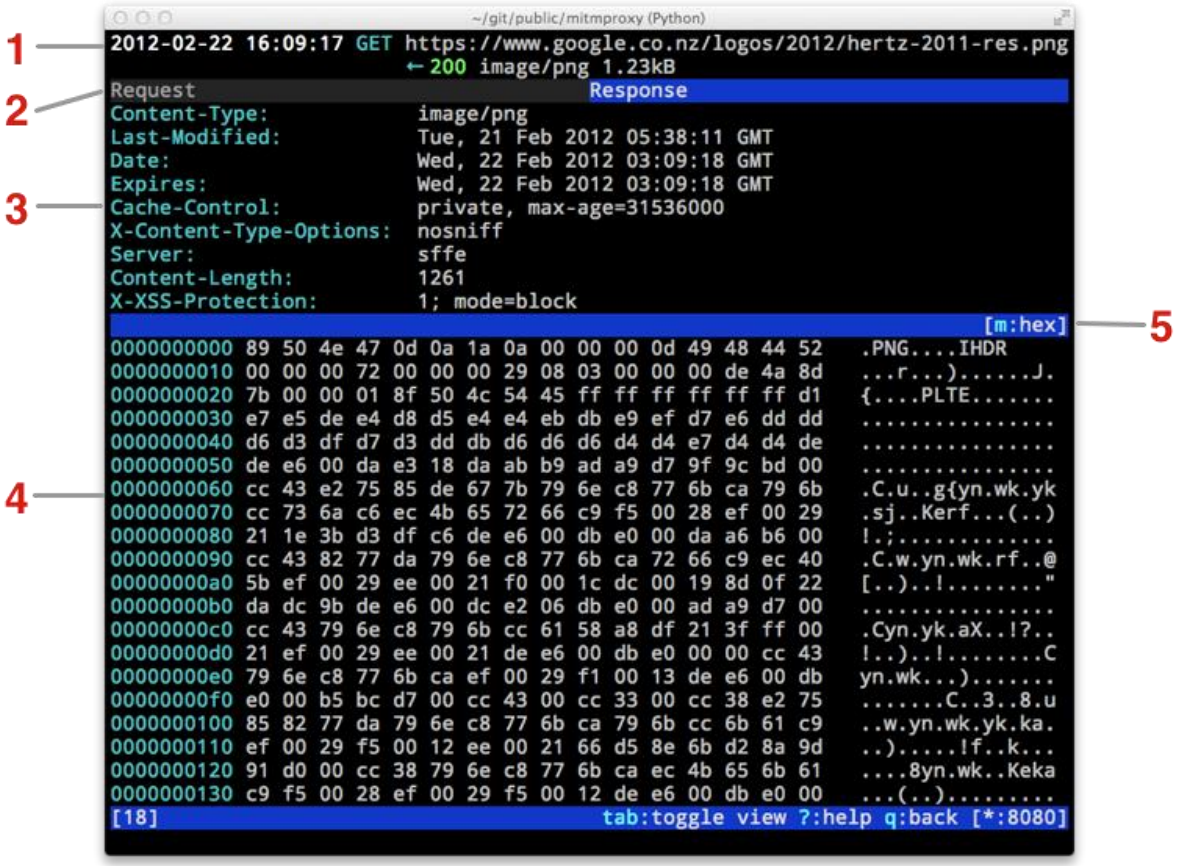


Slika 5. *Lista tokova* (Flow List)

- 1: GET zahtjev, vraća 302 *Redirect* odgovor
- 2: GET zahtjev, vraća 16.75kb text/html podataka
- 3: Ponovljeni zahtjev

- 4: Presrećeni tokovi su označeni narančastim tekstom. Korisnik može mijenjati takve tokove, i onda ih prihvatiti (koristeći *a* tipku). U ovom slučaju, presjeli smo zahtjev na njegovom putu prema serveru
- 5: Presrećemo odgovor od servera na putu prema klijentu
- 6: Zapisnik događaja može se uključiti i isključiti tipkom *e*
- 7: Broj tokova
- 8: Razne informacije o stanju mitmproxyja. U ovom slučaju, uzorak presretanja je postavljen na ".*"
- 9: Vrata na kojima mitmproxy osluškuje

2.5.2. Pregled toka



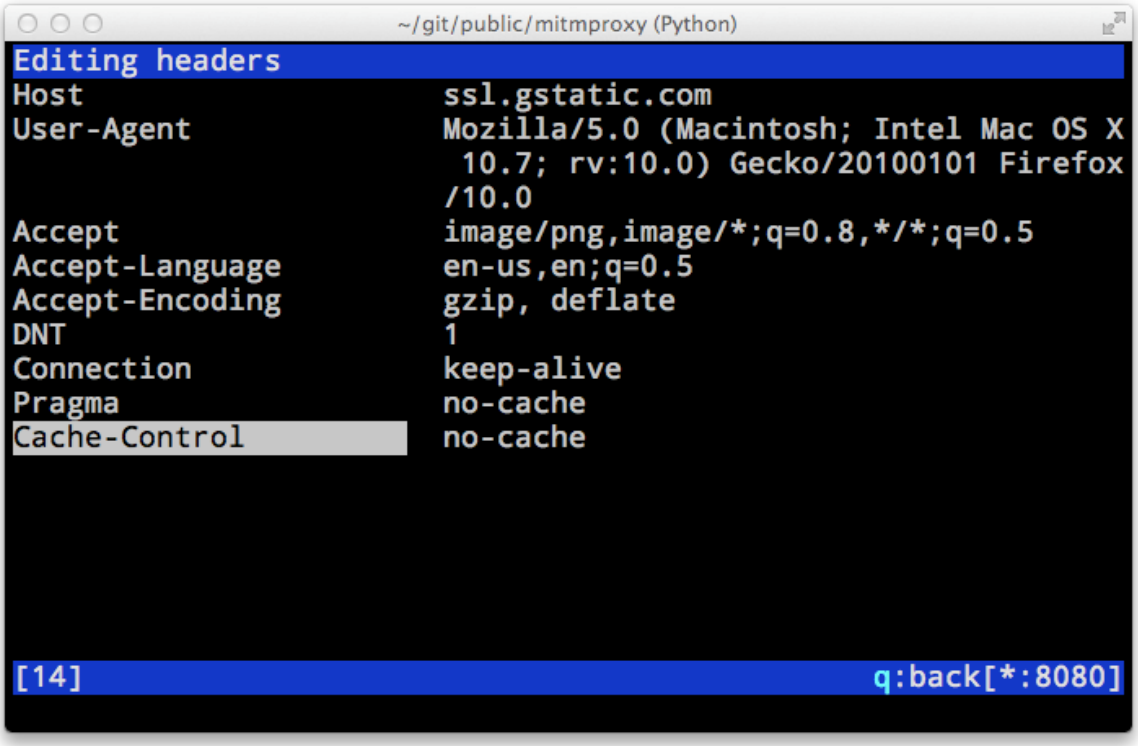
Slika 6. *Pregled toka* (Flow View)

- 1: Opis toka
- 2: Kartice zahtjeva/odgovora, pomoću kojih vidimo koji dio toka promatramo.

U našem primjeru, promatramo odgovor. Pritiskom tipke *tab* se možemo prebaciti između zahtjeva i odgovora

- **3**: *Headeri* (zaglavlja)
- **4**: *Body* (tijelo)
- **5**: Indikator načina pogleda. U našem primjeru, promatramo tijelo odgovora u ***hex*** pogledu. Ostali načini pregleda su ***pretty*** (*user-friendly* pogled) i ***raw***, koji daje točan prikaz bez ikakvih promjena. Između načina pogleda se prebacuje pritiskom tipke *m*

2.5.3. Mrežni editor



Slika 7. *Mrežni editor* (Grid Editor)

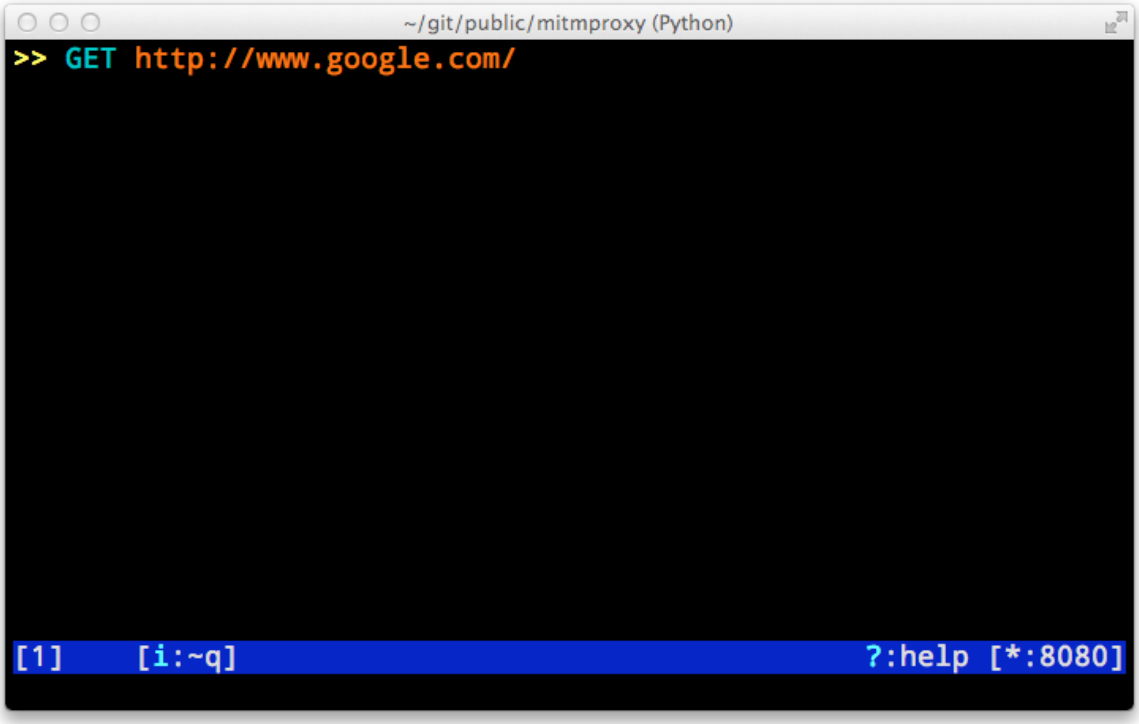
Mrežni editor se trenutno koristi u četiri dijela mitmproxyja:

- pri editiranju zaglavlja od zahtjeva ili odgovora (*e* za **e**ditiranje, zatim *h* za zaglavlja (***h**eadere*) u pregledu toka)
- pri editiranju upitnog (*query*) stringa (*e* za **e**ditiranje, zatim *q* za upit (***q**uery*) u pregledu toka)
- pri editiranju URL-enkodirane forme (engl. *Uniform Resource Locator*)

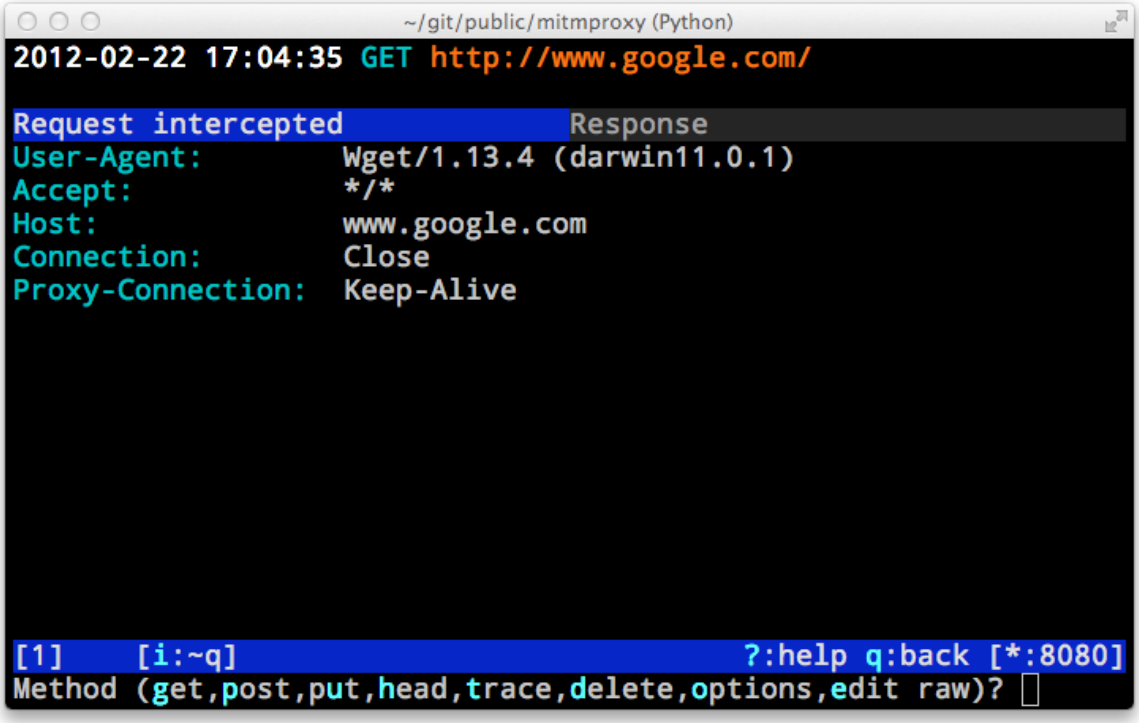
(*e* za editiranje, zatim *f* za **f**ormu u pregledu toka)

- pri editiranju zamjenskih uzoraka (*R* globalno)

2.5.4. Primjeri presretanja veze



Slika 8. *Presretanje zahtjeva*



Slika 9. *Pregled i promjena zahtjeva*

3. HTTP/1.1 vs HTTP/2

HyperText Transfer Protocol (HTTP) je internetski protokol aplikacijskog sloja koji definira format i način razmjene poruka između klijenta i poslužitelja. Protokol HTTP može biti izveden na bilo kojoj mreži temeljenoj na internetskim protokolima i može prenositi različite vrste podataka, a proširiv je i prema novim formatima podataka.
HTTP definira dvije vrste poruka – zahtjev i odgovor. Zahtjev definira metodu, resurs i protokol, dok odgovor sadrži ishod zahtjeva opisan statusnim kodom, i ovisno o vrsti zahtjeva i ishodu, sadržaj traženog resursa.
Od 1999. godine je važeća verzija protokola HTTP verzija HTTP/1.1, specificirana u RFC-u 2616.

3.1. SPDY

SPDY (*Speedy*) je otvoreni internetski protokol razvijen prvenstveno na Googleu za transport web-sadržaja. SPDY manipulira HTTP prometom, s jasnim ciljevima smanjenja vremena učitavanja stranica i poboljšanja sigurnosti. SPDY smanjeno vrijeme učitavanja stranice postiže kompresijom, multipleksiranjem te prioriterizacijom.
SPDY ne mijenja HTTP, već mijenja način kako su HTTP zahtjevi i odgovori poslani *over the wire*. SPDY efektivno služi kao tunel između protokola HTTP i HTTPS. Od 6. mjeseca 2012., grupa ljudi koja je razvijala SPDY je javno objavila da radi prema standardizaciji.
HTTP/2 je druga velika verzija internetskog protokola HTTP, a bazira se na SPDY-u. Razvila ga je Hypertext Transfer Protocol radna grupa iz IETF-a (engl. *Internet Engineering Task Force*).

3.2. HTTP/2 u odnosu na HTTP/1.1

Glavna razlika između dviju spomenutih verzija protokola HTTP je u načinu na koji se podaci grupiraju u okvire i prenose između klijenta i poslužitelja.

HTTP/2 je donio sljedeće velike promjene u odnosu na verziju 1.1:

- binarni je protokol
- omogućeno je multipleksiranje više *streamova* preko jedne veze. To znači da klijent može poslati više zahtjeva na istoj vezi, a poslužitelj može odgovarati u bilo kojem redoslijedu, odnosno kako odgovori postanu spremni
- HPACK kompresija zaglavlja koja uvelike sprječava redundantnost zaglavlja koja je sveprisutna kod ranijih verzija protokola HTTP
- *server push* – poslužitelji mogu unaprijed poslati resurse koji će se uskladištiti iako ih klijent nije eksplicitno zatražio. To omogućuje poslužitelju da „predvidi“ resurse koje će klijent zatražiti sljedeće te tako uštedi jedan obilazak (*RTT – Round Trip Time*)
- prioritizacija zahtjeva
- kontrola toka na razini veze i na razini *streama*

HTTP/2 je također unio različite vrste okvira (DATA, HEADERS, PRIORITY, RST_STREAM, SETTINGS, PUSH_PROMISE, PING, GOAWAY, WINDOW_UPDATE, CONTINUATION).
U ovom završnom radu se neće dublje razmatrati noviteti HTTP/2, s obzirom da bi takav pothvat zahtijevao više desetaka stranica, već će se usredotočiti na cilj implementacije *osnovne* podrške za HTTP/2 za program mitmproxy.

4. Protokol HTTP u mitmproxyju

Mitmproxy je napisan pomoću skriptnog jezika Python, a temelji se na knjižnici *libmproxy*. Protokol HTTP je u mitmproxyju podržan u *http.py* u mapi *libmproxy/protocol*, te sadrži klase koje opisuju HTTP zahtjeve, odgovore, i razne metode koje mitmproxy koristi za enkodiranje, dekodiranje, spremanje stanja itd. Kako bi se to ostvarilo, vrši se *import* svakojakih modula i klasa – kako onih podržanih od strane Pythona, tako ručno napisanih *handlera*.

```
from __future__ import absolute_import
import Cookie
import urllib
import urlparse
import time
import copy
from email.utils import parsedate_tz, formatdate, mktime_tz
import threading
from netlib import http, tcp, http_status, http_cookies
import netlib.utils
from netlib import odict
from .tcp import TCPHandler
from .primitives import KILL, ProtocolHandler, Flow, Error
from ..proxy.connection import ServerConnection
from .. import encoding, utils, controller, stateobject, proxy
```

Posebno su zanimljive klase *HTTPRequest* i *HTTPResponse*, koje su specijalizirani slučaj klase *HTTPMessage*. Klasa HTTPRequest tako brine o strukturi samoga zahtjeva, vrijednostima zaglavlja, stvaranju zahtjeva koji se dalje prosljeđuju poslužitelju, zamjeni određenih vrijednosti s obzirom na proslijeđeni uzorak, te mnogim drugim slučajevima. Slijedi primjer jedne takve metode koja priprema zaglavlja za slanje unutar HTTP zahtjeva *upstream* poslužitelju:

```
def _assemble_headers(self):
    headers = self.headers.copy()
    for k in self._headers_to_strip_off:
        del headers[k]
    if 'host' not in headers and self.scheme and self.host and self.port:
        headers["Host"] = [utils.hostport(self.scheme,
                                          self.host,
                                          self.port)]

    # If content is defined (i.e. not None or CONTENT_MISSING), we always
    # add a content-length header.
    if self.content or self.content == "":
        headers["Content-Length"] = [str(len(self.content))]

    return headers.format()
```

Analizirajmo ovu metodu redak po redak.

```
def _assemble_headers(self):
```

Ova linija koda je poprilično jasna – radi se o inicijalizaciji metode. Ključnom riječju **def** stvaramo metodu imena _**assemble_headers**, koja kao parametar prima instancu objekta **HTTPRequest**. Nastavimo dalje.

```
headers = self.headers.copy()
```

Stvaramo novu varijablu *headers* te u nju kopiramo vrijednosti zaglavlja HTTPRequesta proslijeđenog našoj metodi.

```
for k in self._headers_to_strip_off:
      del headers[k]
```

Ova for-petlja iterira kroz listu _**headers_to_strip_off** u kojoj se nalaze zaglavlja koja želimo maknuti prije nego proslijedimo zahtjev poslužitelju, te briše neželjena zaglavlja iz maloprije stvorene lokalne varijable *headers*.

```
if 'host' not in headers and self.scheme and self.host and self.port:
```

Ova linija kaže da ako u našim zaglavljima nema zaglavlja *host*, te ako su definirane shema, *host* i vrata, treba činiti sljedeći isječak koda:

```
headers["Host"] = [utils.hostport(self.scheme,
                                  self.host,
                                  self.port)]
```

Dakle, koristeći strukturu nalik rječniku postavljamo vrijednost zaglavlja *Host* na rezultat poziva metode **hostport** koja se nalazi u *libmproxy/utils.py*. Ta metoda vraća *host* ako se radi o (80, 'http') ili (443, 'https') paru vrijednosti (vrata, shema), a u suprotnom slučaju vraća par (host, vrata) kao dva stringa odvojena dvotočkom.

```
# If content is defined (i.e. not None or CONTENT_MISSING), we always
# add a content-length header.
if self.content or self.content == "":
            headers["Content-Length"] = [str(len(self.content))]
```

Komentari kažu da ako je sadržaj, odnosno *content* definiran, dodajemo zaglavlje *content-length*. If izjava provjerava da li postoji spomenuti sadržaj, ili je eventualno prazan string (odnosno `None`) – ukoliko je, vrijednost zaglavlja *Content-Length* postavljamo na `str(len(self.content))`, što je duljina sadržaja prebačena u string zapis. Konačno, zadnja linija metode je **return** naredba koja vraća zaglavlja u standardno formatiranom obliku, koja glasi:

```
return headers.format()
```

Kako bi izgledala odgovarajuća metoda implementirana za **HTTP/2**? Ovako.

```
def _assemble_headers(self):
        headers = []
        for k in self.keys()
            headers.append(k)
        for k in self._headers_to_strip_off:
            del headers[k]
        # Host HTTP header is no more in HTTP/2
        # We use :authority pseudo header for 'host' and 'port'


        # If content is defined (i.e. not None or CONTENT_MISSING),
we always
        # add a content-length header.
        if self.content or self.content == "":
            headers["Content-Length"] = [str(len(self.content))]


        return headers.format()
```

Treba imati na umu da ova metoda ne bi bila smještena u klasu **HTTP_2_ Request**, već klasu koja se oslanja na baznu klasu **hyper.common.headers. HTTPHeaderMap**, te bi _assemble_headers(self) učitavalo objekt spomenute *hyper* klase radi lakšeg manipuliranja elementima zaglavlja.

4.1. Ideja za HTTP/2 podršku

Iako je struktura koda mitmproxyja protkana načinom ponašanja svojstvenim HTTP/1.x verzijama, poanta protokola HTTP/2 je da podržava HTTP/1.1, te služi kao svojevrsna nadogradnja. U skladu s tim, odlučio sam promjene svesti na minimum i napraviti *http2.py* u folderu *libmproxy/protocol*. Za pomoć sam koristio *hyper* modul, koji ima implementiranu logiku HTTP/2 za Python. Iako je kod mitmproxyja dobro organiziran, podosta je međuzavisan – za potpuno transparentnu podršku protokola HTTP/2, bilo bi potrebno „pročačkati“ veći dio datoteka i već postojećih klasa (iako to tako ne bi trebalo biti). *HTTP_2_Handler*, *HTTP_2_Message, HTTP_2_Request, HTTP_2_Response* su četiri glavne klase koje bi se trebale implementirati za osnovnu HTTP/2 podršku na mitmproxyju.

Za potrebe ovog završnog rada, a i s obzirom da se često spominju zahtjevi, izdvojit ću dio implementacije **HTTP_2_Message**, koja daje temelj strukturi i ponašanju HTTP/2 poruke (bazne klase HTTP_2_Request i HTTP_2_Response).

5. Implementacija HTTP/2 podrške za mitmproxy

5.1. HTTP_2_Message

```
class HTTP_2_Message(stateobject.StateObject):
    """
    Base class for HTTP_2_Request and HTTP_2_Response
    """

    def __init__(self, headers, content, timestamp_start=None,
                 timestamp_end=None):
        self.headers = headers
        """@type: odict.ODictCaseless"""
        self.content = content
        self.timestamp_start = timestamp_start
        self.timestamp_end = timestamp_end


    _stateobject_attributes = dict(
        headers=tuple,
        content=str,
        timestamp_start=float,
        timestamp_end=float
    )
    _stateobject_long_attributes = {"content"}

    def get_state(self, short=False):
        ret = super(HTTP_2_Message, self).get_state(short)
        if short:
            if self.content:
                ret["contentLength"] = len(self.content)
            elif self.content == CONTENT_MISSING:
                ret["contentLength"] = None
            else:
                ret["contentLength"] = 0
        return ret

    def get_decoded_content(self):
        """
            Returns the decoded content based on the current Content-Encoding
            header.
            Doesn't change the message iteself or its headers.
        """
        ce = self.headers.get_first("content-encoding")
        if not self.content or ce not in encoding.ENCODINGS:
            return self.content
        return encoding.decode(ce, self.content)
```

```
def decode(self):
    """
        Decodes content based on the current Content-Encoding header, then
        removes the header. If there is no Content-Encoding header, no
        action is taken.

        Returns True if decoding succeeded, False otherwise.
    """
    ce = self.headers.get_first("content-encoding")
    if not self.content or ce not in encoding.ENCODINGS:
        return False
    data = encoding.decode(ce, self.content)
    if data is None:
        return False
    self.content = data
    del self.headers["content-encoding"]
    return True

def encode(self, e):
    """
        Encodes content with the encoding e, where e is "gzip", "deflate"
        or "identity".
    """
    # FIXME: Error if there's an existing encoding header?
    self.content = encoding.encode(e, self.content)
    self.headers["content-encoding"] = [e]

def size(self, **kwargs):
    """
        Size in bytes of a fully rendered message, including headers and
        HTTP lead-in.
    """
    hl = len(self._assemble_head(**kwargs))
    if self.content:
        return hl + len(self.content)
    else:
        return hl

def copy(self):
    c = copy.copy(self)
    c.headers = self.headers.copy()
    return c
```

```
def replace(self, pattern, repl, *args, **kwargs):
    """
        Replaces a regular expression pattern with repl in both the headers
        and the body of the message. Encoded content will be decoded
        before replacement, and re-encoded afterwards.

        Returns the number of replacements made.
    """
    with decoded(self):
        self.content, c = utils.safe_subn(
            pattern, repl, self.content, *args, **kwargs
        )
    c += self.headers.replace(pattern, repl, *args, **kwargs)
    return c

def _assemble_first_line(self):
    """
    Returns the assembled request/response line
    """
    raise NotImplementedError()  # pragma: nocover

def _assemble_headers(self):
    """
    Returns the assembled headers
    """
    raise NotImplementedError()  # pragma: nocover

def _assemble_head(self):
    """
    Returns the assembled request/response line plus headers
    """
    raise NotImplementedError()  # pragma: nocover

def assemble(self):
    """
    Returns the assembled request/response
    """
    raise NotImplementedError()  # pragma: nocover
```

5.2. def send_connect_request

```
def send_connect_request(conn, host, port, update_state=True):

    """path = :path pseudo header"""
    upstream_request = hyper.putrequest('CONNECT', path)
    """returns a stream ID for the request"""

    resp = hyper.get_response(upstream_request)
    """returns an HTTP20Response object"""

    if resp.status != 200:
        raise proxy.ProxyError(resp.status,
                               "Cannot establish SSL " +
                               "connection with upstream proxy: \r\n" +
                               str(resp.assemble()))

    """fix resp.assemble() and state.append() for HTTP/2"""

    if update_state:
        conn.state.append(("http", {
            "state": "connect",
            "host": host,
            "port": port}
        ))
    """new state for http2 connections? host+port = authority"""
    return resp
```

5.3. HTTP_2_Response

```
class HTTP_2_Response(hyper.HTTP20Response):

    def __init__(
            self,
            headers,
            h2stream,
            content,
            status):

        assert isinstance(headers, odict.ODictCaseless) or headers is None
        hyper.HTTP20Response.__init__(
            self,
            headers,
            h2stream
        )

        self.status = status

        self.is_replay = False
        self.stream = False
        """
        we need to tell apart the mitmproxy file stream from HTTP/2 stream
        hence, h2stream
        """

        _stateobject_attributes = HTTP_2_Message._stateobject_attributes.copy()
        _stateobject_attributes.update(
        status=int
        )
```

6. Zaključak

Mitmproxy je modularno razvijen program, no unatoč tome, HTTP se provlači kroz sve pore njegovih klasa, *handlera*, pamćenja stanja i slično. To daje zaključiti da iako se novi protokol može implementirati neovisno o ostalim protokolima kojima mitmproxy pruža podršku, interakcija tog protokola sa nekim stavkama mitmproxyja komplicira stvari te zahtijeva suradnju s dva programera koji su mitmproxy i napisali (te ga nastavljaju pisati) – Aldom Cortesiem i Maximilianom Hilsom.

Kroz ovaj završni rad sam se detaljno upoznao sa principom rada protokola HTTP, a posebno HTTP/2, proučio kod programa mitmproxy kako bi našao čim jednostavniji način za implementaciju danog protokola, te konačno okušao se u implementaciji istog. Činjenica je da je nova verzija protokola HTTP velik korak naprijed za komunikaciju na Internetu. Međutim, implementacija će barem u dogledno vrijeme biti spora i otežana načinom razmišljanja koji nudi HTTP/1.x, a koji se nenamjerno probio u ostale dijelove kodova programa.

7. Literatura