Software documentation for forward proxy with TTL caching, blacklisting, and upstream and downstream analysis

Student : James R

Introduction :

Proxies can serve a number of purposes to the user. In our case we were building a forward proxy with functionality for caching, blacklisting and analysis of connections. A forward proxy accepts an incoming request if it comes from an unblacklisted user to its endpoint. This sounds like an unnecessary hop for communication between the client and resource they are accessing however proxying can offer plenty of advantages. In our case these advantages include :

- Rejecting client requests. This is always useful in networking. We can stop malicious addresses from trying to access our resources which in turn can potentially save us some computational power and bandwidth in processing a response for this malicious user.
- Caching of resources. This means we localise content closer in the network to the client. So instead of fetching content from the source, we can instead send back a local copy of the content if it is up to the date to the client, saving us bandwidth because we no longer need to forward the request to the source and forward the response to the client
- Security. This point was kind of glanced over in the project description however it makes sense that if we use an intermediary machine to route requests and responses through we could be increasing our security, especially if our proxy is specialized for analysing responses for viruses etc.
- Analysis. My proxy is threaded, which means we can let each thread maintain a timer that records how long it takes to forward the request to the source of the requested content and send the reply back to the client. We can also check to see how large the replies are, cache these replies, and then analyse how much bandwidth we are saving by using cached content for replies instead of forwarding the request to the endpoint.

High Level Overview :
Protocol Design :
1) Establishing a TTL Cache : Time to live (TTL) is the time that an object is stored in a caching system before it’s deleted or refreshed. In the context of CDNs, TTL typically refers to content caching, which is the process of storing a copy of your website resources (e.g., images, prices, text) on CDN proxies to improve page load speed and reduce origin server bandwidth consumption. Now i did have a problem with the TTL Cache that you can import in python. Its very restrictive and only allows you to assign a fixed lifespan to all cached content. I extended this functionality to allow myself to set individual lifespans to each piece of content i would cache, according to **Access-Control-Max-Age** in the headers of content.

2) Establishing proxy endpoint : The proxy needs a way of listening for connections from users. We do this with sockets. We first find a port to listen on. If we are running the proxy in HTTPS mode (where we want to send a request to the proxy over HTTPS channel) we will use port 443 as it is a HTTPS port. Else we can use majority of the remaining ports. We also need an IP address for this port to bind too. In our case we will be using localhost. Now that we have a socket listening on a port on localhost (our machine) we have an endpoint to send requests from our browser or management terminal to. I have made sure that this socket is **a ssl socket so that the connection to the proxy is using https. We can see that our user connected to the proxy over HTTPS in the terminal from :**

```
ssl loaded!! certfile= ./ssl/certificate.pem keyfile= ./ssl/key.pem
```

And in the browser we can see our connection with the proxy is HTTPS secure by :

```
localhost:8086/http://www.py4inf.com/code/romeo.txt
```

I have included a way of running the proxy so that connection can be over HTTP instead because it may be awkward for people to load my certificate into their browsers authorities.

3) Handling incoming requests : We receive requests through the socket the proxy is listening on. We accept this connection like so :

```
conn, addr = s.accept() #Accept connection from client browser
```

The return value is a pair (conn, address) where conn is a new socket object usable to send and receive data on the connection, and address is the address bound to the socket on the other end of the connection. If we are running the proxy in HTTPS mode (where we want to send our request to the proxy over HTTPS) we need to wrap the proxies connection to the browser in an SSL socket as such :

```
conn = ssl_wrap_socket(conn, ssl_version, keyfile, certfile, ciphers)
```

If we want the browser to communicate to the proxy over HTTP we can just use the socket as it is.

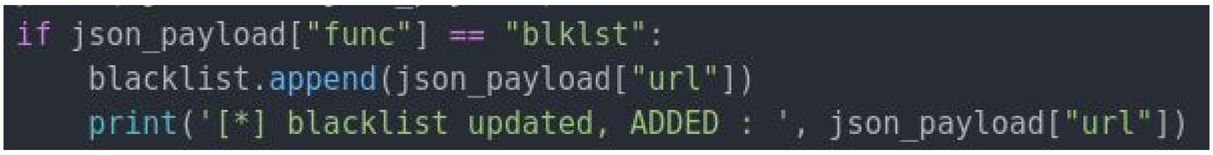
4) Proxy receiving incoming requests from client / browser : Data is transferred in a stream of bytes, not sent in packets when sent through sockets. This is very important and requires attention for sending information through sockets. We can set the buffer size to whatever we like, however it is usually set to 1024, or any value between 8k and 12k. The importance of buffer size with sockets will be explained further below. This is how we receive our data : `data = conn.recv(buffer_size) #Recieve client data`

5) Receiving client (browser) request : This part in the request handling is rudimentary. The only part of importance i found is reformatting our get request to align with HTTP/S standards as follows `b'GET / HTTP/1.1\r\nHost:www.example.com\n\n'` (basically formatting the resource and endpoint and creating a byte array out of the result)

We need to check if the request we have received if from the proxy manager or if its a normal client request.

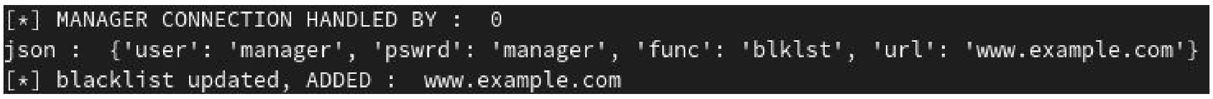
```
if '"user": "manager", "pswrd": "manager"' in data.decode() :
    start_new_thread(handleManagerReq, (decodeData,threadID))
    threadID += 1
```

If we have received a request from the manager we need to handle this request differently. We first determine what type of request the manager is sending to the proxy like so :



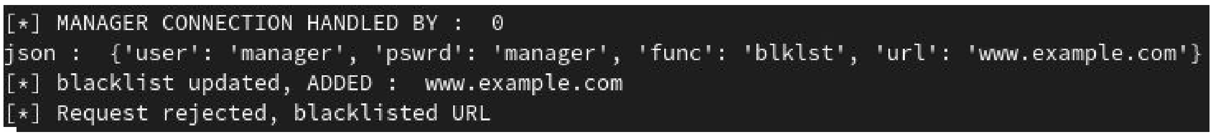
```
if json_payload["func"] == "blklst":
    blacklist.append(json_payload["url"])
    print('[*] blacklist updated, ADDED : ', json_payload["url"])
```

This says that if the “func” (function) that the manager wants is “blklst” (blacklist) then we add the url to the blocked URL’s. Lets run our proxy and send it a blacklist request from the manager :



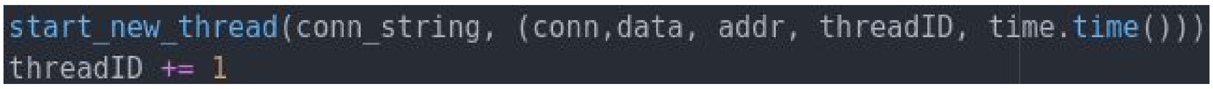
```
[*] MANAGER CONNECTION HANDLED BY :  0
json :  {'user': 'manager', 'pswrd': 'manager', 'func': 'blklst', 'url': 'www.example.com'}
[*] blacklist updated, ADDED :  www.example.com
```

Here we can see the manager request handled by thread 0. We see the func = ‘blklst’ and we see www.example.com is being blacklisted. Now lets send a request to the blacklisted site and see what happens :



```
[*] MANAGER CONNECTION HANDLED BY :  0
json :  {'user': 'manager', 'pswrd': 'manager', 'func': 'blklst', 'url': 'www.example.com'}
[*] blacklist updated, ADDED :  www.example.com
[*] Request rejected, blacklisted URL
```

6) Creating a threaded proxy to handle multiple clients : Pythons _thread module is really good for this because we can assign a thread to run a function and when the function exits the thread will die. So we can make a function for handling regular requests and one for handling manager requests. Whenever we recieve a request we offload it to a new thread and keep the main socket the proxy is connected to running so it can still handled more client requests as such :

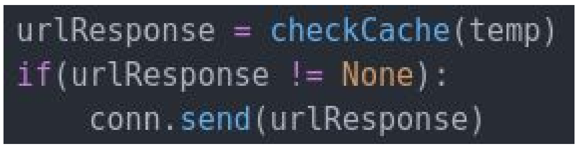


```
start_new_thread(conn_string, (conn,data, addr, threadID, time.time()))
threadID += 1
```

Or for manager requests we create a new thread and pass it the function as such :

7) Forming our request : the proxy needs to form our request for the endpoint. The result will differ depending on if the request from the browser is trying to access HTTP or HTTPS.

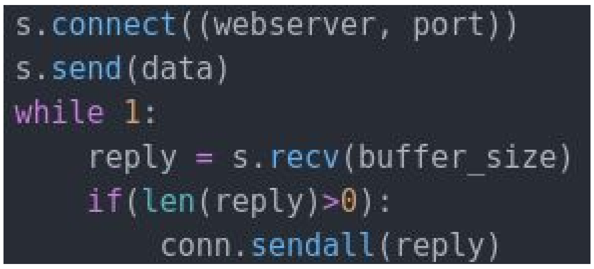
8) Checking the cache : This is done before forwarding http requests to the requested server. If we have a valid value in the cache associated with the request we send that back to the browser / client like this :



```
urlResponse = checkCache(temp)
if(urlResponse != None):
    conn.send(urlResponse)
```

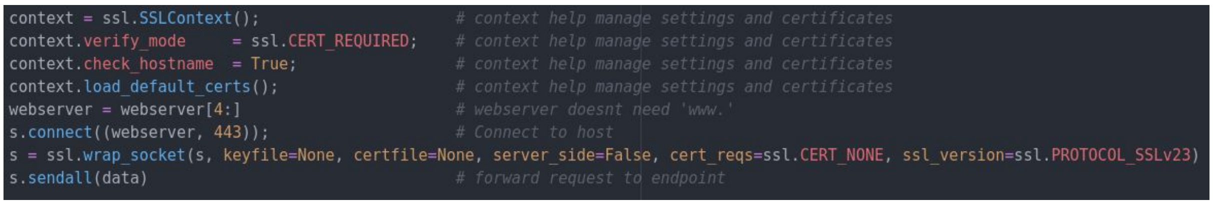
If we don't have a cached response for this request we connect with the requested server, send our request, receive back our data and forward that data back the the client/browser in streams like so :

8) Connecting the proxy to the requested server : If the proxy is forwarding a request to a HTTP resource / server we can simple connect the proxy socket to the web server like so :



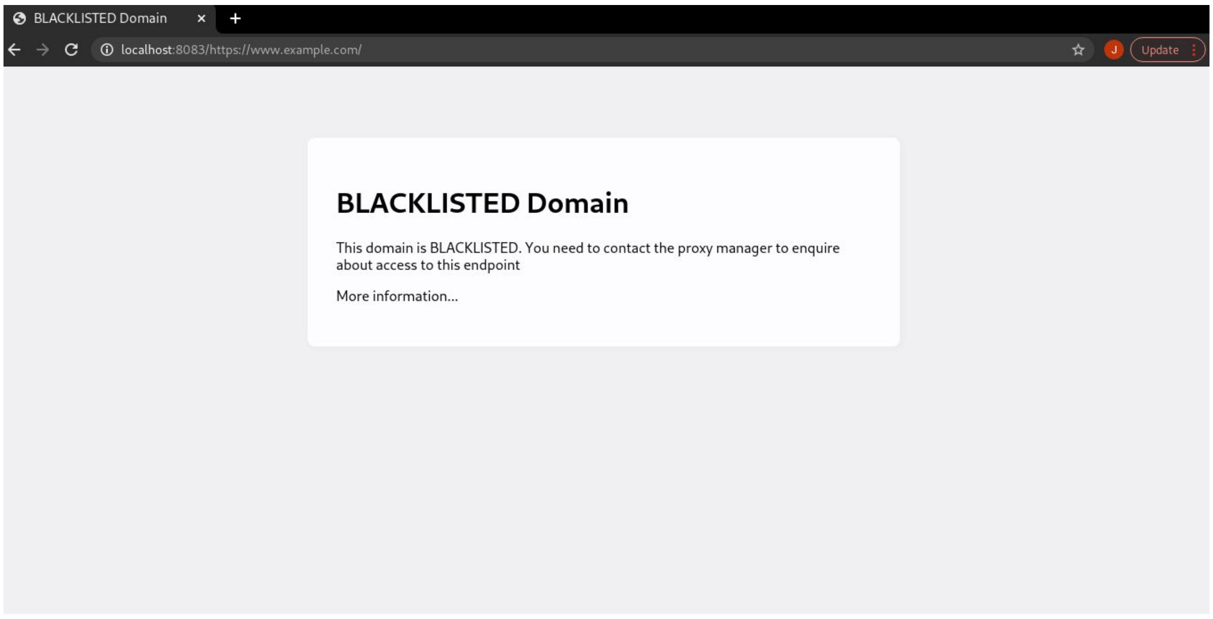
```python
s.connect((webserver, port))
s.send(data)
while 1:
    reply = s.recv(buffer_size)
    if(len(reply)>0):
        conn.sendall(reply)
```

If the proxy is forwarding a request to a HTTPS resource / server we will need to wrap the socket up as a SSL socket with additional parameters to tell the socket how to manage certificates and settings so that it can communicate securely :



```python
context = ssl.SSLContext();                         # context help manage settings and certificates
context.verify_mode     = ssl.CERT_REQUIRED;        # context help manage settings and certificates
context.check_hostname  = True;                     # context help manage settings and certificates
context.load_default_certs();                       # context help manage settings and certificates
webserver = webserver[4:]                           # webserver doesnt need 'www.'
s.connect((webserver, 443));                        # Connect to host
s = ssl.wrap_socket(s, keyfile=None, certfile=None, server_side=False, cert_reqs=ssl.CERT_NONE, ssl_version=ssl.PROTOCOL_SSLv23)
s.sendall(data)                                     # forward request to endpoint
```

9) Blacklisting URL's : If our proxy receives a request for a blacklisted URL it will not fetch the resource from the server. Instead it will send this html response back to the client telling them they have tried to connect with a blacklisted endpoint which is not allowed and to contact the proxy manager for further information.

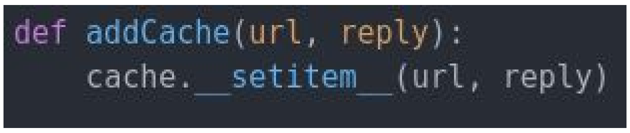


BLACKLISTED Domain

This domain is BLACKLISTED. You need to contact the proxy manager to enquire about access to this endpoint

More information...

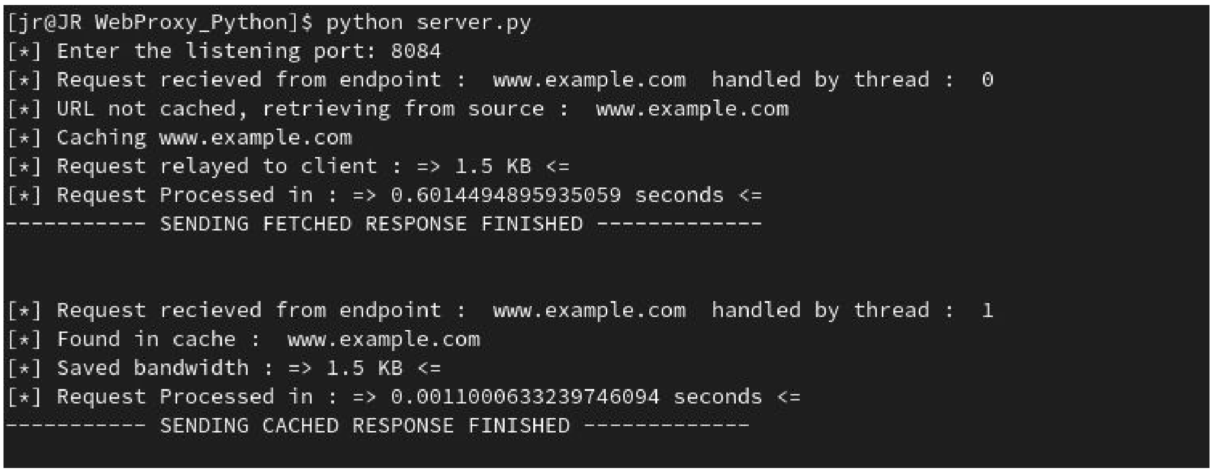
10) Caching HTTP responses : We only cache responses to HTTP requests in the proxy as such :



```python
def addCache(url, reply):
    cache.__setitem__(url, reply)
```

For my particular cache, you can also add a time value to __setitem__ which is the value in seconds that the cache entry should be alive for. However if you leave the value out it will be set to a default value. This is to simulate a Time To Live cache that only keeps content in the cache if it is still "relevant" ie. there are no old / stale responses stored in the cache

10) Displaying Bandwidth & time saved : If we are able to used a cached response for the clients request we can calculate the amount of data this would have required to download by finding the size of the bytes that are cached and return that to the client as such :



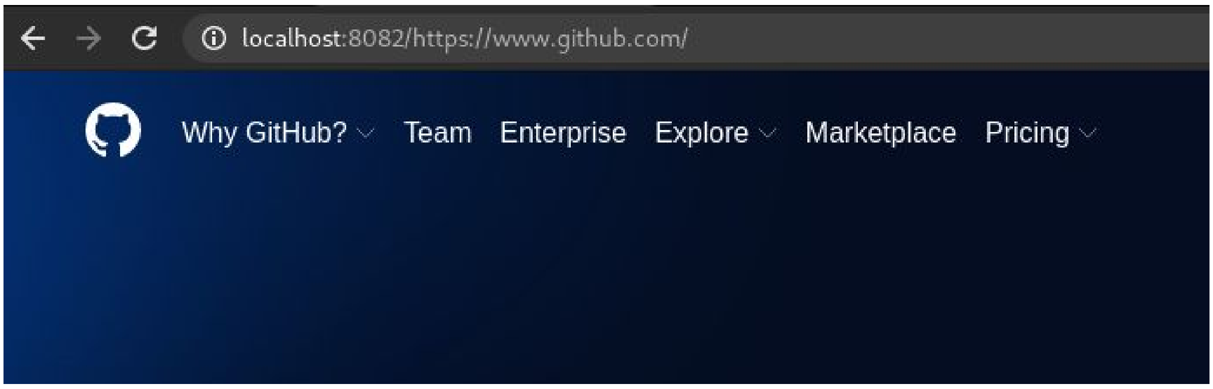
```
[jr@JR WebProxy_Python]$ python server.py
[*] Enter the listening port: 8084
[*] Request recieved from endpoint :  www.example.com  handled by thread :  0
[*] URL not cached, retrieving from source :  www.example.com
[*] Caching www.example.com
[*] Request relayed to client : => 1.5 KB <=
[*] Request Processed in : => 0.6014494895935059 seconds <=
----------- SENDING FETCHED RESPONSE FINISHED -------------


[*] Request recieved from endpoint :  www.example.com  handled by thread :  1
[*] Found in cache :  www.example.com
[*] Saved bandwidth : => 1.5 KB <=
[*] Request Processed in : => 0.0011000633239746094 seconds <=
----------- SENDING CACHED RESPONSE FINISHED -------------
```

We can see for the first request is not cached so we have to fetch that from the endpoint. However when the client tries to connect to this endpoint for the second time the proxy will find an appropriate cached response and will save a lot of time and bandwidth by sending the saved response to the client.
To calculate the time used for a response, for uncached responses we find the difference between the time the request was received by the proxy - the time the last backed was received and sent back to the client / browser. For cached responses we can find the difference between the time we received the request in the proxy - the time it takes to find the cached response and stream the last byte of it back to the client / browser

Running the server.py script :

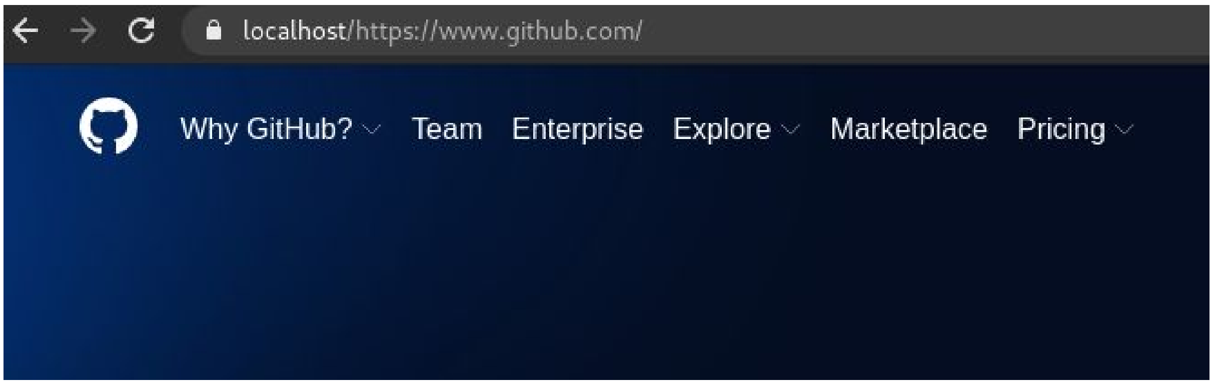
The easiest way to run the server is in http mode. HTTP mode means our connection between the browser / client will be over HTTP however we can still access HTTPS websites with http mode. This mode is run as following

$ python server.py http
Send a request to the proxy as such : http://localhost:80/https://www.github.com/

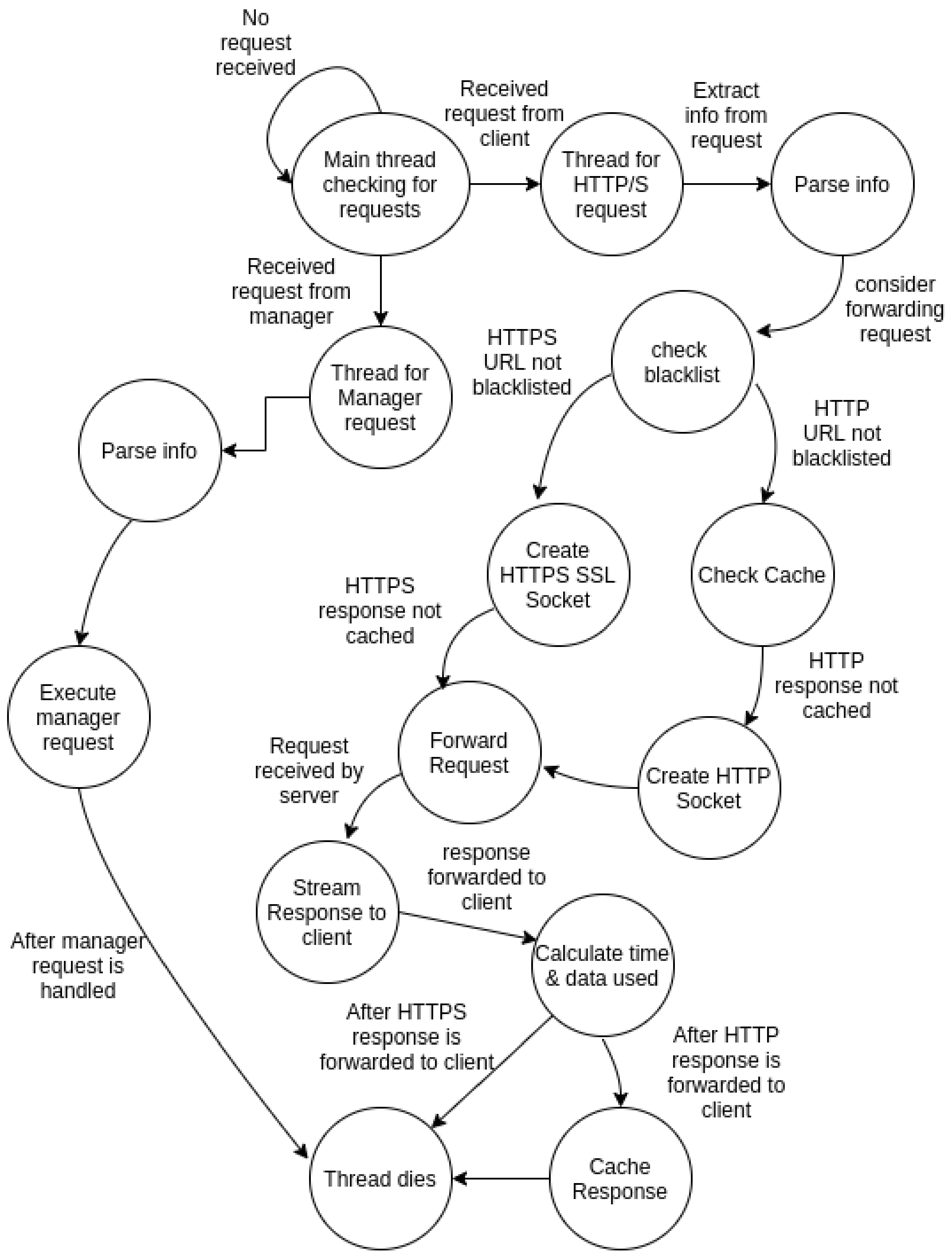


We can see that the connect connection with the proxy server on localhost is unsecure however the resource the proxy requests and sends back to the client is over HTTPS

If we want to communicate with the proxy over HTTPS we run the server in HTTPS mode as such :
$ python server.py https
You can see now that we use https instead of http in the url
If we send the following request to the proxy : https://localhost:443/https://www.github.com/
We get a secure connection as such :



Deterministic finite automata for procedure :



No request received
Main thread checking for requests
Received request from client
Thread for HTTP/S request
Extract info from request
Parse info
Received request from manager
Thread for Manager request
consider forwarding request
check blacklist
HTTPS URL not blacklisted
HTTP URL not blacklisted
Parse info
Create HTTPS SSL Socket
Check Cache
HTTPS response not cached
HTTP response not cached
Execute manager request
Forward Request
Create HTTP Socket
Request received by server
Stream Response to client
response forwarded to client
Calculate time & data used
After manager request is handled
After HTTPS response is forwarded to client
After HTTP response is forwarded to client
Thread dies
Cache Response

Code :

```
import socket, sys, os, time, json, time, ssl
from cachetools import Cache, TTLCache
from _thread import *

class TTLItemCache(TTLCache):
  def __setitem__(self, key, value, cache_setitem=Cache.__setitem__, ttl=None):
    super(TTLItemCache, self).__setitem__(key, value)
    if ttl:
      link = self._TTLCache__links.get(key, None)
```

```
            if link:
                link.expire += ttl - self.ttl

try:
    this_port = input("[*] Enter the listening port: ")
    listening_port = int(os.environ.get("PORT", this_port))
except KeyboardInterrupt:
    print("\n[*] User has requested an interrupt")
    print("[*] Application Exiting.....")
    sys.exit()

max_conn = 5 #Maximum connections queues
buffer_size = 16000 #Maximum socket's buffer size
cache = TTLItemCache(maxsize=30,ttl=99000) #Initializing my TTL cache
blacklist = []  # Intializing my blacklisted URLS
ssl_version = None
certfile = "./ssl/certificate.pem"
keyfile = "./ssl/key.pem"
ciphers = None
option_test_switch = 0 # to test, change to 1
checker = "favicon"
version_dict = {
    "tlsv1.0" : ssl.PROTOCOL_TLSv1,
    "tlsv1.1" : ssl.PROTOCOL_TLSv1_1,
    "tlsv1.2" : ssl.PROTOCOL_TLSv1_2,
    "sslv23"  : ssl.PROTOCOL_SSLv23,
    "sslv3"   : ssl.PROTOCOL_SSLv23,
}

def start():    # Main Program
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Initializing the socket
        s.bind(('', listening_port)) #Binding the socket to listen at the port
        s.listen(max_conn) #Start listening for connections
    except Exception: #Will be executed if anything fails
        sys.exit(2)

    threadID = 0
    if (sys.argv[1] == 'https'):
        print("[$] socket wrap as ssl socket \n")
    elif (sys.argv[1] == 'http'):
        print("[$] no socket wrap needed \n")
    while 1:
        conn, addr = s.accept() #Accept connection from client browser
        if (sys.argv[1] == 'https'):
            conn = ssl_wrap_socket(conn, ssl_version, keyfile, certfile, ciphers)   # convert
socket to ssl socket
        try:
            data = conn.recv(buffer_size) #Recieve client data
            # print("data : ", data.decode())
            decodeData = data.decode().split('\n')
            req = decodeData[0]
            if '"user": "manager", "pswrd": "manager"' in data.decode() : #IF WE RECIEVE A
PROXY MANAGEMENT REQUEST
                start_new_thread(handleManagerReq, (decodeData,threadID))
                threadID += 1
            else :
                if not checker in req :   #IF WE RECIEVE A NORMAL REQUEST
                    start_new_thread(conn_string, (conn,data, addr, threadID, time.time()))
                    threadID += 1
        except KeyboardInterrupt:
            s.close()
            print("\n[*] Proxy server shutting down....")
            print("[*] Have a nice day... ")
            sys.exit(1)

def conn_string(conn, data, addr, threadID, startTime):
    first_line = data.decode().split('\n')[0]
    url = first_line
    http_pos = url.find("://")

    if(http_pos==-1):
        temp=url
    else:
        temp = url[(http_pos+3):]

    port_pos = temp.find(":")
    webserver_pos = temp.find("/")
    if webserver_pos == -1:
        webserver_pos = len(temp)
    webserver = ""
    port = -1
    if(port_pos == -1 or webserver_pos < port_pos):
        port = 80
        webserver = temp[:webserver_pos]
    else:
        port = int((temp[(port_pos+1):])[:webserver_pos-port_pos-1])
        webserver = temp[:port_pos]
    webserver =  webserver.strip('HTTP')
    if "/http:/" in first_line :
        get = str('GET '+temp.replace(webserver,'')+'\n')
        host = 'Host: '+webserver+'\n\n'
        dataPackage = ((get+host).encode())
    else :
        get = str('GET '+temp.replace(webserver,'')+'\n')
        host = 'Host: '+webserver[4:]
        dataPackage = (get+host+"\r\nConnection: close\r\n\r\n").encode()
        print("dp : ", dataPackage)
    fullURL = first_line.replace('GET /http://', '').replace(' HTTP/1.1', '').strip()
    if fullURL not in blacklist:
        proxy_server(webserver, port, conn, first_line, dataPackage, first_line, threadID,
startTime)
    else :
        print("[*] Request rejected, blacklisted URL")
        while True:
            rawString = "HTTP/1.1 200 OK\n"+"Content-Type:
text/html\n"+"\n"+blacklistHTML+"\n"
            html = rawString.encode('utf-8')
            conn.send(html)
            conn.close()
            break;


def proxy_server(webserver, port, conn, addr, data, temp, threadID, startTime):
    fullReply = bytearray()
    timeDifference = 0.0
    try:
        print("[*] Request recieved from endpoint : " , webserver, " handled by thread : ",
threadID)
        s= socket.socket(socket.AF_INET, socket.SOCK_STREAM);
        if "/https:" in temp :
            print("[*] opening socket on SSL port 443")
            context = ssl.SSLContext();                    # context help manage settings and
certificates
            context.verify_mode     = ssl.CERT_REQUIRED;    # context help manage settings
and certificates
            context.check_hostname  = True;                # context help manage settings and
certificates
            context.load_default_certs();                  # context help manage settings and
certificates
            webserver = webserver[4:]                      # webserver doesnt need 'www.'
            s.connect((webserver, 443));                   # Connect to host
            s = ssl.wrap_socket(s, keyfile=None, certfile=None, server_side=False,
cert_reqs=ssl.CERT_NONE, ssl_version=ssl.PROTOCOL_SSLv23)
            s.sendall(data)                                # forward request to endpoint

            while True:                            # Streaming back bytes
                new = s.recv(buffer_size)          #
                if(len(new)>0):
                    fullReply += new
                    timeDifference = (time.time()-startTime)
                    conn.sendall(new)
                else:
                  break
```

```
          dar = float(len(fullReply))
          dar = float(dar/1024)
          dar = "%.3s" % (str(dar))
          dar = "%s KB" % (dar)
          print("[*] Request relayed to client : => %s <=" % (str(dar)))
          print("[*] Request Processed in : => %s seconds <=" % timeDifference)
          print("----------- SENDING FETCHED RESPONSE FINISHED & NOT CACHING
-------------\n\n")
      else :
          # "http request recieved "
          urlResponse = checkCache(temp)
          if(urlResponse != None):                      # CHECK IF THE URL IS CACHED BEFORE
CONSIDERING FETCHING IT
              conn.send(urlResponse)
              dar = float(len(urlResponse))
              dar = float(dar/1024)
              dar = "%.3s" % (str(dar))
              dar = "%s KB" % (dar)
              print("[*] Saved bandwidth : => %s <=" % (str(dar)))
              print("[*] Request Processed in : => %s seconds <= " %
(str(time.time()-startTime)))
              print("----------- SENDING CACHED RESPONSE FINISHED -------------\n\n")
          else :
              s.settimeout(2)
              s.connect((webserver, port))
              s.send(data)
              while 1:
                  reply = s.recv(buffer_size)
                  if(len(reply)>0):
                      conn.sendall(reply)
                      fullReply += reply
                      timeDifference = (time.time()-startTime)
                  else:
                      break

  except (BlockingIOError, socket.timeout, OSError, ssl.SSLError):
      print("[*] Caching " + temp)
      addCache(temp, fullReply)
      dar = float(len(fullReply))
      dar = float(dar/1024)
      dar = "%.3s" % (str(dar))
      dar = "%s KB" % (dar)
      print("[*] Request relayed to client : => %s <=" % (str(dar)))
      print("[*] Request Processed in : => %s seconds <=" % timeDifference)
      print("----------- SENDING FETCHED RESPONSE FINISHED -------------\n\n")
      pass
  except socket.error:
      print("Error occured in thread")
```

```
            s.close()
            conn.close()
            sys.exit(1)
        except Exception as e:
            print("Undiagnosed error : ", e)
        s.close()
        conn.close()

def checkCache(url):
    try :
        urlResponse = cache[url]
        print("[*] Found in cache : ", url)
        return urlResponse
    except KeyError as e:
        print('[*] URL not cached, retrieving from source : ', url)


def addCache(url, reply):
    cache.__setitem__(url, reply)

def handleManagerReq(req, threadID):

    print("[*] MANAGER CONNECTION HANDLED BY : ",threadID)
    payload = req[len(req)-1]   # extracts string containing json payload
    json_acceptable_payload = payload.replace("'", "\"") # creates json payload
    json_payload = json.loads(json_acceptable_payload)
    print('json : ', json_payload)
    if json_payload["func"] == "blklst":
        blacklist.append(json_payload["url"])
        print('[*] blacklist updated, ADDED : ', json_payload["url"])
    if json_payload["func"] == "rmblk":
        blacklist.remove(json_payload["url"])
        print('[*] blacklist updated, REMOVED : ', json_payload["url"])
    if json_payload["func"] == "usrBan":
        blacklist.remove(json_payload["url"])
        print('[*] blacklist updated, REMOVED : ', json_payload["url"])

def ssl_wrap_socket(sock, ssl_version=None, keyfile=None, certfile=None, ciphers=None):

    #1. init a context with given version(if any)
    if ssl_version is not None and ssl_version in version_dict:
        #create a new SSL context with specified TLS version
        sslContext = ssl.SSLContext(version_dict[ssl_version])
        if option_test_switch == 1:
            print("ssl_version loaded!! =", ssl_version)
    else:
        #if not specified, default
        sslContext = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
    if ciphers is not None:
        #if specified, set certain ciphersuite
        sslContext.set_ciphers(ciphers)
        if option_test_switch == 1:
            print("ciphers loaded!! =", ciphers)

    #server-side must load certfile and keyfile, so no if-else
    sslContext.load_cert_chain(certfile, keyfile)
    print("ssl loaded!! certfile=", certfile, "keyfile=", keyfile)

    try:
        return sslContext.wrap_socket(sock, server_side = True)
    except ssl.SSLError as e:
        print("wrap socket failed!")
        print(traceback.format_exc())

blacklistHTML = """
    <!doctype html>
    <html> <head> <title>BLACKLISTED Domain</title>
        <meta charset="utf-8" />
        <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1" />
        <style type="text/css">
        body {
            background-color: #f0f0f2;
            margin: 0;
            padding: 0;
            font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI", "Open
Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;}
        div {
            width: 600px;
            margin: 5em auto;
            padding: 2em;
            background-color: #fdfdff;
            border-radius: 0.5em;
            box-shadow: 2px 3px 7px 2px rgba(0,0,0,0.02);
        }
        a:link, a:visited {
            color: #38488f;
            text-decoration: none;
        }
        @media (max-width: 700px) {
            div {
                margin: 0 auto;
                width: auto;
            }
        }
        </style>
    </head>
    <body>
    <div>
        <h1>BLACKLISTED Domain</h1>
        <p>This domain is BLACKLISTED. You need to contact the proxy manager to enquire
about access to this endpoint</p>
        <p><a ">More information...</a></p>"""

start()
```