HTTP & HTTPS Web Proxy

Cian Jinks

Contents

1 Overview

1.1 Proxy Design

In this section I will discuss the overall design of my proxy before going into specifics about how I implemented HTTP request relaying, HTTPS tunnelling using websocket connections, a response cache, url blocking and timing/bandwidth data collection.

On program start, a proxy thread is launched running the follow function:

```
PROXY_HOST = "127.0.01"
PROXY_PORT = 8080

def proxy():
    # Await Connection
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind((PROXY_HOST, PROXY_PORT))
    sock.listen(1)
    debug_print(f"[-] Awaiting connection to proxy server on {PROXY_HOST}:{PROXY_PORT}...")

    # Receive Requests
    while True:
        conn, addr = sock.accept()
        data = conn.recv(4096)
        thread = threading.Thread(target=handle_connection, args=(data, conn, ))
        thread.start()
```

This function is the main entry point of my proxy server. I begin by setting up a TCP socket to receive connections from clients on localhost port 8080. Once it is setup the proxy loops forever waiting for incoming connections. Whenever a connection is received it creates a "handle connection" thread. This where we see the concept of a **threaded** proxy which can handle multiple requests simultaneously.

Handling connections works as follows:

```
def handle_connection(data, conn):
    if data:
        # Parse HTTP packet
        http_request = HTTPRequest(data)
        if http_request.https:
            tunnel(conn, http_request)
        else:
            relay(conn, http_request)
```

Firstly, the raw packet data is passed to the constructor of a `HTTPRequest` class. This class parses the raw data based on a typical HTTP request format and stores important information in variables such as *method*, *url*, *headers*, *etc.*

The proxy then has to differentiate between whether the packet wants to begin a HTTPS tunnel (this would be a CONNECT method packet) or whether it is a simple HTTP request that needs to be relayed. In the event of a HTTPS request, a tunnel is setup between the client and the server for them to talk securely over using TCP sockets. In the event of a HTTP request, the request is relayed to the server, the response is collected by the proxy and then relayed back to the client. The proxy also makes sure to cache the responses for each given url as well as pass all requests to the Management Console. Lastly, if a request is already in the cache, the cached response will be returned and if a request's url is blocked a HTTP error is returned.

This concludes a very high level overview of my proxy design. The next sections will cover each concept mentioned here in more detail.

1.2 HTTPS Tunnelling (Websockets)

When a client wants to use HTTPS to talk to a server through my proxy it will send a HTTP request with a CONNECT method. When my proxy reads that the method is CONNECT it treats this as a special case and sets up a Websocket between the client and server for them to pass messages over securely like so:

```
def tunnel(client_conn, http_request):

    # <Blocked URLs are checked here normally>

    # If not blocked, continue as normal
    server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_sock.connect((http_request.connect_host, http_request.port))

    response = b"HTTP/1.1 200 OK\r\n\r\n"
    client_conn.sendall(response)

    # Websocket
    conns = [client_conn, server_sock]
    while True:
        recvlist, _, error = select.select(conns, [], conns, 3)
        if error or not recvlist:
            break
        for r in recvlist:
            other = conns[1] if r is conns[0] else conns[0]
            data = r.recv(4096)
            if not data:
                break
            other.sendall(data)
    server_sock.close()
    client_conn.close()
    debug_print(f"[-] Closed HTTPS tunnel!")
```

First the proxy responds to the client with a HTTP 200 OK, letting it know the websocket has been set up and it is good to go. Then the actual implementation of the web socket uses python's `select` module to do I/O multiplexing with the client and server socket. This allows them to send messages back and forth without the need for delays caused by polling in the python code.

1.3 Management Console

I implemented a management console for the proxy using python's `curses` library. The management console enables you to view all HTTP requests that the proxy has received and inspect them in detail. It also allows you to block specific URLs. A demonstration of these features is shown in the assignment video I made.

To pass information between the proxy and the management console the program makes use of shared memory with thread locks like so:

```
cache_lock = threading.Lock()
cache = {}

requests_lock = threading.Lock()
requests = []

blocked_lock = threading.Lock()
blocked_urls = []
```

This allows the proxy to place all requests it has received in the `requests` array and the management console can then read this array to obtain all the information it needs. The proxy will also check the `blocked_urls` whenever it receives a request. This is an array populated by the management console thread.

1.4 Caching

The caching system for my proxy is fairly simple. It uses a python dictionary mapping request URLs to cached response data. Because the proxy can handle multiple requests simultaneously the cache is protected with a thread lock as well.

When a new request comes into the proxy from a client, the proxy will grab the lock and check to see if the URL is within the cache. (Note that this only happens for HTTP requests as HTTPS request are encrypted and cannot be cached):

```
# Relay packets when using HTTP
def relay(conn, http_request):

    # <Check blocked URLs occurs here as well>

    with cache_lock:
        if http_request.url in cache:
            request_status = REQUEST_CACHED

    response = b""
    if request_status == REQUEST_CACHED:
        # Caching
        debug_print(f"[-] Using cached website for HTTP")
        response = cache[http_request.url]
    else:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.settimeout(1)
        sock.connect((http_request.connect_host, http_request.port))
        sock.sendall(http_request.raw_data)
        total_bytes = total_bytes + len(http_request.raw_data)
        response = receive_http_response(sock)
        cache[http_request.url] = response
        sock.close()
    conn.sendall(response)
    total_bytes = total_bytes + len(response)
    conn.close()
```

If the URL is within the cache it will return the cached response. Otherwise it will talk to the server to retrieve the response, cache this response and then relay the response back to the client.

1.5 Blocking URLs

The `blocked_urls` array is populated by the management console as shown in the video. Every time the proxy receives a HTTP request it will check the blocked URLs list to see if the request's url is contained within like so:

```
def relay(conn, http_request):
    with blocked_lock:
        if http_request.url in blocked_urls:
            request_status = REQUEST_BLOCKED
    with requests_lock:
        debug_print(f"[-] Added packet to requests!")
        requests.append((datetime.now().strftime("%H:%M:%S"), http_request, request_status))
    if request_status == REQUEST_BLOCKED:
        debug_print(f"[-] Failed - URL is blocked")
        response = b"HTTP/1.1 403 Forbidden\r\n\r\n"
        conn.sendall(response)
        conn.close()
        return

    # Otherwise continue as normal
```

As can be seen, if the URL is blocked an HTTP 403 Forbidden response is returned to the client. Otherwise, the `relay` function proceeds as normal. Note that this will also work for HTTPS connections as to setup a tunnel a HTTP request with the URL and CONNECT method is given to the proxy.

1.6 Timing and Bandwidth Data

To gather timing and bandwidth data I wrapped the `relay` function for HTTP requests with a profiling function:

```
def profile_relay(conn, http_request):
    start = time.perf_counter()
    total_bytes_trans = relay(conn, http_request)
    end = time.perf_counter()
    time_taken_s = (end - start)
    debug_print(f"Relayed HTTP request and response in {time_taken_s * 1000}ms -
                  Transferred {total_bytes_trans / time_taken_s} bytes per second")
```

As shown in my demonstration video, this revealed that by adding a cache my proxy's efficieny had improved *dramatically*. Most normal HTTP relays were taking roughly **30-70ms** each. However, when the response was cached it took **under 1ms** to return the response to the client. (This data may have been skewed by the fact my client was running on the same machine as the proxy).

2 Code

Here is the code for my Proxy and Management Console in its entirety. Coming from my `Proxy.py` .

```
import socket
import os
import threading
import select
import curses
import time
from datetime import datetime
from typing import List, Tuple

# ---------------------------------------- Debug ----------------------------------------
DEBUG_MODE = False

def debug_print(fmt):
    if DEBUG_MODE:
        print(fmt)

# ---------------------------------------- Shared Resources ----------------------------
cache_lock = threading.Lock()
cache = {}

requests_lock = threading.Lock()
requests = []
REQUEST_NONE = 0
REQUEST_CACHED = 1
REQUEST_BLOCKED = 2

blocked_lock = threading.Lock()
blocked_urls = []

# ---------------------------------------- Management Console ---------------------------

CONSOLE_WINDOW_ID = False
BLOCKED_WINDOW_ID = True
```

```
def draw_keymap(window):
    window.erase()
    window.border()
    window.addstr(0, int(curses.COLS / 4) - 4, " Keymap ")
    window.addstr(1, 1, "[Q] - Quit")
    window.addstr(2, 1, "[S] - Switch window")
    window.addstr(3, 1, "[R] - Refresh requests list")
    window.addstr(4, 1, "[B] - Block URL (of selected packet)")
    window.addstr(5, 1, "[U] - Unblock URL")
    window.addstr(6, 1, "[Ent/Esc] - View detailed packet info")
    window.addstr(7, 1, "[Up/Down Arrow] - Select packet or URL")
    window.refresh()

def draw_blocked(window, selected_url, focused_window, block_min, block_max):
    window.erase()
    window.border()
    window.addstr(0, int(curses.COLS / 4) - 7, " Blocked URLs ")

    # List of blocked URLs
    b_pos = 1
    for b in range(block_min, block_max + 1):
        window.addstr(b_pos, 1, "[" + str(b) + "] " + blocked_urls[b].decode(),
            curses.A_REVERSE if (b == selected_url and focused_window == BLOCKED_WINDOW_ID)
                             else curses.A_NORMAL)
        b_pos = b_pos + 1
    window.refresh()

def draw_request_string(window, y, x, attrib, time, request, request_status):
    type = "HTTPS" if request.https else "HTTP"
    window.addstr(y, x, f"[{time}] ", attrib)
    window.addstr(y, x + 11, f"{type}  ", attrib)
    window.addstr(y, x + 17, f"--> ", attrib)
    window.addstr(y, x + 21, f"{request.method.decode()}
        {request.url.decode()} {request.version.decode()}", attrib)
    window.addstr(y, curses.COLS - 14, f"({request.length} bytes)", curses.A_NORMAL)
    if request_status == REQUEST_BLOCKED:
        window.addstr(y, curses.COLS - 22, f"BLOCKED", curses.A_NORMAL)
    elif request_status == REQUEST_CACHED:
        window.addstr(y, curses.COLS - 25, f"USED CACHE", curses.A_NORMAL)

def draw_console(window, selected_packet, focused_window, request_min, request_max):
    window.erase()
    window.border()
    window.addstr(0, int(curses.COLS / 2) - 22,
        f" Management Console - {len(requests)} Total Packets ")

    # List of requests
    r_pos = 1
    for r in range(request_min, request_max + 1):
        current_request = requests[r]
        draw_request_string(window, r_pos, 1,
            curses.A_REVERSE if (r == selected_packet and focused_window == CONSOLE_WINDOW_ID)
                             else curses.A_NORMAL,
                             current_request[0], current_request[1], current_request[2])
        r_pos = r_pos + 1
    window.refresh()
```

```
def draw_details(window, selected_packet):
    request = requests[selected_packet]

    window.erase()
    window.border()
    window.addstr(0, int(curses.COLS / 2) - 8, f" Request Details ")
    window.addstr(1, 1, f"General Information", curses.A_REVERSE | curses.A_UNDERLINE)
    window.addstr(2, 1, f"Timestamp - {request[0]}")
    window.addstr(3, 1, f"Length (in bytes) - {request[1].length}")
    window.addstr(4, 1, f"Method - {request[1].method.decode()}")
    window.addstr(5, 1, f"URL - {request[1].url.decode()}")
    window.addstr(6, 1, f"Version - {request[1].version.decode()}")

    window.addstr(8, 1, f"HTTP Headers", curses.A_REVERSE | curses.A_UNDERLINE)

    h_pos = 9
    for k,v in request[1].headers.items():
        name = k.decode()
        value = v.decode()
        window.addstr(h_pos, 1, f"{name}: {value}")
        h_pos = h_pos + 1


    window.addstr(h_pos + 1, 1, f"Raw Request Data", curses.A_REVERSE | curses.A_UNDERLINE)
    window.addstr(h_pos + 2, 1, f"{request[1].raw_data}")
    window.refresh()

def app(stdscr):
    # Options
    curses.curs_set(0)

    # Window Creation
    console_window = curses.newwin(curses.LINES - int(curses.LINES / 3), curses.COLS, 0, 0)
    keymap_window = curses.newwin(int(curses.LINES / 3), int(curses.COLS / 2),
                                  curses.LINES - int(curses.LINES / 3), 0)
    blocked_window = curses.newwin(int(curses.LINES / 3), int(curses.COLS / 2),
                                   curses.LINES - int(curses.LINES / 3), int(curses.COLS / 2))
    focused_window = False # False for console, True for blocked urls
    details_window = curses.newwin(curses.LINES, curses.COLS, 0, 0)
    details_page = False
    selected_packet = 0
    selected_url = 0

    max_urls = int(curses.LINES / 3) - 2 # Max number of URLs displayable
    block_min = 0
    block_max = len(blocked_urls) - 1 if len(blocked_urls) < max_urls else max_urls - 1

    max_requests = curses.LINES - int(curses.LINES / 3) - 2 # Max number of packets displayable
    request_min = 0
    request_max = (len(requests) - 1) if (len(requests) < max_requests) else (max_requests - 1)

    while True:
        stdscr.clear()
        stdscr.refresh()
        if not details_page:
            # Draw Windows
            draw_console(console_window, selected_packet,
```

```
                    focused_window, request_min, request_max)
        draw_keymap(keymap_window)
        draw_blocked(blocked_window, selected_url, focused_window, block_min, block_max)

        # Handle Input
        c = stdscr.getch()
        if c == ord('q'):
            break
        elif c == ord('s'):
            focused_window = not focused_window
        elif c == ord('r'):
            # Update requests bounds
            if request_max < max_requests:
                request_max =
                    (len(requests) - 1) if (len(requests) < max_requests)
                                        else (max_requests - 1)
        elif c == ord('b') and focused_window == CONSOLE_WINDOW_ID:
            with blocked_lock:
                blocked_urls.append(requests[selected_packet][1].url)
            # Update blocked url bounds
            if block_max < max_urls:
                block_max =
                    (len(blocked_urls) - 1) if (len(blocked_urls) < max_urls)
                                            else (max_urls - 1)
        elif c == ord('u') and focused_window == BLOCKED_WINDOW_ID:
            with blocked_lock:
                blocked_urls.pop(selected_url)
            if selected_url > len(blocked_urls) - 1:
                selected_url = len(blocked_urls) - 1
            # Update blocked url bounds
            if block_max < max_urls:
                block_max =
                    (len(blocked_urls) - 1) if (len(blocked_urls) < max_urls)
                                            else (max_urls - 1)
        elif c == curses.KEY_DOWN:
            if focused_window == CONSOLE_WINDOW_ID and selected_packet < len(requests) - 1:
                selected_packet = selected_packet + 1
                if selected_packet > request_max:
                    request_max = request_max + 1
                    request_min = request_min + 1
            elif focused_window == BLOCKED_WINDOW_ID and selected_url < len(blocked_urls) - 1:
                selected_url  = selected_url + 1
                if selected_url > block_max:
                    block_max = block_max + 1
                    block_min = block_min + 1
        elif c == curses.KEY_UP:
            if focused_window == CONSOLE_WINDOW_ID and selected_packet > 0:
                selected_packet = selected_packet - 1
                if selected_packet < request_min:
                    request_max = request_max - 1
                    request_min = request_min - 1
            elif focused_window == BLOCKED_WINDOW_ID and selected_url > 0:
                selected_url = selected_url - 1
                if selected_url < block_min:
                    block_max = block_max - 1
                    block_min = block_min - 1
        elif c == 10: # Enter
```

```
                if len(requests) > 0:
                    details_page = True
        else:
            draw_details(details_window, selected_packet)
            # Handle Input
            c = stdscr.getch()
            if c == ord('q'):
                break
            elif c == 27: # Escape
                details_page = False

# ---------------------------------------- PROXY ----------------------------------------

class HTTPRequest:
    method: str
    url: str
    version: str
    headers: dict

    port: int # 80 for http and 443 for https
    https: bool
    length: int
    connect_host: bytes # Parse URL for socket connection

    def __init__(self, raw_data):
        self.raw_data = raw_data
        self.length = len(raw_data)
        self.parse()
        self.print()


    def parse(self):
        split_data = self.raw_data.split(b"\r\n")
        request_line = split_data[0].split(b" ")

        url = []
        self.https = False

        headers = {}
        for i in range(1, len(split_data)):
            if (split_data[i] == b""):
                break
            header = split_data[i].split(b": ")
            headers[header[0]] = header[1]

        if b"http://" in request_line[1]:
            url.append(request_line[1][7:])
        elif b":" in request_line[1]:
            self.https = True
            url = request_line[1].split(b":")
        else:
            url.append(request_line[1])

        self.method = request_line[0]
        self.url = url[0]
        self.version = request_line[2]
        self.headers = headers
```

```
        self.port = 443 if self.https else 80

        if b"Host" in self.headers:
            if b":" in self.headers[b"Host"]:
                url = self.headers[b"Host"].split(b":")
                self.connect_host = url[0]
            else:
                self.connect_host = self.headers[b"Host"]
        else:
            self.connect_host = self.url

    def print(self):
        debug_print('[-] HTTPParser Information')
        debug_print("[-] ----------------------------------------")
        debug_print(f"Method - {self.method}")
        debug_print(f"URL - {self.url}")
        debug_print(f"Version - {self.version}")
        h = 0
        for k,v in self.headers.items():
            debug_print(f"Header {h} - {k}: {v}")
            h = h + 1
        debug_print("[-] ----------------------------------------")

def get_content_length(response):
        result = []
        len_index = response.find(b"Content-Length")
        if len_index != -1:
            len_index = len_index + 15
            i = len_index
            while chr(response[i]) != '\r':
                character = chr(response[i])
                if character.isdigit():
                    result.append(chr(response[i]))
                i = i + 1
            return int(''.join(result))
        return -1

def receive_http_response(sock):
    response = b""
    content_length = -1
    try:
        while True:
            if content_length != -1 and len(response) >= content_length:
                break
            chunk = sock.recv(4096)
            response = response + chunk
            if content_length == -1 and b"Content-Length" in response:
                content_length = get_content_length(response)
            if len(chunk) == 0:
                break
    except TimeoutError:
        debug_print("[-] Timeout occurred")
    sock.close()
    debug_print(f"[-] Received response of length {len(response)}")
    return response
```

```
# Relay packets when using HTTP
def relay(conn, http_request):
    # Track total transferred bytes
    total_bytes = 0

    # Handle checking cached and blocked list
    request_status = REQUEST_NONE
    with cache_lock:
        if http_request.url in cache:
            request_status = REQUEST_CACHED
    with blocked_lock:
        if http_request.url in blocked_urls:
            request_status = REQUEST_BLOCKED
    with requests_lock:
        debug_print(f"[-] Added packet to requests!")
        requests.append((datetime.now().strftime("%H:%M:%S"), http_request, request_status))
    if request_status == REQUEST_BLOCKED:
        debug_print(f"[-] Failed - URL is blocked")
        response = b"HTTP/1.1 403 Forbidden\r\n\r\n"
        conn.sendall(response)
        conn.close()
        return

    response = b""
    if request_status == REQUEST_CACHED:
        # Caching
        debug_print(f"[-] Using cached website for HTTP")
        response = cache[http_request.url]
    else:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.settimeout(1)
        sock.connect((http_request.connect_host, http_request.port))
        sock.sendall(http_request.raw_data)
        total_bytes = total_bytes + len(http_request.raw_data)
        response = receive_http_response(sock)
        cache[http_request.url] = response
        sock.close()
    conn.sendall(response)
    total_bytes = total_bytes + len(response)
    conn.close()
    debug_print(f"[-] Finished relaying packet...")
    return total_bytes

# Create a TCP tunnel when using HTTPS
def tunnel(client_conn, http_request):

    # Handle checking blocked list
    request_status = REQUEST_NONE
    with blocked_lock:
        if http_request.url in blocked_urls:
            request_status = REQUEST_BLOCKED
    with requests_lock:
        debug_print(f"[-] Added packet to requests!")
        requests.append((datetime.now().strftime("%H:%M:%S"), http_request, request_status))
    debug_print(f"[-] Received HTTPS Request - Creating Tunnel")
    if request_status == REQUEST_BLOCKED:
        debug_print(f"[-] Failed - URL is blocked")
```

```
        response = b"HTTP/1.1 403 Forbidden\r\n\r\n"
        client_conn.sendall(response)
        client_conn.close()
        return

    # If not blocked, continue as normal
    server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_sock.connect((http_request.connect_host, http_request.port))

    response = b"HTTP/1.1 200 OK\r\n\r\n"
    client_conn.sendall(response)

    # Websocket
    conns = [client_conn, server_sock]
    while True:
        recvlist, _, error = select.select(conns, [], conns, 3)
        if error or not recvlist:
            break
        for r in recvlist:
            other = conns[1] if r is conns[0] else conns[0]
            data = r.recv(4096)
            if not data:
                break
            other.sendall(data)
    server_sock.close()
    client_conn.close()
    debug_print(f"[-] Closed HTTPS tunnel!")

def profile_relay(conn, http_request):
    start = time.perf_counter()
    total_bytes_trans = relay(conn, http_request)
    end = time.perf_counter()
    time_taken_s = (end - start)
    debug_print(f"Relayed HTTP request and response in {time_taken_s * 1000}ms -
                    Transferred {total_bytes_trans / time_taken_s} bytes per second")

def handle_connection(data, conn):
    if data:
        debug_print(f"------------------------------------------------------------")
        debug_print(f"[-] Received connection - parsing packet")
        # Parse HTTP packet
        http_request = HTTPRequest(data)
        if http_request.https:
            tunnel(conn, http_request)
        else:
            profile_relay(conn, http_request)
        debug_print(f"------------------------------------------------------------")

PROXY_HOST = "127.0.01"
PROXY_PORT = 8080

def proxy():
    # Await Connection
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind((PROXY_HOST, PROXY_PORT))
    sock.listen(1)
```

```
    debug_print(f"[-] Awaiting connection to proxy server on {PROXY_HOST}:{PROXY_PORT}...")

    # Receive Requests
    while True:
        conn, addr = sock.accept()
        data = conn.recv(4096)
        thread = threading.Thread(target=handle_connection, args=(data, conn, ))
        thread.start()

def main():
    # Start Proxy
    pt = threading.Thread(target=proxy)
    pt.start()

    # Start Management Console
    if not DEBUG_MODE:
        curses.wrapper(app)
        print(f"[-] Shutting down proxy...")
        os._exit(0)

if __name__ == "__main__":
    main()
```