CS - 433

Computer Networks

Final Report

Team ID - 9

Mahesh Dange	20110050
Mumuksh Tayal	20110116
Pramod Limbore	20110101
Shriyash Mandavekar	20110106

Submitted to: Prof. Sameer Kulkarni

Acknowledgement

We would like to express our sincere gratitude to our Teaching Assistant, Dhyey Thummar, for his invaluable support and guidance throughout the duration of this project. His unwavering commitment and expertise have played a pivotal role in shaping the direction of our work. Our TA's dedication in discussing various approaches and concepts have been instrumental in clarifying intricate details, refining our strategies, and providing valuable insights that significantly enhanced the quality of our project. We extend our heartfelt appreciation to our TA for the indispensable contribution to the successful completion of this project.

Additionally, we would like to extend our deep appreciation to Professor Sameer Kulkarni for providing us with the opportunity to undertake this project under his guidance. Professor Kulkarni's mentorship and encouragement have been instrumental in the realisation of this endeavour. His expertise and commitment to fostering a learning environment have enriched our understanding of the subject matter and inspired us to pursue excellence in our work. We are truly thankful for the guidance and encouragement provided by Professor Kulkarni, which has been a cornerstone in the successful execution of this project.

Table of Contents

Development of a Comprehensive URL Filtering Tool

About the Project:

In today's digital era, ensuring a secure online environment is vital. This project focuses on web URL filtering, aiming to understand and implement effective filtering techniques. Our approach involves exploring different architectures, evaluating their pros and cons, and understanding the properties of data packets across various protocols. The goal is to build a practical URL filtering system based on a grounded understanding of these elements.

The project involves a methodical examination of URL filtering, dissecting different approaches. We've delved into the practical aspects of data packet layers and architectural choices, emphasising a straightforward and focused exploration. The insights gained contribute to the development of a functional and well-informed URL filtering solution.

Various approaches and challenges:

A) Intercept TCP connection with the server by setting the RST flag in the TCP header (client-side):

With this architecture, we only focus on blocking certain blocklisted IP addresses. With this architecture, we only focus on blocking certain blocklisted IP addresses stored in the database. This is done by setting the RST flag in one of the packets' TCP headers to terminate the TCP connection between the personal device and the server. However, we witnessed multiple challenges while implementing it with socket programming. Although there seemingly exists a method (libnet_write()) that could help send the TCP packet, it is not very trivial to implement it, as intervening and sending a packet from a socket that is already involved in an established connection with the server is not approach that is used very commonly. Moreover, only blocking the IP addresses may not be enough in a lot of cases as most websites nowadays use load balancing and host the website on multiple servers at a time, so even if we block one of the IP addresses to that website, there may be some other server that could serve the files to the website.

B) Build a DNS server for URL querying:

To implement URL blocking, an approach could be to build a DNS server and configure the router to use this DNS server. So if the user tries accessing some blocked website, the DNS itself won't serve the query request for that URL. However, a potential challenge would be that a user can anyday switch the DNS from the router configurations. A potential solution for this is to lock the router's configuration so that the user isn't able to switch the DNS server. However, this can again be bypassed by directly using an IP address for the website instead of using the URL.

C) Implement a proxy server:

Another approach to filter web URLs could be to use a proxy server that intercepts the outgoing requests. If an ingress request URL or an IP address exists in the blocklist, the proxy server can easily intercept the request by not forwarding it any further. This is the most practical and secure way to implement URL filtering.

We tried to build a proxy server which just focuses on the URL blacklisting based on the IP addresses. We successfully built such a proxy server which can block certain HTTP urls which are blacklisted. However, we are only able to build it for the HTTP requests. While building a proxy server that can handle HTTPS requests, we faced some problems, such as needing to handle the SSL/TLS encryption, which was challenging. Also, managing the concurrent connections and handling multiple clients simultaneously is a little difficult.

D) Tunnelling over proxy server using CONNECT method:

This final approach also uses a proxy server, however, it targets to filter encrypted data packets over HTTPS, which uses SSL/TLS encryption to hide the contents of the data packet (encryption acts between Application layer (HTTP) and Transport layer (TCP)). This approach utilises a rather uncommonly spoken about HTTP method, CONNECT. Using this method, the proxy server facilitates a direct connection between the client to the host server over HTTPS protocol white ensuring none of the packets get intercepted in the middle or on the proxy. This approach is pretty relevant in the industry and is quite widely used as well.

Methodology:

After trying out all different types of approaches, we finally deduce down to primarily using a proxy server approach for our project of URL filtering.

Implementation of Proxy Server

We successfully implemented the proxy server approach to block certain URLs. Let's look into the code.

1. We first created a socket using `socket` and bound it to localhost(`127.0.0.1`). It listens to the queries on the localhost and port 8080. This socket is used to accept client connections and initiate communication with clients.

```
int proxy_socket = socket(AF_INET, SOCK_STREAM, 0);
// ... (binding and listening)
```

2. It enters a continuous loop to accept incoming clients. For each accepted communication, if there are no errors, it prints information about the client's address. It then calls the handle_client function to process the client's request.

```
int client_socket = accept(proxy_socket, (struct sockaddr*)&client_addr, &client_addr_len);
// ... (handling client request using handle_client function)
```

3. The function handle_client then stores the request in the request buffer. By parsing the Host field code, it extracts and prints out the URL. The URL is then checked against the blacklisted domains. (for, eg. example.com)

```
void handle_client(int client_socket) {
    char request[4096];
    // Extraction URL and checking if it is blacklisted
}
```

4. If the URL is blacklisted, then it sends a forbidden response to the client and closes the client connection. The code does not close the proxy socket, which continues listening for new incoming connections.

```
if (strstr(request, BLACKLISTED_URL) != NULL) {
        printf("URL blocked: %s\n", BLACKLISTED_URL);
        const char* response = "HTTP/1.1 403 Forbidden\r\nContent-Length: 19\r\n\r\nAccess Denied: URL
blocked\r\n";
        send(client_socket, response, strlen(response), 0);
        close(client_socket);
        return;
    }
```

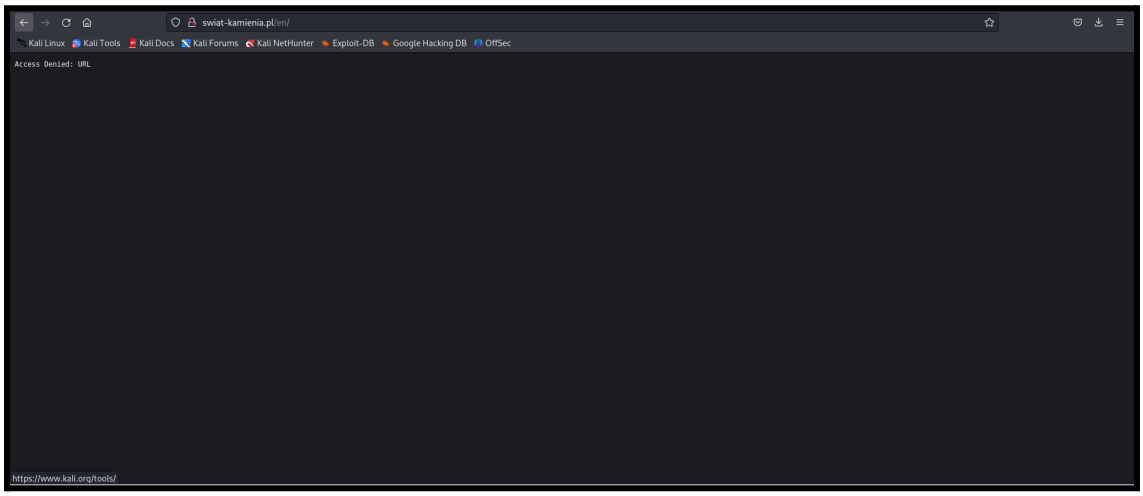
5. If the URL is not blacklisted, the code retrieves the target server's IP address based on the URL extracted from the client's request. It creates a new socket (target_socket) for communication with the target server and establishes a connection. It receives the response from the target server and forwards it back to the client.

```
// Forward the request to the target server
send(target_socket, request, bytes_received, 0);
```

Setting and testing the proxy server

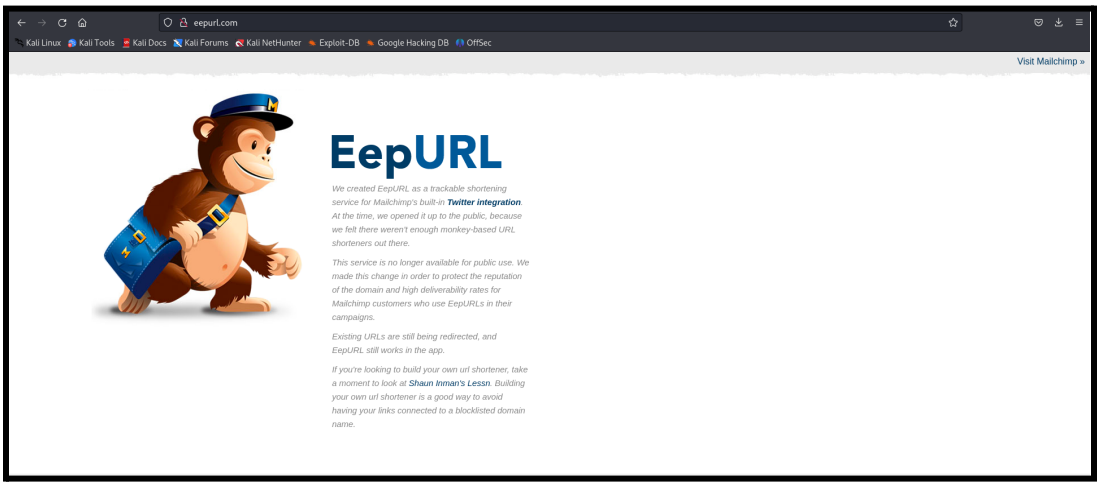
1. To set the proxy server in Mozilla Firefox, open network settings and use the localhost or `127.0.0.1` and port number 8080.

2. The above proxy server implementation works only for HTTP requests; hence, we are configuring only HTTP proxy. Now, you can test any HTTP URL on the browser.
   For, eg. If we block one HTTP url such as http://swiat-kamienia.pl/en/. Then, we get the following response.



3. If the URL is not blacklisted, then it gets forwarded to the target server.

   For example, The http://eepurl.com/ is not blacklisted and can be viewed easily.



Log and response from the server

```
Forwarding request to the target server...
Error connecting to the target server: Connection refused
Done handling client request
Connection accepted from 127.0.0.1:45232
Handling client request...
Inside handle_client
Inside while loop
Extracted URL: eepurl.com
IP address: 23.15.194.151 & eepurl.com
Forwarding request to the target server:
Sending request to target server
Received request from client: GET http://eepurl.com/ HTTP/1.1
Host: eepurl.com
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Cookie: ak_bmsc=602611A72308AF28A0F21899DDF76EBD~000000000000000000000000000000~YAAQzowsMUVYhK
LtMMKiaLSnlf6WewGx3SwHuBO9NIlP9jIABMZhEvOBX50Zb50NdTfwxUsG0PL8TVXF2/+KksNJrqQw7jsLrjQ7eWoWsdWx
cemzhqbl9nqal9MSh5dFBwpVJ7s4WlLV
Upgrade-Insecure-Requests: 1
```

Simple HTTP Proxy:

a. In case of blocklisted URL:

1. TCP Connection Establishment:

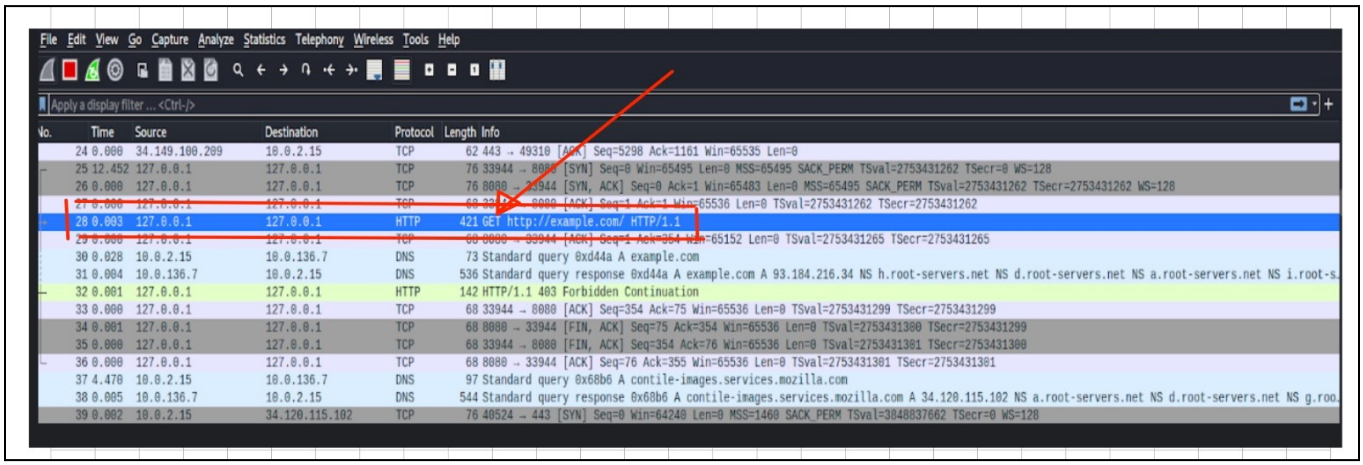
- TCP connection is established with the custom HTTP proxy.
- Connection details: Local host (127.0.0.1) and destination port (8080).

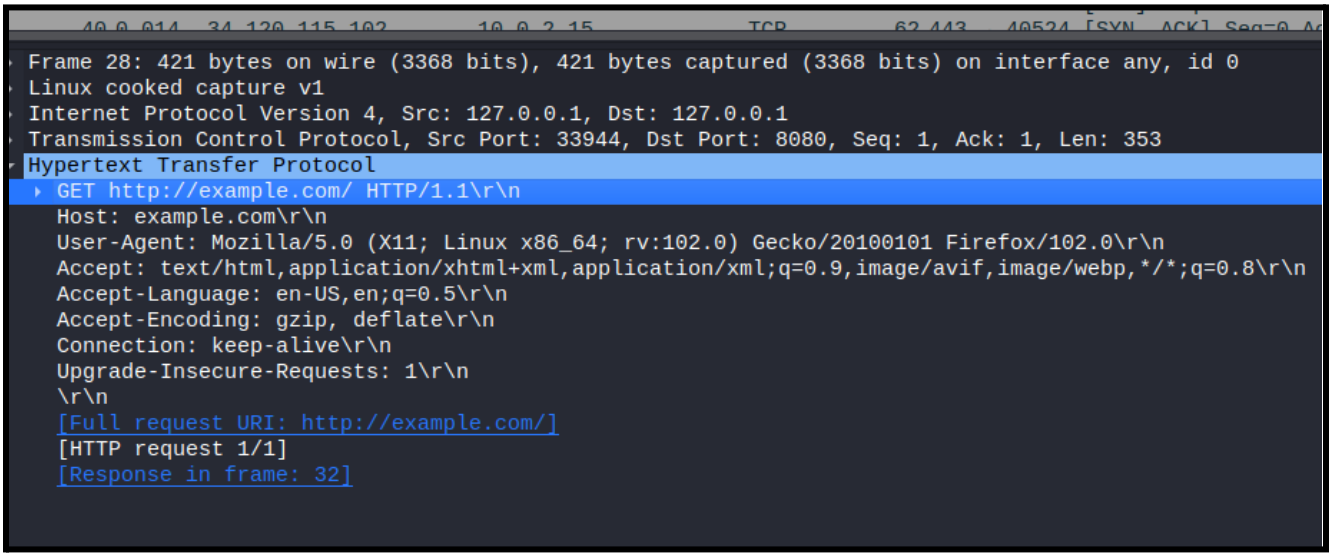
2. Sending GET Request:

- A GET request is sent over the established TCP connection to the proxy.

We can double-check our idea by examining a packet capture that I took while making the request. The first image you see displays the actual GET request sent to our proxy.

GET request from client to proxy server:

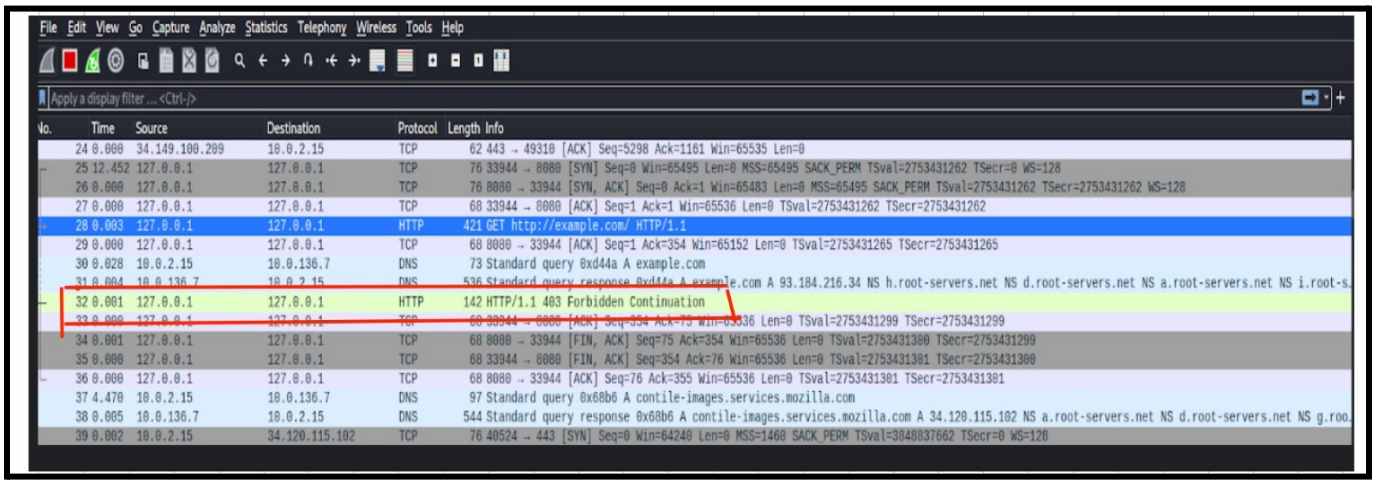


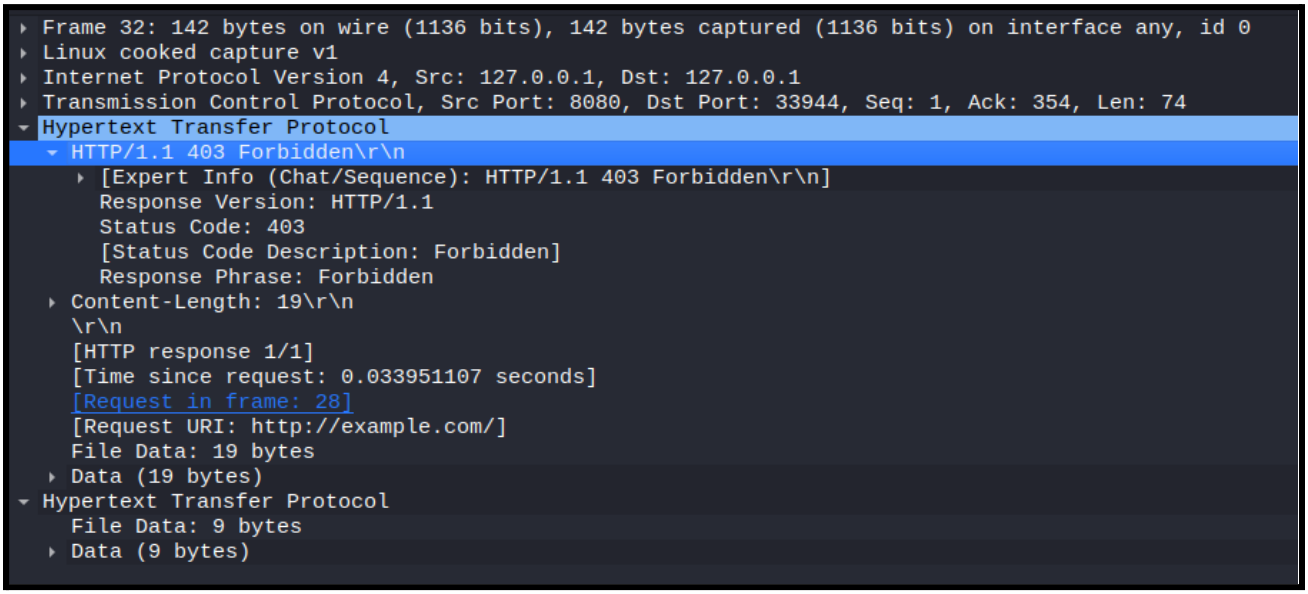


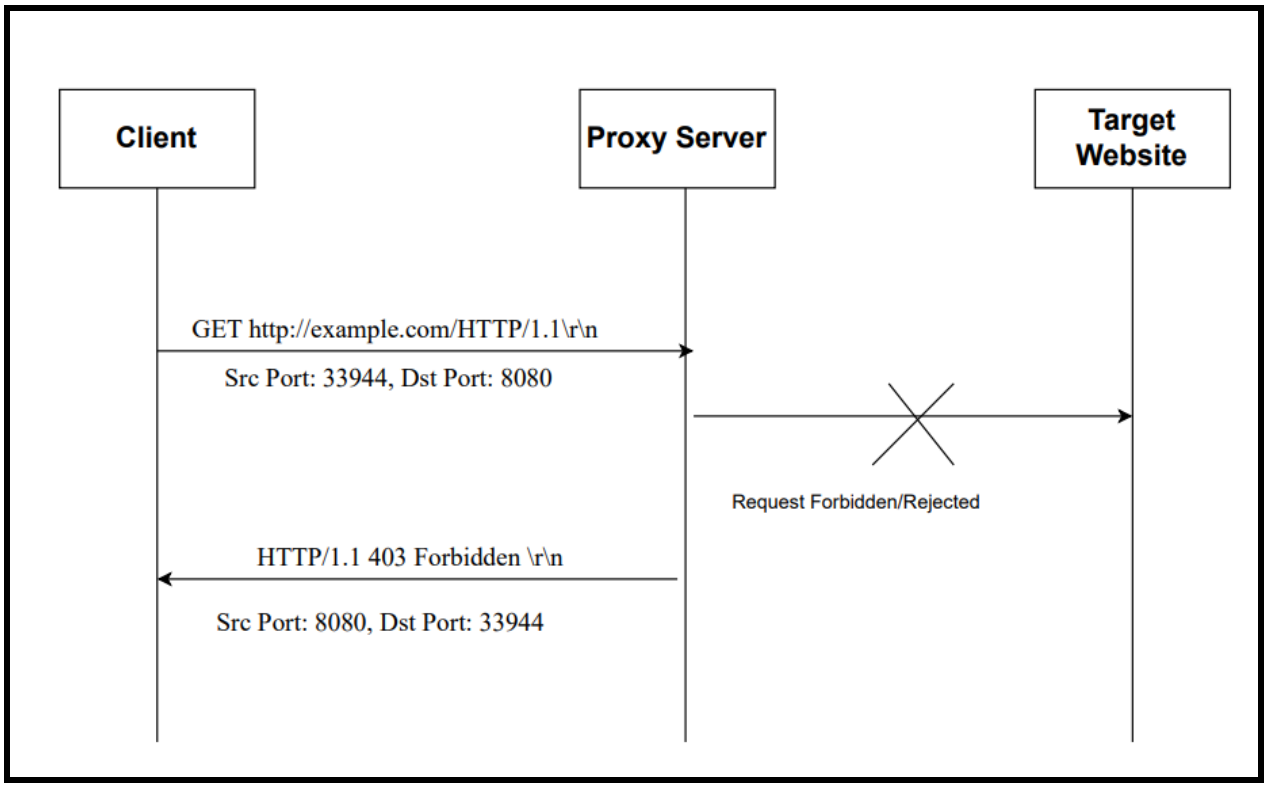
3.Proxy Functionality:

- custom HTTP proxy processes the incoming GET request.
- It is not able to generate its own GET request and forwards it to the target server (example.com).

Request is not sent from HTTP proxy to server in case of blocklisted URL:







b. In case of non-blocklisted URL:

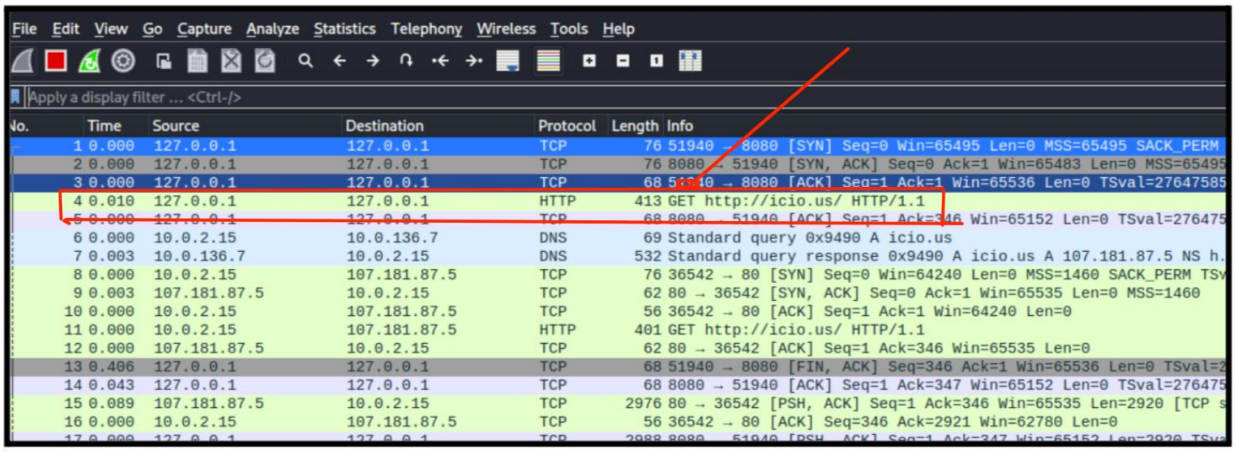
1.TCP Connection Establishment:

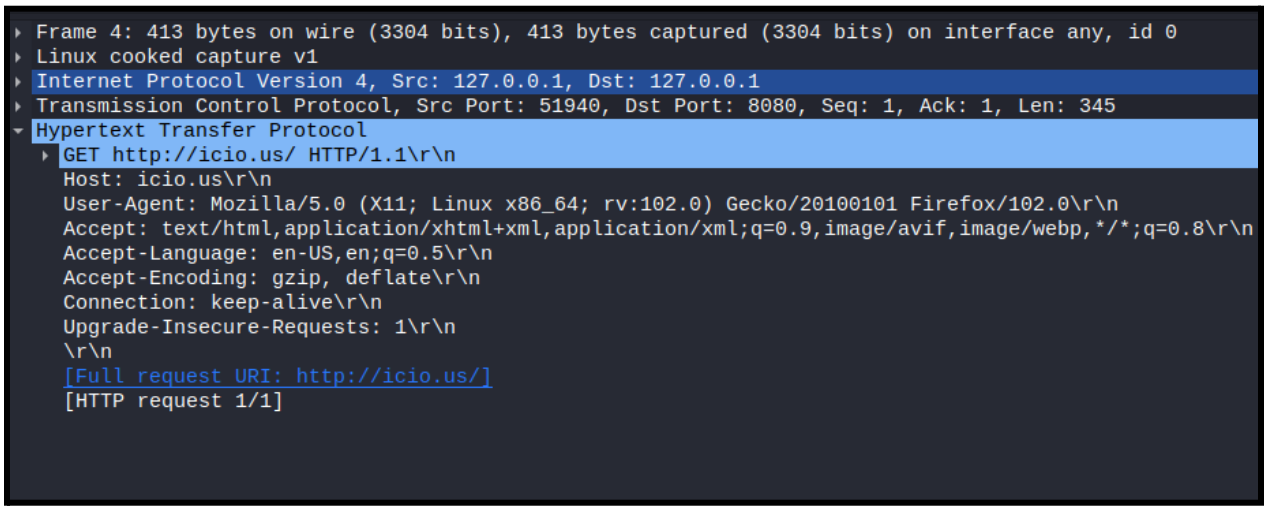
- TCP connection is established with the custom HTTP proxy.
- Connection details: Local host (127.0.0.1) and destination port (8080).

2.Sending GET Request:

- A GET request is sent over the established TCP connection to the proxy.

GET request from client to proxy server:

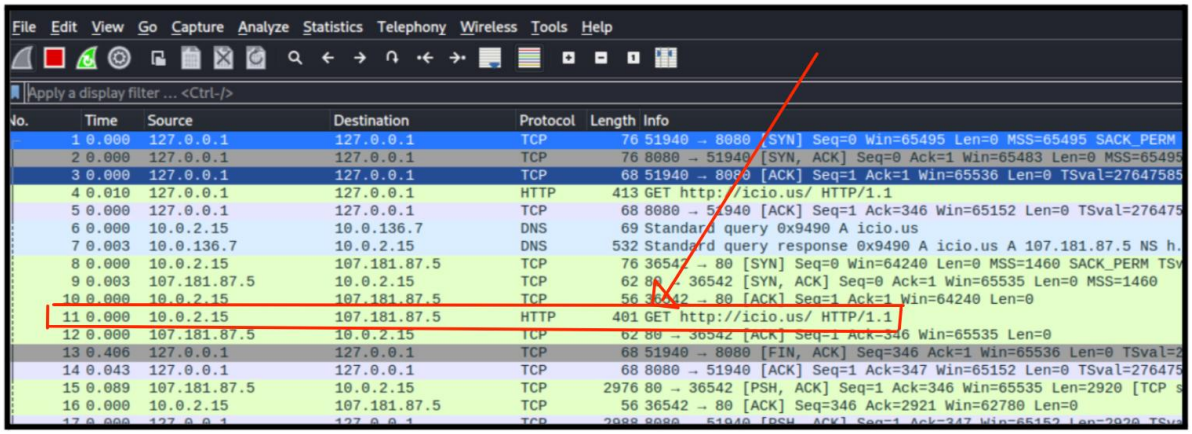


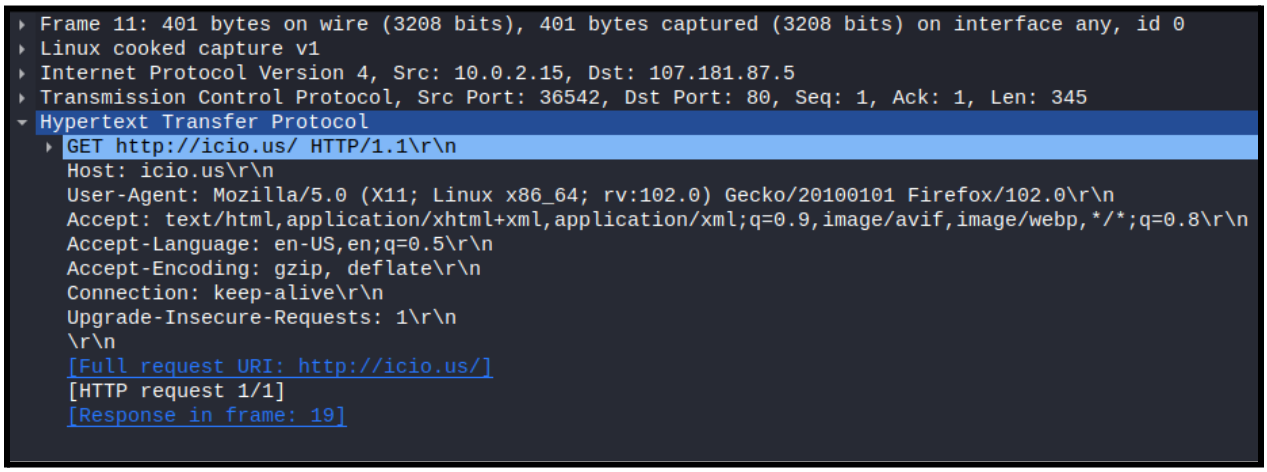


3.Proxy Functionality:

- custom HTTP proxy processes the incoming GET request.
- It then generates its own GET request and forwards it to the target server (icio.us).

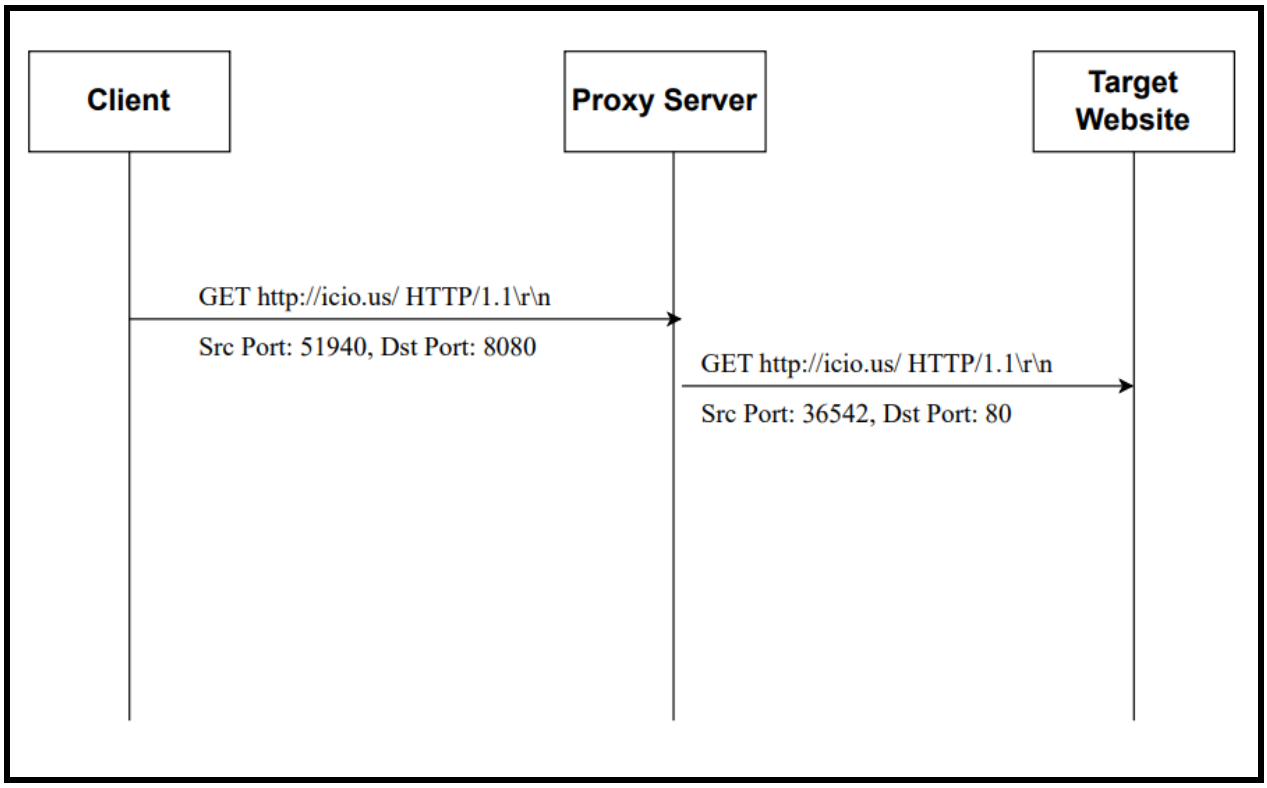
GET request is sent from HTTP proxy to target server in case of non-blocklisted URL:





4.Response Handling:

- The response from icio.us is received by the proxy.
- The proxy processes the response, adds extra headers if needed, and prepares it for forwarding.



5.Response Sent to Client:

- The processed response is sent back to the client over the established TCP connection.

Limitations for using the above approach:

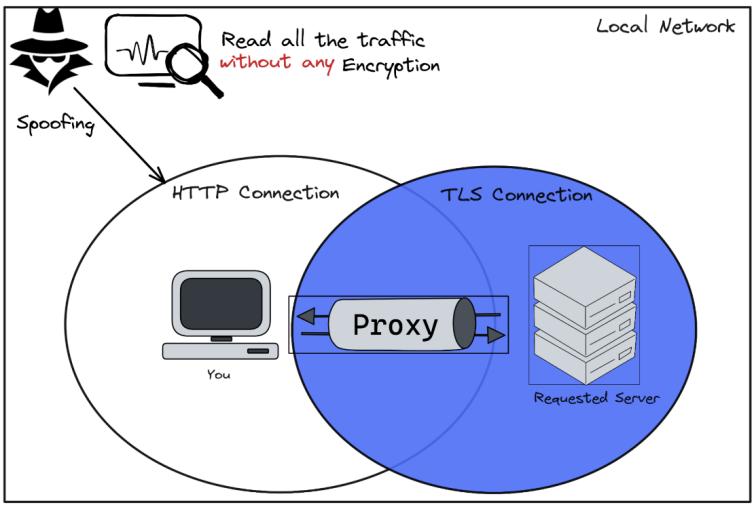
1. **Limited to HTTP**: The proxy server in the provided code is designed to handle HTTP traffic only. It parses HTTP requests and forwards them to the target server. It does not handle HTTPS traffic, as the code does not include mechanisms for decrypting and inspecting encrypted SSL/TLS traffic. This limitation is due to the secure nature of HTTPS, where the contents of the communication are encrypted, and transparent interception would require additional steps, such as SSL/TLS decryption.
2. **Basic URL parsing**: The code performs basic parsing to extract the URL from the "Host" field in the HTTP request. It may not handle all possible variations and edge cases of HTTP requests.

HTTPS Tunnelling using HTTP CONNECT method :

Understanding the Risks of Using an HTTP Proxy Without Encryption:

When you use an HTTPS proxy, all the data passing through it is visible to the proxy. If you completely trust the proxy server, this might not be a concern.

However, if the proxy server is using a non-TLS (non-secure) connection, it implies that all your data is exposed on the network. Many local proxies operate over HTTP without a secure TLS connection. Developers often use such proxies for debugging or intercepting requests.

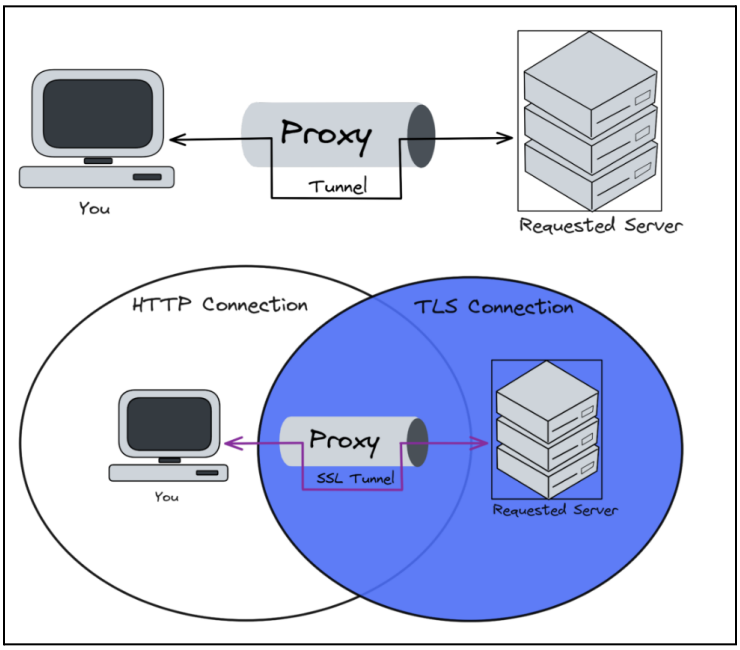


In the scenario illustrated above, even if the destination you're connecting to uses a secure TLS connection, the request is sent to the proxy using plain HTTP. This means that anyone on your local network can intercept and read the data packets. This situation becomes especially risky when using public Wi-Fi networks.

That's why opting for HTTP Tunneling is a more favourable approach and also, at present, the most widely used approach when transferring data packets over an HTTP Proxy.

HTTP CONNECT (Tunnelling) approach:

This approach also involves building a proxy server. Handling the HTTP requests involves parsing requests and passing such requests to the server handling the blacklisting part, and forwarding it back to the client. For that, only a built-in HTTP server and client are required. On the other hand, HTTPS is different as it requires a technique called HTTP CONNECT tunnelling. First, the client must send a request using the HTTP CONNECT method to set up the tunnel between the client and the destination server. When a tunnel consisting of two TCP connections is ready, the client starts a regular TLS handshake with the destination server through the tunnel passing via the proxy to establish a secure connection and later send requests and receive responses.

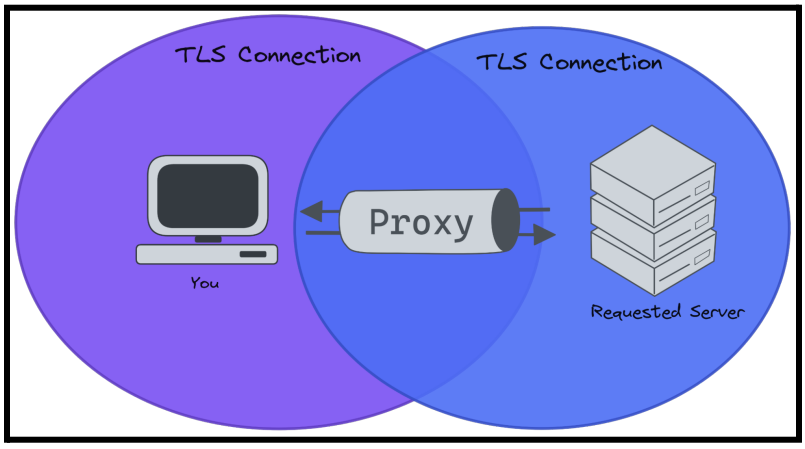


Let's say the client wishes to communicate with the server over Websockets or HTTPS. The client is aware that proxies are used. Simple HTTP requests cannot be used since the client needs to establish a secure connection with the HTTPS server or wants to use another protocol over a TCP connection (WebSockets). The technique that we can use in such cases is HTTP connect tunnelling. It tells the proxy server to establish a TCP connection between the proxy server and

the destination server. Hence, the proxy server won't terminate the SSL, but it simply forwards the data between the client and the destination server so that they can establish a secure connection.

Steps:

1. First, we need to accept the incoming client connections. For each client, intercept the initial part of the request to identify if it's an HTTP CONNECT request.
2. When a client sends an HTTP CONNECT request, we extract the destination address from the request and implement SSL/TLS handling to correctly manage HTTPS proxy traffic. This is the address the client wants to establish a connection with.
3. After that, we check whether the destination address is in a predefined blacklist. If the destination is found in the blacklist, we respond to the client with a message indicating that the request is forbidden.
4. If the destination is not blacklisted, we establish a connection to the destination server. This involves creating a new socket and connecting to the destination server on the specified port.
5. Once the connection to the destination server is established, there is a need to set up bidirectional communication between the client and the destination.



Limitations for using the above approach:

1. **Limited to HTTP CONNECT Method:** The approach relies on the HTTP CONNECT method for tunnelling through the proxy. Some applications may not use this method, so the proxy might not effectively filter all types of traffic.
2. **No support for HTTP/2 or higher versions:** While HTTP/1.1 messages are plain text and relatively easy to parse, HTTP/2 messages are binary. Also, HTTP/2 uses header compression, allowing more efficient use of network resources. Due to the above reasons, the above implementation does not work for HTTP/2 or higher versions of HTTP.

Some Additional Features Implemented:

1. Implemented LRU cache for caching requests and their responses. The proxy caches the most recent requests on the proxy server itself to improve the performance of the network.
2. Customisable blocklist of URLs (a - add new URL to list, r - remove existing URL from list, l - view the current list)
3. Command line based arguments passed to customise the functionality of the Proxy server (‘-p’ choose a port to listen on, q - quit the proxy anytime (closes the proxy port))

Video Presentation Link:
https://drive.google.com/file/d/1REiO-XjlKh5cazJkbwQ3OILfIYdCKT07/view?usp=sharing

References:

1. “HTTP Proxy Tunneling.” *Rafael Gonzaga*, https://blog.rafaelgss.com.br/http-proxy-tunneling Accessed 12 November 2023.
2. Łowicki, Michał. “HTTP(S) Proxy in Golang in less than 100 lines of code.” *Medium*, https://medium.com/@mlowicki/http-s-proxy-in-golang-in-less-than-100-lines-of-code-6a51c2f2c38c Accessed 12 November 2023.
3. *Why No HTTPS? The World's Largest Websites Not Redirecting Insecure Requests to HTTPS*, https://whynohttps.com/ Accessed 12 November 2023.
4. HTTP CONNECT method documentation for how to use: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/CONNECT Accessed 12 November 2023.

5. Kamath, Aditya. “curl and proxies | by Aditya Kamath | Medium.” *Aditya Kamath*, 9 July 2021, https://akamath32.medium.com/curl-and-proxies-b4445feb58db . Accessed 14 November 2023.