Report | Lab 6

CS378 - Computer Networks Lab

Omkar Shirpure (22B0910)

Contents

1 Part 1: Investigating HTTP Traffic with Wireshark

a

The client's browser is running `HTTP/1.1` and the server's browser too, is running `HTTP 1.1` version.

b

The client's IP address is `192.168.1.102` and the server's IP address is `128.119.245.12`

c

The status code returned from the server to the client's browser is `200 (OK)`.

d

The HTML file which the client is retrieving was last modified at `23 Sep 2003 05:29:00 GMT` at the server.

e

A total of `73 bytes` of content is being returned to the client's browser. (This doesn't include the size of headers, along with which it is a total of 439 `bytes`)

2 Part 2: Basic Authentication over HTTP

a

The first response received from the server was `HTTP/1.1 401 Authorization Required`

This happens because the GET request for the /ethereal-labs/protected_pages/lab2-5.html from the client is protected and requires Authorization.

b

The second request succeeds because the second request from the client contains an additional field as `Authorization: Basic ZXRoLXN0dWRlbnRzOm5ldHdvcmtz`
which satisfies the requirements to access the protected file i.e., credentials as
`Credentials: eth-students:networks`.

c

The credentials received are
`Credentials: eth-students:networks`.

Since the ID/username and password are separated by ':'
username : **eth-students**
password : **networks**

3 Part 3: Analysing and Decrypting TLS Traffic

Public-key cryptography basically allows two parties to communicate securely without prior contact by using a pair of keys: a public key and a private key. The sender encrypts a message using the recipient's public key, ensuring that only the recipient can decrypt it with their private key. This process protects the message from eavesdroppers.

Also, the sender can sign the message with their private key, allowing the recipient to verify its authenticity with the sender's public key. This establishes a secure communication channel over a public network.

a

Ciphers common between both SSL end points:
`ECDHE-RSA-AES256-GCM-SHA384`

b

The protocol was used for communication is `TLSv1.2`

c

The master key used was
`Master-Key: 9F9A0F19A02BDDBE1A05926597D622CCA06D2AF416A28AD9C03163B87FF1B0C6 7824BBDB595B32D8027DB566EC04FB25`

4 Part 4: Capturing Your Own Traffic

a HTTP Capture

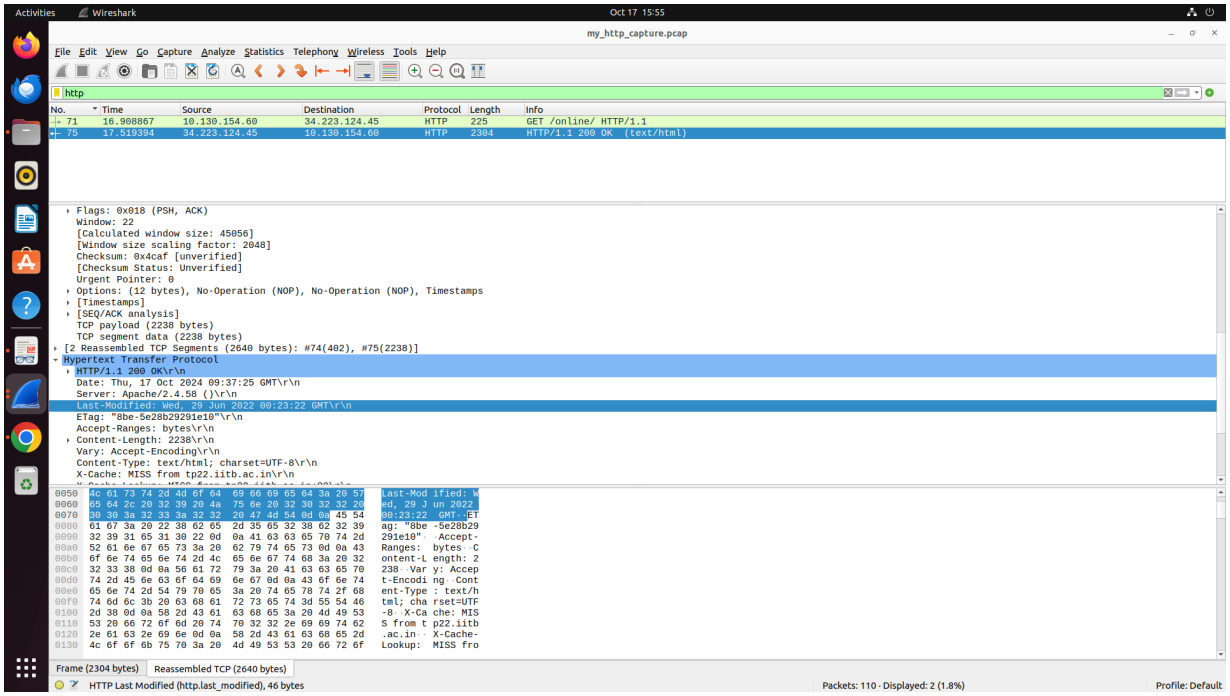

Figure 1: HTTP capture analyzed in Wireshark showing the last modified time.

The last modified time was
`Wed, 29 Jun 2022 00:23:22 GMT`

This can be seen in the HTTP section of the response packet from the server to the original GET request from client, along with the other parameters such as `response code`, `message`, `server`, etc in Wireshark.

b UDP Capture

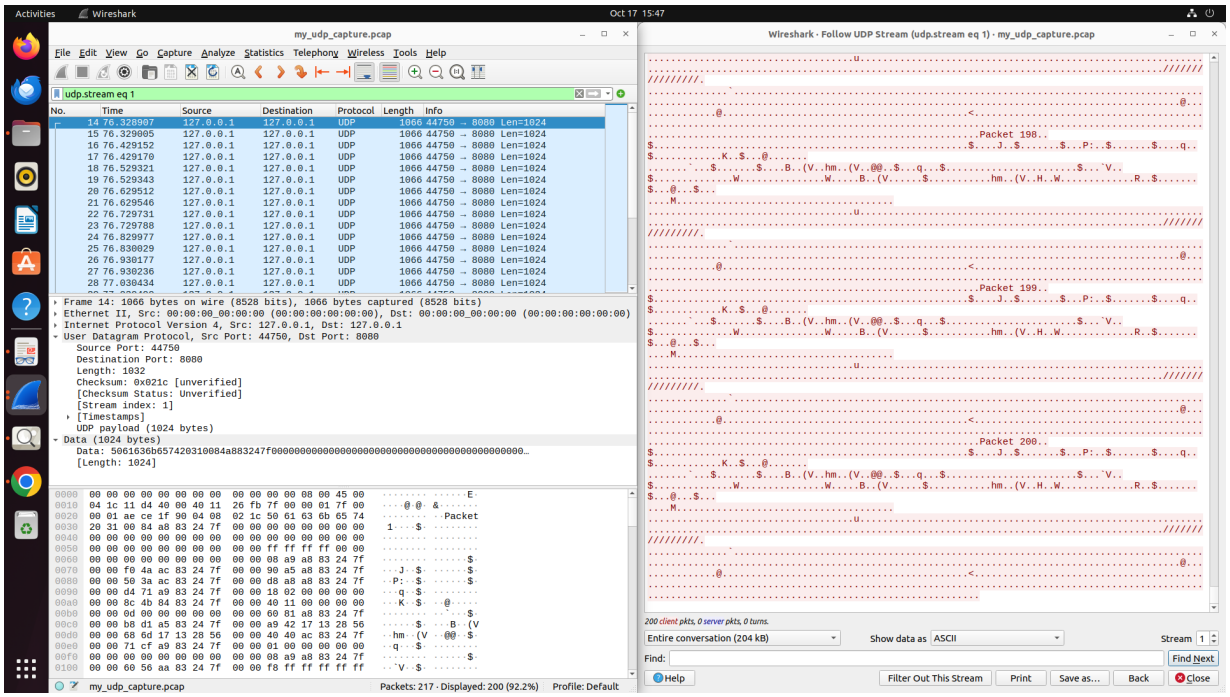

Figure 2: UDP capture in Wireshark showing the packets and data within packets

The UDP capture is run in the loop-back interface for 100 pairs of packets (a total of 200 packets). The above screenshot shows the packet data when followed the UDP stream.

5 Part 5: Using Scapy for Packet Manipulation

a ICMP Echo Request

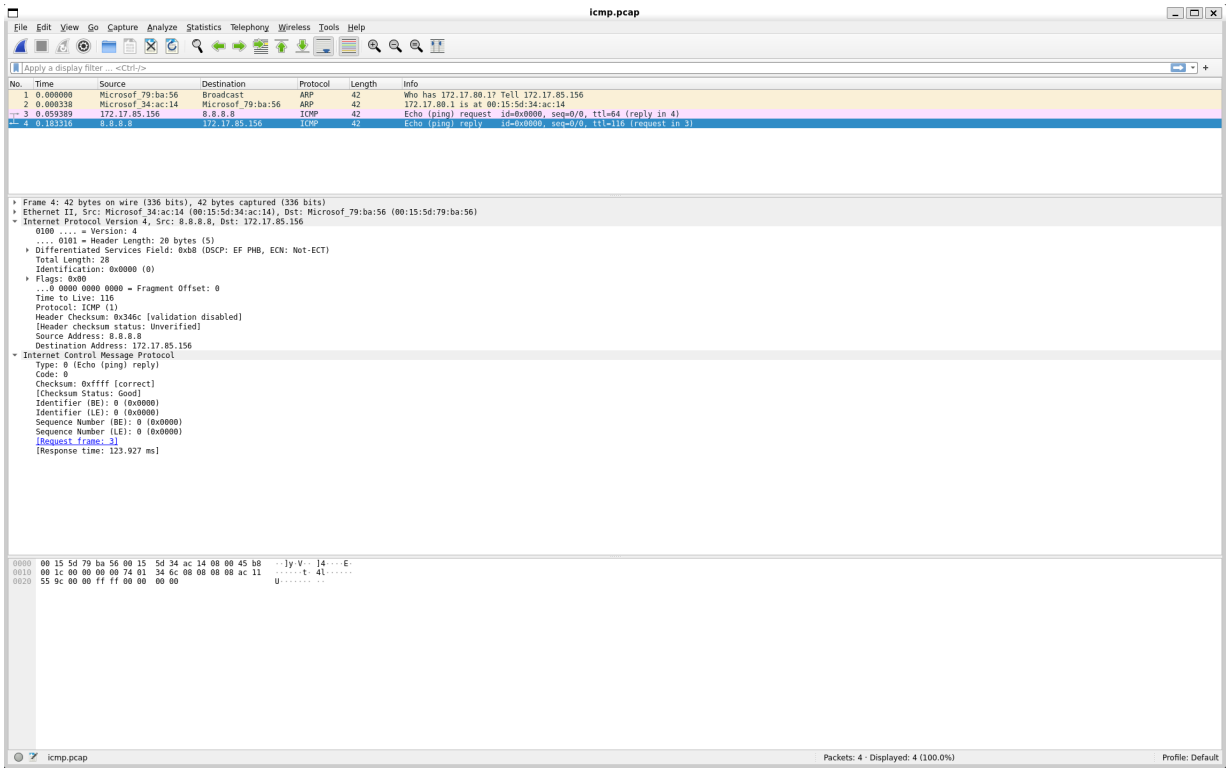

Figure 3: ICMP Echo request and response

```
omkar@OS:/mnt/c/Users/omkar/Desktop/Sem5/CS378-Networks_Lab/Lab06/Submission$ sudo python3 scapy_icmp.py
Sending ICMP Echo Request to 8.8.8.8...
Received ICMP Echo Reply from 8.8.8.8:
###[ IP ]###
  version   = 4
  ihl       = 5
  tos       = 0xb8
  len       = 28
  id        = 0
  flags     =
  frag      = 0
  ttl       = 116
  proto     = icmp
  chksum    = 0x346c
  src       = 8.8.8.8
  dst       = 172.17.85.156
  \options   \
###[ ICMP ]###
     type      = echo-reply
     code      = 0
     chksum    = 0xffff
     id        = 0x0
     seq       = 0x0
     unused    = b''
```

Figure 4: ICMP Echo response received

There are two ICMP packets going to and fro within 172.17.85.156 and 8.8.8.8. One of them being sent from 172.17.85.156 to 8.8.8.8 which is ICMP echo ping request. We then receive an ICMP echo ping reply from the target IP address 8.8.8.8.

b DNS query

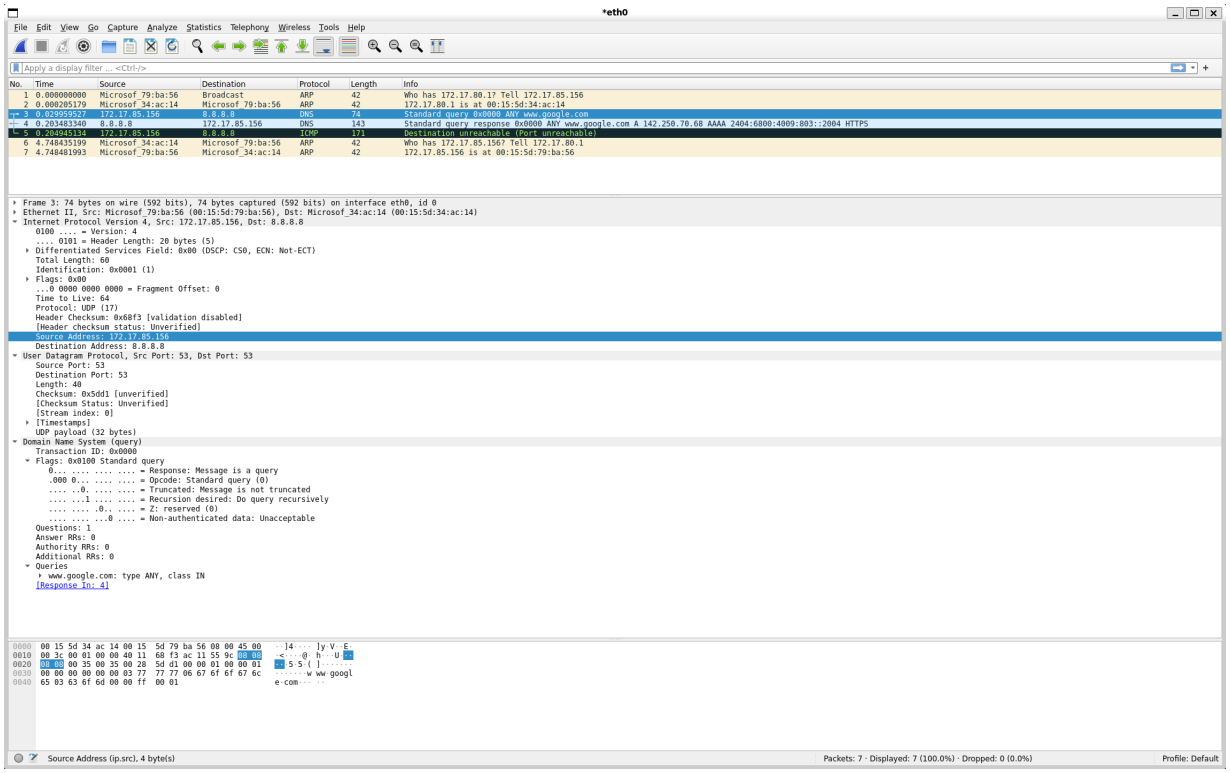

Figure 5: DNS query request and response highlighted

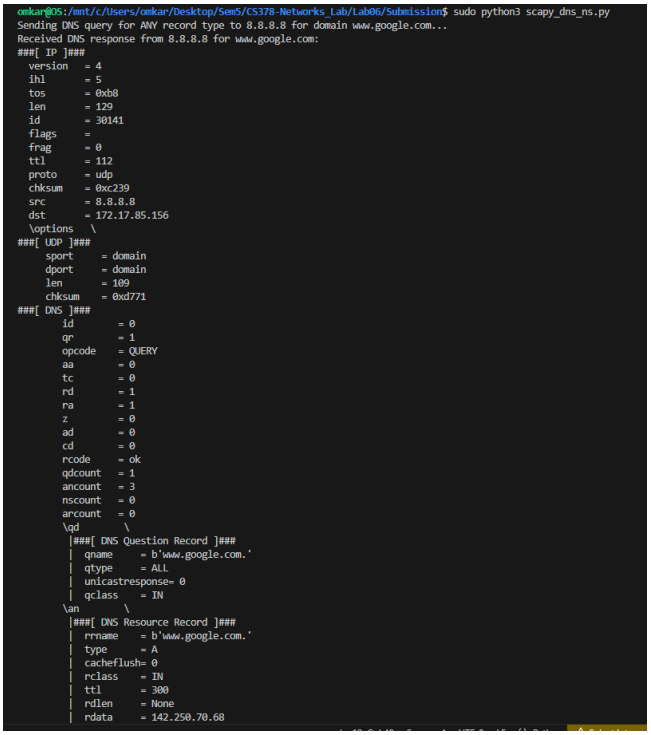

Figure 6: DNS query response received

We send a DNS query request with an non-standard record type (ANY in this case). In the highlighted part in the screenshot, we can see the `ANY` record type. We also receive a DNS response from the targeted IP address.

c TCP SYN packet with a spoofed source IP address

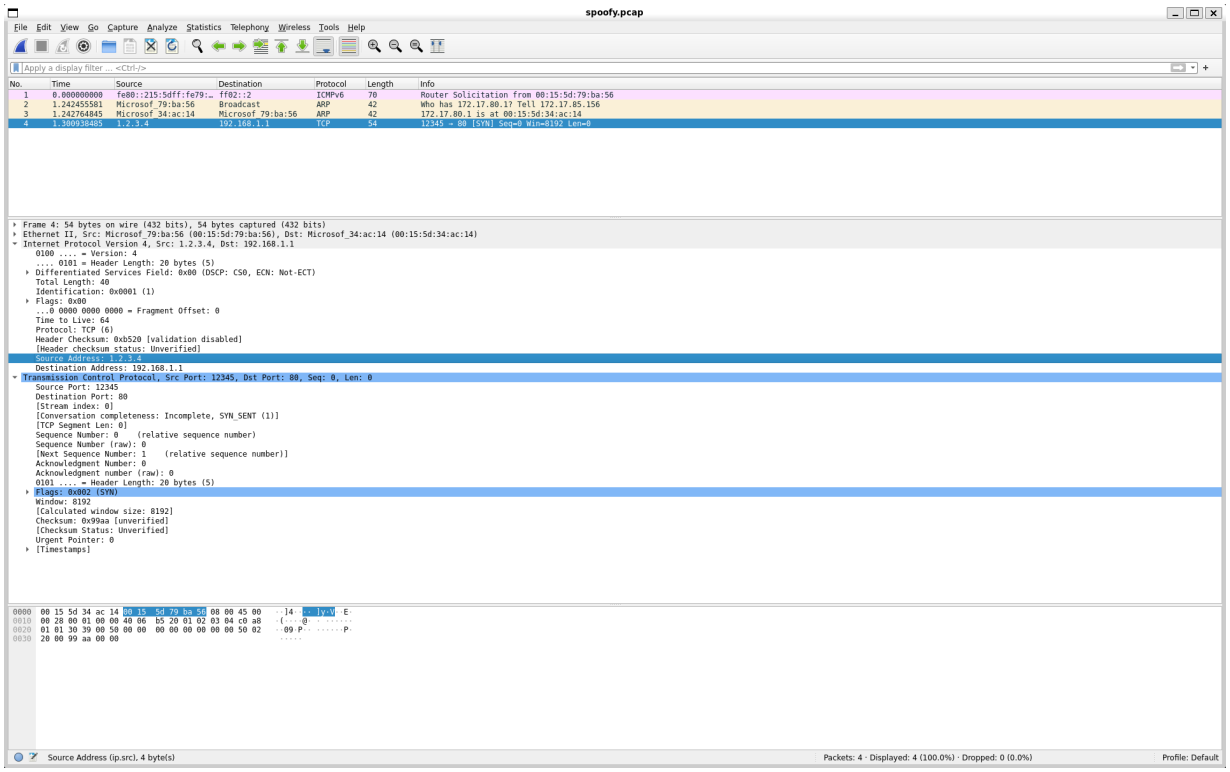

Figure 7: TCP packet with a spoofed source IP address (`1.2.3.4`)

As it can also be seen in the code, we send the TCP packet with spoofed IP address `1.2.3.4` to a random (choosable) destination. We can also verify that in the source address of the captured packet that we sent (highlighted in the above screenshot).