# Lab Assignment-1

## Part 1

### Wireshark

Wireshark is a software tool that can capture and examine *packet traces*. A packet trace is a record of traffic at a location on the network, that is, the traffic seen by some network interface (e.g., an Ethernet or WiFi adapter). The trace is a log of the bits that make up each packet seen, along with timestamps indicating when they were seen. Capturing traces requires root privilege. It's easy enough to do if you have root, so you might want to try it on your own machine. For this assignment, we start from an already captured trace file and use Wireshark only to help investigate its contents.

### Trace File

Uploaded in Shared drive and Moodle

### Warm-up: Inspecting the Trace

The Wireshark GUI has three main sections, as shown in the figure below. In the top panel is a list of the packets in the trace. The bottom two panels show details on a single packet, selected by clicking on one in the top panel. The middle panel shows header fields - each protocol layer adds a header that encapsulates the information passed down by the layer above. The bottom panel shows the raw bytes that make up the packet. (Note that we're using "packet" as a general term here. Strictly speaking, a unit of information at the link layer (which is what is captured in the trace) is called a *frame*. At the network layer (IP), the unit is called a *packet*, at the transport layer a *datagram* or a *segment* (depending), and at the application layer a *message*. We'll often use packet out of habit, though.)

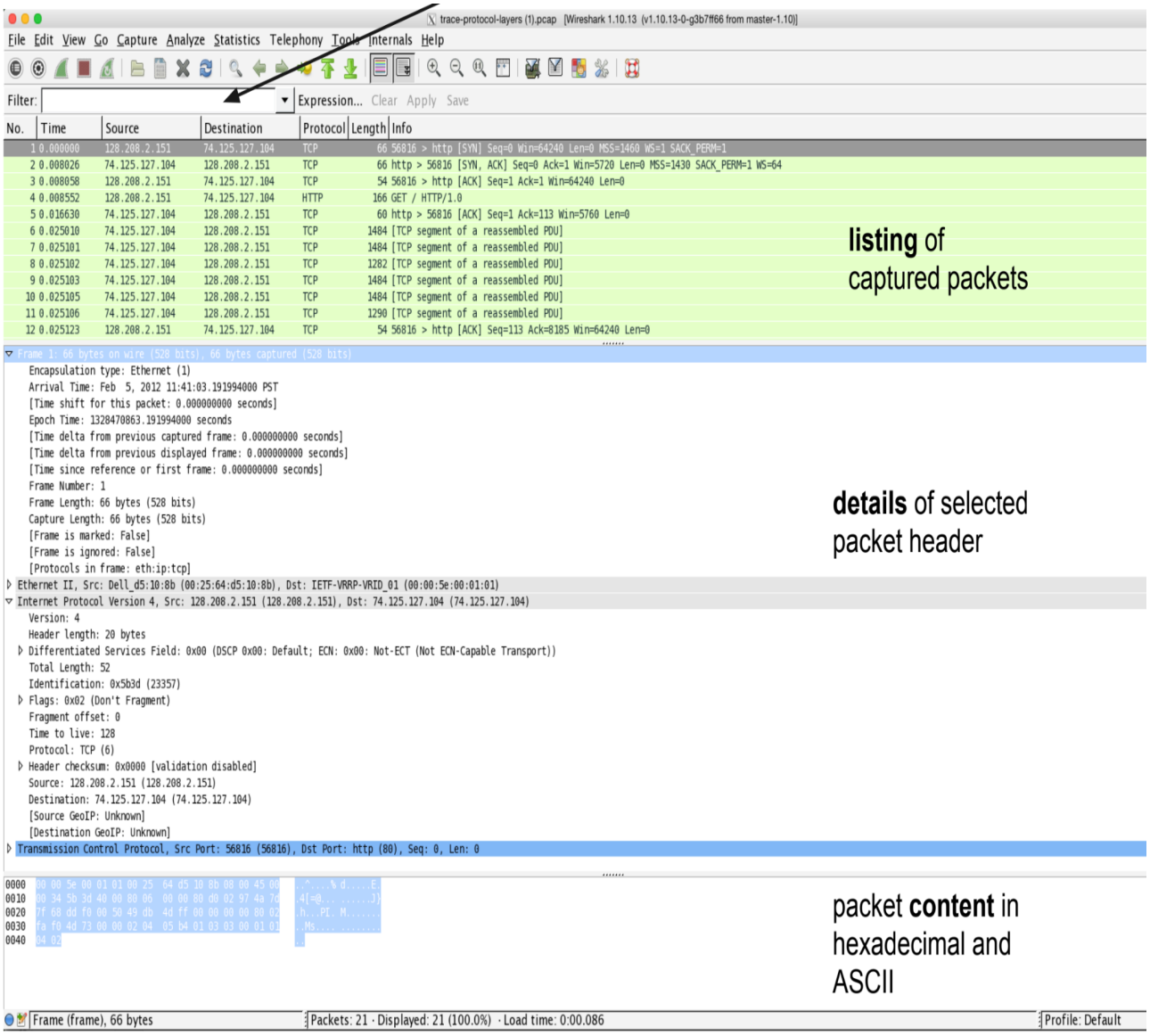


Figure 1: Example Wireshark session

Open the trace file in Wireshark. Select a packet for which the Protocol column is HTTP and the Info column says it is a GET. This is a packet that carries a web (HTTP) request, for instance as sent from your browser to a web server. Let's have a closer look to see how the packet structure reflects the protocols that are in use.

Since we are fetching a web page, we know that the protocol layers being used are as shown below. That is, HTTP is the application layer protocol used to fetch URLs. Like many Internet applications, it runs on top of the TCP transport layer, which itself runs on top of the IP network layer. IP runs on top of some link/physical layer protocols, depending on the physical network.

These are typically combined by Wireshark and displayed as Ethernet, if the trace was captured on a wireless interface, or 802.11, if it was captured on a wireless interface.

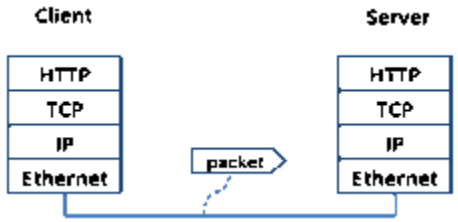


Figure 2: Protocol stack for a web fetch

With the HTTP GET packet selected you can examine the protocol header for each layer, using the middle panel. You can expand the information for each layer by clicking on the + expander or icon to see details about the information it provides.

- The first Wireshark block is the Frame. Its bits (header plus *payload* (data)) are all the bits of that transmission unit. Expanding the Frame line (in the middle panel) provides overall information about the packet, including when it was captured and how many bits long it is.
- The second block is Ethernet. Expanding it provides details about the contents of this frame's Ethernet header.
- Then come IP, TCP, and HTTP, in that order. Each layer transmits a header, used to communicate between the protocol agents at that layer at each end, and encapsulates the data and headers of the layers above. Note that the order of the headers is from the bottom of the protocol stack upwards. This is because as packets are passed down the stack, the header information of each lower layer protocol is added to the front of the information from the higher layer protocol. That is, the lower layer protocols come first in the packet on the wire.

Now find another HTTP message, the response from the server to your computer, and look at the structure of its packets. The packet whose Info field contains "200 OK" is the final packet carrying the server's HTTP response message. (That is, the HTTP response is so large that it is carried as many network packets.) In our trace, there are two blocks synthesized by Wireshark and shown in its middle panel, as seen in the next figure. These blocks provide you, the Wireshark user, with potentially useful information, but the information they contain is not encoded in this particular packet (or, at least, not fully contained in this packet).

- The first extra block, "[11 reassembled TCP segments ]", is a "reassembly" of the complete HTTP response using all the packets into which it was broken. Each of these packets is shown earlier in the trace.
- The second extra block, "Line-based text data", describes the HTTP data contained in the HTTP message. (The full message is an HTTP header plus this data.) In our case the data is of type text/html. (Wireshark understands this by parsing the HTTP header, the same way that your browser understands it.) The data is displayed using a format appropriate for its type.

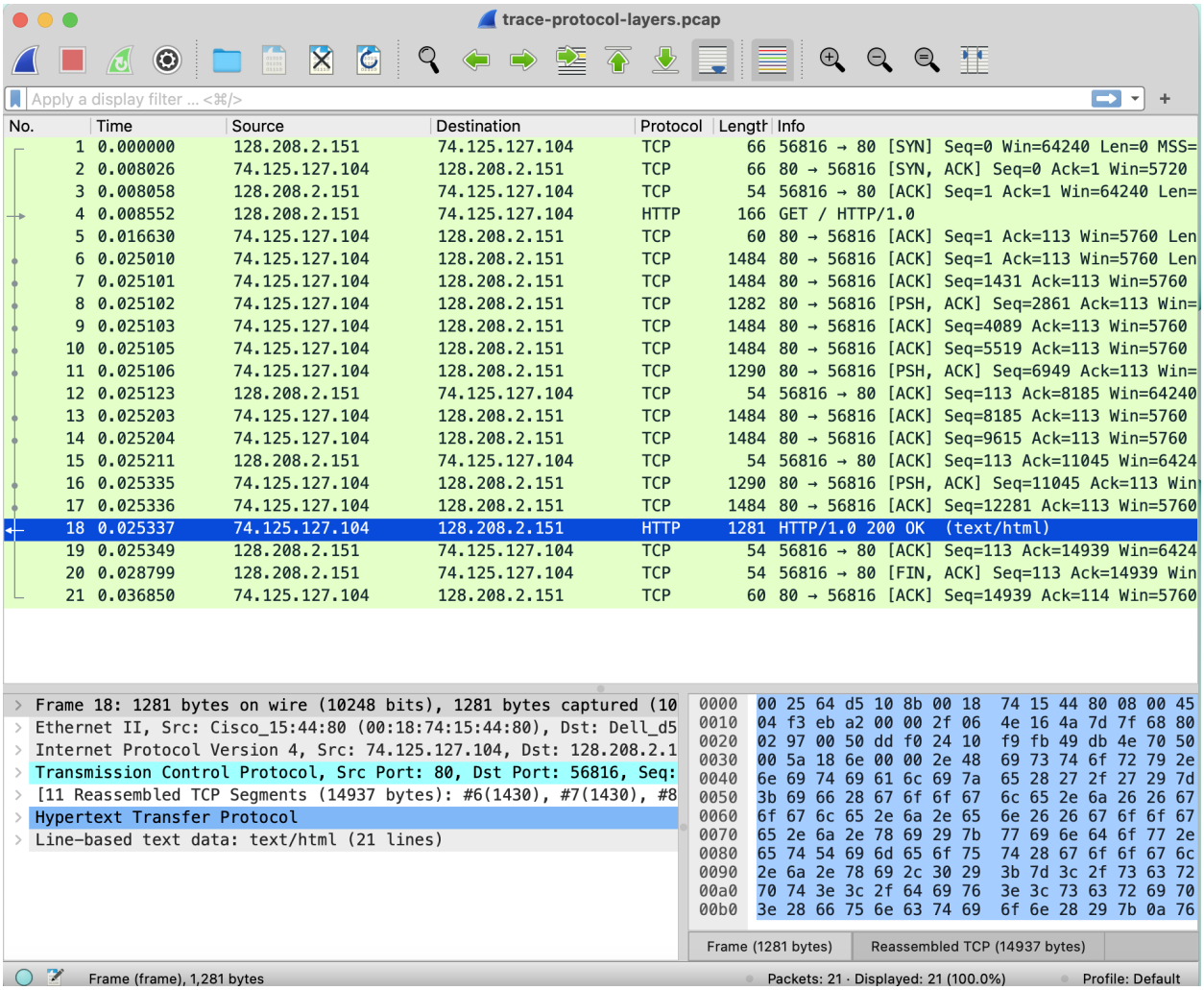


Figure 3: Inspecting a HTTP 200 OK response

### Question 1: Packet Structure Diagram

Draw a figure of the HTTP GET packet (packet 4 in the trace) that shows the position and size in bytes of the HTTP, TCP, IP and Ethernet protocol headers. Your figure can simply show the

overall packet as a long, thin rectangle. Leftmost elements are the first sent on the wire. On this drawing, show for each protocol layer the byte range containing the protocol header. If the topmost layer has data, it will be contained in the final segment of the packet. In that case, show its byte range as well. (So, your diagram partitions the bytes of the packets into many protocol headers and possibly one data segment.)

To work out sizes, observe that when you select a protocol block in the middle panel by clicking on it, Wireshark highlights the bytes it corresponds to in the packet in the lower panel and displays their length at the bottom of the window.

Q1: Hand in your packet drawing.

**Questions 2-3: Protocol Overhead**

Estimate the download protocol overhead, or percentage of the download bytes taken up by protocol overhead. To do this, consider HTTP data (headers and message) to be useful data for the network to carry, and lower layer headers (TCP, IP, and Ethernet) to be the overhead. We would like this overhead to be small, so that most bits are used to carry content that applications care about. To work this out, first look at only the packets in the download direction for a single web fetch. (The GET travels upstream. The other direction is downstream.) The packets should start with a short TCP packet described as a SYN ACK, which is the beginning of a connection. They will be followed by mostly longer packets in the middle (of roughly 1 to 1.5KB), of which the last one is an HTTP packet. This is the main portion of the download. And they will likely end with a short TCP packet that is part of ending the connection. For each packet, you can inspect how much overhead it has in the form of Ethernet / IP / TCP headers, and how much useful HTTP data it carries in the TCP payload. You may also look at the HTTP packet in Wireshark to learn how much data is in the TCP payloads over all download packets.

Q2: Estimate the download protocol overhead on packet 7 in the given trace.

Q3: Estimate the download protocol overhead for the entire HTTP response, as defined above.

**Question 4-6: Demultiplexing Keys**

When an Ethernet frame arrives at a computer, the Ethernet layer must hand the packet it contains to the next higher layer to be processed. There can be many "next higher layers" installed on any particular system, and the act of finding the right one to hand any particular incoming packet to is called *demultiplexing*. We know that in our case the higher layer is IP. But how does the Ethernet protocol know this? We have the same issue at the IP layer -- IP must be able to determine that the contents of an IP message is a TCP packet so that it can hand it to the TCP protocol to process. The answer is that protocols use fields in their headers indicating what the next higher level protocol is. These fields, called *demultiplexing keys*, are filled in by the protocol layer on the sender side and are read by the protocol layer on the receiving side, since

the path up through the layers on the receiver should be the same as the path down through the layers on the sender.

Look at the Ethernet and IP headers of a download packet in detail to answer the following questions:

Q4: Which Ethernet header field is the demultiplexing key indicating that the next higher layer is IP? What value is used in this field to indicate IP?

Q5: Which IP header field is the demultiplexing key indicating that the next higher layer is TCP? What value is used in this field to indicate TCP?

Q6: Why doesn't TCP's header contain a demultiplexing key? How can TCP know to deliver the data to HTTP on the receiving side?

## Part 2

**The Basic HTTP GET/response interaction**

Let's begin our exploration of HTTP by downloading a very simple HTML file - one that is very short, and contains no embedded objects. Do the following:

1. Start up your web browser.
2. Start up the Wireshark packet sniffer, as described in the Introductory lab (but don't yet begin packet capture). Enter "http" (just the letters, not the quotation marks, and in lower case) in the display-filter-specification window, so that only captured HTTP messages will be displayed later in the packet-listing window. (We're only interested in the HTTP protocol here, and don't want to see the clutter of all captured packets).
3. Wait a bit more than one minute (we'll see why shortly), and then begin Wireshark packet capture.
4. Enter the following to your browser
   http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file1.html
   Your browser should display the very simple, one-line HTML file.
5. Stop Wireshark packet capture.

Your Wireshark window should look similar to the window shown in Figure 4.

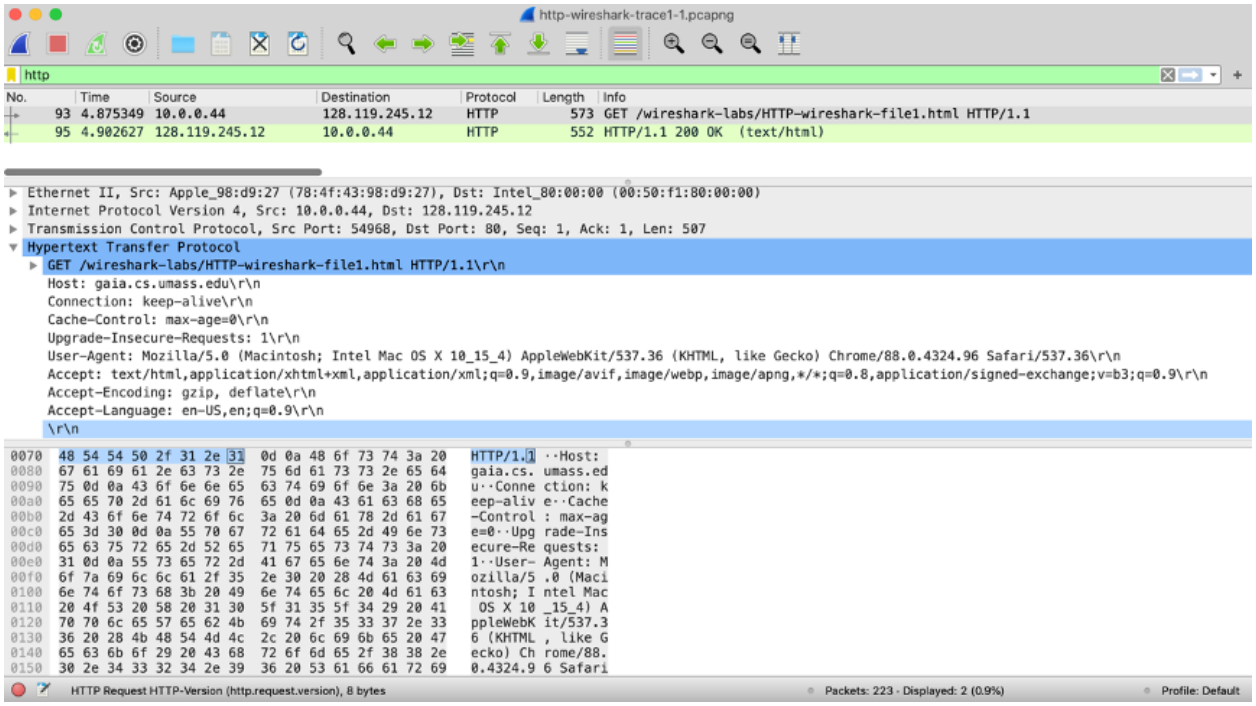


**Figure 4:** Wireshark Display after http://gaia.cs.umass.edu/wireshark-labs/ HTTP-wireshark-file1.html has been retrieved by your browser

The example in Figure 4 shows in the packet-listing window that two HTTP messages were captured: the GET message (from your browser to the gaia.cs.umass.edu web server) and the response message from the server to your browser. The packet-contents window shows details of the selected message (in this case the HTTP OK message, which is highlighted in the packet-listing window). Recall that since the HTTP message was carried inside a TCP segment, which was carried inside an IP datagram, which was carried within an Ethernet frame, Wireshark displays the Frame, Ethernet, IP, and TCP packet information as well. We want to minimize the amount of non-HTTP data displayed (we're interested in HTTP here, and will be investigating these other protocols is later labs), so make sure the boxes at the far left of the Frame, Ethernet, IP and TCP information have a plus sign or a right-pointing triangle (which means there is hidden, undisplayed information), and the HTTP line has a minus sign or a down-pointing triangle (which means that all information about the HTTP message is displayed).

By looking at the information in the HTTP GET and response messages, answer the following questions.

1. Is your browser running HTTP version 1.0, 1.1, or 2? What version of HTTP is the server running?
2. What languages (if any) does your browser indicate that it can accept to the server?
3. What is the IP address of your computer? What is the IP address of the gaia.cs.umass.edu server?

4. What is the status code returned from the server to your browser?
5. When was the HTML file that you are retrieving last modified at the server?
6. How many bytes of content are being returned to your browser?
7. By inspecting the raw data in the packet content window, do you see any headers within the data that are not displayed in the packet-listing window? If so, name one.

In your answer to question 5 above, you might have been surprised to find that the document you just retrieved was last modified within a minute before you downloaded the document. That's because (for this particular file), the **gaia.cs.umass.edu** server is setting the file's last-modified time to be the current time, and is doing so once per minute. Thus, if you wait a minute between accesses, the file will appear to have been recently modified, and hence your browser will download a "new" copy of the document.

2. **The HTTP CONDITIONAL GET/response interaction**

Recall from Section 2.2.5 of the text, that most web browsers perform object caching and thus often perform a conditional GET when retrieving an HTTP object. Before performing the steps below, make sure your browser's cache is empty[3]. Now do the following:

- Start up your web browser, and make sure your browser's cache is cleared, as discussed above.
- Start up the Wireshark packet sniffer
- Enter the following URL into your browser
  http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file2.html
  Your browser should display a very simple five-line HTML file.
- Quickly enter the same URL into your browser again (or simply select the refresh button on your browser)
- Stop Wireshark packet capture, and enter "http" (again, in lower case without the quotation marks) in the display-filter-specification window, so that only captured HTTP messages will be displayed later in the packet-listing window.

Answer the following questions:

8. Inspect the contents of the first HTTP GET request from your browser to the server. Do you see an "IF-MODIFIED-SINCE" line in the HTTP GET?
9. Inspect the contents of the server response. Did the server explicitly return the contents of the file? How can you tell?
10. Now inspect the contents of the second HTTP GET request from your browser to the server. Do you see an "IF-MODIFIED-SINCE:" line in the HTTP GET? If so, what information follows the "IF-MODIFIED-SINCE:" header?
11. What is the HTTP status code and phrase returned from the server in response to this second HTTP GET? Did the server explicitly return the contents of the file? Explain.

3. **Retrieving Long Documents**

In our examples thus far, the documents retrieved have been simple and short HTML files. Let's next see what happens when we download a long HTML file. Do the following:

- Start up your web browser, and make sure your browser's cache is cleared, as discussed above.
- Start up the Wireshark packet sniffer
- Enter the following URL into your browser
  http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file3.html
  Your browser should display the rather lengthy US Bill of Rights.
- Stop Wireshark packet capture, and enter "http" in the display-filter-specification window, so that only captured HTTP messages will be displayed.

In the packet-listing window, you should see your HTTP GET message, followed by a multiple-packet TCP response to your HTTP GET request. Make sure your Wireshark display filter is cleared so that the multi-packet TCP response will be displayed in the packet listing.

This multiple-packet response deserves a bit of explanation. The HTTP response message consists of a status line, followed by header lines, followed by a blank line, followed by the entity body. In the case of our HTTP GET, the entity body in the response is the *entire* requested HTML file. In our case here, the HTML file is rather long, and at 4500 bytes is too large to fit in one TCP packet. The single HTTP response message is thus broken into several pieces by TCP, with each piece being contained within a separate TCP segment. In recent versions of Wireshark, Wireshark indicates each TCP segment as a separate packet, and the fact that the single HTTP response was fragmented across multiple TCP packets is indicated by the "TCP segment of a reassembled PDU" in the Info column of the Wireshark display.

Answer the following questions:

12. How many HTTP GET request messages did your browser send? Which packet number in the trace contains the GET message for the Bill or Rights?
13. Which packet number in the trace contains the status code and phrase associated with the response to the HTTP GET request?
14. What is the status code and phrase in the response?
15. How many data-containing TCP segments were needed to carry the single HTTP response and the text of the Bill of Rights?

4. **HTML Documents with Embedded Objects**

Now that we've seen how Wireshark displays the captured packet traffic for large HTML files, we can look at what happens when your browser downloads a file with embedded objects, i.e., a file that includes other objects (in the example below, image files) that are stored on another server(s).

Do the following:

- Start up your web browser, and make sure your browser's cache is cleared, as discussed above.
- Start up the Wireshark packet sniffer
- Enter the following URL into your browser
  http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file4.html
  Your browser should display a short HTML file with two images. These two images are referenced in the base HTML file. That is, the images themselves are not contained in the HTML; instead the URLs for the images are contained in the downloaded HTML file. Your browser will have to retrieve these logos from the indicated web sites.
- Stop Wireshark packet capture, and enter "http" in the display-filter-specification window, so that only captured HTTP messages will be displayed.

Answer the following questions:

16. How many HTTP GET request messages did your browser send? To which Internet addresses were these GET requests sent?
17. Can you tell whether your browser downloaded the two images serially, or whether they were downloaded from the two websites in parallel? Explain.

**5. HTTP Authentication**

Finally, let's try visiting a web site that is password-protected and examine the sequence of HTTP messages exchanged for such a site. The URL **http://gaia.cs.umass.edu/wireshark-labs/protected_pages/HTTP-wireshark-file5.html** is password protected. The username is "wireshark-students" (without the quotes), and the password is "network" (again, without the quotes). So let's access this "secure" password-protected site. Do the following:

- Make sure your browser's cache is cleared, as discussed above, and close down your browser. Then, start up your browser
- Start up the Wireshark packet sniffer

- Enter the following URL into your browser
  http://gaia.cs.umass.edu/wireshark-labs/protected_pages/HTTP-wireshark-file5.html
  Type the requested username and password into the pop up box.
- Stop Wireshark packet capture, and enter "http" in the display-filter-specification window, so that only captured HTTP messages will be displayed later in the packet-listing window.

Now let's examine the Wireshark output. You might want to first read up on HTTP authentication by reviewing the easy-to-read material on "HTTP Access Authentication Framework" at http://frontier.userland.com/stories/storyReader$2159

Answer the following questions:

18. What is the server's response (status code and phrase) in response to the initial HTTP GET message from your browser?
19. When your browser sends the HTTP GET message for the second time, what new field is included in the HTTP GET message?

The username (wireshark-students) and password (network) that you entered are encoded in the string of characters (d2lyZXNoYXJrLXN0dWRlbnRzOm5ldHdvcms=) following the "Authorization: Basic" header in the client's HTTP GET message. While it may appear that your username and password are encrypted, they are simply encoded in a format known as Base64 format. The username and password are *not* encrypted! To see this, go to http://www.motobit.com/util/base64-decoder-encoder.asp and enter the base64-encoded string d2lyZXNoYXJrLXN0dWRlbnRz and decode. *Voila!* You have translated from Base64 encoding to ASCII encoding, and thus should see your username! To view the password, enter the remainder of the string Om5ldHdvcms= and press decode. Since anyone can download a tool like Wireshark and sniff packets (not just their own) passing by their network adaptor, and anyone can translate from Base64 to ASCII (you just did it!), it should be clear to you that simple passwords on WWW sites are not secure unless additional measures are taken.

---

*Source: 1. Computer Networking: A Top-Down Approach, J.F. Kurose and K.W. Ross*

*2. Some part of the assignment is taken from the Computer Networks course offered by Prof Arvind Krishnamurthy. Many thanks to him.*