Wireshark Lab 1: Getting Started and HTTP

Answer the following questions, based on your Wireshark experimentation. [The questions numbered 1 through 20 below should be answered and handed in].

1. List the 3 different protocols that appear in the protocol column in the unfiltered packet-listing window in step h) above.

*Answer:* Some of the protocols listed in the screenshot below are UDP, TCP, ARP, ICMP, MDNS, and STUN. (Note you weren’t asked to do a screenshot, but here is mine):
2. How long did it take from when the HTTP GET message was sent until the HTTP OK reply was received? (By default, the value of the Time column in the packet-listing window is the amount of time, in seconds, since Wireshark tracing began. To display the Time field in time-of-day format, select the Wireshark View pull down menu, then select Time Display Format, then select Time-of-day.)

**Answer:** As shown in the screen shot below (you didn’t have to provide this), the GET was sent at 11.300694 and the reply was received at 11.301658. The delay was thus 0.000964 secs
3. What is the Internet address of the gaia.cs.umass.edu (also known as www.net.cs.umass.edu)? What is the Internet address of your computer?

*Answer:* As shown in the screen shot below (you didn’t have to provide this), the IP address of gaia.cs.umass.edu is 128.119.245.145; the IP address of my laptop is 128.119.66.142

4. Print the two HTTP messages displayed in step j) above. To do so, select *Print* from the Wireshark *File* command menu, and select the “Selected Packet Only” and “Print as displayed” radial buttons, and then click OK. [Note: You can cut and paste the HTTP message text and include this in the writeup that you hand in]

*Answer:* The print out of the two HTTP messages are below:

HTTP GET message:

```plaintext
HTTP GET /wireshark-labs/INTRO-wireshark-file1.html HTTP/1.1
Host: gaia.cs.umass.edu
```
Part II: HTTP

Now that you’ve taken Wireshark out for a test run and have a feel for what it can do, let’s use it to study HTTP!
You’ll now explore several aspects of the HTTP protocol: the basic GET/response interaction, HTTP message formats, retrieving large HTML files, retrieving HTML files with embedded objects, and HTTP authentication and security. Before beginning these labs, you might want to review Section 2.2 of the text.

II.1. 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.

5.  Is your browser running HTTP version 1.0 or 1.1?  What version of HTTP is the server running?
6.  What languages (if any) does your browser indicate that it can accept to the server?
7.  What is the IP address of your computer?  Of the gaia.cs.umass.edu server?
8.  What is the status code returned from the server to your browser?
9.  When was the HTML file that you are retrieving last modified at the server?
10.  How many bytes of content are being returned to your browser?
Answer: The following screen shots showing the HTTP GET and HTTP reply answer these questions:

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

Answer the following questions:

11. 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?
12. Inspect the contents of the server response. Did the server explicitly return the contents of the file? How can you tell?
13. 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?
14. 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.
Answer: Below are a screen shot and the four captures packets (two GETs and two REPLIES, in chronological order) that answer the questions above:

First GET, then a reply, then another identical GET, then a reply (304 not modified)

There is no IF-MODIFIED-SINCE in the first GET
II.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
  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 response to your HTTP GET request. This multiple-packet response
deserves a bit of explanation. Recall from Section 2.2 (see Figure 2.9 in the text) that 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 (see Figure 1.24 in the text). Each TCP segment is
recorded as a separate packet by Wireshark, and the fact that the single HTTP response
was fragmented across multiple TCP packets is indicated by the “Continuation” phrase
displayed by Wireshark. We stress here that there is no “Continuation” message in
HTTP!

Answer the following questions:
  15. How many HTTP GET request messages were sent by your browser?
  16. How many data-containing TCP segments were needed to carry the single HTTP
      response?
  17. What is the status code and phrase associated with the response to the HTTP GET
      request?
  18. Are there any HTTP status lines in the transmitted data associated with a TCP-
      induced “Continuation”?

In our answer below, we use the http-ethereal-trace-3 packet trace file. The HTTP GET
for the long document is packet 8 in the trace (at t=4.623732); the HTTP OK reply is
packet 14 (at t=6.680432).
The HTTP repl7 carrying the text of the Bill of Rights are packets 10, 11, and 13. If you look into the ASCII content of packet 10, you can see the beginning of the text of the Bill of Rights. Note that packet 12 is a client-to-server TCP ACK.

Answers the above questions:

- How many HTTP GET request messages did your browser send? Answer: 1.
- Which packet number in the trace contains the GET message for the Bill of Rights? Answer: 8.
- Which packet number in the trace contains the status code and phrase associated with the response to the HTTP GET request? Answer: packet 10.
- What is the status code and Phrase in the response? Answer: 200 (OK)
- How many data-containing TCP segments were needed to carry the single HTTP response and the text of the Bill of Rights? Answer: three packets (10, 11, 13 in the trace)

II.4 HTTP Authentication

Finally, let’s try visiting a web site that is password-protected and examine the sequence of HTTP message 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
  \url{http://gaia.cs.umass.edu/wireshark-labs/protected_pages/HTTP-wireshark-file5.html}
  Type the requested user name 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.

(\textit{Note:} If you are unable to run Wireshark on a live network connection, you can use the \texttt{http-ethereal-trace-5} packet trace to answer the questions below; see footnote 1.)

Now let’s examine the Wireshark output. You might want to first read up on HTTP authentication by reviewing the Wikipedia entry at \url{http://en.wikipedia.org/wiki/Basic_access_authentication}

Answer the following questions:
19. What is the server’s response (status code and phrase) in response to the initial HTTP GET message from your browser? \textit{Answer:} Packet 6 in the trace contains the first GET and packet 9 contains the REPLY. The server’s in packet 9 is: 401 Authorization Required

20. When your browser’s sends the HTTP GET message for the second time, what new field is included in the HTTP GET message? \textit{Answer:} The HTTP GET includes the Authorization: Basic: field

The username (wireshark-students) and password (network) that you entered are encoded in the string of characters (d2lyZXNoYXJrLXN0dWRlbRzOm5ldHdvcms=) 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 \textit{not} encrypted! To see this, go to \url{http://www.hcidata.info/base64.htm#b64} and enter the base64-encoded string d2lyZXNoYXJrLXN0dWRlbRz and press decode. \textit{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.

Fear not! As we will see in Chapter 7, there are ways to make WWW access more secure. However, we’ll clearly need something that goes beyond the basic HTTP authentication framework!