Lab 1: Applications

Objective

Having gotten our feet wet with the Wireshark packet sniffer in the introductory lab, we're now ready to use Wireshark to investigate protocols in operation. In this lab, we'll 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.

Step 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. Do the following:

- Start up the Wireshark packet sniffer, as described in the Introductory lab (but don't yet begin packet capture) on an Ethernet connection. Filter for "http" traffic to or from your computer, 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. We also want to be cognizant of other's privacy.
- Using your browser, display the following page. If you've tried this step before, you may need to make sure your browser's cache is empty. (Command-Option-E in Safari. Chrome, try Tools → Clear Private Data. Others are on your own.[1]) If you run a local proxy, please disable it.

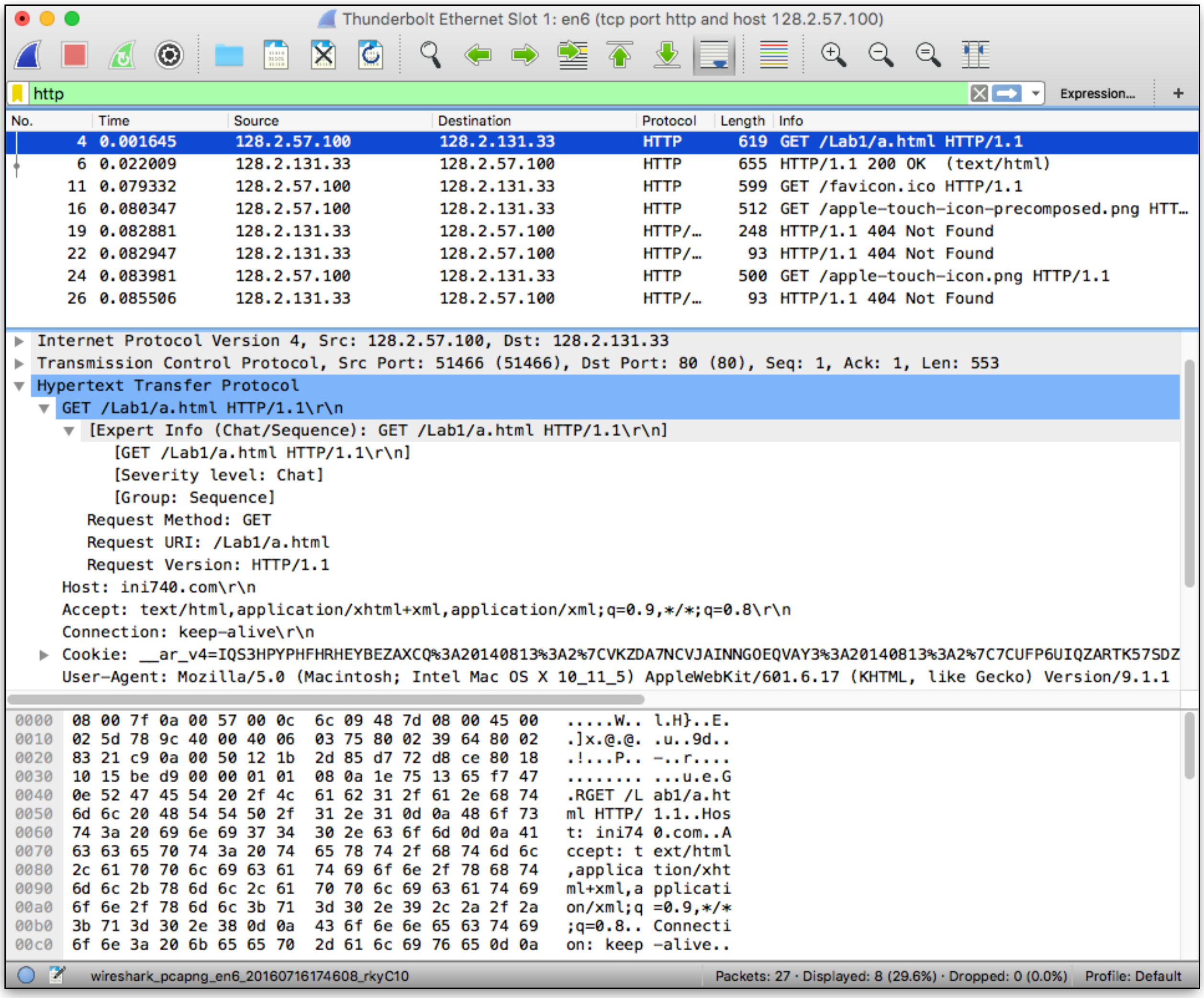
  http://ini740.rocks/Lab1/a.html

  Your browser should display the very simple, one-line HTML file.
- Stop Wireshark packet capture.
- Your Wireshark window should look similar to the window shown in Figure 1.

The example in Figure 1 shows in the packet-listing window that eight HTTP messages were captured: four GET messages and four responses. Two of the messages are of interest, the rest are attempts by my browser to get favicons and other decorative bangles. You might have similar attempts or even HTTP requests generated by other software you had running at the time (dropbox, for instance). As many utilities use HTTP, it is very possible that you will have similar messages that are not germane to this

[1] Wikipedia's page at wikipedia.org/wiki/Wikipedia:REFRESH describes how to clear the cache on just about any browser

lab. Inspect them to determine their source and then ignore them for the rest of the lab.[2]


Thunderbolt Ethernet Slot 1: en6 (tcp port http and host 128.2.57.100)

http

```
No.   Time      Source          Destination     Protocol  Length  Info
    4 0.001645  128.2.57.100    128.2.131.33    HTTP       619 GET /Lab1/a.html HTTP/1.1
    6 0.022009  128.2.131.33    128.2.57.100    HTTP       655 HTTP/1.1 200 OK  (text/html)
   11 0.079332  128.2.57.100    128.2.131.33    HTTP       599 GET /favicon.ico HTTP/1.1
   16 0.080347  128.2.57.100    128.2.131.33    HTTP       512 GET /apple-touch-icon-precomposed.png HTT…
   19 0.082881  128.2.131.33    128.2.57.100    HTTP/…     248 HTTP/1.1 404 Not Found
   22 0.082947  128.2.131.33    128.2.57.100    HTTP/…      93 HTTP/1.1 404 Not Found
   24 0.083981  128.2.57.100    128.2.131.33    HTTP       500 GET /apple-touch-icon.png HTTP/1.1
   26 0.085506  128.2.131.33    128.2.57.100    HTTP/…      93 HTTP/1.1 404 Not Found
```

```
▶ Internet Protocol Version 4, Src: 128.2.57.100, Dst: 128.2.131.33
▶ Transmission Control Protocol, Src Port: 51466 (51466), Dst Port: 80 (80), Seq: 1, Ack: 1, Len: 553
▼ Hypertext Transfer Protocol
  ▼ GET /Lab1/a.html HTTP/1.1\r\n
    ▼ [Expert Info (Chat/Sequence): GET /Lab1/a.html HTTP/1.1\r\n]
          [GET /Lab1/a.html HTTP/1.1\r\n]
          [Severity level: Chat]
          [Group: Sequence]
       Request Method: GET
       Request URI: /Lab1/a.html
       Request Version: HTTP/1.1
    Host: ini740.com\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
    Connection: keep-alive\r\n
  ▶ Cookie: __ar_v4=IQS3HPYPHFHRHEYBEZAXCQ%3A20140813%3A2%7CVKZDA7NCVJAINNGOEQVAY3%3A20140813%3A2%7C7CUFP6UIQZARTK57SDZ
    User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_5) AppleWebKit/601.6.17 (KHTML, like Gecko) Version/9.1.1
```

```
0000  08 00 7f 0a 00 57 00 0c  6c 09 48 7d 08 00 45 00   .....W.. l.H}..E.
0010  02 5d 78 9c 40 00 40 06  03 75 80 02 39 64 80 02   .]x.@.@. .u..9d..
0020  83 21 c9 0a 00 50 12 1b  2d 85 d7 72 d8 ce 80 18   .!...P.. -..r....
0030  10 15 be d9 00 00 01 01  08 0a 1e 75 13 65 f7 47   ........ ...u.e.G
0040  0e 52 47 45 54 20 2f 4c  61 62 31 2f 61 2e 68 74   .RGET /L ab1/a.ht
0050  6d 6c 20 48 54 54 50 2f  31 2e 31 0d 0a 48 6f 73   ml HTTP/ 1.1..Hos
0060  74 3a 20 69 6e 69 37 34  30 2e 63 6f 6d 0d 0a 41   t: ini74 0.com..A
0070  63 63 65 70 74 3a 20 74  65 78 74 2f 68 74 6d 6c   ccept: t ext/html
0080  2c 61 70 70 6c 69 63 61  74 69 6f 6e 2f 78 68 74   ,applica tion/xht
0090  6d 6c 2b 78 6d 6c 2c 61  70 70 6c 69 63 61 74 69   ml+xml,a pplicati
00a0  6f 6e 2f 78 6d 6c 3b 71  3d 30 2e 39 2c 2a 2f 2a   on/xml;q =0.9,*/*
00b0  3b 71 3d 30 2e 38 0d 0a  43 6f 6e 6e 65 63 74 69   ;q=0.8.. Connecti
00c0  6f 6e 3a 20 6b 65 65 70  2d 61 6c 69 76 65 0d 0a   on: keep -alive..
```

wireshark_pcapng_en6_20160716174608_rkyC10   Packets: 27 · Displayed: 8 (29.6%) · Dropped: 0 (0.0%)   Profile: Default

Figure 1: Wireshark display after ini740.rocks/Lab1/a.html has been retrieved.

The packet-contents window shows details of the selected message (in this case the HTTP GET 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 information for

[2] On my local network, I get tons of SSDP messages, which are sent in HTTP format, so they show up when I filter on HTTP. To get rid of SSDP messages, add "and not (udp.port eq 1900)" to your filter.

Frame, Ethernet, IP and TCP lines are collapsed (by clicking the plus sign or twisting the triangle on the left of each line). Similarly, make sure the HTTP line displays all the information about each HTTP message (by similarly twisting the triangle or clicking to get a minus sign).

By looking at the information in the HTTP GET and response messages, answer the following questions. When answering any questions in this lab, describe your work. You don't need to provide screenshots for everything, but we need to ensure you understand your answers. That's a difficult task, given that your particular network capture will be unique. If you just give us a number (or whatever), we can't check your understanding. If we can't check your understanding, we can't give you points for it. So, give us a description of the number (or whatever) and where you found it. You are encouraged to give us a screenshot if that will clarify where you got the answer.

1. Is your browser running HTTP version 1.0 or 1.1? What version of HTTP is the server running? (2 points)
2. What languages (if any) does your browser indicate that it can accept? (2 points)
3. What was the round-trip-time for the request (i.e. time between sending the request and capturing the response)? (3 points)
4. What is the status code returned from the server to your browser? What does that status code mean? (2 points)
5. When was the HTML file that you are retrieving last modified at the server? (2 points)
6. How many bytes of content are being returned to your browser? (2 points)
7. How many bytes are in the HTTP header (just the header, don't include data)? (3 points)

Step 2: Retrieving Long Documents

Let's next see what happens when we download a longer HTML file. Do the following:

- Start up the Wireshark packet capture with our normal filter.
- Enter the following URL into your browser.

  http://www.ietf.org/rfc/rfc2616.txt
- Your browser should display the HTTP RFC, which is a moderately lengthy document.
- Stop Wireshark packet capture.

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 422KBytes 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 "Reassembled TCP Segments" phrase displayed by Wireshark.

Answer the following questions (again, make sure to explain your answers thoroughly):

8. How many HTTP GET request messages were sent by your browser? (2 points)
9. How many TCP segments were needed to carry the single HTTP response? (3 points)
10. How many bytes of overhead was generated in TCP to transport the response? What percentage is the TCP overhead of the entire TCP + HTTP + Data transmission? (Yes, we are deliberately ignoring IP and Ethernet for this question). Make sure to explain what you think is overhead. (6 points)
11. What is the status code and phrase associated with the response to the HTTP GET request? (2 points)
12. Are there any HTTP status lines in the transmitted data associated with a TCP-induced "Reassembly of TCP Segments?" (3 points)

Step 3: 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 the Wireshark packet capture with your (by now) favorite filter.
- Enter the following URL into your browser (after clearing the cache, if necessary)

  http://ini740.rocks/Lab1/b.html
- Your browser should display a short HTML file with three images. The first two images are hosted on the ini740.rocks web server. The third is hosted elsewhere. Your browser will have to retrieve these image files from the indicated web sites using separate GET requests.
- Stop Wireshark packet capture.

Answer the following questions:

13. How many HTTP GET request messages were sent by your browser? To which addresses (domain and IP) were these GET requests sent? (7 points)
14. Can you tell whether your browser downloaded the images serially, or whether they were downloaded from the two web sites in parallel? Explain. (7 points)

Step 4: The Conditional GET

Recall from Section 2.2.6 of the text, that most web browsers perform object caching and thus perform a conditional GET when retrieving an HTTP object. Let's see that in action.

Now do the following:

- Start up the Wireshark packet capture with your (by now) favorite filter.
- Enter the following URL into your browser:

  http://ini740.rocks/Lab1/a.html
- Enter the same URL into your browser again (or simply select the refresh button on your browser)
- Stop Wireshark packet capture, and examine the messages you've caught.

Answer the following questions:

15. 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? (2 points)

16. Inspect the contents of the server response. Did the server explicitly return the contents of the file? How can you tell? (4 points)

17. 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? (If not, try again. When I tried this under Safari 4.0, I discovered that it won't send a conditional get in the instances you would think it should. I was able to force one by quitting Safari and restarting it. If you are still having trouble, try loading the b.html file again. Safari uses a conditional get to request the image files). Chrome does send the header we are looking for. (2 points)

18. 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. (4 points)

Step 5: Form Submission

We've talked about using HTTP to get web pages -- that is, for the browser to request data be transmitted from the server. How about data flow in the opposite direction? What happens when you fill in an HTML form and press "submit?"

Do the following:

- Start up the Wireshark packet capture with your (by now) favorite filter.
- Enter the following URL into your browser:

  http://ini740.rocks/Lab1/c.html
- In the page that gets downloaded, you'll see an extremely simple form with two fields, mimicking a login page. Fill in any data you'd like in the two fields and press the submit button.
- Stop Wireshark packet capture, and examine the messages you've caught.

Answer the following questions:

19. You did notice that the two text fields on the form acted differently, right? How did the browser handle the password field differently from the username field? (3 points)

20. Examine the second request from the browser. How is it different from the GET request messages we've seen? Where in the message is the data you typed in the text fields? (4 points)

21. The data you typed in the username field is being sent with a key of *user*. Shouldn't it be *username*? The stuff in the password field is sent as *password*. Why is it sent differently? (Hint: The answer to this question can be found from an inspection of the first reply message from the server, among other places.) (2 points)

Step 6: Authentication

Clearly the login form of the previous step isn't sufficient to actually protect anyone. We saw the password is sent in the clear. Let's try a bit more sophisticated mechanism.

Do the following:

- Start up the Wireshark packet capture with your (by now) favorite filter.
- Enter the following URL into your browser:

  http://ini740.rocks/Lab1/Lab1_protected/d.html
- You will have to enter a username and password before the page will be displayed. Use the username *SmartStudent* and the password *PoorPassword*.
- After you see the authenticated page, stop Wireshark packet capture, and examine the messages you've caught.

Answer the following questions:

22. Is there anything different or strange in the first GET message sent from the browser? At this point in time, does the browser know the webpage is protected? (2 points)

23. Examine the first response message. What is the status code? What is the content of this message (i.e. the data section) (3 points)

24. Examine the next GET message from the browser. The **`Authorization`** header looks like:

`Authorization: Basic U21hcnRTdHVkZW50OlBvb3JQYXNzd29yZA==`

"Basic" is the authorization type. It matches the **`WWW-Authenticate`** header in the first response message.

The **`U21...A==`** string looks interesting. Is it the encrypted password? Copy it into the Base64 Decoder at:

http://www.opinionatedgeek.com/dotnet/tools/Base64Decode.

What do you find? If you twist the triangle on the Authorization header in Wireshark, you'll discover the same thing. What is Base64? Does it protect the password from people with Wireshark? (8 points)

Step 7: Another Application Protocol

Now that we've had plenty of time to mess around with HTTP, let's take a short look at the DNS protocol. This section builds upon your knowledge of dig from HW#1. If you haven't yet completed part 2 of the homework, you might want to complete it first.

Do the following, answering the interleaved questions as you go:

- Start up the Wireshark packet capture, but this time filter for DNS packets headed to and from your own IP address.
- In a separate terminal window, invoke dig. Use dig to find out the IP address of www.purple.com.

25. What is the IP address of purple.com? Which DNS server was the request sent to? (2 points)

26. Examine the DNS traffic in Wireshark. How many DNS requests (queries) were sent from your computer? Explain the purpose of each. Do you see differences if you request recursion or not? (4 points)

- Ask dig for the IP address of a random (but likely to exist) website (really, make it random. Don't just pick Facebook. Ask for any single English word (I'd guess that most words in the dictionary have been picked up by somebody, perhaps just a domain squatter)). Use Dig to ask the same question again. The reason for the randomness, is that all students in the course are likely using the same small set of nameservers, so any website I gave you would probably be cached by a classmate's requests.

27. Can you tell if the first request was cached or if the nameserver went out and found it somewhere? What's your evidence? If you think it was cached, repeat the experiment until you find a non-cached domain name. (3 points)
28. Can you tell if the second request was cached? (3 points)
29. My computer sends lots of DNS queries with type = AAAA. Does yours? What are those? (2 points)
30. Do something with Wireshark that isn't listed in this assignment so far. Document it. You will be graded on creativity and your description. This is your chance to play around and learn something different. (6 points)

Complete

You're done now. Feel free to play around with Wireshark and learn a bit more. Submit your answers as a single PDF file to Gradescope. Your answers are due BEFORE the start of class on the date specified on the website. No late submissions will be accepted.