Alexander Miller
Collaborators: AC Fields (acfields) & Ryan Li (rli3)
Assignment 3

Step 2

Using Wireshark to snoop on IP packets moving between a client and a server can tell us a lot about the server, the client, the network they're utilizing, and the data they are sharing. In the case of the server, by dissecting the IP packets we can figure out the details of its Certificate; we can figure out the domain name of the specified IP by resolving it with wireshark, the algorithm it uses to generate signatures (sha256 in the case of the server GCSSP is interacting with), the CA the server got its certificate from (Let's Encrypt), the start and expiration date of its certificate (4/20/20), and its certificate ID.
We can also find things out about the client and its network. For example we can figure out the system that GCSSP is using (A Mac running OSX 10.14 and an Intel CPU), its IP address (192.128.1.2), the ports that are open on the client during transmission (i.e. 56393), and the browser it's using (Firefox). Furthermore, by analyzing the packets coming in we can both estimate the network bandwidth by analyzing the limit of the Window sizes designated in the packets as well as the type of data being sent from both the size of the packets coming in and the method of transmission (for example, since the client isn't connected to the server via UDP but via TCP, we can guess that it's not sending and receiving a phone call but files). From all of this it is possible to figure out that GCSSP is browsing pages from a domain called 'isc.cs.uchicago.edu' at the IP '128.135.24.28. I also know that it is generating traffic encrypted through TLSv1.2 and that it's sending TCP packets carrying snippets of files.

Step 3

First, I filtered the packets I collected from monitoring GCSSP's traffic with the display filter:

```
(ip.dst eq 128.135.24.28) && (tcp.flags.ack == 1 || tcp.flags.syn == 1)
&& (tcp.flags.reset  == 0)  &&
( not tcp.analysis.duplicate_ack and not tcp.analysis.retransmission
 and  not tcp.analysis.out_of_order)
```

I did this because in order to start parsing GCSSP's traffic and figure out what types of data its receiving, I only need to read its ACK counts from the server. I also don't care about packets that are confusing (out of order packets, packets that don't tell me about SEQ and

ACK data, Reset packets, and duplicate packets) so I throw them out before I start parsing them. Now, onto parsing.
My python script does a few things to organize the packet data into an understandable format:

1. It filters packets so that I only process packets that I care about (Admittedly taken care of by the initial display filter in Wireshark)
2. It checks how long after the previous packet the current packet was sent; if the delay is not longer than three seconds, it associates the packet with a conversation on its given port and associates that packet with a page (if a packet is sent within 3 seconds of the last packet, it is considered as being within the same page)
3. If a packet is sent after three seconds have passed since the last packet was sent, it marks the end of a page.
4. Once the end of a page is reached, it sorts that page by port number, a prints it. It also finishes associating the conversations occurring on the active ports and summarizes the following information: for a given active port it estimates the amount of data that GCSSP received from the server on that port (measured in three different ways). It then outputs that for all ports, sums it, and outputs that data in regards to the page. Finally, it marks the conversations that happened on those ports as finished and saves them to be outputted later (this was once useful but stopped being so; summarizes all the data sent on a port in a given timeframe)

I processed all of GCSSP's traffic like this and put it in a file called 'captures.txt.'

Step 4

FLAG:
8438626844355492081
7225508486349866589
2738121144371683593
6853336026951060780
3534038669776643623
2680005981675510488
4235353008706140423
5468348021594000424
347348042349246400
487495247933867962

In order to collect the flag I measured my traffic at the domain 'isc.cs.uchicao.edu.;' for a given page I clicked on all of the links present on a page and measured the resulting traffic with Wireshark. I then put that traffic through the same display filter as the GCSSP traffic and then parsed that data using the same python script that I used in Step 3. I then matched that output to that generated by GCSSP's traffic and matched which link generated traffic equivalent to that of GCSSP at a given point. I then proceeded to that link and repeated that process.

I had a few difficulties with this process. For one, it was hard to filter out FIN, ACK packets from the general data; they were somewhat confusing to work through but could be filtered out. Another problem was that it was hard to measure the data content of small pages; I relied on trial and error for these pages. This difficulty was mitigated by directly comparing SEQ and ACK numbers and seeing where close match ups occurred; automated analysis of such packets is then possible. A final difficulty I had was automating data measurements; often, data transfers didn't start when consecutive SEQ numbers occurred on a given port (as I assumed). I mitigated this by taking multiple data measurements of a conversation, generating a bound, and comparing the bounds of network traffic.

Sources:

https://blog.catchpoint.com/2017/05/12/dissecting-tls-using-wireshark/
https://www.cybrary.it/0p3n/how-to-use-the-wireshark-cyber-security-tool/
https://stackabuse.com/reading-and-writing-json-to-a-file-in-python/
https://www.tutorialspoint.com/python/python_command_line_arguments.htm
https://stackoverflow.com/questions/9535954/printing-lists-as-tabular-data
https://stackoverflow.com/questions/10480806/compare-dictionaries-ignoring-specific-keys