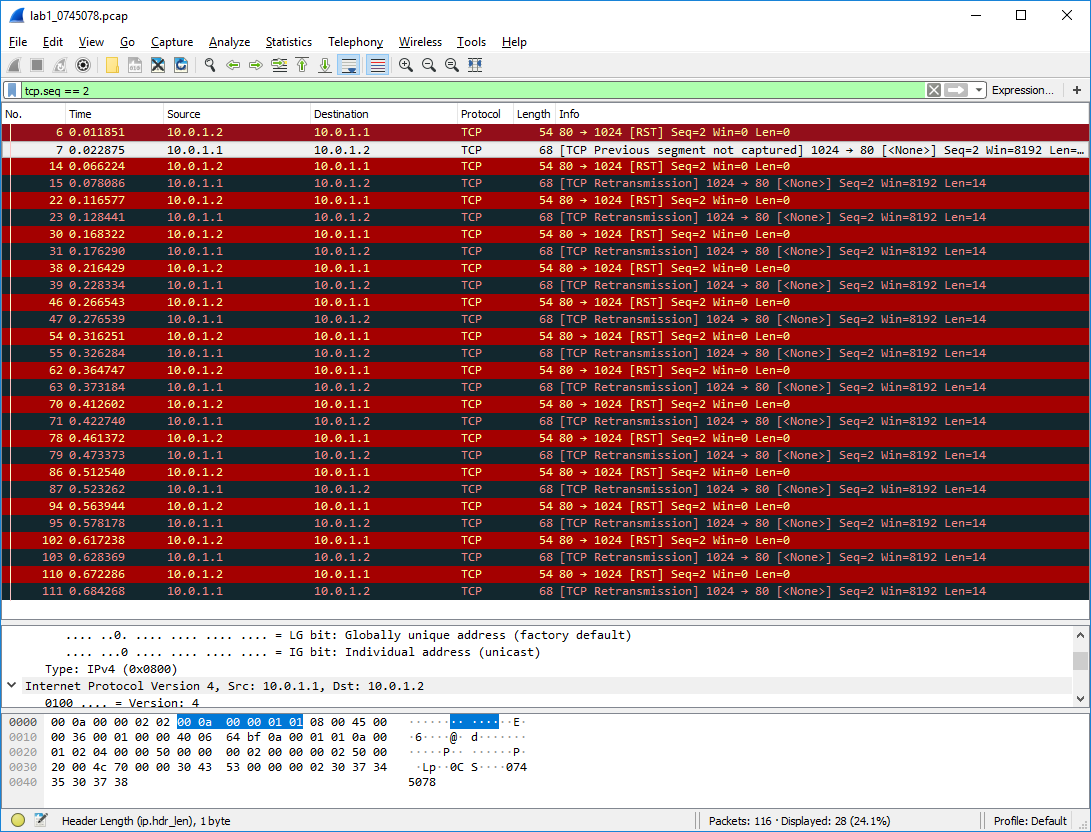
**NCTU CN2018 LAB 1 – PACKET MANIPULATION VIA SCAPY**

Student Name: Thomas Ng Student ID: 0745078 Department: CS

**PART A**

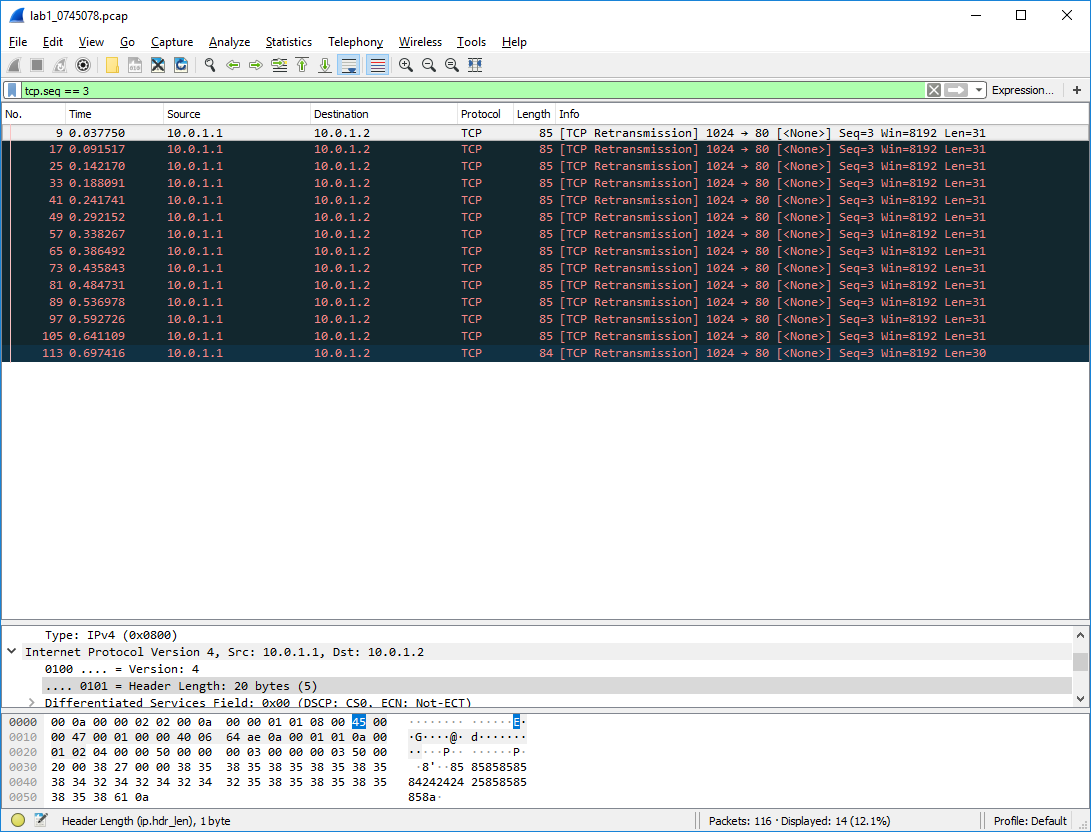
1. tcp.seq == 2

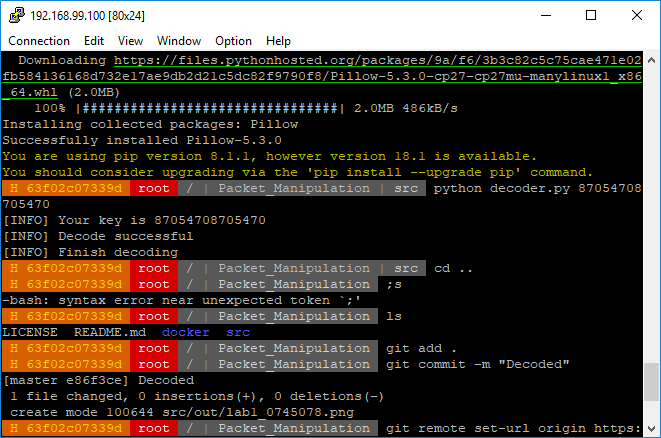
Looking at sender.py, we see that when a packet with customized header is sent, the tcp sequence number is 2 (tcp = TCP(sport = src\_port, dport = dst\_port, flags = '', seq = 3, ack = ack)). We can use this information to determine that we should be looking out for packets that have tcp sequence 2 in order to filter out these packets with customized header. In the display filter, we enter “tcp.seq == 2” as our filter.

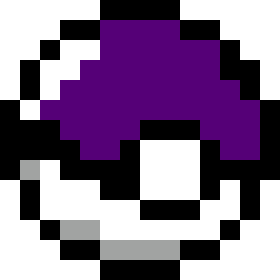


1. tcp.seq == 3

Looking at sender.py, we see that when a packet with the secret payload is sent, the tcp sequence number is 3 (tcp = TCP(sport = src\_port, dport = dst\_port, flags = '', seq = 3, ack = ack)). We can use this information to determine that we should be looking out for packets that have tcp sequence 3 in order to filter out these packets with the secret payload. In the display filter, we enter “tcp.seq == 3” as our filter. We then look at the TCP segement data to look for the first digit of the secret payload. We iterate this 14 times for all of the 14 packets containing the secret payload, with the 14 first-digits forming our secret code.



1. After forming our 14 digit secret code from the earlier steps, we now have the secret key to be decoded. We use our secret key with decoder.py (python decoder.py 87054708705470) and run the python script to obtain the following secret payload.



Bonus:

1. I have learnt how to create my own protocol through this lab. In this lab, I learnt how to configure the sender and receiver to send our secret data while specifying my own IP protocol in Python. I also learnt how to use display filters in Wireshark to filter out the packets that I am looking for, as well as looking for hints for what filters to use.
2. I had difficulties accessing the Docker image as I was using the wrong IP address for my SSH connection. Instead of the IP address specified inside kitematic, I was trying to SSH to 127.0.0.1 on port 9487, which was not where the Docker image was running on. I also faced difficulties trying to commit my Docker image, as I was not running the commands on the Docker Toolbox CLI.

**PART B**

Base Files

* I cloned the necessary files for Docker image and python scripts from the provided GitHub repository
* I then created my own repository and pushed the downloaded files into it

Task 1: Environment Setup

* I first set up the configuration for the Dockerfile by firstly getting the base Linux image. I then updated the software repository and installed the necessary softwares that are missing (tcpdump). I then installed scapy before configuring the port (port 22) on which the container will be listening to and then running cloning from my GitHub Repository.
* Once the configuration for Dockerfile has been set up, I then proceed to build the docker image from Dockerfile using “docker build”. Once that was done, a container is then built using “docker run” while mapping port 9487 of the docker host to port 22 of the local machine.
* Once this was all done, we then connect to the docker host through SSH using PieTTY and beginning modifying main.sh to create our namespace. Using main.sh, we created our network namespaces h1 and h2, which would be used for the sender and receiver respectively. After configuring the interface, IP address and gateway of h1 and h2 respectively, we execute main.sh to build the namespace after changing the permissions of the script.

Task 2: Defining Protocol

* After setting up the environment, I proceeded with defining my own protocol. In my custom protocol, I defined my own name for the protocol, as well as the fields in my protocol, such as index, department, gender and id.

Task 3: Sending Packets

* Once I am done with defining my own protocol, I then modified sender.py to send our custom packets. Through sender.py, I can establish a TCP connection and send individual packets with customized header and secret payload. In sender.py, each sequence of the packet sending has not just their unique packet, but also their own TCP sequence (1 to 3), source and destination port, and flags

Task 4: Sniffing packets

* After configuring sender.py, I then configured receiver.py to receive and sniff the packets sent by sender.py. In receiver.py, I set the source IP address to be receiving the packets from and the destination interface to sniff packets on. I would then dumb the sniffed packet into a PCAP file that I would use later on.

Task 5: Running Sender and Receiver

* Using tmux, I created multiple terminals in the session to run both sender.py and receiver.py separately. receiver.py was ran first to ensure that the listener is running before any packets are sent out. This prevents any packets from being missed out in the window that receiver.py is not running if sender.py is ran first. Once receiver.py is running, I then ran sender.py to send the packets over. Once I have saved the output (PCAP file and recv\_secret.txt) from receiver.py to ./src/out, the PCAP file can be viewed in the terminal using tcpdump.

Task 6: Pushing files

* Once I had the necessary files, I then saved my current docker image using “docker commit” before logging in to Docker registry to push the image onto it. I then added the Dockerfile onto GitHub with “git add .” and pushing it up onto my GitHub repository after committing my changes.

Task 7: Loading PCAP via Wireshark

* After downloading and installing Wireshark, I then opened the PCAP file In Wireshark to analyze the captured packets. Since this was done on the same local machine, I had no need to clone my files on the GitHub repository. Furthermore, for the analyzing of the PCAP file in Wireshark, only the PCAP file that was created by receiver,py is needed.