Final Project Architecture: PyreShark:

Some suggestions for a future report of this type:

1. Include an overall architecture diagram, with annotation and explanation (full sentences).
2. Explain all acronyms, abbreviations, classes, etc.  Such as dict FO, MF.
3. For pseudo code, include module/method descriptions.
4. Two source code files were submitted, without explanation, pyre\_shark.py, traffic-sniffing.py. At a minimum, each file should have a header with summary and author’s names.

Python Wireshark:

* Read incoming and outgoing traffic on TCP protocol
* Parse Ip Fragments on a given TCP Stream ("tcp is a byte stream”)
  + TCP splits its message into sequence fragments
* TCP connection is established, and the client sends (ACK=True, ACK\_Num=Y+1), meaning the first byte it expects to receive is #(Y+1)
* Bytes sent on a TCP stream are received “in order” they are sent
  + Read Bytes as is in the stream

Packet Reconstruction:

* python doesn't support buffer building
  + Build a Buffer fragment
* eth\_addr being passed in (as bytes, like b"\x08\x00'\x98\x88" and then when it tries to call ord(a[0])
* create a dict (a dict of dicts, it maps from the ip\_id to a dict of fragments, so each ip\_id gets multiple fragments)
* manually feed data fragments into dict
* create gaps (space)
  + (dict keys that map to None) if the data wasn't big enough to fill the gap and if the offset started in the middle of the gap, then leave a gap behind it.
  + The last gap, don't leave another gap after if it's passed the more\_frags = 0
* Bytes are received in order they are sent, order bytes in the order received.
* Reconstructed order:
  + initialize the defrag dict for given ip\_id
  + start with this:
    - [None, b'abc', b'def', None, b'ghi', None, b'jkl', b'mno', b'pqr', None]
  + output this:
    - [ [b'abc', b'def'], [b'ghi'], [b'jkl', b'mno', b'pqr'] ]
  + subdivide the list into a list of lists, splitting where there's a “None”
    - find first section that is gap AND FO >= section.idx
* Pass the whole sequence to reassemble the reassembled fragments (Create one long array)
  + "pull from IP packet pool"
* flush the TCP buffer every so often instead of waiting for TCP to "end"
  + flush with LHS of gap, reassign that entry in the dict
* callback for operation on successful reconstruction
* Implement decode()
  + decoding the TCP data, which includes { app-layer headers, app-layer data }
  + replace "problem" characters with that question-mark-in-a-diamond that you get for characters that can't be parsed
  + Unparsed data: set Content-Type: text/html, and then just output "unsupported content-type" or "unsupported application protocol"

Pseudocode:

insert ({ FO, MF, data }):

find first section that is gap AND FO >= section.idx

next\_idx <-- idx(next section)

if not last section

else INFINITY

if FO + len(data) > next\_idx:

throw error

if FO = section.idx

section <-- data

else

# FO > section.idx

insert (data, idx = FO) after section

set section ref to point at newly inserted (data, idx = FO)

if MF and FO + len(data) < next\_idx:

insert (None, idx = FO + len(data)) after section

# section should contain "data"

Text

Description automatically generated

Summary:

This application is a python script that will read a TCP stream, then reconstruct the original payload on the application layer. It is capable of not only reading the byte stream and Ip fragment data, but also reconstructing each sequence of fragments as the data is being sent on a Network. The packet reconstruction works by capturing the TCP packets, decoding, and constructing a new sequence of data that is identical to the original. When data or content is unsupported it will display the approximate up supported message. All this can be output to an output file that will display the necessary metadata that was captured, such as source port, destination port, sequence number, TCP header length, and the Data in hex. The decoder takes the hex data fragments and decodes to decimal and displays in readable text.