PKS project 1

Packet analyzer

Márk Bartalos

Seminars with**: Ing. Lukáš Mastiľak**
Seminars on**: Thu 14:00-15:50**

Contents

Introduction

As task we got building a packet analyzer which has similar functionality as the popular Wireshark program.

Programming environment

As programming environment, I chose Python 3.10.8 and Visual Studio Code. I choose Python because it will be a great fit for this task with its versatility ease of use and many libraries.

Application can be started using the command: ***python main.py***

It can be also supplied extra parameters such as: ***python main.py -p PROTOCOL FILE_NAME*** (Note that file name can only be supplied if protocol filter is used)

Note: input files have to be in the Capture directory

User interface

As this is a command line application user interface is only the command used to run the application. All data is written into YAML by design.

Used libraries

- scapy.all - for reading the packet capture files
- scapy.compat – for getting the raw data from the read files
- ruamel.yaml – for formatting and writing YAML

Implementation of task 1

For task 1 we had to implement basic analyzation of packets, we had to extract useful information from them such as their source and destination MAC address, frame type, length of the frame and so on. This extracted we had to serialize to YAML and save to a file and the end of each packet we had to append a hexa frame which is basically the full packet in hexadecimal numbers.

The main analysis was done in the analyze_frames() method

```python
    # Classify packet type
    if int(packet_length, base=16) > 1500:
        frame_type = "Ethernet II"
    elif packet_type_length == "aaaa":
        frame_type = "IEEE 802.3 LLC & SNAP"
    elif packet_type_length == "ffff":
        frame_type = "IEEE 802.3 RAW"
    else:
        frame_type = "IEEE 802.3 LLC"

    packet_object = {
        "frame_number": i + 1,
        "frame_type": frame_type,
        "len_frame_pcap": real_frame_length,
        "len_frame_medium": len_frame_medium,
        "dst_mac": dest_mac,
        "src_mac": src_mac,
    }
```

Here we classify the packet based on its length and packet type which is from the packet itself.

After identifying the packet we create the packet object.

Each packet type has its modify function which fills out its type specific properties.

```python
    if frame_type == "Ethernet II":
        packet_object = modify_ethernet_object(
            packet=packet,
            packet_length=packet_length,
            packet_object=packet_object,
            ipv4_history=ipv4_history,
            filter=filter,
            filter_type=filter_type,
            offset=frame_jump,
        )
    elif frame_type == "IEEE 802.3 LLC" and filter == "":
        packet_object = modify_iee_llc(
            packet=packet, packet_object=packet_object, offset=frame_jump
        )
    elif frame_type == "IEEE 802.3 LLC & SNAP" and filter == "":
        packet_object = modify_iee_llc_snap(
            packet=packet, packet_object=packet_object, offset=frame_jump
        )
    elif filter != "":
        packet_object = None
```

ETHERNET II specific modify function

```python
if ether_type == "IPv4":
    src_ip = f"{int(packet[ip_offset:ip_offset+2],base=16)}.{int(packet[ip_offset+2:ip_offset+4],base=16)}.{int(packet[ip_offset+4:
    dst_ip = f"{int(packet[ip_offset+8:ip_offset+10],base=16)}.{int(packet[ip_offset+10:ip_offset+12],base=16)}.{int(packet[ip_offs
    protocol = f"{dictionaries['ipProtocolTypes'][str(packet[protocol_offset:protocol_offset+2])]}".strip()
    src_port = int(packet[port_offset : port_offset + 4], base=16)
    dst_port = int(packet[port_offset + 4 : port_offset + 8], base=16)

    if src_ip in ipv4_history:
        ipv4_history[src_ip]["number_of_sent_packets"] = (
            ipv4_history[src_ip]["number_of_sent_packets"] + 1
        )
    else:
        ipv4_history[src_ip] = {"number_of_sent_packets": 1}

    packet_object["protocol"] = protocol

    packet_object["src_ip"] = src_ip
    packet_object["dst_ip"] = dst_ip
    if src_port > 0 and dst_port > 0:
        packet_object["src_port"] = src_port
        packet_object["dst_port"] = dst_port

    if protocol == "TCP":
        dictType = "tcpPortTypes"
    elif protocol == "UDP":
        dictType = "udpPortTypes"
```

For setting properties such as ether type or app protocol I used .txt files which are loaded into a dictionary at the start of the program

External file structure

```
Protocols
    EtherTypes.txt
    ICMPTypes.txt
    PID.txt
    protocols-all.txt
    Protocols.txt
    SAP.txt
    TCPports.txt
    TFTPCodes.txt
    UDPports.txt
```

EtherTypes.txt

rotocols > EtherTypes.txt

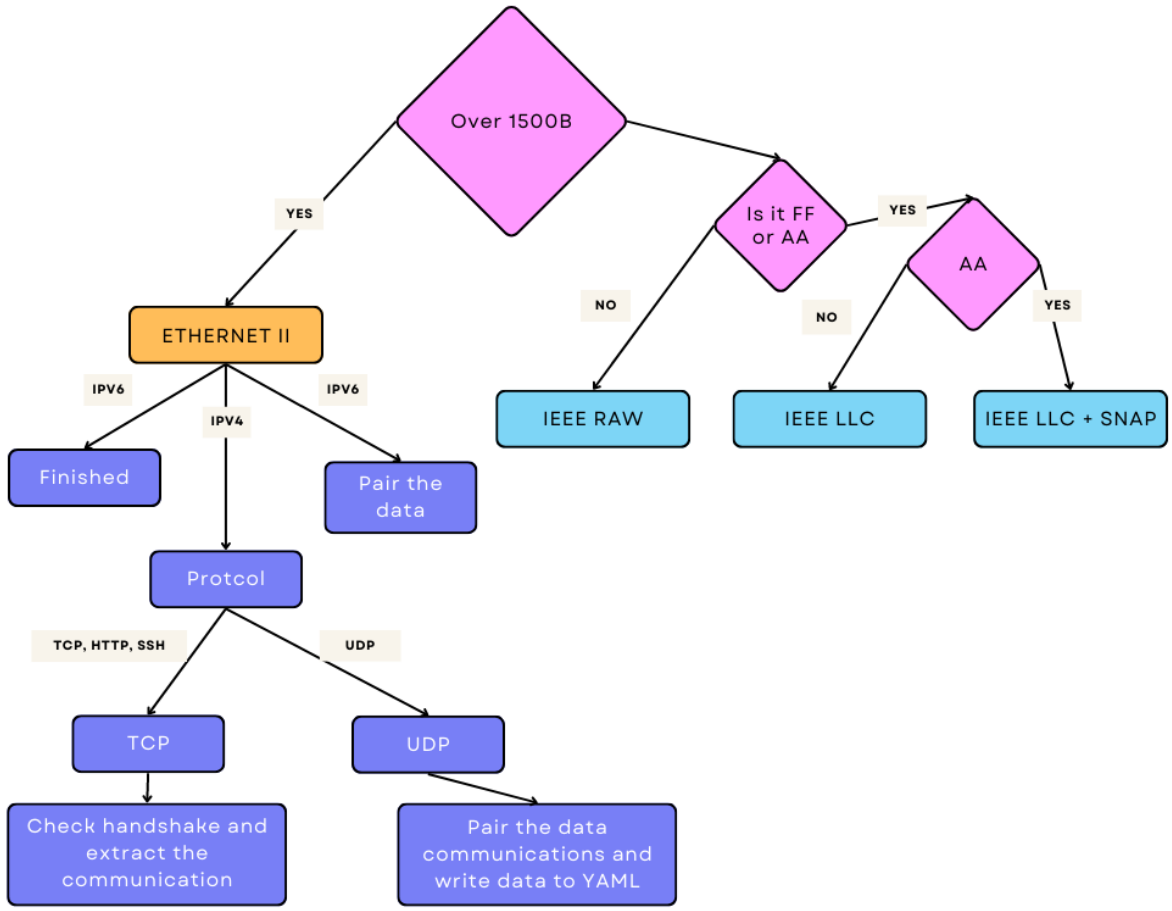
```
0200:XEROX PUP
0201:PUP Addr Trans
0800:IPv4
0801:X.75 Internet
0805:X.25 Level 3
0806:ARP
8035:Reversed ARP
809b:Appletalk
80f3:Apple Talk AARP (Kinetics)
8100:IEEE 802.1Q VLAN-tagged frames
8137:Novell IPX
86dd:IPv6
880b:PPP
8847:MPLS
88cc:LLDP
8848:MPLS with upstream-assigned label
8863:PPPoE Discovery Stage
8864:PPPoE Session Stage
9000:ECTP
```

```python
def load_dictionaries():
    load_dictionary("Protocols/EtherTypes.txt", dictionaries["etherTypes"])
    load_dictionary("Protocols/SAP.txt", dictionaries["sapTypes"])
    load_dictionary("Protocols/PID.txt", dictionaries["pidTypes"])
    load_dictionary("Protocols/Protocols.txt", dictionaries["ipProtocolTypes"])
    load_dictionary("Protocols/UDPports.txt", dictionaries["udpPortTypes"])
    load_dictionary("Protocols/TCPports.txt", dictionaries["tcpPortTypes"])
    load_dictionary("Protocols/ICMPTypes.txt", dictionaries["icmpTypes"])
    load_dictionary("Protocols/TFTPCodes.txt", dictionaries["tftpCodes"])
```

```python
def load_dictionary(file_path, dictionary):
    file = open(file_path, "r")
    for line in file:
        splitLine = line.split(":")
        if len(splitLine) > 1:
            dictionary[splitLine[0]] = splitLine[1].strip()
    file.close()
```

```python
def modify_iee_llc(packet, packet_object, offset=0):
    dsap_offset = 28 + offset
    # dsap_num = packet[dsap_offset : dsap_offset + 2]
    ssap_num = packet[dsap_offset + 2 : dsap_offset + 4]

    try:
        ssap = f"{dictionaries['sapTypes'][ssap_num]}".strip()
    except:
        ssap = ssap_num
    packet_object["sap"] = ssap

    return packet_object


def modify_iee_llc_snap(packet, packet_object, offset=0):
    pid_offset = 40 + offset
    pid_num = packet[pid_offset : pid_offset + 4]
    try:
        pid = f"{dictionaries['pidTypes'][pid_num]}".strip()
    except:
        pid = pid_num

    packet_object["pid"] = pid

    return packet_object
```

Flowchart of analyzing packet processing



Implementaton of task two

Here we had to fill out more and protocol specific information to each protocol in the ETHERNET II frame. Example for ARP:

```python
if ether_type == "ARP":
    arp_ip_offset = 56 + offset
    arp_dst_ip_offset = 76 + offset
    arpcode_offset = 40 + offset
    src_ip = f"{int(packet[arp_ip_offset:arp_ip_offset+2],base=16)}
    dst_ip = f"{int(packet[arp_dst_ip_offset:arp_dst_ip_offset+2],b
    arpcode = int(packet[arpcode_offset : arpcode_offset + 4], base
    packet_object["src_ip"] = src_ip
    packet_object["dst_ip"] = dst_ip
    if arpcode == 1:
        packet_object["arp_opcode"] = "REQUEST"
    else:
        packet_object["arp_opcode"] = "REPLY"
```

All the well know protocol names are loaded from files and retrieved from a dictionary

```python
if protocol == "TCP":
    dictType = "tcpPortTypes"
elif protocol == "UDP":
    dictType = "udpPortTypes"

try:
    app_protocol = f"{dictionaries[dictType][str(src_port)]}".strip()
    packet_object["app_protocol"] = app_protocol
except:
    try:
        app_protocol = f"{dictionaries[dictType][str(dst_port)]}".strip()
        packet_object["app_protocol"] = app_protocol
    except:
        app_protocol = ""
```

Here we first get the type of protocol to be able the retrieve the appropriate dictionary, because TCP and UDP well know ports are in different txt files.

Analysis of frames

```python
if ether_type == "ICMP":
    icmp_type_offet = 68 + offset
    icmp_code = int(packet[icmp_type_offet : icmp_type_offet + 2])
    try:
        packet_object["icmp_type"] = dictionaries["icmpTypes"][str(icmp_code)]
    except:
        packet_object["icmp_type"] = None
```

Settings the ICMP type from the ICMP header

```
packet_object["protocol"] = protocol

packet_object["src_ip"] = src_ip
packet_object["dst_ip"] = dst_ip
if src_port > 0 and dst_port > 0:
    packet_object["src_port"] = src_port
    packet_object["dst_port"] = dst_port
```

Setting the IP and the port for IPV4

```
packet_object["dst_ip"] = dst_ip
if arpcode == 1:
    packet_object["arp_opcode"] = "REQUEST"
else:
    packet_object["arp_opcode"] = "REPLY"
```

Settings arpcode for arp.

Implementation of task three

Here we had to create statistics of IPV4 packets. I did this using a dictionary where the key was the sender's ip address and if the same sender sent a packet the packet count was incremented.

Then at the end selected the largest sent packet number and using a for loop I search up the ip addresses which sent the same amount of packets.

```
    frames_database["ipv4_senders"] = []
    for node in ipv4_history:
        frames_database["ipv4_senders"].append(
            {
                "node": node,
                "number_of_sent_packets": ipv4_history[node][
                    "number_of_sent_packets"
                ],
            }
        )
    frames_database["max_send_packets_by"] = find_max_sent_packets(
        ipv4_history=ipv4_history
    )
```

```python
def find_max_sent_packets(ipv4_history):
    max = 0
    for node in ipv4_history:
        if ipv4_history[node]["number_of_sent_packets"] > max:
            max = ipv4_history[node]["number_of_sent_packets"]
    max_send_packets = []
    for node in ipv4_history:
        if ipv4_history[node]["number_of_sent_packets"] == max:
            max_send_packets.append(node)

    return max_send_packets
```

Task four

Filtering

Filtering is done using -p argument. When passing a protocol to the filter the filter checks if the protocol exists from external files, then gets the correct type if the protocol is a nested protocol such as HTTP. Then passes these arguments to the analyze_frames() function. It also supports adding file name as additional argument when using -p.

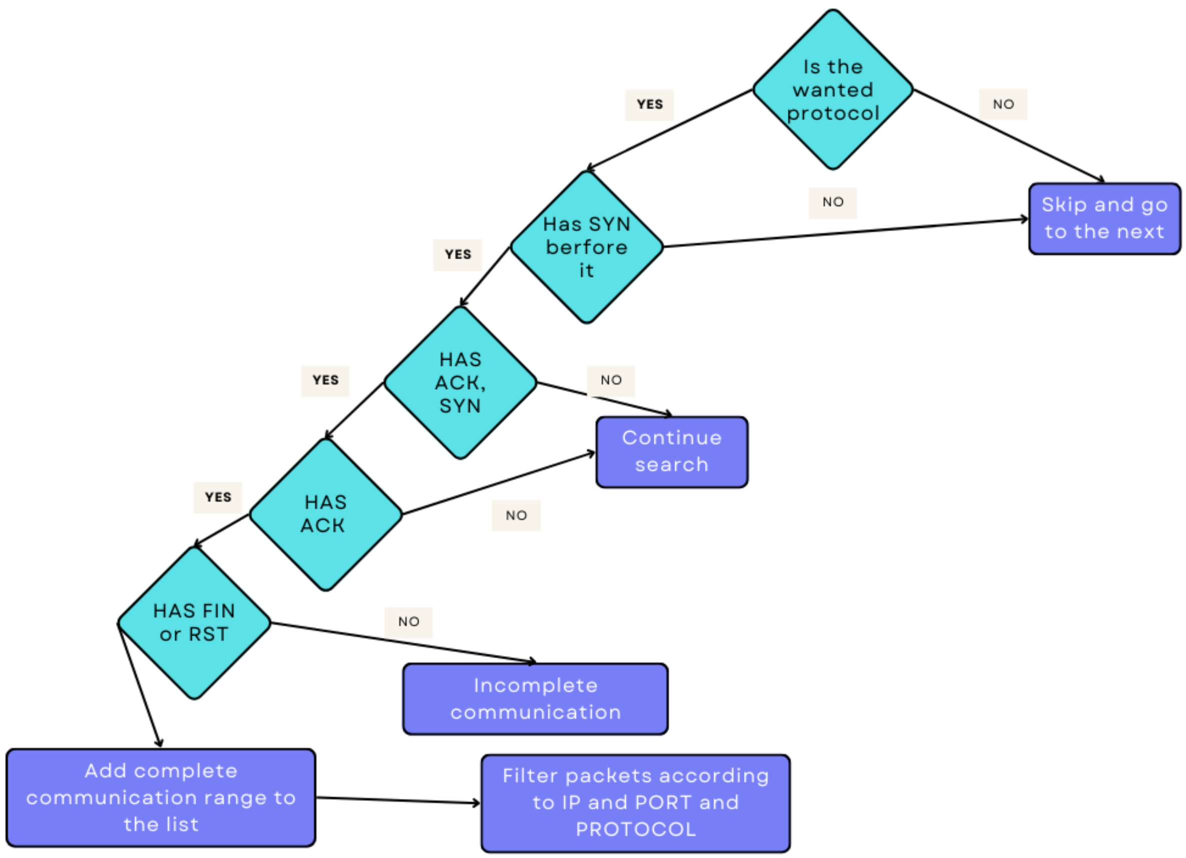
```python
    if sys.argv[1] != "-p":
        print("Incorrect argument. Try -p")
        exit()

    if not check_protocol_exists(filter):
        print("Protocol doesnt exists")
        exit()

    for key in dictionaries:
        if filter in dictionaries[key].values():
            if key == "tcpPortTypes":
                filter_type = "TCP"
            elif key == "udpPortTypes":
                filter_type = "UDP"
            elif key == "etherTypes":
                filter_type = "Ether"
            break
    if pcap != None:
        print(pcap)
        analyze_frames(pcap_file=pcap, filter=filter, filter_type=filter_type)
    else:
        analyze_frames(filter=filter, filter_type=filter_type)
```

After that the packeges are filtered in the modify_ethernet_object () function.

```python
if is_filtering and filter_type == "TCP":
    try:
        if packet_object["protocol"] != "TCP":
            return None
    except:
        return None

if is_filtering and filter_type == "UDP":
    try:
        if packet_object["protocol"] != "UDP":
            return None
    except:
        return None

if is_filtering and filter_type == "Ether":
    if ether_type != filter:
        return None

if is_filtering and filter_type == "IP":
    try:
        if packet_object["protocol"] != filter:
            return None
    except:
        return None
```

This is done for every packet one by one.

TCP handshake and communication detection



First, I checked the handshake then if a complete connection was the result I added the range of packets inside of that connection, then in the next step I filtered all those packets according to IP, PORT and filter PROTOCOL which was passed into the function.

UDP communication pairing

With each TFTP packet I added a connection object to the connection list, after the TFTP packet, I saved the ports and IP addresses for the next packet and all following packets were compared to these ports and IP addresses, if they did match than they were part of the same connection. Communications which were not preceded by TFTP packet were only paired by port and ip address. The first IP and port addresses were saved and if the next matched the IPs and the ports than it belonged to the same communication.

```python
#TFT packet
if "app_protocol" in packet and packet["app_protocol"] == "TFTP":
    print(packet["frame_number"])
    previously_added = True
    communications.append({
            "number_comm": len(communications) + 1,
            "src_comm": src_ip,
            "dst_comm": dst_ip,
            "packets": [deepcopy(packet)]
        })
    continue

if previously_added:
    previously_added = False
    comm_src_port = packet["src_port"]
    comm_dst_port = packet["dst_port"]
    comm_src_ip = packet["src_ip"]
    comm_dst_ip = packet["dst_ip"]
    comm_packet_index = 1

    communications[len(communications)-1]["packets"].append(packet)
elif (comm_src_ip == src_ip and comm_src_port == packet["src_port"]) or (comm_dst_ip == src_ip and comm_dst_port =
    communications[len(communications)-1]["packets"].append(packet)
```

ARP

In the case of ARP I looped through the ARP packages one by one, if a REQUEST packet is found the program starts another loop from that packet and searches until it finds a REPLY for that exact Ip address. If along the way it finds a REQUEST packet with the same addresses it adds to a temporary list and these are added when a current REPLY if found in that way supporting N*REQUEST. Added REPLY and REQUEST (after the first one) are removed from the list.
Then these complete communications are appended to the return object, which will contain all communications and will be returned by the function.

```python
packets = [deepcopy(packet)]

for same_packet in temp_same_requests:
    packets.append(deepcopy(same_packet))
    frames_database["packets"].remove(same_packet)

packets.append(deepcopy(sub_packet))

return_frames["complete_comms"].append(
    {
        "number_comm": len(return_frames["complete_comms"]) + 1,
        "src_comm": src_ip,
        "dst_comm": dst_ip,
        "packets": packets,
    }
)
response_found = True
frames_database["packets"].remove(sub_packet)
break
```

ICMP

For ICMP protocol I first when filtering I filled out the ICMP type where this field existed, then similar to method used for ARP with a for loop I went through the loop and for each request I tried to find a response with a nested loop, if a reply was found the communication was added to the complete communications list and the reply packet was deleted.

For every request packet it tries to find and answer packet meaning the destination and the source ips of the source are flipped and contains the keyword “Reply”.

```python
found_pair = False
for j in range(i,len(frames_database["packets"])):
    resp_packet = frames_database["packets"][j]
    if "src_ip" not in resp_packet or "dst_ip" not in resp_packet:
        continue

    if src_ip == resp_packet["dst_ip"] and dst_ip == resp_packet["src_ip"] and resp_packet["icmp_type"] != None and "Reply" in
        return_frames["complete_comms"].append({
                    "number_comm": len(return_frames["complete_comms"]) + 1,
                    "src_comm": src_ip,
                    "dst_comm": dst_ip,
                    "packets": [deepcopy(packet), deepcopy(resp_packet)],
        })
        frames_database["packets"].remove(resp_packet)
        found_pair = True
        break

if not found_pair:
    return_frames["partial_comms"].append({
                "number_comm": len(return_frames["partial_comms"]) + 1,
                "packets": [deepcopy(packet)],
        })

i = i+1
```

Final results

All task were successfully completed and implemented. Every output yaml file was tested with the validator and were successfully validated. For validation I used a simple **tester.py** python script which tested the different protocols and filters on each capture file and then validated the results using the validator.

Further extensibility

The program can be easily extended by additional scripts. Due to the codes more procedural nature in most cases the man.py file will have to be modified. New protocol support can be easily added by appending to the protocol text files. New protocol and header analyzation also can be added pretty easily by calling it in the analyze_frames() function.

Examples of YAML exports and "interface"

UDP TFTP communication


```yaml
---
name: PKS2022/23
pcap_name: trace-15.pcap
filter_name: UDP
complete_comms:
  - number_comm: 1
    src_comm: 12.0.0.1
    dst_comm: 12.0.0.5
    packets:
      - frame_number: 23
        frame_type: Ethernet II
        len_frame_pcap: 70
        len_frame_medium: 74
        dst_mac: 02:00:4C:4F:4F:50
        src_mac: CC:08:09:D4:00:00
        ether_type: IPv4
        protocol: UDP
        src_ip: 12.0.0.1
        dst_ip: 12.0.0.5
        src_port: 49917
        dst_port: 69
        app_protocol: TFTP
        hexa_frame: |
          02 00 4C 4F 4F 50 CC 08 09 D4 00 00 08 00 45 00
          00 38 00 00 00 00 FF 11 A3 AF 0C 00 00 01 0C 00
          00 05 C2 FD 00 45 00 24 86 3B 00 01 63 6D 65 2D
          67 75 69 2D 34 2E 31 2E 30 2E 31 2E 74 61 72 00
          6F 63 74 65 74 00
        opcode: File not found
      - frame_number: 24
        frame_type: Ethernet II
        len_frame_pcap: 558
        len_frame_medium: 562
        dst_mac: CC:08:09:D4:00:00
        src_mac: 02:00:4C:4F:4F:50
        ether_type: IPv4
        protocol: UDP
        src_ip: 12.0.0.5
        dst_ip: 12.0.0.1
        src_port: 1510
        dst_port: 49917
        hexa_frame: |
```

TCP communication using FTP-CONTROL filter


```yaml
---
name: PKS2022/23
pcap_name: eth-4.pcap
filter_name: FTP-CONTROL
complete_comms:
  - number_comm: 1
    src_comm: 192.168.1.33
    dst_comm: 193.0.6.140
    packets:
      - frame_number: 3
        frame_type: Ethernet II
        len_frame_pcap: 62
        len_frame_medium: 66
        dst_mac: 00:02:CF:AB:A2:4C
        src_mac: 00:14:38:06:E0:93
        ether_type: IPv4
        protocol: TCP
        src_ip: 192.168.1.33
        dst_ip: 193.0.6.140
        src_port: 3742
        dst_port: 21
        app_protocol: FTP-CONTROL
        hexa_frame: |
          00 02 CF AB A2 4C 00 14 38 06 E0 93 08 00 45 00
          00 30 14 67 40 00 80 06 5D 0B C0 A8 01 21 C1 00
          06 8C 0E 9E 00 15 0E 97 37 4F 00 00 00 00 70 02
          FF FF A5 30 00 00 02 04 05 B4 01 01 04 02
      - frame_number: 4
        frame_type: Ethernet II
        len_frame_pcap: 62
        len_frame_medium: 66
        dst_mac: 00:14:38:06:E0:93
        src_mac: 00:02:CF:AB:A2:4C
        ether_type: IPv4
        protocol: TCP
        src_ip: 193.0.6.140
        dst_ip: 192.168.1.33
        src_port: 21
        dst_port: 3742
        app_protocol: FTP-CONTROL
        hexa_frame: |
          00 14 38 06 E0 93 00 02 CF AB A2 4C 08 00 45 00
          00 30 D4 15 40 00 F1 06 2C 5C C1 00 06 8C C0 A8
          01 21 00 15 0E 9E 9A 70 A0 AF 0E 97 37 50 70 12
          10 2C 5B 24 00 00 02 04 05 64 04 02 00 00
      - frame_number: 5
        frame_type: Ethernet II
        len_frame_pcap: 54
```

No filter – trace-26.pcap


```
      73 2C 20 49 6E 63 2E 0A 43 6F 6D 70 69 6C 65 64
      20 4D 6F 6E 20 32 38 2D 4A 61 6E 2D 31 33 20 31
      30 3A 31 30 20 62 79 20 70 72 6F 64 5F 72 65 6C
      5F 74 65 61 6D 08 10 46 61 73 74 45 74 68 65 72
      6E 65 74 30 2F 32 34 0E 04 00 14 00 04 10 0C 05
      01 0A 14 1E FE 03 00 00 00 01 00 FE 06 00 80 C2
      01 00 01 FE 09 00 12 0F 01 03 6C 00 00 10 00 00
  - frame_number: 2
    frame_type: IEEE 802.3 LLC
    len_frame_pcap: 60
    len_frame_medium: 64
    dst_mac: 01:80:C2:00:00:00
    src_mac: 00:16:47:02:24:1A
    sap: STP
    hexa_frame: |
      01 80 C2 00 00 00 00 16 47 02 24 1A 00 26 42 42
      03 00 00 00 00 00 80 01 00 16 47 02 24 00 00 00
      00 00 80 01 00 16 47 02 24 00 80 1A 00 00 14 00
      02 00 0F 00 00 00 00 00 00 00 00 00
  - frame_number: 3
    frame_type: Ethernet II
    len_frame_pcap: 60
    len_frame_medium: 64
    dst_mac: 00:16:47:02:24:1A
    src_mac: 00:16:47:02:24:1A
    ether_type: ECTP
    hexa_frame: |
      00 16 47 02 24 1A 00 16 47 02 24 1A 90 00 00 00
      01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      00 00 00 00 00 00 00 00 00 00 00 00
  - frame_number: 4
    frame_type: IEEE 802.3 LLC
    len_frame_pcap: 60
    len_frame_medium: 64
    dst_mac: 01:80:C2:00:00:00
    src_mac: 00:16:47:02:24:1A
    sap: STP
    hexa_frame: |
      01 80 C2 00 00 00 00 16 47 02 24 1A 00 26 42 42
      03 00 00 00 00 00 80 01 00 16 47 02 24 00 00 00
```