

Documentation PKS

Author: Zaitsev Artem

Assignment 1

External libraries:

```
$ pip freeze
PyYAML==6.0.1
ruamel.yaml==0.17.33
ruamel.yaml.clib==0.2.7
scapy==2.5.0
yamale==4.0.4
```

The proposed mechanism for protocol analysis on individual layers

Actually on each individual layer mechanism for protocol analysis is almost the same. We take bytes which represents a protocol (for example 12-13 bytes of the whole frame) and convert from byte-code to HEX, finally we compare it with a list of possible protocols. Sometimes we have to do some additional computation, for example for 12-13 bytes we should convert it to decimal and compare to 1500, if it is smaller than 1500 it is IEEE 802.3 something and those 2 bytes represent length of whole frame, while if it is bigger than 1500 it is Ethernet II and those bytes represent Ether-type.

In python to convert from bytes to HEX I used `f'{byte:02x}'` -> string format and to convert from HEX to Decimal I used `int(HEX, 16)`.

My solution consists of 4 .py files:

- main.py

- analyzer.py
- filters.py
- Frame.py

Frame.py is a class which represents pcap packet's frame. It contains constructor method - `__init__` which calls every needed method to identify needed information from byte code of frame.

Except for usual information (byte code and number) we are passing dictionaries `first_check` and `second_check` to `__init__`. Those are dictionaries which contains information from external files.

I believe that code in `__init__` is mostly self explanatory:

If we need some info we call method from which we can get that info.

There are methods in `Frame.py` which are needed to get some info from byte code:

- `get_destination_from_byte_code()`
- `get_source_from_byte_code()`
- `check_type(first_check)`
- `ether_type_check(second_check)`
- `check_tcp_flags()`
- `check_tftp_opcode()`
- `check_icmp()`
- `check_arp_opcode()`
- `create_hex_code()`

`check_type()` is used for checking if frame is Ethernet II or IEEE 802.3 or IEEE 802.3 LLC or 802.3 LLC & SNAP.

It returns frame type, sap and pid if frame has one.

`ether_type_check()` is used for checking ether type.

It returns ether type, IPv4 protocol, source and destination IPs and ports and TCP/UDP protocol – if it is possible to define.

`Check_icmp()` is used for checking ICMP protocol frame info.

It returns ICMP type, id and seq numbers if frame has this info.

I believe all other methods above are self explanatory as they return only stuff which is defined in their names.

`check_arp_opcode()` returns arp opcode, `check_tftp_flags` returns tftp flags, etc...

Every other method in `Frame.py` are getters:

There is only one interesting getter:

`get_item()`

This method constructs and returns `yaml.map()` object representing frame.

filters.py consist of 4 filter functions:

- `tcp_filter(frame_list, protocol_name)`
- `tftp_filter(frame_list)`
- `icmp_filter(frame_list)`
- `arp_filter(frame_list)`

`tcp_filter(frame_list, protocol_name)`

1) Using `frame_list` and `protocol_name` we create a list of frames with app protocol defined by `protocol_name`.

- 2) Using new `frame_list` we find every unique pair of `source_ip:source_port – destination_ip:destination_port`
- 3) Using `ip_pairs_list` we create dictionary where key = `ip_pair` and value – list of frames where this `ip_pair` is found.
- 4) In each list of frames from dictionary we are checking first 3 or 4 frames to do 3-way or 4-way handshake to establish communication
- 5) In each list of frames from dictionary we are checking last 4 frames to do 4-way handshake to terminate communication.
- 6) If last frame of list has RST flag – communication was terminated with RESET
- 7) If 4 == true and (5 == true or 6 == true) we add communication `frame_list` to `coms_dict` as a complete communication
- 8) If 4 == false this means communication was never established in this pcap file which means it is `partial_com` and we add it to `coms_dict` as partial communication
- 9) If 5 == false and 6 == false this means communication was never terminated in this pcap file which means it is `partial_com` and we add it to `coms_dict` as partial communication
- 10) When every value from dictionary is checked we return `coms_dict`.

tftp_filter(frame_list)

- 1) Using `frame_list` we create a list of frames with `ipv4_protocol == UDP`
- 2) Using new `frame_list` we find every unique pair of `source_ip:source_port – destination_ip:destination_port`
 - 2.1) To define ports correctly we check `tftp_opcode()`, if `opcode == read` we take ports from next frame, if `opcode == write` we take ports from frame after next frame.

- 3) Using `ip_pairs_list` we create dictionary where `key = ip_pair` and `value` – list of frames where this `ip_pair` is found.
- 4) In each list of frames we check if the last frame has `err` or `ack tftp opcode`.
- 5) In each list of frames we check if second to last frame length is equal to first data frame length and if second to last frame not equal to first data frame (It is done to check if all data frames where transferred, as the last data frame must have length < first(every other) data frame)
- 6) If (4 == true and 5 == true) or 4 == false we add list of frames to `coms_dict` as partial communication
- 7) If 4 == true and 5 == false we add list of frames to `coms_dict` as complete communication
- 8) When every value from dictionary is checked we return `coms_dict`.

`icmp_filter(frame_list)`

- 1) Using `frame_list` we create a list of frames with `ipv4_protocol == ICMP`
- 2) Using new frame list we create ip pairs in case `icmp type == "Reply", "Request" or "Time Exceeded"` otherwise we add frame to `frames_dict` with frame number as a key.
- 3) Using `ip_pairs_list` we create dictionary where `key = ip_pair` and `value` – list of frames where this `ip_pair` is found.
 - 3.1) While creating pairs if we find frame with Request type we save it's `ip_pair` to `previous_ip_pair`
 - 3.2) If we find frame with Time Exceeded we load saved previous `ip_pair` from `previous_ip_pair` to `ip_pair` and continue to do as we would do in 3.
(that way we save Time Exceeded icmp frames to same communication with it's requests)

- 4) We move every value from dictionary to frames_dict with a first frame number as a key.
- 5) We sort frames_dict to sort communications
- 6) If length of communication can be divided by 2 and starts with Requests and finishes with Reply or Time Exceeded -> it is completed communication, so we add it to coms_dict as complete communication.
- 7) Else we add this communication to coms_dict as partial communication.
- 8) When every value from dictionary is checked we return coms_dict.

arp_filter(frame_list)

- 1) Using frame_list we create a list of frames with ether_type == ARP
- 2) Using new frame_list we find each Request arp frame and save it's destination ip to list
- 3) Using arp frame list we find frames with Request opcode and if it's destination ip address is in destination ip list we add this frame to the list inside destination ip dictionary with destination ip key.
- 4) Using arp frame list we find frames with Reply opcode and if it's source ip address is in destination ip list we add this frame to the list inside destination ip dictionary with destination ip key.
- 5) We count difference between amount of Reply and Request opcodes in which list inside dictionary.
- 6) If diffence is lower than 0 -> there are more Replies from the ip than Request to it. We delete all Requests from that list and add it coms_dict as partial communication

- 7) If difference is higher or equal to 0 -> there are equal amount or more Requests than Replies. We delete all Requests from the end of the list and add them to part_com_requests until we find Reply and add this list to coms_dict as complete communication
- 8) If there are elements in part_com_requests we add it to coms_dict as partial communication.
- 9) When every value from dictionary is checked we return coms_dict.

All filters have +- same logic with some differences for ip_pair creating

analyzer.py – there are some functions to print our output to yaml file, read pcap file, read external files. Call Frame constructor to parse frame from bytes. Call filters to filter frames:

- get_name(pcap_path)
 - print_in_yaml(pcap_path, mode)
 - senders_yaml (sender)
 - read_protocols()
 - communication_item(key, value, mode)
 - communication_block(comp_coms,mode)
- get_name – return file's name without path.

print_in_yaml – main function of analyzer – takes path to pcap and mode. Reads pcap file and gives packets to for-cycle in which every frame of file is parsed and counts

senders if filter mode is not set it prints all read frames to yaml output file and also prints senders.

If filter mode is set than it calls specified filter passes it's return value to communication_block , then prints communication_block return values to yaml output

senders_yaml – takes sender as tuple and returns nested item as Yaml.map() to

read_protocols – simply reads protocols from external files and returns it to print_in_yaml where it is combined into 2 lists of dictionaries and passed to functions where are needed.

Communication_item – takes key and value – decrypt key into number of communication, adds value's info to yaml.map() nested item, and returns it.

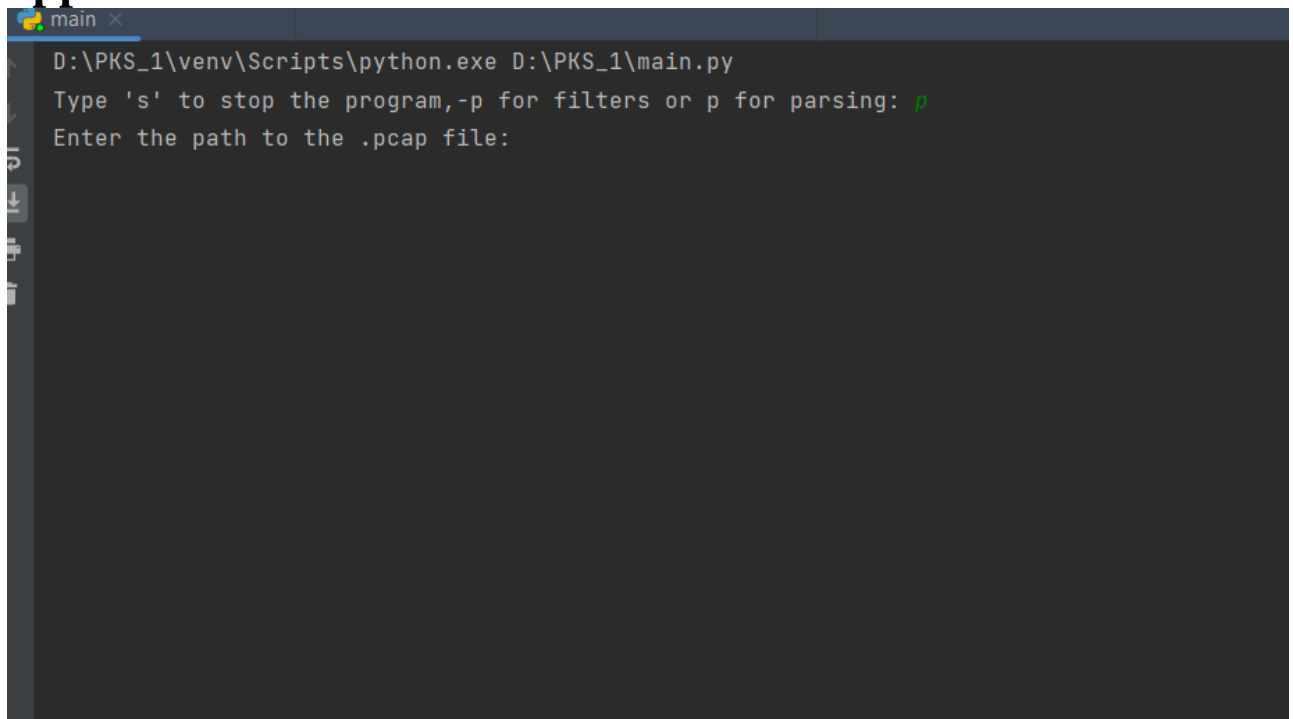
Communication_block takes coms_dict and, evaluate whether communication from coms_dict is partial or complete communication, appends return of communication_item to communication_seq if communication is complete.

If communication is partial – adds 1 partial communication to part_communication_seq if filter – one of TCP, else adds all partial communications to sequence, returns sequences.

main.py:
main():

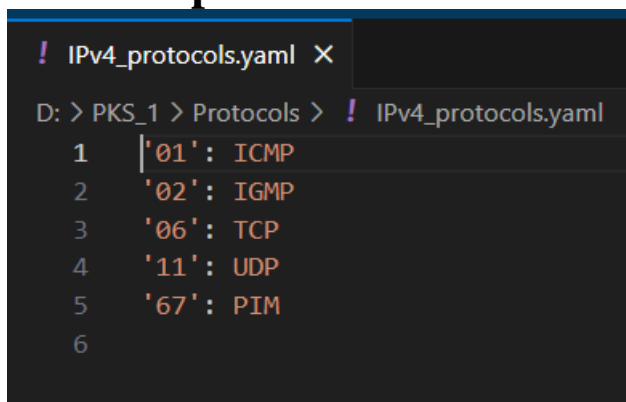
waits 's' to be inserted to stop program, if '-p' is given asks for Filter name and pcap_path relative to "Input\" and calls print_in_yaml with filter, if 'p' is given asks for pcap_path relative to "Input\" and calls print_in_yaml without filter.

User interface of program is represented as console application.



```
main x
D:\PKS_1\venv\Scripts\python.exe D:\PKS_1\main.py
Type 's' to stop the program, -p for filters or p for parsing: p
Enter the path to the .pcap file:
```

An example of external file:



```
! IPv4_protocols.yaml x
D: > PKS_1 > Protocols > ! IPv4_protocols.yaml
1  |'01': ICMP
2  |'02': IGMP
3  |'06': TCP
4  |'11': UDP
5  |'67': PIM
6
```

Chosen implementation environment

For implementation in Python I used PyCharm IDE.
I like it. That's why I choose it.

Evaluation and possible extensions:

If I was asked how I would evaluate my own solution and implementation, I would say that it's organized enough and easy to extend. Eventhough I believe there might be some parts of the code which could be written better, for example using different data structures, as for the whole solution only 3 "default" python data structures were used (list, dictionary and tuple). I believe in some places a queue could be used as well. Nevertheless, I believe most of my solution is done good and efficient enough.

Representing every Frame as an object of class Frame seems efficient for me as for example, if in the process of filtering I'd need some information about protocol on the specific layer, I don't have to look for specific bytes and analyze them. I can just ask "What is this layer protocol value?" and get an answer from class.

User guide:

All pcap files which you want to use in program you should insert to Input folder.

All yaml outputs will be in Output/"MODE"

-p is made through console app not through flag and argparse.