Beginners Guide to TShark (Part 3)

February 28, 2020 By Raj Chandel

This is the third instalment in the Beginners Guide to TShark Series. Please find the first and second instalments below.

- **Beginners Guide to TShark (Part 1)**
- **Beginners Guide to TShark (Part 2)**

TL; DR

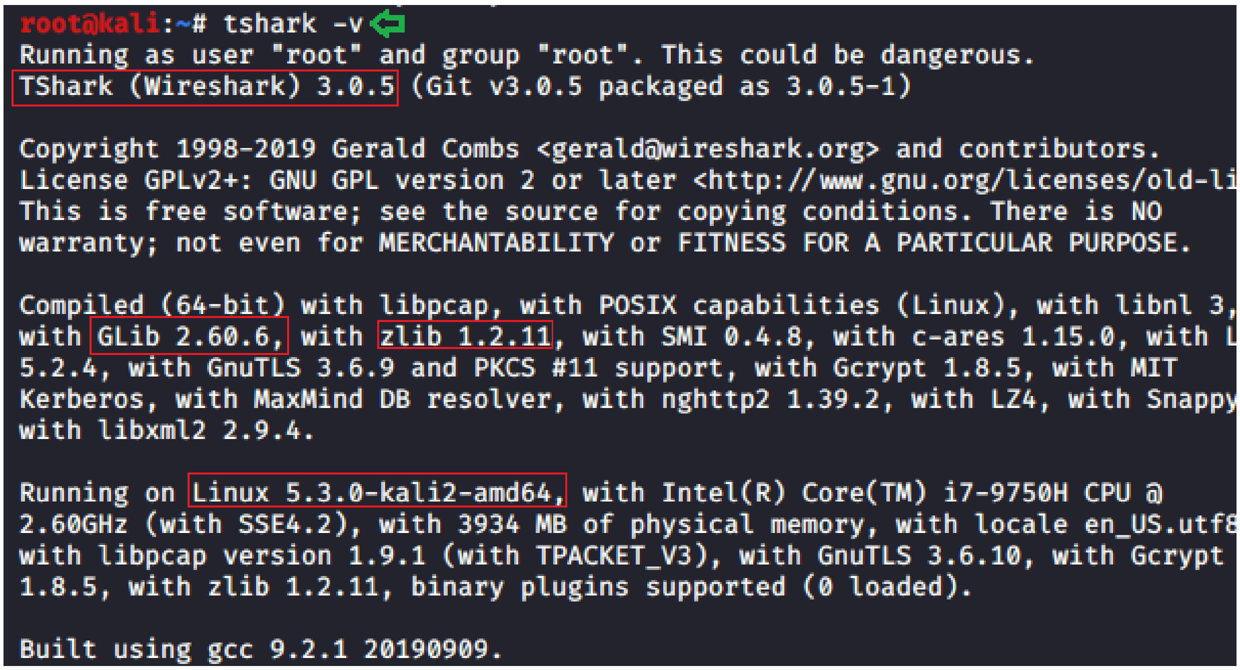
In this part, we will understand the reporting functionalities and some additional tricks that we found while tinkering with TShark.

Table of Content

Version Information

Let’s begin with the very simple command so that we can understand and correlate that all the practicals performed during this article and the previous articles are of the version depicted in the image given below. This parameter prints the Version information of the installed TShark.

```
tshark -v
```



```
root@kali:~# tshark -v
Running as user "root" and group "root". This could be dangerous.
TShark (Wireshark) 3.0.5 (Git v3.0.5 packaged as 3.0.5-1)

Copyright 1998-2019 Gerald Combs <gerald@wireshark.org> and contributors.
License GPLv2+: GNU GPL version 2 or later <http://www.gnu.org/licenses/old-li
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Compiled (64-bit) with libpcap, with POSIX capabilities (Linux), with libnl 3,
with GLib 2.60.6, with zlib 1.2.11, with SMI 0.4.8, with c-ares 1.15.0, with L
5.2.4, with GnuTLS 3.6.9 and PKCS #11 support, with Gcrypt 1.8.5, with MIT
Kerberos, with MaxMind DB resolver, with nghttp2 1.39.2, with LZ4, with Snappy
with libxml2 2.9.4.

Running on Linux 5.3.0-kali2-amd64, with Intel(R) Core(TM) i7-9750H CPU @
2.60GHz (with SSE4.2), with 3934 MB of physical memory, with locale en_US.utf8
with libpcap version 1.9.1 (with TPACKET_V3), with GnuTLS 3.6.10, with Gcrypt
1.8.5, with zlib 1.2.11, binary plugins supported (0 loaded).

Built using gcc 9.2.1 20190909.
```

Reporting Options

During any Network capture or investigation, there is a dire need of the reports so that we can share the findings with the team as well as superiors and have a validated proof of any activity inside the network. For the same reasons, TShark has given us a beautiful option (**-G**). This option will make the TShark print a list of several types of reports that can be generated. Official Manual of TShark used the word Glossaries for describing the types of reports.

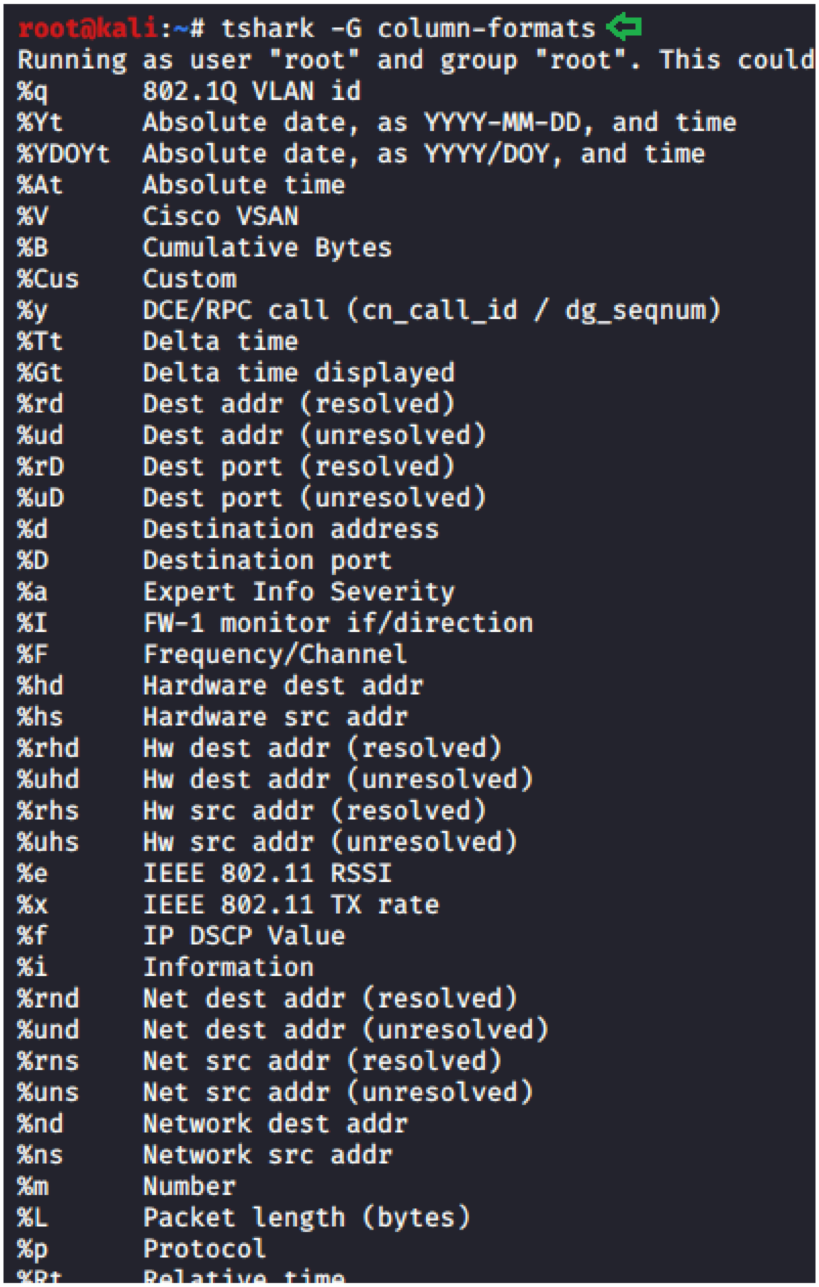
```
tshark -G help
```

```
root@kali:~# tshark -G help
Running as user "root" and group "root". This could be dangerous.
TShark (Wireshark) 3.0.5 (Git v3.0.5 packaged as 3.0.5-1)

Usage: tshark -G [report]

Glossary table reports:
  -G column-formats        dump column format codes and exit
  -G decodes               dump "layer type"/"decode as" associations and exit
  -G dissector-tables      dump dissector table names, types, and properties
  -G elastic-mapping       dump ElasticSearch mapping file
  -G fieldcount            dump count of header fields and exit
  -G fields                dump fields glossary and exit
  -G ftypes                dump field type basic and descriptive names
  -G heuristic-decodes     dump heuristic dissector tables
  -G plugins               dump installed plugins and exit
  -G protocols             dump protocols in registration database and exit
  -G values                dump value, range, true/false strings and exit

Preference reports:
  -G currentprefs          dump current preferences and exit
  -G defaultprefs          dump default preferences and exit
  -G folders               dump about:folders
```

Column Formats

From our previous practicals, we saw that we have the Column Formats option available in the reporting section of TShark. To explore its contents, we ran the command as shown in the image given below. We see that it prints a list of wildcards that could be used while generating a report. We have the VLAN id, Date, Time, Destination Address, Destination Port, Packet Length, Protocol, etc.

```
tshark -G column-formats
```


```
root@kali:~# tshark -G column-formats
Running as user "root" and group "root". This could
%q      802.1Q VLAN id
%Yt     Absolute date, as YYYY-MM-DD, and time
%YDOYt  Absolute date, as YYYY/DOY, and time
%At     Absolute time
%V      Cisco VSAN
%B      Cumulative Bytes
%Cus    Custom
%y      DCE/RPC call (cn_call_id / dg_seqnum)
%Tt     Delta time
%Gt     Delta time displayed
%rd     Dest addr (resolved)
%ud     Dest addr (unresolved)
%rD     Dest port (resolved)
%uD     Dest port (unresolved)
%d      Destination address
%D      Destination port
%a      Expert Info Severity
%I      FW-1 monitor if/direction
%F      Frequency/Channel
%hd     Hardware dest addr
%hs     Hardware src addr
%rhd    Hw dest addr (resolved)
%uhd    Hw dest addr (unresolved)
%rhs    Hw src addr (resolved)
%uhs    Hw src addr (unresolved)
%e      IEEE 802.11 RSSI
%x      IEEE 802.11 TX rate
%f      IP DSCP Value
%i      Information
%rnd    Net dest addr (resolved)
%und    Net dest addr (unresolved)
%rns    Net src addr (resolved)
%uns    Net src addr (unresolved)
%nd     Network dest addr
%ns     Network src addr
%m      Number
%L      Packet length (bytes)
%p      Protocol
%Rt     Relative time
```

Decodes

This option generates 3 Fields related to Layers as well as the protocol decoded. There is a restriction enforced for one record per line with this option. The first field that has the "s1ap.proc.sout" tells us the layer type of the network packets. Followed by that we have the value of selector in decimal format. At last, we have the decoding that was performed on the capture. We used the head command as the output was rather big to fit in the screenshot.

```
tshark -G decodes | head
```

```
root@kali:~# tshark -G decodes | head
Running as user "root" and group "root". This could be dangerous.
s1ap.proc.sout  17      s1ap
s1ap.proc.sout  3       s1ap
s1ap.proc.sout  6       s1ap
s1ap.proc.sout  23      s1ap
s1ap.proc.sout  9       s1ap
s1ap.proc.sout  48      s1ap
s1ap.proc.sout  43      s1ap
s1ap.proc.sout  29      s1ap
s1ap.proc.sout  4       s1ap
s1ap.proc.sout  21      s1ap
```

Dissector Tables

Most of the users reading this article are already familiar with the concept of Dissector. If not, in simple words Dissector is simply a protocol parser. The output generated by this option consists of 6 fields. Starting from the Dissector Table Name then the name is used for the dissector table in the GUI format. Next, we have the type and the base for the display and the Protocol Name. Lastly, we have the decode as a format.

```
root@kali:~# tshark -G dissector-tables
Running as user "root" and group "root". This could be dangerous.
amqp.version    AMQP versions   FT_UINT8        BASE_DEC        AMQP    Decode As supported
ansi_637.tele_id        ANSI IS-637-A Teleservice ID    FT_UINT8        BASE_DEC        ANSI IS-637-A Te
ted
ansi_a.ota      IS-683-A (OTA)  FT_UINT8        BASE_DEC        ANSI BSMAP      Decode As not supported
ansi_a.pld      IS-801 (PLD)    FT_UINT8        BASE_DEC        ANSI BSMAP      Decode As not supported
ansi_a.sms      IS-637-A (SMS)  FT_UINT8        BASE_DEC        ANSI BSMAP      Decode As not supported
ansi_map.ota    IS-683-A (OTA)  FT_UINT8        BASE_DEC        ANSI MAP        Decode As not supported
ansi_map.pld    IS-801 (PLD)    FT_UINT8        BASE_DEC        ANSI MAP        Decode As not supported
ansi_map.tele_id        IS-637 Teleservice ID   FT_UINT8        BASE_DEC        ANSI MAP        Decode A
ansi_tcap.nat.opcode    ANSI TCAP National Opcodes      FT_UINT16       BASE_DEC        ANSI_TCAP
ansi_tcap.ssn   ANSI SSN        FT_UINT8        BASE_DEC        TCAP    Decode As not supported
arcnet.protocol_id      ARCNET Protocol ID      FT_UINT8        BASE_HEX        ARCNET  Decode As not su
aruba_erm.type  Aruba ERM Type  FT_NONE ARUBA_ERM       Decode As supported
atm.aal2.type   ATM AAL_2 type  FT_UINT32       BASE_DEC        ATM     Decode As supported
atm.aal5.type   ATM AAL_5 type  FT_UINT32       BASE_DEC        ATM     Decode As not supported
atm.cell_payload.vpi_vci        ATM Cell Payload VPI VCI        FT_UINT32       BASE_DEC        ATM
atm.reassembled.vpi_vci ATM Reassembled VPI VCI FT_UINT32       BASE_DEC        ATM     Decode As not su
awdl.tag.number AWDL Tags       FT_UINT8        BASE_DEC        AWDL    Decode As not supported
ax25.pid        AX.25 protocol ID       FT_UINT8        BASE_HEX        AX.25   Decode As not supported
bacapp.vendor_identifier        BACapp Vendor Identifier        FT_UINT8        BASE_HEX        BACapp
bacnet.vendor   BACnet Vendor Identifier        FT_UINT8        BASE_HEX        BACnet  Decode As not su
bacp.option     PPP BACP Options        FT_UINT8        BASE_DEC        PPP BACP        Decode As not su
bap.option      PPP BAP Options FT_UINT8        BASE_DEC        PPP BAP Decode As not supported
bcp_ncp.option  PPP BCP NCP Options     FT_UINT8        BASE_DEC        PPP BCP NCP     Decode As not su
bctp.tpi        BCTP Tunneled Protocol Indicator        FT_UINT32       BASE_DEC        BCTP    Decode A
```

Elastic Mapping

Mapping is the outline of the documents stored in the index. Elasticsearch supports different data types for the fields in a document. The elastic-mapping option of the TShark prints out the data stored inside the ElasticSearch mapping file. Due to a large amount of data getting printed, we decided to use the head command as well.

```
tshark -G elastic-mapping | head
```

```
root@kali:~# tshark -G elastic-mapping | head
Running as user "root" and group "root". This could be dangerous.
{
  "template": "packets-*",
  "settings": {
    "index.mapping.total_fields.limit": 1000000
  },
  "mappings": {
    "pcap_file": {
      "dynamic": false,
      "properties": {
        "timestamp": {
```

Field Count

There are times in a network trace, where we need to get the count of the header fields travelling at any moment. In such scenarios, TShark got our back. With the fieldcount option, we can print the number of header fields with ease. As we can observe in the image given below that we have 2522 protocols and 215000 fields were pre-allocated.

```
tshark -G fieldcount
```

```
root@kali:~# tshark -G fieldcount
Running as user "root" and group "root". This could be dangerous
There are 214494 header fields registered, of which:
        0 are deregistered
        2522 are protocols
        16070 have the same name as another field

215000 fields were pre-allocated.

The header field table consumes 1679 KiB of memory.
The fields themselves consume 15081 KiB of memory.
```

Fields

TShark can also get us the contents of the registration database. The output generated by this option is not as easy to interpret as the others. For some users, they can use any other parsing tool for generating a better output. Each record in the output is a protocol or a header file. This can be differentiated by the First field of the record. If the Field is P then it is a Protocol and if it is F then it’s a header field. In the case of the Protocols, we have 2 more fields. One tells us about the Protocol and other fields show the abbreviation used for the said protocol. In the case of Header, the facts are a little different. We have 7 more fields. We have the Descriptive Name, Abbreviation, Type, Parent Protocol Abbreviation, Base for Display, Bitmask, Blurb Describing Field, etc.

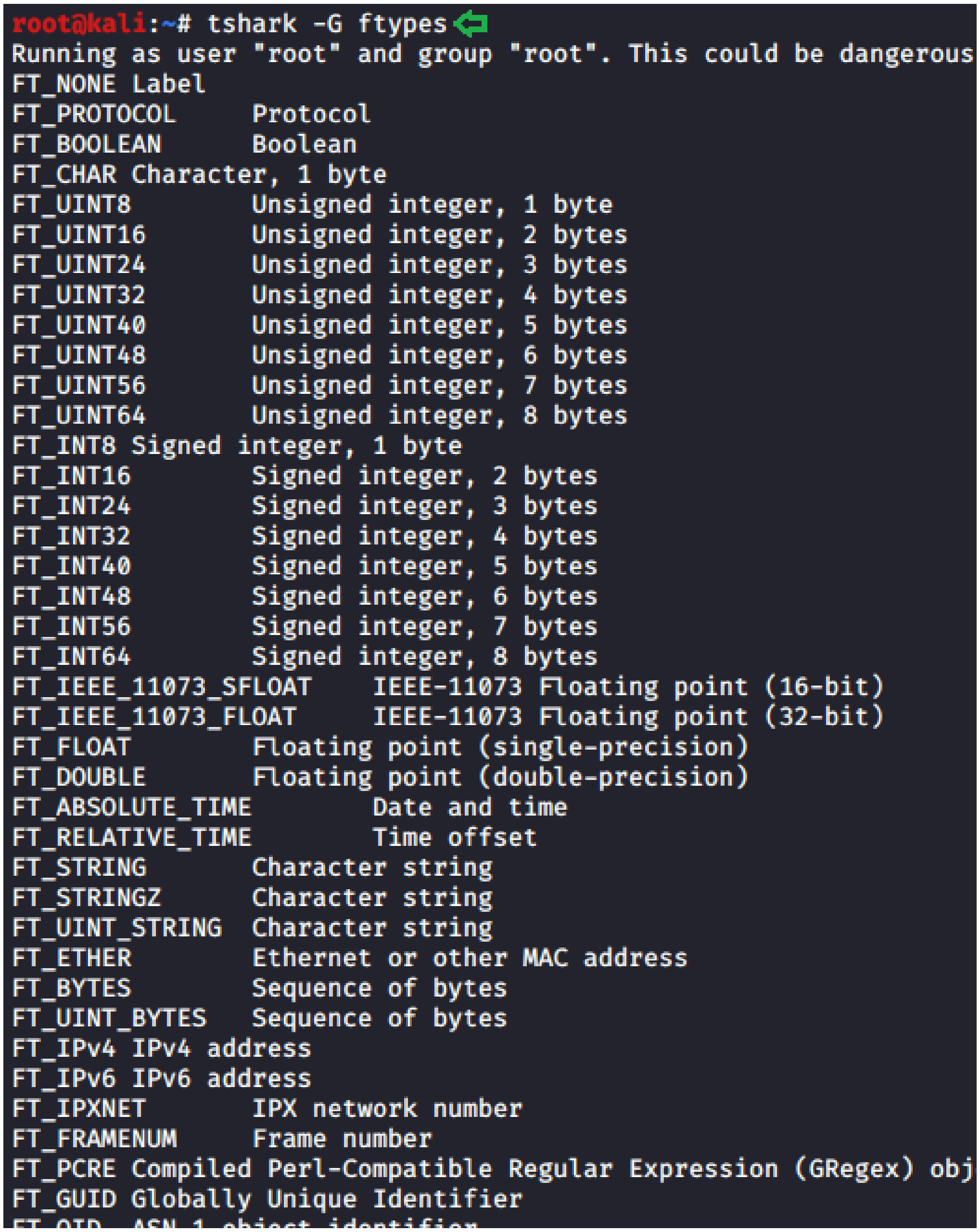
```
tshark -G fields | head
```

```
root@kali:~# tshark -G fields | head
Running as user "root" and group "root". This could be dangerous.
P       Short Frame     _ws.short
P       Malformed Packet        _ws.malformed
P       Unreassembled Fragmented Packet _ws.unreassembled
F       Dissector bug   _ws.malformed.dissector_bug     FT_NONE _ws.malformed
F       Reassembly error        _ws.malformed.reassembly        FT_NONE _ws.malformed
F       Malformed Packet (Exception occurred)   _ws.malformed.expert    FT_NONE _ws.ma
P       Type Length Mismatch    _ws.type_length
F       Trying to fetch X with length Y _ws.type_length.mismatch        FT_NONE _ws.ty
P       Number-String Decoding Error    _ws.number_string.decoding_error
F       Failed to decode number from string     _ws.number_string.decoding_error.faile
x0
```

Fundamental Types

TShark also helps us generate a report centralized around the fundamental types of network protocol. This is abbreviated as ftype. This type of report consists of only 2 fields. One for the FTYPE and other for its description.

```
tshark -G ftypes
```



```
root@kali:~# tshark -G ftypes
Running as user "root" and group "root". This could be dangerous
FT_NONE Label
FT_PROTOCOL     Protocol
FT_BOOLEAN      Boolean
FT_CHAR Character, 1 byte
FT_UINT8        Unsigned integer, 1 byte
FT_UINT16       Unsigned integer, 2 bytes
FT_UINT24       Unsigned integer, 3 bytes
FT_UINT32       Unsigned integer, 4 bytes
FT_UINT40       Unsigned integer, 5 bytes
FT_UINT48       Unsigned integer, 6 bytes
FT_UINT56       Unsigned integer, 7 bytes
FT_UINT64       Unsigned integer, 8 bytes
FT_INT8 Signed integer, 1 byte
FT_INT16        Signed integer, 2 bytes
FT_INT24        Signed integer, 3 bytes
FT_INT32        Signed integer, 4 bytes
FT_INT40        Signed integer, 5 bytes
FT_INT48        Signed integer, 6 bytes
FT_INT56        Signed integer, 7 bytes
FT_INT64        Signed integer, 8 bytes
FT_IEEE_11073_SFLOAT    IEEE-11073 Floating point (16-bit)
FT_IEEE_11073_FLOAT     IEEE-11073 Floating point (32-bit)
FT_FLOAT        Floating point (single-precision)
FT_DOUBLE       Floating point (double-precision)
FT_ABSOLUTE_TIME        Date and time
FT_RELATIVE_TIME        Time offset
FT_STRING       Character string
FT_STRINGZ      Character string
FT_UINT_STRING  Character string
FT_ETHER        Ethernet or other MAC address
FT_BYTES        Sequence of bytes
FT_UINT_BYTES   Sequence of bytes
FT_IPv4 IPv4 address
FT_IPv6 IPv6 address
FT_IPXNET       IPX network number
FT_FRAMENUM     Frame number
FT_PCRE Compiled Perl-Compatible Regular Expression (GRegex) obj
FT_GUID Globally Unique Identifier
```

Heuristic Decodes

Sorting the Dissectors based on the heuristic decodes is one of the things that need to be easily and readily available. For the same reason, we have the option of heuristic decodes in TShark. This option prints all the heuristic decodes which are currently installed. It consists of 3 fields. First, one representing the underlying dissector, the second one representing the name of the heuristic decoded and the last one tells about the status of the heuristic. It will be T in case it is heuristics and F otherwise.

```
tshark -G heuristic-decodes
```

```
root@kali:~# tshark -G heuristic-decodes
Running as user "root" and group "root".
rtsp    rtp     F
sctp    sip     T
sctp    nbap    T
sctp    jxta    T
udp     xml     F
udp     wol     T
udp     wg      T
udp     waveagent       T
udp     wassp   F
udp     udt     T
udp     teredo  F
udp     stun    T
udp     srt     T
udp     sprt    T
udp     skype   F
udp     sip     T
udp     rtps    T
udp     rtp     F
udp     rtcp    T
udp     rpcap   T
udp     rpc     T
udp     rlm     T
udp     rlc-nr  F
udp     rlc-lte F
udp     rlc     F
udp     rftap   T
udp     reload-framing  T
udp     reload  T
udp     redbackli       T
udp     raknet  T
udp     quic    T
udp     proxy   T
udp     pktgen  T
udp     peekremote      T
udp     pdcp-nr F
```

Plugins

Plugins are a very important kind of option that was integrated with Tshark Reporting options. As the name states it prints the name of all the plugins that are installed. The field that this report consists of is made of the Plugin Library, Plugin Version, Plugin Type and the path where the plugin is located.

```
tshark -G plugins
```

```
root@kali:~# tshark -G plugins
Running as user "root" and group "root". This could be dangerous.
ethercat.so              0.1.0   dissector       /usr/lib/x86_64-linux-gnu/wireshark/plugins/3
gryphon.so               0.0.4   dissector       /usr/lib/x86_64-linux-gnu/wireshark/plugins/3
irda.so                  0.0.6   dissector       /usr/lib/x86_64-linux-gnu/wireshark/plugins/3
mate.so                  1.0.1   dissector       /usr/lib/x86_64-linux-gnu/wireshark/plugins/3
opcua.so                 1.0.0   dissector       /usr/lib/x86_64-linux-gnu/wireshark/plugins/3
profinet.so              0.2.4   dissector       /usr/lib/x86_64-linux-gnu/wireshark/plugins/3
stats_tree.so            0.0.1   dissector       /usr/lib/x86_64-linux-gnu/wireshark/plugins/3
transum.so               2.0.4   dissector       /usr/lib/x86_64-linux-gnu/wireshark/plugins/3
unistim.so               0.0.2   dissector       /usr/lib/x86_64-linux-gnu/wireshark/plugins/3
usbdump.so               0.0.1   file type       /usr/lib/x86_64-linux-gnu/wireshark/plugins/3
wimax.so                 1.2.0   dissector       /usr/lib/x86_64-linux-gnu/wireshark/plugins/3
wimaxasncp.so            0.0.1   dissector       /usr/lib/x86_64-linux-gnu/wireshark/plugins/3
wimaxmacphy.so           0.0.1   dissector       /usr/lib/x86_64-linux-gnu/wireshark/plugins/3
```

Protocols

If the users want to know the details about the protocols that are recorded in the registration database then, they can use the protocols parameter. This output is also a bit less readable so that the user can take the help of any third party tool to beautify the report. This parameter prints the data in 3 fields. We have the protocol name, short name, and the filter name.

```
tshark -G protocols | head
```

```
root@kali:~# tshark -G protocols | head
Running as user "root" and group "root". This could be dangerous.
Lua Dissection  Lua Dissection  _ws.lua
Expert Info     Expert  _ws.expert
IEC 60870-5-104-Apci    104apci 104apci
IEC 60870-5-104-Asdu    104asdu 104asdu
29West Protocol 29West  29west
Pro-MPEG Code of Practice #3 release 2 FEC Protocol     2dparityfec     2dparityfec
3Com XNS Encapsulation  3COMXNS 3comxns
3GPP2 A11       3GPP2 A11       a11
IPv6 over Low power Wireless Personal Area Networks     6LoWPAN 6lowpan
802.11 radio information        802.11 Radio    wlan_radio
```

Values

Let’s talk about the values report. It consists of value strings, range strings, true/false strings. There are three types of records available here. The first field can consist of one of these three characters representing the following:

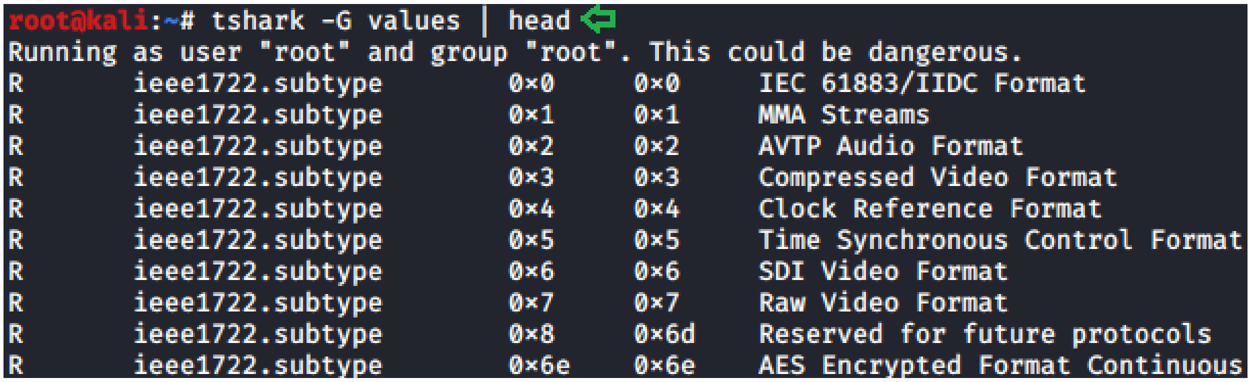
V: Value Strings

R: Range Strings

T: True/False Strings

Moreover, in the value strings, we have the field abbreviation, integer value, and the string. In the range strings, we have the same values except it holds the lower bound and upper bound values.

```
tshark -G values | head
```

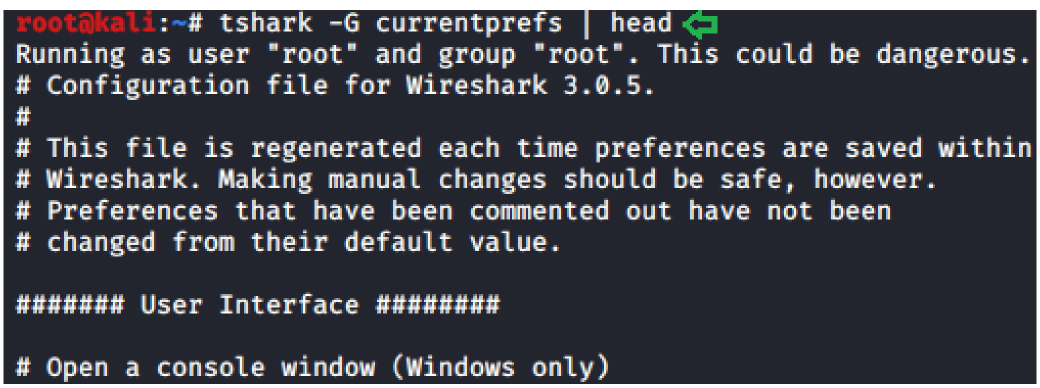


```
root@kali:~# tshark -G values | head
Running as user "root" and group "root". This could be dangerous.
R       ieee1722.subtype        0x0     0x0     IEC 61883/IIDC Format
R       ieee1722.subtype        0x1     0x1     MMA Streams
R       ieee1722.subtype        0x2     0x2     AVTP Audio Format
R       ieee1722.subtype        0x3     0x3     Compressed Video Format
R       ieee1722.subtype        0x4     0x4     Clock Reference Format
R       ieee1722.subtype        0x5     0x5     Time Synchronous Control Format
R       ieee1722.subtype        0x6     0x6     SDI Video Format
R       ieee1722.subtype        0x7     0x7     Raw Video Format
R       ieee1722.subtype        0x8     0x6d    Reserved for future protocols
R       ieee1722.subtype        0x6e    0x6e    AES Encrypted Format Continuous
```

Preferences

In case the user requires to revise the current preferences that are configured on the system, they can use the currentprefs options to read the preference saved in the file.

```
tshark -G currentprefs | head
```



```
root@kali:~# tshark -G currentprefs | head
Running as user "root" and group "root". This could be dangerous.
# Configuration file for Wireshark 3.0.5.
#
# This file is regenerated each time preferences are saved within
# Wireshark. Making manual changes should be safe, however.
# Preferences that have been commented out have not been
# changed from their default value.

####### User Interface ########

# Open a console window (Windows only)
```

Folders

Suppose the user wants to manually change the configurations or get the program information or want to take a look at the lua configuration or some other important files. The users need the path of those files to take a peek at them. Here the folders option comes a little handy.

```
tshark -G folders
```

```
root@kali:~# tshark -G folders
Running as user "root" and group "root". This could be dangerous.
Temp:                   /tmp
Personal configuration: /root/.config/wireshark
Global configuration:   /usr/share/wireshark
System:                 /etc
Program:                /usr/bin
Personal Plugins:       /root/.local/lib/wireshark/plugins/3.0
Global Plugins:         /usr/lib/x86_64-linux-gnu/wireshark/plugins/3.0
Personal Lua Plugins:   /root/.local/lib/wireshark/plugins
Global Lua Plugins:     /usr/lib/x86_64-linux-gnu/wireshark/plugins
Extcap path:            /usr/lib/x86_64-linux-gnu/wireshark/extcap
MaxMind database path:  /usr/share/GeoIP
MaxMind database path:  /var/lib/GeoIP
MaxMind database path:  /usr/share/GeoIP
MaxMind database path:  /var/lib/GeoIP
```

Since we talked so extensively about TShark, It won't be justice if we won't talk about the tool that is heavily dependent on the data from TShark. Let's talk about PyShark.

PyShark

It is essentially a wrapper that is based on Python. Its functionality is that allows the python packet parsing using the TShark dissectors. Many tools do the same job more or less but the difference is that this tool can export XMLs to use its parsing. You can read more about it from its **GitHub** page.

Installation

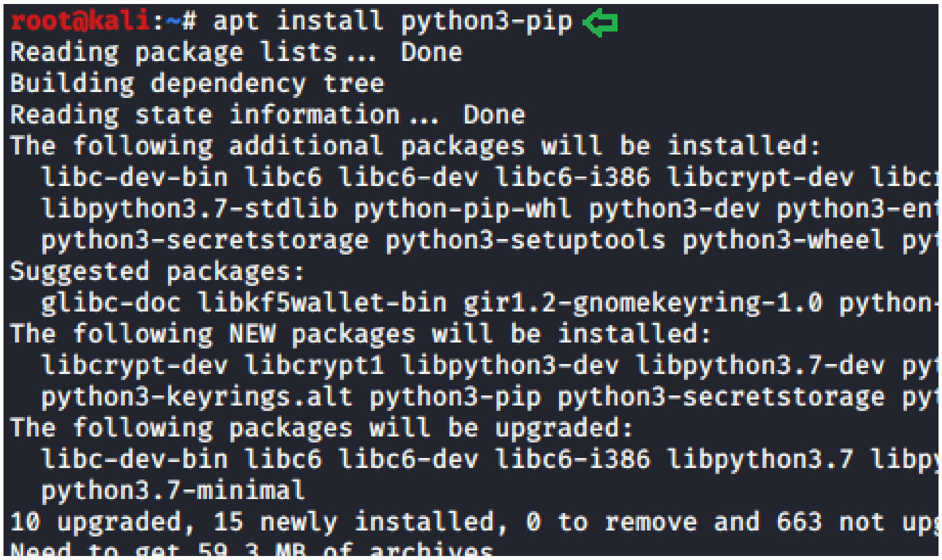
As the PyShark was developed using Python 3 and we don't Python 3 installed on our machine. We installed Python3 as shown in the image given below.

```
apt install python3
```

```
root@kali:~# apt install python3
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  libpython3-stdlib python3-minimal
Suggested packages:
  python3-doc python3-venv
The following packages will be upgraded:
  libpython3-stdlib python3 python3-minimal
3 upgraded, 0 newly installed, 0 to remove and 673 not upgrade
Need to get 119 kB of archives.
After this operation, 1,024 B of additional disk space will be
Do you want to continue? [Y/n] y
Get:1 http://ftp.harukasan.org/kali kali-rolling/main amd64 py
Get:2 http://ftp.harukasan.org/kali kali-rolling/main amd64 py
Get:3 http://ftp.harukasan.org/kali kali-rolling/main amd64 li
Fetched 119 kB in 10s (11.7 kB/s)
```

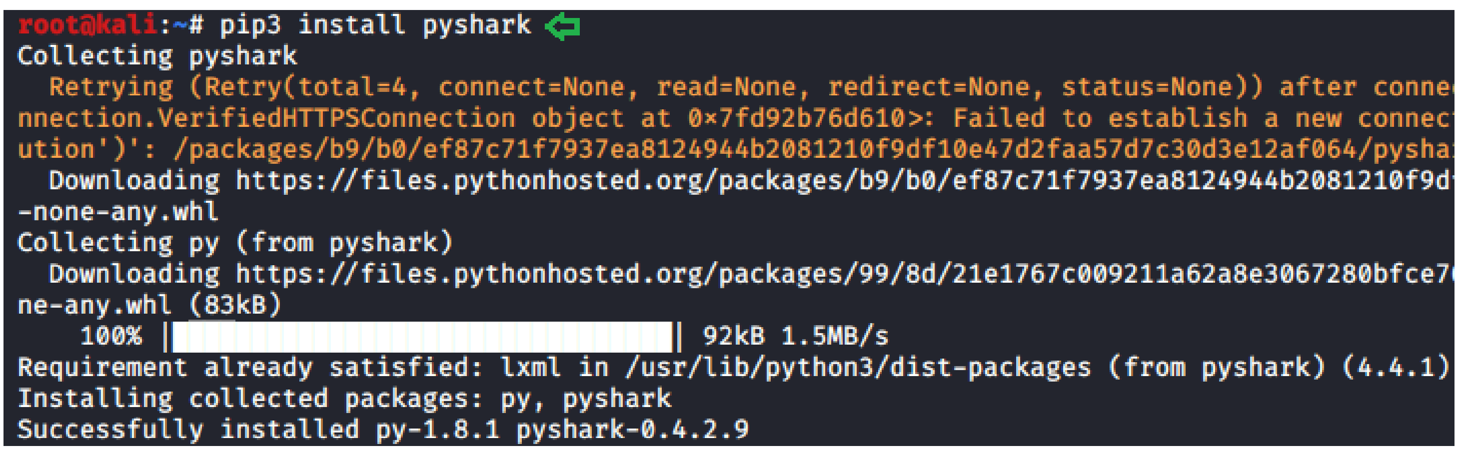
PyShark is available through the pip. But we don't have the pip for python 3 so we need to install it as well.

```
apt install python3-pip
```



Since we have the python3 with pip we will install pyshark using pip command. You can also install PyShark by cloning the git and running the setup.

```
pip3 install pyshark
```



Live Capture

Now to get started, we need the python interpreter. To get this we write python3 and press enter. Now that we have the interpreter, the very first thing that we plan on doing is importing PyShark. Then we define network interface for the capture. Followed by that we will define the value of the timeout parameter for the capture.sniff function. At last, we will begin the capture. Here we can see that in the timeframe that we provided PyShark captured 9 packets.

```
python3
import pyshark
capture = pyshark.LiveCapture(interface='eth0')
capture.sniff(timeout=5)
capture
```

```
root@kali:~# python3
Python 3.7.6 (default, Jan 19 2020, 22:34:52)
[GCC 9.2.1 20200117] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pyshark
>>> capture = pyshark.LiveCapture(interface='eth0')
>>> capture.sniff(timeout=5)
>>> capture
<LiveCapture (9 packets)>
```

Pretty Representation

There are multiple ways in which PyShark can represent data inside the captured packet. In the previous practical, we captured 9 packets. Let's take a look at the first packet that was captured with PyShark. Here we can see that we have a layer-wise analysis with the ETH Layer, IP Layer, and the TCP Layer.

```
capture[1].pretty_print()
```

```
>>> capture[1].pretty_print()
Layer ETH:
        Destination: 1c:5f:2b:59:e1:24
        Address: 1c:5f:2b:59:e1:24
        .... ..0. .... .... .... .... = LG bit: Globally unique address (facto
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
        Source: 00:0c:29:d5:b7:2d
        Type: IPv4 (0x0800)
        Address: 00:0c:29:d5:b7:2d
        .... ..0. .... .... .... .... = LG bit: Globally unique address (facto
        .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
Layer IP:
        0100 .... = Version: 4
        .... 0101 = Header Length: 20 bytes (5)
        Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
        0000 00.. = Differentiated Services Codepoint: Default (0)
        .... ..00 = Explicit Congestion Notification: Not ECN-Capable Transpor
        Total Length: 52
        Identification: 0x4b7c (19324)
        Flags: 0x4000, Don't fragment
        0... .... .... .... = Reserved bit: Not set
        .1.. .... .... .... = Don't fragment: Set
        ..0. .... .... .... = More fragments: Not set
        ...0 0000 0000 0000 = Fragment offset: 0
        Time to live: 64
        Protocol: TCP (6)
        Header checksum: 0x62cb [validation disabled]
        Header checksum status: Unverified
        Source: 192.168.0.137
        Destination: 13.35.190.40
Layer TCP:
        Source Port: 38820
        Destination Port: 443
        Stream index: 1
        TCP Segment Len: 0
        Sequence number: 1    (relative sequence number)
        Next sequence number: 1    (relative sequence number)
        Acknowledgment number: 1    (relative ack number)
        1000 .... = Header Length: 32 bytes (8)
        Flags: 0x010 (ACK)
        000. .... .... = Reserved: Not set
        ...0 .... .... = Nonce: Not set
        .... 0... .... = Congestion Window Reduced (CWR): Not set
        .... .0.. .... = ECN-Echo: Not set
        .... ..0. .... = Urgent: Not set
        .... ...1 .... = Acknowledgment: Set
        .... .... 0... = Push: Not set
```

Captured Length Field

In our capture, we saw some data that can consist of multiple attributes. These attributes need fields to get stored. To explore this field, we will be using the dir function in Python. We took the packet and then defined the variable named pkt with the value of that packet and saved it. Then using the dir function we saw explored the fields inside that particular capture. Here we can see that we have the pretty_print function which we used in the previous practical. We also have one field called captured_length to read into that we will write the name of the variable followed by the name of the field with a period (.) in between as depicted in the image below.

```
capture[2]
pkt = capture[2]
pkt
dir(pkt)
pkt.captured_length
```

```
>>> capture[2]
<TCP Packet>
>>> pkt = capture[2]
>>> pkt
<TCP Packet>
>>> dir(pkt)
['__bool__', '__class__', '__contains__', '__delattr__', '__dict__', '__dir__', '__doc__', '__e
'__getattribute__', '__getitem__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_su
module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__se
sshook__', '__weakref__', '_packet_string', 'captured_length', 'eth', 'frame_info', 'get_multip
, 'interface_captured', 'ip', 'layers', 'length', 'number', 'pretty_print', 'show', 'sniff_time
']
>>> pkt.captured_length
'66'
```

Layers, Src and Dst Fields

As we listed the fields in the previous step we saw that we have another field named layers. We read its contents as we did earlier to find out that we have 3 layers in this capture. Now to look into the individual layer, we need to get the fields of that individual layer. For that, we will again use the dir function. We used the dir function on the ETH layer as shown in the image given below. We observe that we have a field named src which means source, dst which means destination. We checked the value on those fields to find the physical address of the source and destination respectively.

```
pkt.layers
pkt.eth.src
pkt.eth.dst
pkt.eth.type
```

```
>>> pkt.layers
[<ETH Layer>, <IP Layer>, <TCP Layer>]
>>> dir(pkt.eth)
['DATA_LAYER', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__
e__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__
e__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__', '__sizeof__',
 '_field_prefix', '_get_all_field_lines', '_get_all_fields_with_alternates', '_g
_sanitize_field_name', 'addr', 'addr_resolved', 'dst', 'dst_resolved', 'field_na
d_value', 'ig', 'layer_name', 'lg', 'pretty_print', 'raw_mode', 'src', 'src_reso
>>> pkt.eth.src
'1c:5f:2b:59:e1:24'
>>> pkt.eth.dst
'00:0c:29:d5:b7:2d'
>>> pkt.eth.type
'0x00000800'
```

For our next step, we need the fields of the IP packet. We used the dir function on the IP layer and then we use src and dst fields here on this layer. We see that we have the IP Address as this is the IP layer. As the Ethernet layer works on the MAC Addresses they store the MAC Addresses of the Source and the Destination which changes when we come to the IP Layer.

```
dir(pkt.ip)
pkt.ip.src
pkt.ip.dst
pkt.ip.pretty_print()
```

```
>>> dir(pkt.ip)
['DATA_LAYER', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq_
e__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le_
e__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__', '__sizeof__', '_
 '_field_prefix', '_get_all_field_lines', '_get_all_fields_with_alternates', '_get_
_sanitize_field_name', 'addr', 'checksum', 'checksum_status', 'dsfield', 'dsfield_d
lags', 'flags_df', 'flags_mf', 'flags_rb', 'frag_offset', 'get', 'get_field', 'get_
 'id', 'layer_name', 'len', 'pretty_print', 'proto', 'raw_mode', 'src', 'src_host',
>>> pkt.ip.src
'13.35.190.40'
>>> pkt.ip.dst
'192.168.0.137'
>>> pkt.ip.pretty_print()
Layer IP:
        0100 .... = Version: 4
        .... 0101 = Header Length: 20 bytes (5)
        Differentiated Services Field: 0x10 (DSCP: Unknown, ECN: Not-ECT)
        0001 00.. = Differentiated Services Codepoint: Unknown (4)
        .... ..00 = Explicit Congestion Notification: Not ECN-Capable Transport (0)
        Total Length: 52
        Identification: 0x2e26 (11814)
        Flags: 0x4000, Don't fragment
        0... .... .... .... = Reserved bit: Not set
        .1.. .... .... .... = Don't fragment: Set
        ..0. .... .... .... = More fragments: Not set
        ...0 0000 0000 0000 = Fragment offset: 0
        Time to live: 248
        Protocol: TCP (6)
        Header checksum: 0xc810 [validation disabled]
        Header checksum status: Unverified
        Source: 13.35.190.40
        Destination: 192.168.0.137
```

Similarly, we can use the dir function and the field's value on any layer of the capture. This makes the investigation of the capture quite easier.

Promisc Capture

In previous articles we learned about the promisc mode that means that a network interface card will pass all frames received up to the operating system for processing, versus the traditional mode of operation wherein only frames destined for the NIC's MAC address or a broadcast address will be passed up to the OS. Generally, promiscuous mode is used to "sniff" all traffic on the wire. But we got stuck when we configured the network interface card to work on promisc mode. So while capturing traffic on TShark we can switch between the normal capture and the promisc capture using the –p parameter as shown in the image given below.

```
ifconfig eth0 promisc
ifconfig eth0
tshark -i eth0 -c 10
tshark -i eth0 -c  10 -p
```

```
root@kali:~# ifconfig eth0 promisc
root@kali:~# ifconfig eth0
eth0: flags=4419<UP,BROADCAST,RUNNING,PROMISC,MULTICAST>  mtu 1500
        inet 192.168.0.137  netmask 255.255.255.0  broadcast 192.168.0.255
        inet6 fe80::20c:29ff:fed5:b72d  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:d5:b7:2d  txqueuelen 1000  (Ethernet)
        RX packets 67816  bytes 85545596 (81.5 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 30726  bytes 2463013 (2.3 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@kali:~# tshark -i eth0 -c 10
Running as user "root" and group "root". This could be dangerous.
Capturing on 'eth0'
    1 0.000000000 192.168.0.137 → 35.169.2.62  TLSv1.2 164 Application Data
    2 0.000142943 192.168.0.137 → 107.23.176.98 TLSv1.2 164 Application Data
    3 0.236904732  35.169.2.62 → 192.168.0.137 TLSv1.2 187 Application Data
    4 0.236921665 192.168.0.137 → 35.169.2.62  TCP 66 40520 → 443 [ACK] Seq=99 Ack=122 W
    5 0.242952531 107.23.176.98 → 192.168.0.137 TLSv1.2 187 Application Data
    6 0.242967301 192.168.0.137 → 107.23.176.98 TCP 66 41152 → 443 [ACK] Seq=99 Ack=122
    7 1.343354460  192.168.0.6 → 224.0.0.251  IGMPv2 60 Membership Report group 224.0.0.
    8 2.842606464  192.168.0.6 → 224.0.0.252  IGMPv2 60 Membership Report group 224.0.0.
    9 6.807673972 192.168.0.137 → 34.213.241.62 TCP 66 51094 → 443 [ACK] Seq=1 Ack=1 Win
   10 7.100843807 34.213.241.62 → 192.168.0.137 TCP 66 [TCP ACKed unseen segment] 443 →
10 packets captured
root@kali:~# tshark -i eth0 -c 10 -p
Running as user "root" and group "root". This could be dangerous.
Capturing on 'eth0'
    1 0.000000000 34.213.241.62 → 192.168.0.137 TLSv1.2 97 Encrypted Alert
    2 0.000019158 192.168.0.137 → 34.213.241.62 TCP 66 51094 → 443 [ACK] Seq=1 Ack=32 Wi
    3 0.000222027 192.168.0.137 → 34.213.241.62 TLSv1.2 97 Encrypted Alert
    4 0.000288786 192.168.0.137 → 34.213.241.62 TCP 66 51094 → 443 [FIN, ACK] Seq=32 Ack
    5 0.289883135 34.213.241.62 → 192.168.0.137 TCP 66 [TCP Previous segment not capture
    6 0.289903932 34.213.241.62 → 192.168.0.137 TCP 66 [TCP Out-Of-Order] 443 → 51094 [F
    7 0.289914338 192.168.0.137 → 34.213.241.62 TCP 66 51094 → 443 [ACK] Seq=33 Ack=33 W
    8 4.120921966 192.168.0.137 → 35.169.2.62  TLSv1.2 165 Application Data
    9 4.121065015 192.168.0.137 → 107.23.176.98 TLSv1.2 164 Application Data
   10 4.394954971  35.169.2.62 → 192.168.0.137 TLSv1.2 188 Application Data
10 packets captured
```