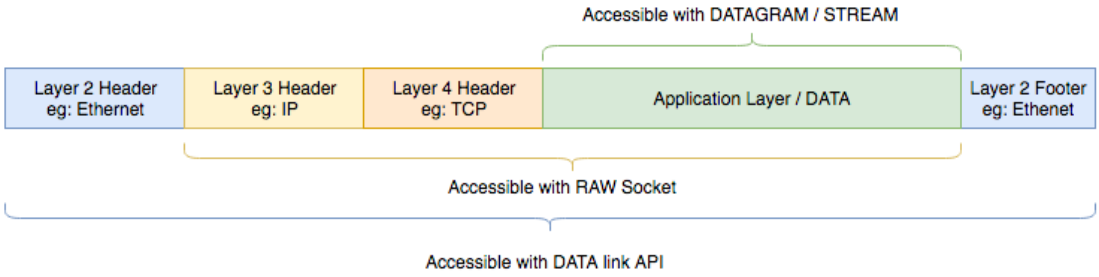
Packet sniffer in Python

A simple packet sniffer in Python can be created with the help socket module. We can use the raw socket type to get the packets. A raw socket provides access to the underlying protocols, which support socket abstractions. Since raw sockets are part of the internet socket API, they can only be used to generate and receive IP packets.



Getting ready

As some behaviors of the socket module depend on the operating system socket API and there is no uniform API for using a raw socket under a different operating system, we need to use a Linux OS to run this script. So, if you are using Windows or macOS, please make sure to run this script inside a virtual Linux environment. Also, most operating systems require root access to use raw socket APIs.

How to do it...

Here are the steps to create a basic packet sniffer with `socket` module:

1. Create a new file called `basic-packet-sniffer-linux.py` and open it in your editor.
2. Import the required modules:

```
import socket
```

3. Now we can create an `INET` raw socket:

```
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_TCP)
```

Both reading and writing to a raw socket require creating a raw socket first. Here we use the `INET` family raw socket. The family parameter for a socket describes the address family of the socket. The following are the address family constants:

- `AF_LOCAL`: Used for local communication
- `AF_UNIX`: Unix domain sockets
- `AF_INET`: IP version 4

- `AF_INET6`: IP version 6
- `AF_IPX`: Novell IPX
- `AF_NETLINK`: Kernel user-interface device
- `AF_X25`: Reserved for X.25 project
- `AF_AX25`: Amateur Radio AX.25
- `AF_APPLETALK`: Appletalk DDP
- `AF_PACKET`: Low-level packet interface
- `AF_ALG`: Interface to kernel crypto API

The next parameter passed is the type of the socket. The following are the possible values for the socket type:

- `SOCK_STREAM`: Stream (connection) socket
- `SOCK_DGRAM`: Datagram (connection-less) socket
- `SOCK_RAW`: RAW socket
- `SOCK_RDM`: Reliably delivered message
- `SOCK_SEQPACKET`: Sequential packet socket
- `SOCK_PACKET`: Linux-specific method of getting packets at the development level

The last parameter is the protocol of the packet. This protocol number is defined by the **Internet Assigned Numbers Authority** (**IANA**). We have to be aware of the family of the socket; then we can only choose a protocol. As we selected `AF_INET` (IPV4), we can only select IP-based protocols.

4. Next, start an infinite loop to receive data from the socket:

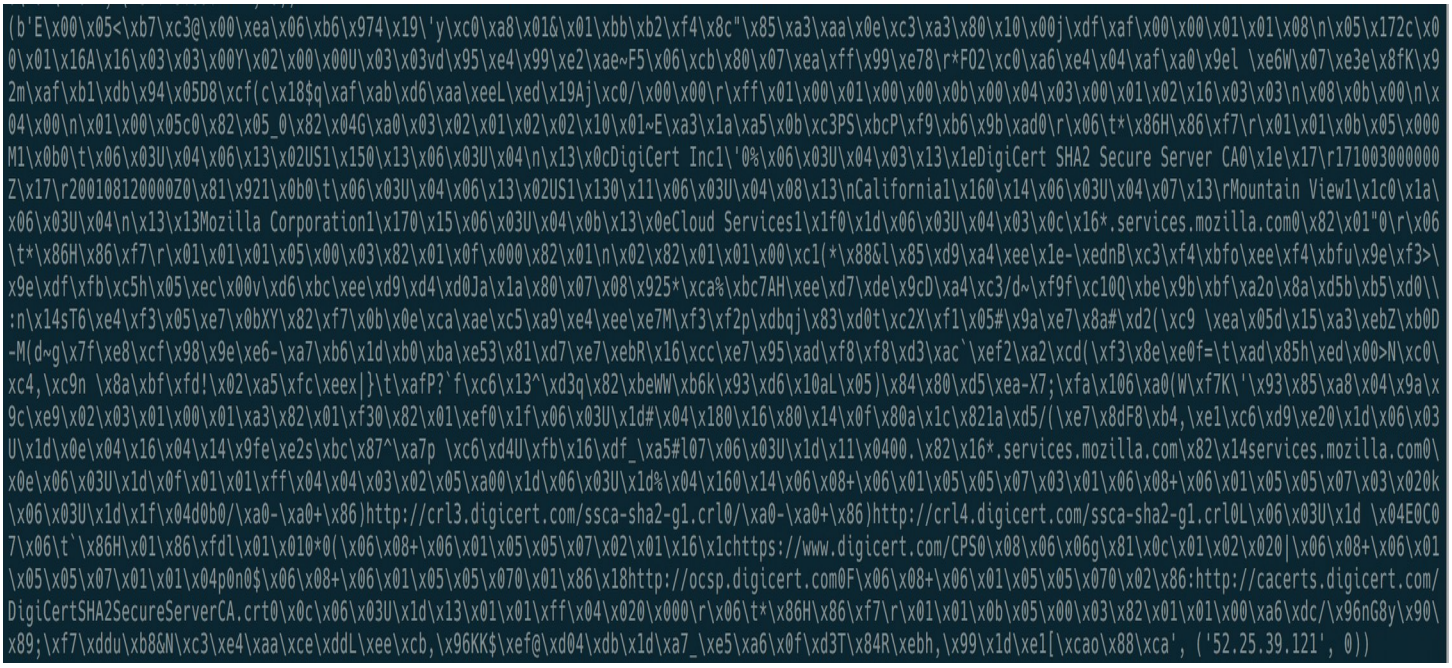
```
while True:
  print(s.recvfrom(65565))
```

The `recvfrom` method in the socket module helps us to receive all the data from the socket. The parameter passed is the buffer size; `65565` is the maximum buffer size.

5. Now run the program with Python:

```
sudo python3 basic-packet-sniffer-linux.py
```

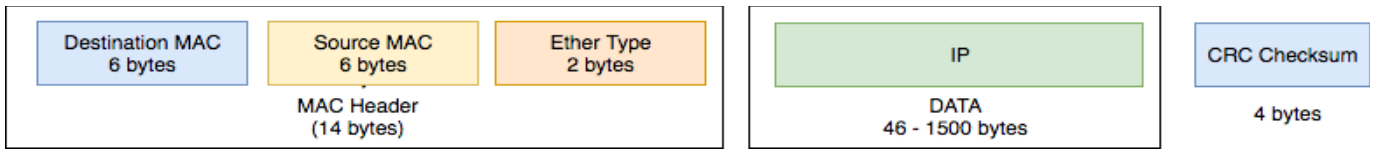
The result will be as follows:



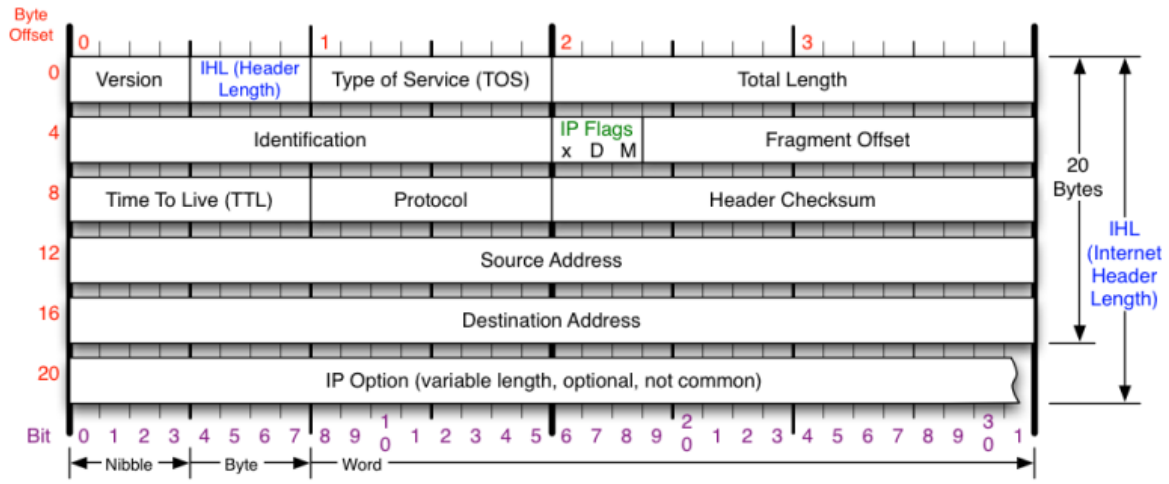
Parsing the packet

Now we can try to parse the data that we sniffed, and unpack the headers. To parse a packet, we need to have an idea of the Ethernet frame and the packet headers of the IP.

The Ethernet frame structure is as follows:



The first six bytes are for the **Destination MAC** address and the next six bytes are for the **Source MAC**. The last two bytes are for the **Ether Type**. The rest includes **DATA** and **CRC Checksum**. According to RFC 791, an IP header looks like the following:



The IP header includes the following sections:

- **Protocol Version (four bits)**: The first four bits. This represents the current IP protocol.
- **Header Length (four bits)**: The length of the IP header is represented in 32-bit words. Since this field is four bits, the maximum header length allowed is 60 bytes. Usually the value is `5`, which means five 32-bit words: *5 * 4 = 20 bytes*.
- **Type of Service (eight bits)**: The first three bits are precedence bits, the next four bits represent the type of service, and the last bit is left unused.
- **Total Length (16 bits)**: This represents the total IP datagram length in bytes. This a 16-bit field. The maximum size of the IP datagram is 65,535 bytes.
- **Flags (three bits)**: The second bit represents the **Don't Fragment** bit. When this bit is set, the IP datagram is never fragmented. The third bit represents the **More Fragment** bit. If this bit is set, then it represents a fragmented IP datagram that has more fragments after it.
- **Time To Live (eight bits)**: This value represents the number of hops that the IP datagram will go through before being discarded.
- **Protocol (eight bits)**: This represents the transport layer protocol that handed over data to the IP layer.
- **Header Checksum (16 bits)**: This field helps to check the integrity of an IP datagram.
- **Source and destination IP (32 bits each)**: These fields store the source and destination address, respectively.

Refer to the RFC 791 document for more details on IP headers: https://tools.ietf.org/html/rfc791

How to do it...

Following are the steps to parse a packet:

1. Create a new file called `basic-parse-packet-packet-linux.py` and import the modules required to parse the packets:

```
from struct import *
import sys
```

2. Now we can create a function to parse the Ethernet header:

```
def ethernet_head(raw_data):
    dest, src, prototype = struct.unpack('! 6s 6s H', raw_data[:14])
    dest_mac = get_mac_addr(dest)
    src_mac = get_mac_addr(src)
    proto = socket.htons(prototype)
    data = raw_data[14:]
    return dest_mac, src_mac, proto, data
```

Here we use the `unpack` method in the `struct` module to unpack the headers. From the Ethernet frame structure, the first six bytes are for the destination MAC, the second 6 bytes are for the source MAC, and the last unsigned short is for the Ether Type. Finally, the rest is data. So, this function returns the destination MAC, source MAC, protocol, and data.

3. Now we can create a main function and, in the `ethernet_head()`, parse this function and get the details:

```
def main():
    s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(3))
    while True:
        raw_data, addr = s.recvfrom(65535)
        eth = ethernet(raw_data)
        print('\nEthernet Frame:')
        print('Destination: {}, Source: {}, Protocol: {}'.format(eth[0], eth[1],
eth[2]))

main()
```

4. Now we can check the data section in the Ethernet frame and parse the IP headers. We can create another function to parse the `ipv4` headers:

```
def ipv4_head(raw_data):
    version_header_length = raw_data[0]
    version = version_header_length >> 4
    header_length = (version_header_length & 15) * 4
    ttl, proto, src, target = struct.unpack('! 8x B B 2x 4s 4s', raw_data[:20])
    data = raw_data[header_length:]
    return version, header_length, ttl, proto, src, target, data
```

As per the IP headers, we will unpack the headers using the `unpack` method in `struct`, and return the `version`, `header_lentgth`, `ttl`, protocol source, and destination IPs.

5. Now update `main()` to print the IP headers:

```
def main():
    s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(3))
    while True:
```

```
        raw_data, addr = s.recvfrom(65535)
        eth = ethernet(raw_data)
        print('\nEthernet Frame:')
        print('Destination: {}, Source: {}, Protocol: {}'.format(eth[0], eth[1],
eth[2]))
        if eth[2] == 8:
            ipv4 = ipv4(ethp[4])
            print( '\t - ' + 'IPv4 Packet:')
            print('\t\t - ' + 'Version: {}, Header Length: {}, TTL:
{},'.format(ipv4[1], ipv4[2], ipv4[3]))
            print('\t\t - ' + 'Protocol: {}, Source: {}, Target:
{}'.format(ipv4[4], ipv4[5], ipv4[6]))
```

6. Currently, the IP addresses printed are not in a readable format, so we can write a function to format them:

```
def get_ip(addr):
    return '.'.join(map(str, addr))
```

Make sure to update the `ipv4_head` function to format the IP address by adding the following lines before returning the output:

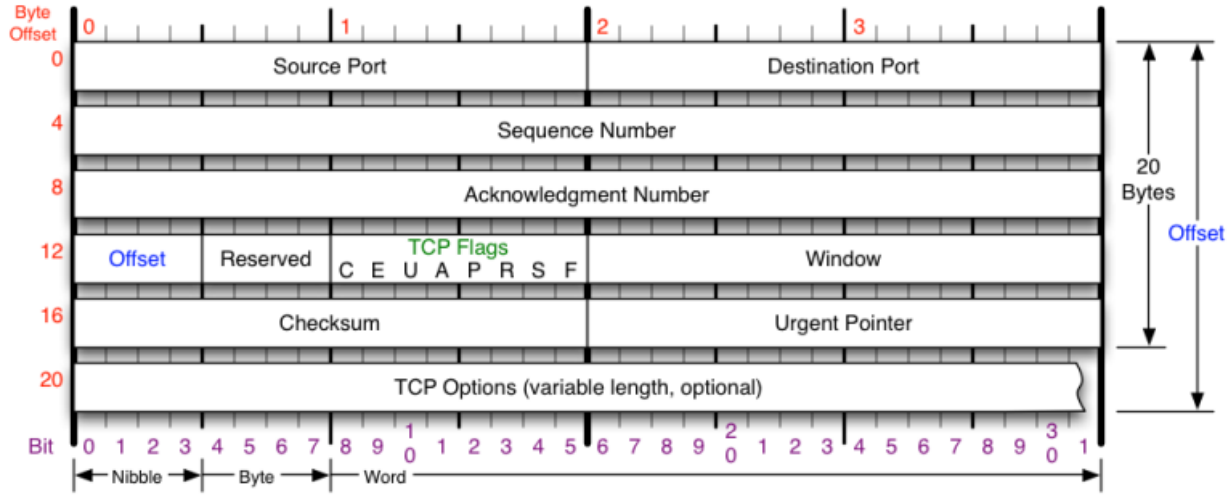
```
src = get_ip(src)
target = get_ip(target)
```

7. Now that we have the internet layer unpacked, the next layer we have to unpack is the transport layer. We can determine the protocol from the protocol ID in the IP header. The following are the protocol IDs for some of the protocols:
    - **TCP**: 6
    - **ICMP**: 1
    - **UDP**: 17
    - **RDP**: 27

8. Next, we can create a function to unpack the TCP packets:

```
def tcp_head( raw_data):
    (src_port, dest_port, sequence, acknowledgment, offset_reserved_flags) =
struct.unpack(
        '! H H L L H', raw_data[:14])
    offset = (offset_reserved_flags >> 12) * 4
    flag_urg = (offset_reserved_flags & 32) >> 5
    flag_ack = (offset_reserved_flags & 16) >> 4
    flag_psh = (offset_reserved_flags & 8) >> 3
    flag_rst = (offset_reserved_flags & 4) >> 2
    flag_syn = (offset_reserved_flags & 2) >> 1
    flag_fin = offset_reserved_flags & 1
    data = raw_data[offset:]
    return src_port, dest_port, sequence, acknowledgment, flag_urg, flag_ack,
flag_psh, flag_rst, flag_syn, flag_fin, data
```

The TCP packets are unpacked according to the TCP packet header's structure:

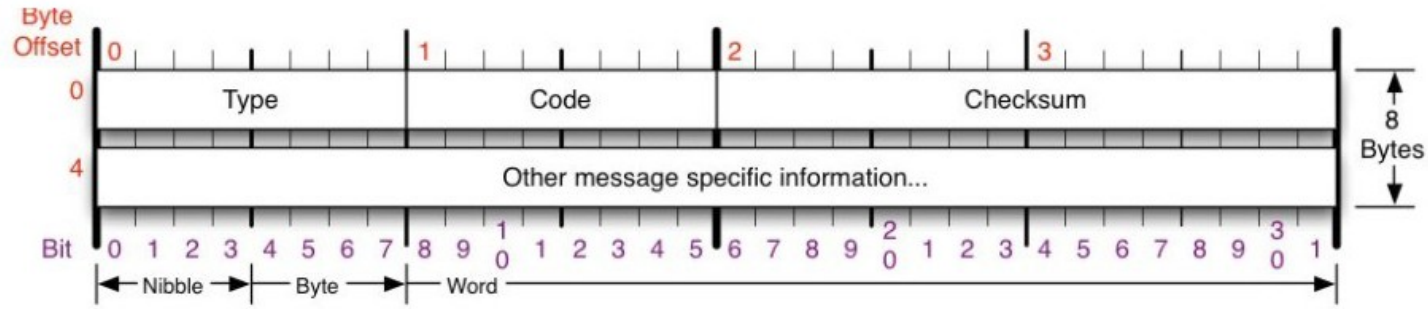


9. Now we can update `main()` to print the TCP header details. Add the following lines inside the `ipv4` section:

```
if ipv4[4] == 6:
    tcp = tcp_head(ipv4[7])
    print(TAB_1 + 'TCP Segment:')
    print(TAB_2 + 'Source Port: {}, Destination Port: {}'.format(tcp[0], tcp[1]))
    print(TAB_2 + 'Sequence: {}, Acknowledgment: {}'.format(tcp[2], tcp[3]))
    print(TAB_2 + 'Flags:')
    print(TAB_3 + 'URG: {}, ACK: {}, PSH:{}'.format(tcp[4], tcp[5], tcp[6]))
    print(TAB_3 + 'RST: {}, SYN: {}, FIN:{}'.format(tcp[7], tcp[8], tcp[9]))
    if len(tcp[10]) > 0:
         # HTTP
        if tcp[0] == 80 or tcp[1] == 80:
             print(TAB_2 + 'HTTP Data:')
                 try:
                    http = HTTP(tcp[10])
                    http_info = str(http[10]).split('\n')
                    for line in http_info:
                       print(DATA_TAB_3 + str(line))
                 except:
                       print(format_multi_line(DATA_TAB_3, tcp[10]))
                 else:
                      print(TAB_2 + 'TCP Data:')
                      print(format_multi_line(DATA_TAB_3, tcp[10]))
```

10. Similarly, update the functions to unpack the UDP and ICMP packets.

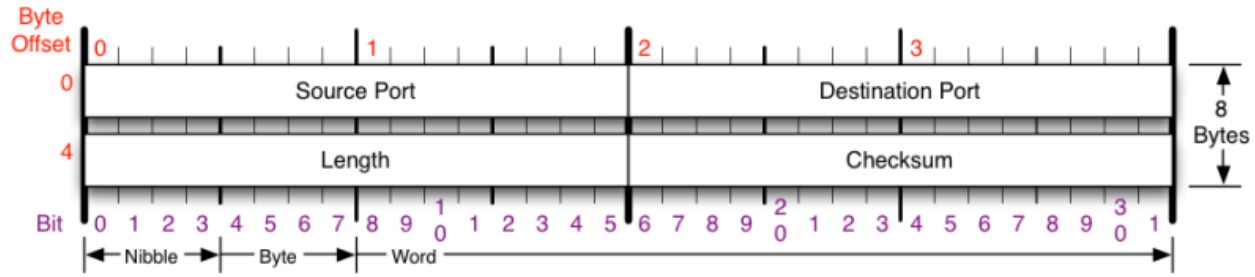
The packets are unpacked according to the packet header structure. Here is the packet header structure for ICMP:



According to the diagram, we can unpack the packet using the following code:

```
elif ipv4[4] == 1:
    icmp = icmp_head(ipv4[7])
    print('\t -' + 'ICMP Packet:')
    print('\t\t -' + 'Type: {}, Code: {}, Checksum:{},'.format(icmp[0], icmp[1],
icmp[2]))
    print('\t\t -' + 'ICMP Data:')
    print(format_multi_line('\t\t\t', icmp[3]))
```

Here is the packet header structure for UDP:



As we did for the ICMP, we can unpack the UDP packet headers as follows:

```
elif ipv4[4] == 17:
    udp = udp_head(ipv4[7])
    print('\t -' + 'UDP Segment:')
    print('\t\t -' + 'Source Port: {}, Destination Port: {}, Length:
{}'.format(udp[0], udp[1], udp[2]))
```

Now save and run the script with the required permission:

```
sudo python3 basic-parse-packet-linux.py
```

The output will print all the packets that were sniffed. So, it will continue printing until we stop it with a keyboard interrupt. The output will be as follows:

```
Ethernet Frame:
Destination: 01:00:5E:7F:FF:FA, Source: 8C:3A:E3:4C:54:82, Protocol: 8
        -IPv4 Packet:
                -Version: 4, Header Length: 20, TTL: 1,
                -Protocol: 17, Source: 192.168.1.34, Target: 239.255.255.250
        -UDP Segment:
                -Source Port: 53426, Destination Port: 1900, Length: 18295

Ethernet Frame:
Destination: 8C:85:90:1B:90:37, Source: 08:3E:8E:04:78:F1, Protocol: 8
        -IPv4 Packet:
                -Version: 4, Header Length: 20, TTL: 64,
                -Protocol: 6, Source: 192.168.1.37, Target: 192.168.1.35
        -TCP Segment:
                -Source Port: 22, Destination Port: 61389
                -Sequence: 235473480, Acknowledgment: 851396349
                -Flags:
                        -URG: 0, ACK: 1, PSH: 1
                        -RST: 0, SYN: 0, FIN:0
                -TCP Data:
                       \x6f\xcb\x08\x57\x74\x33\xbf\xe9\x6f\x9e\x67\x97\x09\x31\x91\x93\xdc\x49\x0a
                       \xc6\x33\x09\xb7\xf0\x11\x83\x1a\xd8\xbb\x05\xd6\x46\x0a\x4d\x26\x12\x54\x77
                       \x84\x7e\x67\xd0\xd5\x38\x80\xbf\x37\x35\x56\x1b\xaa\x86\x93\x8f\xaf\x41\x93
                       \x40\xcd\x6f\xd9\x55\x7a\x0f\xf3\xd8\xca\xe3\xf1\xa6\x9f\xe9\xde\x7e\x75\x33
                       \xeb\xe8\x5d\x5d\x37\x28\x86\x61\x30\xe8\x60\x59\x6e\x1b\xa6\x0c\x90\x70\x98
                       \xfd\x36\x6e\x20\xcb\x19\xf9\x52\x1d\x17\xbf\x57\xe3\x9d\x2e\x3c\xfe\x9e\xe0
                       \x7f\x3a\x08\x5b\x82\x65\x96\x7d\x79\xb1\x8a\x12\x44\x93\x91\x51\x3f\x6a

Ethernet Frame:
Destination: 8C:85:90:1B:90:37, Source: 08:3E:8E:04:78:F1, Protocol: 8
        -IPv4 Packet:
                -Version: 4, Header Length: 20, TTL: 64,
                -Protocol: 6, Source: 192.168.1.37, Target: 192.168.1.35
        -TCP Segment:
                -Source Port: 22, Destination Port: 61389
                -Sequence: 235473612, Acknowledgment: 851396349
                -Flags:
                        -URG: 0, ACK: 1, PSH: 1
                        -RST: 0, SYN: 0, FIN:0
                -TCP Data:
                       \xee\xf2\x41\x13\x99\x88\x45\xef\xcb\xd5\x1d\x78\x25\x6d\x35\x7f\xd5\x9b\x9f
                       \x22\xfb\xe0\xbf\xad\xa7\x86\xf8\xe0\x42\x7d\x8a\xe1\x62\x37\x74\x4a\xb6\x89
                       \xeb\x1e\x47\xa1\xfe\x24\xbc\x1e\x3d\x82\x81\x83\x9f\xb1\xfe\x75\x7f\x45\x91
                       \xe2\x3b\x9a\xb4\xe4\x4d\xff\x67\xee\x97\x3f\xdd\x99\x0d\x69\x0b\x58\x30\x59
                       \x9c\xe4\x65\x49\x71\x8c\x20\x72\x35\x6b\x76\x4e\xff\xe7\xe5\x5c\x06\x43\xe0
                       \x9c\xcc\x15\xcc\xef\xad\xd6\x8d\x79\xd3\x11\xcb\xb9\x1f\x34\x7c\xe7\xe2\x5f
                       \xa7\xd3\x5f\x74\x9b\x55\x37\xf2\xd4\x2e\x5a\xe7\x3f\x20\x8a\x31\xaf\x26\xa2
                       \x30\x88\xbe\x9b\x2d\xb1\x6a\x2c\xe9\xa7\x45\x62\xd9\x77\xfc\x29\xff\x60\xde
                       \xf5\x17\x37\x65\x74\x4b\x65\x37\x83\x17\xa7\x31\x1a\x38\x6b\x3c\xa3\x65\x24
                       \xe5\x75\x74\x71\x41\xf7\xc1\xcf\x44\xe7\x53\xbe\x97\x10\x41\xe5\xf7\x19\xf9
                       \xd7\x97\xe0\x45\x27\x4c\x57\x92\x9e\xb0\x2f\xca\xca\xba\xa4\x46\x03\x70\xa5
                       \x7e\xc7\x5f\xa7\x58\xbc\x4d\x57\xcb\x7d\xc5\x16\xf1\x23\x62\x49\xdb\x68\x17
```