Basic Network Sniffer

Building a network sniffer in Python is a great way to understand how data flows and how network packets are structured. Below is a step-by-step guide to creating a basic network sniffer using the "**Scapy"** library, a powerful Python library for network packet manipulation and analysis.

Step 1: Install Required Libraries
First, you need to install the scapy library. You can install it using pip:

```
pip install scapy
```

Step 2: Import Required Modules
Import the necessary modules from scapy:

```
from scapy.all import *
import argparse
```

Step 3: Define a Packet Sniffing Function
Create a function that will be called every time a packet is captured. This function will analyze and display information about the packet.

```
def packet_callback(packet):
    if packet.haslayer(IP):
        ip_src = packet[IP].src
        ip_dst = packet[IP].dst
        proto = packet[IP].proto

        print(f"IP Packet: {ip_src} -> {ip_dst} Protocol: {proto}")

        if packet.haslayer(TCP):
            tcp_sport = packet[TCP].sport
            tcp_dport = packet[TCP].dport
            print(f"TCP Segment: {ip_src}:{tcp_sport} -> {ip_dst}:{tcp_dport}")

        elif packet.haslayer(UDP):
            udp_sport = packet[UDP].sport
            udp_dport = packet[UDP].dport
            print(f"UDP Datagram: {ip_src}:{udp_sport} -> {ip_dst}:{udp_dport}")

        elif packet.haslayer(ICMP):
            icmp_type = packet[ICMP].type
            icmp_code = packet[ICMP].code
            print(f"ICMP Packet: {ip_src} -> {ip_dst} Type: {icmp_type} Code: {icmp_code}")

        print("-" * 50)
```

Step 4: Set Up Argument Parsing
Use argparse to allow the user to specify the network interface to sniff on:

```
def get_arguments():
    parser = argparse.ArgumentParser(description="Network Sniffer")
    parser.add_argument("-i", "--interface", dest="interface", help="Network interface to sniff
on", required=True)
    options = parser.parse_args()
    return options
```

Step 5: Start Sniffing
Finally, start sniffing packets on the specified network interface:

```
def main():
    options = get_arguments()
    print(f"[*] Starting sniffer on interface {options.interface}")
    sniff(iface=options.interface, prn=packet_callback, store=False)

if __name__ == "__main__":
    main()
```

Step 6: Run the Sniffer
Save the script as network_sniffer.py and run it from the command line, specifying the network interface you want to sniff on:

```
python network_sniffer.py -i eth0
```

Replace eth0 with the appropriate network interface for your system (e.g., wlan0 for wireless).

Explanation

- **sniff()**: This function from scapy captures packets. The prn parameter specifies the callback function to be called for each packet.
- **packet_callback()**: This function processes each packet. It checks for IP, TCP, UDP, and ICMP layers and prints relevant information.
- **argparse**: This module is used to handle command-line arguments, allowing the user to specify the network interface.

Additional Features
You can extend this basic sniffer with additional features, such as:

- **Filtering packets**: Use BPF (Berkeley Packet Filter) syntax to filter packets by protocol, port, etc.
- **Saving packets**: Save captured packets to a file for later analysis.
- **Analyzing more protocols**: Add support for analyzing other protocols like DNS, HTTP, etc.

Example of Filtering Packets
To filter only TCP packets on port 80 (HTTP), you can modify the sniff() call as follows:

```
sniff(iface=options.interface, prn=packet_callback, store=False, filter="tcp port 80")
```

Conclusion

This basic network sniffer provides a foundation for understanding network traffic and packet structure. You can expand upon this by adding more sophisticated analysis, and logging, or even integrating it into a larger network monitoring tool. Always ensure you have permission to sniff network traffic, as unauthorized sniffing can be illegal and unethical.