Task 5

Packet Sniffing with Python

report

1. Introduction

Packet sniffing is the process of capturing and analyzing network packets to inspect their contents, such as source and destination IP addresses, protocols, and payloads. It is a critical technique used in network monitoring, security analysis, and troubleshooting. This report examines a Python-based packet sniffer, detailing its functionality, improvements made to the original code, and its practical applications. The sniffer uses raw sockets to capture network traffic, parse packets, and generate reports summarizing captured data.

2. Overview of the Packet Sniffer

The packet sniffer is a Python script designed to capture raw network packets, extract key information (source/destination IPs, protocol, payload), and generate reports in CSV and text formats. It supports both Linux and Windows, with OS-specific handling for raw socket operations.

2.1 Original Code

The original code provided the following functionality:

- **Socket Creation**: Used AF_PACKET on Linux for Ethernet frames and AF_INET on Windows for IP packets.
- **Packet Parsing**: Extracted IPv4 header details (source/destination IPs, protocol) and up to 64 bytes of payload.
- **Reporting**: Generated a CSV (report.csv) and text file (report.txt) summarizing packet counts for IP pairs.
- **Error Handling**: Basic handling for permission and socket errors.

However, it had limitations:

- Limited Windows support due to raw socket constraints.
- No IPv6 support.
- Basic filtering (only IPv4 packets).
- Potential index errors in payload extraction.
- Overwriting reports without timestamps.
- No real-time statistics or protocol name mapping.

2.2 Improved Code

The improved version (provided in the previous response) addresses these limitations and adds significant functionality:

- **Modular Design**: Separated into PacketSniffer, PacketParser, and ReportGenerator classes for maintainability.
- **Command-Line Interface**:
  - Filters for source/destination IP, protocol (TCP, UDP, ICMP), and interface.
  - Options for max packets and payload logging.
- **IPv6 Support**: Parses IPv6 packets on Linux.
- **Real-Time Stats**: Displays packet count and rate every 5 seconds.
- **Robust Error Handling**: Uses logging for console and file output, with detailed permission instructions.
- **Configurability**: Centralized CONFIG dictionary for tunable parameters (e.g., max packets, payload size).
- **Enhanced Reporting**: Timestamped reports, filter details, and empty report handling.

- **Payload Logging**: Optional logging of full payloads to a file.

3. Technical Details

3.1 System Requirements

- **Python**: 3.6 or higher.
- **Libraries**: pandas for report generation; standard libraries (socket, struct, binascii, etc.) for core functionality.
- **Permissions**:
  - Linux: Requires root privileges (sudo).
  - Windows: Requires Administrator privileges.
- **OS Support**: Linux and Windows; other OSes unsupported without modification.

3.2 Key Components

1. **PacketSniffer Class**:
   - Creates raw sockets (AF_PACKET on Linux, AF_INET on Windows).
   - Captures packets using recvfrom with a timeout.
   - Tracks packet counts and provides real-time statistics.

    - Cleans up sockets, disabling promiscuous mode on Windows.
2. **PacketParser Class**:
    - Parses IPv4 and IPv6 packets (IPv6 on Linux only).
    - Extracts source/destination IPs, protocol, and payload (up to configurable size).
    - Handles malformed packets gracefully.
3. **ReportGenerator Class**:
    - Generates CSV and text reports with timestamped filenames.
    - Includes applied filters in reports.
    - Handles empty data sets with proper column headers.

3.3 Workflow

1. **Initialization**: Parse command-line arguments and configure settings.
2. **Socket Setup**: Create a raw socket based on the OS and enable promiscuous mode (Windows).

3. **Packet Capture**: Continuously capture packets, parse them, and apply filters.
4. **Real-Time Stats**: Log packet counts and rates periodically.
5. **Reporting**: On interruption (Ctrl+C), generate reports summarizing IP pairs and packet counts.
6. **Cleanup**: Close sockets and disable promiscuous mode.

3.4 Example Usage

```
# Capture 500 TCP packets from a specific source IP
sudo python3 packet_sniffer.py --protocol TCP --src-ip 192.168.1.100 --max-packets 500


# Log payloads and capture on a specific interface (Windows)
python3 packet_sniffer.py --interface 192.168.1.10 --log-payloads
```

Output Files:

- report_<timestamp>.csv: IP pairs and packet counts.

- report_<timestamp>.txt: Summary with filters and connection details.
- sniffer.log: Detailed logs of capture process.
- payloads.log (if enabled): Hex-encoded payloads.

4. Use Cases

1. **Network Monitoring**:
   - Track traffic patterns between devices.
   - Identify high-traffic IP pairs or protocols.
2. **Security Analysis**:
   - Detect unusual traffic (e.g., unexpected ICMP packets).
   - Analyze payloads for potential malicious content (with --log-payloads).
3. **Troubleshooting**:
   - Diagnose network issues by inspecting packet details.
   - Verify connectivity between specific IPs.
4. **Education**:

- Learn about network protocols and packet structures.
- Experiment with raw socket programming.

5. Limitations and Challenges

1. **Windows Raw Sockets**:
   - Limited to incoming traffic or specific protocols.
   - Promiscuous mode may not work on all adapters.
   - Recommendation: Use npcap with libraries like scapy for better capture.
2. **IPv6 Support**:
   - Limited to Linux due to Windows raw socket constraints.
   - Future improvement: Add IPv6 parsing for Windows with npcap.
3. **Performance**:
   - High traffic may overwhelm the script, especially with payload logging.
   - Mitigation: Adjust max_packets or use a database for storage.

4. **Security Risks**:
    - Capturing packets may expose sensitive data (e.g., unencrypted payloads).
    - Reports and logs should be stored securely.
    - Recommendation: Use responsibly on trusted networks.
5. **Filtering**:
    - Current filters are basic (IP, protocol).
    - Future improvement: Add port-based or regex-based filtering.

6. Security and Ethical Considerations

Packet sniffing can be used for both legitimate and malicious purposes. Ethical considerations include:

- **Permission**: Only capture traffic on networks you own or have explicit permission to monitor.
- **Privacy**: Avoid logging sensitive data (e.g., passwords, personal information).
- **Compliance**: Adhere to local laws and regulations (e.g., GDPR, HIPAA).
- **Mitigation**: Use filters to limit captured data and encrypt logs/reports.

7. Future Improvements

1. **Advanced Filtering**:
   - Support port-based filtering (e.g., HTTP on port 80).
   - Implement regex for payload or IP matching.
2. **Database Integration**:
   - Store packet data in SQLite or PostgreSQL for analysis.
3. **GUI**:
   - Develop a Tkinter or web-based interface for real-time visualization.
4. **Scapy Integration**:
   - Replace raw sockets with scapy for better cross-platform support and protocol decoding.
5. **Protocol Decoding**:
   - Parse application-layer protocols (e.g., HTTP, DNS) for richer insights.

8. Conclusion

The Python packet sniffer provides a lightweight, customizable tool for capturing and analyzing network traffic. The improved version enhances functionality with modular design, IPv6 support, real-time stats, and robust error handling. While it has limitations (e.g., Windows raw socket constraints), it serves as a valuable tool for network monitoring, security analysis, and education. Future enhancements could include advanced filtering, database storage, and GUI support to make it even more versatile.

For further details or to access the source code, refer to the provided Python script (packet_sniffer.py).