Building a Powerful Packet Sniffing Tool with Python

A Practical Guide to Network Traffic Analysis for Developing an Advanced Packet Sniffer

Karthikeyan Nagaraj

.

Follow

Published in
InfoSec Write-ups
5 min read

.

Nov 17, 2024

Listen

Share

More

Image from pyseek.com
Packet sniffing is often associated with cybersecurity, ethical hacking, and network troubleshooting. With Python and libraries like **Scapy**, we can create robust tools for analyzing and understanding the data flowing through networks.

What is Packet Sniffing?

Packet sniffing involves capturing data packets traveling across a network. These packets are the fundamental units of communication between devices. Analyzing them can reveal:

- **Traffic patterns**
- **Anomalies in data flow**
- **Potential threats, such as SYN scans or DoS attacks**

Packet sniffing has legitimate uses in network security and management, but misuse can lead to ethical and legal consequences. Always ensure you have proper authorization before sniffing network traffic.

Usage:

1. Download the Script

wget https://gist.githubusercontent.com/Cyberw1ng/0e85388b0e918b1a6169b867f8dd2be0/raw/9cc80f8a8eb0c7cbeb6f0bd85119d3cb76c948b2/packet_sniffer.py

2. Installing Requirements

pip install scapysudo python3 packet_sniffer.py

```
└─$ sudo python3 packet_sniffer.py
[sudo] password for cyberw1ng:
[+] Starting packet sniffing on interface eth0
    Filter: None
[TCP] 103.239.139.201:443 -> 192.168.0.101:54894
[TCP] 103.239.139.201:443 -> 192.168.0.101:54894
[TCP] 192.168.0.101:54894 -> 103.239.139.201:443
[TCP] 103.239.139.201:443 -> 192.168.0.101:54894
[TCP] 103.239.139.201:443 -> 192.168.0.101:54894
[TCP] 192.168.0.101:54894 -> 103.239.139.201:443
[TCP] 103.239.139.201:443 -> 192.168.0.101:54894
[TCP] 103.239.139.201:443 -> 192.168.0.101:54894
[TCP] 192.168.0.101:54894 -> 103.239.139.201:443
[TCP] 103.239.139.201:443 -> 192.168.0.101:54894
[TCP] 103.239.139.201:443 -> 192.168.0.101:54894
[TCP] 192.168.0.101:54894 -> 103.239.139.201:443
[TCP] 103.239.139.201:443 -> 192.168.0.101:54894
[TCP] 103.239.139.201:443 -> 192.168.0.101:54894
[TCP] 192.168.0.101:54894 -> 103.239.139.201:443
[TCP] 103.239.139.201:443 -> 192.168.0.101:54894
[TCP] 103.239.139.201:443 -> 192.168.0.101:54894
[TCP] 192.168.0.101:54894 -> 103.239.139.201:443
[TCP] 103.239.139.201:443 -> 192.168.0.101:54894
```

Setting Up Your Environment

To begin, you'll need Python installed on your system, along with the **Scapy** library for handling network packets. Install Scapy with:

pip install scapy

Additionally, ensure you run the script with administrator privileges as packet sniffing requires access to network interfaces.

Advanced Python Script for Packet Sniffing

Here's a comprehensive Python script for packet sniffing. It goes beyond basic sniffing by including features like protocol filtering, packet analysis, anomaly detection, and saving packets to a .pcap file.

Code Explanation

1. Importing Required Libraries

```
from scapy.all import *
import os
import time
from datetime import datetime
```

- **scapy.all**: The scapy library is a powerful tool for packet manipulation and analysis. It simplifies tasks like sniffing, dissecting, and crafting packets.
- **os**: Used for interacting with the operating system, such as clearing the screen.
- **time**: Provides time-related functions for delays and timestamps.
- **datetime**: Enables precise timestamping for each packet.

2. Defining the Packet Callback Function

```
def packet_callback(packet):
protocol = "Unknown"
if packet.haslayer(IP):
ip_src = packet[IP].src
ip_dst = packet[IP].dst
else:
ip_src, ip_dst = "N/A", "N/A"
```

- **Purpose**: This function is executed every time a packet is captured. It processes the packet and extracts critical details.
- **haslayer**: Checks if a specific protocol (e.g., IP) exists in the packet.
- **IP Address Extraction**: If the packet contains an IP layer, the source (ip_src) and destination (ip_dst) addresses

are extracted.

3. Protocol Identification

```
if packet.haslayer(TCP):
protocol = "TCP"
elif packet.haslayer(UDP):
protocol = "UDP"
elif packet.haslayer(ICMP):
protocol = "ICMP"
```

- **Identifying Protocols**: The code checks for layers like TCP, UDP, and ICMP to classify the packet by its transport protocol.

4. Displaying Packet Details

```
timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:
%S")
print(f"[{timestamp}] {protocol} Packet: {ip_src} -> {ip_dst}")
```

- **Timestamp**: A human-readable timestamp is added to each captured packet using datetime.now().strftime.
- **Formatted Output**: The information is displayed in a clear and professional format.

5. Storing Packets to a Pcap File

```
global pcap_writer
if pcap_writer is not None:
pcap_writer.write(packet)
```

- **Pcap Writer**: Packets are saved in a .pcap file using Scapy's writer object for offline analysis.
  The pcap_writer is a global variable initialized in the main function.

6. Main Sniffing Function

```
def start_sniffing(interface=None, filter=None):
global pcap_writer
pcap_writer = PcapWriter("captured_traffic.pcap",
append=True, sync=True)
print("[*] Starting packet capture...")
sniff(iface=interface, filter=filter, prn=packet_callback,
store=0)
```

- **start_sniffing**: The core function that initializes packet sniffing.
- **interface**: The network interface to sniff on. If None, it defaults to all interfaces.
- **filter**: A BPF (Berkeley Packet Filter) string to filter traffic. For example, filter="tcp" captures only TCP traffic.
- **prn**: Specifies the callback function (packet_callback) to process each packet.
- **store=0**: Prevents Scapy from storing packets in memory, optimizing resource usage.

7. Interactive User Menu

```
def menu():
print("\n[ Packet Sniffer ]\n")
print("1. Start Sniffing")
print("2. Specify Interface")
print("3. Apply Filter")
print("4. Exit")
return input("\nEnter your choice: ")
```

- **Purpose**: Provides a user-friendly interface for selecting sniffing options.
- **Menu Options**: Users can start sniffing, choose a

specific interface, apply packet filters, or exit.

8. Handling User Input

```
def main():
interface = None
filter = None

while True:
os.system("clear" if os.name == "posix" else "cls")
choice = menu()

if choice == "1":
start_sniffing(interface, filter)
elif choice == "2":
interface = input("\nEnter interface (leave blank for all): ")
elif choice == "3":
filter = input("\nEnter BPF filter (e.g., 'tcp', 'udp port 53'): ")
elif choice == "4":
print("[*] Exiting...")
break
else:
print("[!] Invalid choice!")
time.sleep(2)
```

- **Persistent Menu**: Loops the menu until the user chooses to exit.
- **Custom Options**:
- **Interface Selection**: Allows users to specify a network interface for targeted sniffing.
- **Filter Application**: Accepts BPF filters to refine captured traffic.

- **Exit Option**: Gracefully terminates the program.

9. Running the Script

if __name__ == "__main__":
main()

- Ensures the script runs only when executed directly and not when imported as a module.

Practical Use Cases

1. Detecting SYN Scans

SYN scans are a common reconnaissance technique used by attackers. This script flags potential SYN scans based on unusual SYN traffic patterns.

2. Analyzing Traffic

You can capture and analyze specific traffic, such as:

- **Web Traffic**: Filter by port 80 or 443.
- **DNS Queries**: Filter by port 53 to analyze DNS resolution patterns.

sudo python packet_sniffer.py -i eth0 -f "tcp port 80"

3. Saving Packets for Later Analysis

Captured packets are saved to a .pcap file, which can be analyzed with network analysis tools like **Wireshark**.
sudo python packet_sniffer.py -i wlan0 -o traffic_capture.pcap

Features of the Script

- **Protocol Analysis**: The script detects and logs TCP, UDP, ICMP, and other protocols.
- **Anomaly Detection**:
- SYN scans.
- Large UDP packets.

- Unusually large IP packets.

3. Filtering: Use BPF (Berkeley Packet Filter) to capture specific traffic (e.g., only TCP or UDP).
4. Save to File: Captured packets are saved as a .pcap file for further analysis using tools like Wireshark.
5. Statistics: Displays packet counts by protocol for quick insights.

Tips for Effective Packet Sniffing

- **Understand Your Network**: Familiarize yourself with normal traffic patterns to identify anomalies.
- **Use Filters**: Applying filters reduces noise and focuses on relevant packets.
- **Run with Sudo**: Administrative privileges are often required for packet sniffing.
- **Analyze with Tools**: Use tools like Wireshark to complement your Python script for deeper analysis.

This script provides a strong foundation for building more advanced network monitoring solutions. Experiment with adding more features like HTTP header analysis, payload decryption, or integration with alerting systems.
A YouTube Channel for Cybersecurity Lab's Poc and Write-ups

Cyberw1ng

Learn Cyber Security and Create Awareness ~ cyberwing Stay tuned with me, Subscribe, and Like the Videos... Ask Doubts...

www.youtube.com