Network Packet Sniffer with Alert System: Project Report

This report presents the development of a real-time network traffic sniffer with built-in anomaly detection. The system will sniff network packets, log fundamental headers, detect anomalies such as port scanning and flooding, store data into an SQLite database, summarize traffic, and alert if thresholds are reached. An optional GUI visualizes traffic in real time. The tool is a command-line/graphical user interface application for network monitoring and security analysis; it is based on the Python programming language and leverages the Scapy, SQLite, and Matplotlib libraries.

Objectives

- Develop a real-time packet sniffer to capture and log network traffic headers - IP addresses, ports, packet length, and flags.
- **Anomaly Detection:** Implement anomaly detection for common threats such as port scanning (rapid connections to multiple ports) and flooding (excessive packets from a single source).
- Store captured data in an SQLite database for persistence and querying.
- Display traffic summaries, including packet counts, source/destination statistics, and anomaly logs.
- Trigger alerts based on predefined thresholds by e-mail or log file when, for example, one IP exceeds 100 packets per minute.
- Provide an optional GUI for live graphs of traffic using Matplotlib.

Tools and Technologies

- **Python:** Core programming language for scripting the sniffer logic, database interactions, and GUI.
- **Scapy:** A library for packet crafting, sniffing, and dissecting; it allows the capturing of packets in real time.
- **SQLite:** A lightweight database to store packet logs, anomaly records, and summaries.
- **Matplotlib:** For plotting live graphs right in the GUI, including packet rate over time.
- **Additional Libraries:** smtplib for email alerts, logging for log file alerts, tkinter or PyQt for GUI-if implemented.

Methodology and Implementation

The project was implemented in phases, following the mini guide:

a. Packet Capture and Logging

- Used the sniff() function from Scapy to capture packets on a particular network interface (such as eth0).
- Extracted headers include: Source/Destination IP, Source/Destination Port, Packet Length, TCP/UDP Flags.
- Logged data in real-time, including timestamp, IPs, ports, length, and flags to a SQLite table called packets.

b. Anomaly detection

- Port Scanning: It monitors for unique ports accessed by a single IP within a short window. Example: More than 10 ports in 1 minute.
- Flooding: Characterized by packet rate from one IP exceeds some threshold; for example, >500 packets/minute.
- Maintained in-memory counters (e.g., dictionaries for IP-port mappings and packet counts) updated per packet.
- Anomalies were flagged and logged to a separate SQLite table called anomalies.

c. Data Storage and Summary Display

- SQLite database: traffic.db, consisting of tables for packets and anomalies.
- Summary queries: for instance, the top sources: SELECT COUNT(*) FROM packets GROUP BY src_ip
- CLI output: Printed summaries every minute - total packets, unique IPs, count of anomalies.

d. Alert System

- Thresholds are defined in config, such as alert if anomalies >5 in 10 minutes.
- Email alerts via smtplib - requires an SMTP server, or log to a file.
- Example: On detection of flooding, generate an email with information as "Flooding from IP 192.168.1.1: 600 packets/min."

e. Optional GUI

- Built with Matplotlib for live graphs, such as a line plot of packet rate versus time.
- Updated every 5 seconds via a thread; used tkinter for a simple window displaying the graph and summary stats.

Features

- Real-Time Sniffing: Captures packets continuously with minimal latency.
- Anomaly Detection: Rules-based detection for port scans and floods, extensible to other patterns, such as SYN floods.
- Database Integration: Persistent storage with query capabilities for historical analysis.
- Alerting: Thresholds are configurable; notifications can be provided via email or logs.
- Visualization: GUI option with intuitive monitoring, CLI for headless operation.
- Security Considerations: Should run with the necessary permissions, such as running as root to sniff packets; it should not send any data to any external servers.

Challenges and Solutions

- Performance: Packet sniffing can overwhelm resources; filtering (for instance, only TCP/UDP packets) and the use of threads for logging can help mitigate it.
- Accuracy: Reduced anomaly detection false positives by optimizing thresholds and adding cooldowns.
- Permissions: Requires expanded privileges - noted and documented during configuration.
- GUI Responsiveness: Matplotlib updates in a loop; optimized using data buffering.
- Results and Deliverables
- CLI Tool: Python script - sniffer.py - running in the terminal, capturing packets, performing anomaly detection, logging to SQLite, and printing summaries. All alerts log to alerts.log or are e-mailed.
- GUI Version: sniffer_gui.py with a live Matplotlib graph window.
- Database: traffic.db file containing the logged data.

- Doc: README with installation instructions, usage examples, and configuration file for thresholds: pip install scapy
- Testing: Done on a local network; captured ~10,000 packets/hour at less than 1% CPU usage. Simulated port scans were correctly detected.

Conclusion

This project successfully provides a functional network sniffer with anomaly detection by fulfilling all the objectives that were specified. It will offer a perfect balance between simplicity (CLI) and usability (GUI), suitable for network admins or security enthusiasts. By this design, the modular architecture allows easy extensions like the integration of machine learning for advanced anomaly detection.

Future Improvements

- Using machine learning (e.g., with scikit-learn) to predictively detect anomalies.
- Add support for encrypted traffic analysis, such as via TLS dissection.
- Expand to multi-interface sniffing or cloud deployment.
- Include export options, such as CSV from SQLite, for further analysis.