**Project Report**

Project 1-

**Title: Network Packet Sniffer with Alert System**

**Submitted by: Manish Panjwani**

**Institution: IIT (ISM) Dhanbad**

**Year: 2025**

**1. Abstract**

Network monitoring and anomaly detection play a crucial role in ensuring cybersecurity and identifying malicious activities such as port scanning, SYN flooding, ARP spoofing, and data exfiltration.  
The SnifferWithAlertSystem is a Python-based network monitoring and alerting tool that captures live network traffic using Scapy, stores the data in a SQLite database, and performs automated anomaly detection through pattern analysis and statistical thresholds.  
It provides visualization using a Tkinter GUI and generates alerts for suspicious network behavior, helping analysts quickly respond to potential threats.

**2. Introduction**

With the rise of cyberattacks, real-time network monitoring tools have become essential for defensive and offensive security. Traditional sniffers like Wireshark capture packets but require manual analysis.  
This project automates both packet capture and anomaly detection. The goal is to develop a lightweight system capable of:

* Capturing packets in real-time
* Storing structured network data for later analysis
* Automatically identifying suspicious patterns
* Providing alerts and visualizations to users

**3. Objectives**

1. Capture live network packets and log key metadata (IP, MAC, Protocol, Ports, etc.).
2. Store all packet information in a SQLite database for persistence.
3. Detect common anomalies such as:
   * Port Scans
   * SYN Flood Attacks
   * ICMP Floods / Ping Sweeps
   * ARP Spoofing
   * Data Exfiltration Attempts
4. Export captured data into CSV format for offline analysis.
5. Display the captured data and generated alerts through a GUI interface.

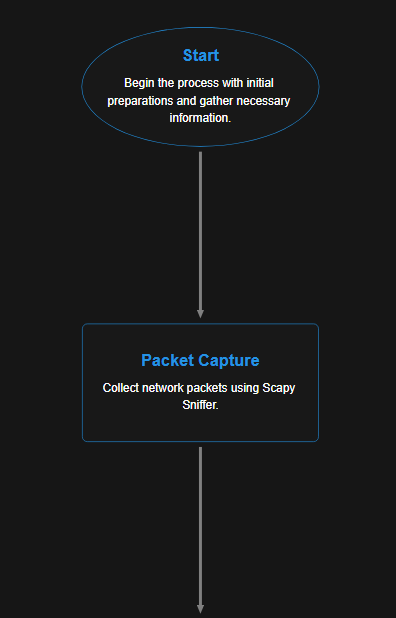
**4. System Requirements**

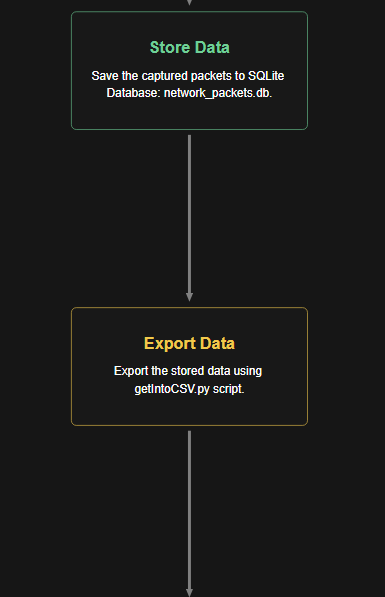
Software Requirements

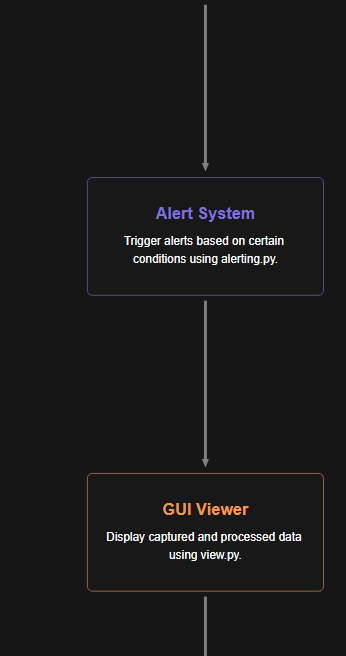
* Python 3.x
* Libraries:
  + scapy
  + pandas
  + sqlite3
  + tkinter
  + datetime

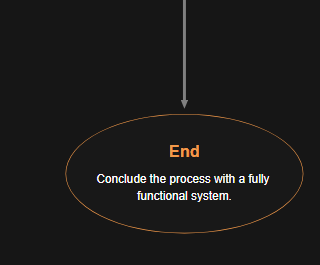
Hardware Requirements

* Any system with Internet connectivity
* Minimum 4GB RAM
* Administrator/root privileges for packet capture
* **5. System Architecture**
* Architecture Flow:









**6. Implementation**

**Modules Overview**

**1. sniffer.py**

* Captures live network traffic using **Scapy**.
* Extracts essential details: source/destination IP, MAC, protocol, ports, packet size.
* Classifies traffic direction (IN/OUT).
* Stores structured data into network\_packets.db using **SQLite**.

**2. getIntoCSV.py**

* Reads packet data from the database.
* Exports data into a CSV file (network\_packets\_export.csv).
* Provides a **Tkinter Treeview GUI** for viewing packet logs.

**3. alerting.py**

* Loads exported CSV data and performs anomaly detection:
  + **Port Scan Detection** → same IP contacting >10 ports within 10 seconds
  + **SYN Flood Detection** → excessive TCP SYN packets
  + **ICMP Flood Detection** → too many ICMP packets
  + **ARP Spoofing Detection** → multiple MACs for one IP
  + **Data Exfiltration** → >10 MB data sent
* Generates alert entries and saves them in detected\_anomalies.csv.

**4. view.py**

* Displays the packet database content in an interactive **Tkinter table**.

**5. requirements.txt**

Specifies dependencies for setting up the environment quickly.

**7. Algorithmic Steps**

**A. Packet Sniffing**

1. Initialize Scapy sniffer.
2. Capture each packet and identify its layer (TCP, UDP, ICMP, ARP).
3. Extract relevant metadata.
4. Determine packet direction.
5. Store into the SQLite database.

**B. Alert System**

1. Load packet logs into Pandas DataFrame.
2. Apply time-based and count-based anomaly rules.
3. Flag potential attack events.
4. Save and display alerts.

**8. Results and Output**

**Sample Console Output (Sniffer)**

[2025-10-27 15:42:10] TCP-OUT: 60 Bytes SRC:112.228.169.106:54023 DST:8.8.8.8:443

[2025-10-27 15:42:12] ICMP-OUT: 64 Bytes SRC:112.238.133.103 DST:8.8.4.4

**Sample Alerts**

| **Anomaly Type** | **Source IP** | **Details** | **Time** |
| --- | --- | --- | --- |
| Port Scan | 112.168.1.100 | 15 unique destination ports in 10 sec | 2025-10-27 14:55:10 |
| ARP Spoofing | 112.168.1.101 | 3 unique MACs for same IP | — |

**GUI View**

Tkinter interface displays all captured packets in a scrollable table.

**9. Advantages**

* Lightweight and real-time.
* Local database logging (no external dependency).
* Detects multiple attack patterns automatically.
* User-friendly visualization.
* Open-source and easily extensible.

**10. Limitations**

* Detection thresholds are static (could be adaptive).
* Limited visualization and alert notification (no sound/email alerts yet).
* Works best on single-interface systems.

**11. Future Enhancements**

1. Integrate real-time alert notifications (sound, email, or Slack).
2. Build a live dashboard using Flask or Streamlit.
3. Add ML-based anomaly detection models.
4. Support for IPv6 analysis and multi-interface capture.
5. Integration with ELK/Prometheus for enterprise use.

**12. Conclusion**

The **SnifferWithAlertSystem** successfully demonstrates real-time packet capture and anomaly detection using Python.  
It acts as a foundation for more advanced network intrusion detection systems (NIDS).  
By combining packet sniffing, database logging, and rule-based alerting, the tool enhances situational awareness and helps security analysts detect suspicious activities efficiently.

**13. References**

* Scapy Documentation
* Wireshark Wiki
* Python SQLite3 Module
* Pandas Documentation

**Project 2-**

**Title**

**Web Application Vulnerability Scanner**

**1. Abstract**

This project implements a lightweight Python-based web application vulnerability scanner that detects common web vulnerabilities (XSS, SQL Injection, SSRF, security misconfiguration, outdated components, etc.). The scanner crawls pages, locates input fields, injects test payloads, analyzes HTTP responses for evidence of vulnerabilities, and exposes results via a simple Flask web UI. Results are logged with evidence and severity to assist security analysts and testers.

**2. Objectives**

* Crawl target web pages and discover user-controllable inputs (forms, query parameters).
* Detect common vulnerabilities: XSS, SQLi, SSRF, misconfigurations (phpinfo), outdated server components, and basic broken-access control checks.
* Provide a Flask-based UI to launch scans and view/save findings.
* Generate human-readable reports (plain text/CSV/JSON) with evidence and risk levels based on OWASP Top 10.

**3. Tools & Libraries**

* Python 3.x
* requests — HTTP client
* BeautifulSoup (bs4) — HTML parsing
* Flask — web UI (recommended)
* argparse, textwrap, re, logging, json

**4. Implementation (Modules)**

**Provided files**

* **main.py** — CLI scanner + OWASP test functions (crawl, inject, analyze).
* **XSS-attack-vectors.txt** — payloads to test XSS.
* **(Recommended)** app.py — Flask UI (not included but described below).
* **Deliverable** — scanner (Python) + optional Flask web interface + reports.

**5. Key Algorithms & Techniques**

**Crawling & Discovery**

* Fetch the target URL, parse HTML for <form> and <input> elements, and extract form method/action and input names.
* Optionally extract links and follow same-origin pages to find more inputs.

**Injection Testing**

* For each discovered input field:
  + Send benign POST/GET requests to get baseline response.
  + Inject payloads from XSS-attack-vectors.txt or SQLi list and compare the response body and headers for evidence (payload reflected, SQL errors, behavioral changes).
* For SSRF, attempt to submit internal URLs (e.g., http://localhost:8080) where the app would fetch the resource server-side.

**Detection patterns**

* XSS evidence: payload or encoded variant reflected in response body, or DOM-change indicators (note: JS-driven XSS needs browser emulation).
* SQLi evidence: database error messages or successful login bypass responses.
* SSRF evidence: server responses including internal content or error text referencing local resources.
* Misconfiguration: existence of phpinfo.php or other debug pages.
* Outdated components: check Server header for known vulnerable versions.

**6. Risk Rating (OWASP Top 10 mapping)**

Use a mapping to label each finding with low, medium, or high. (You already use RISK\_RATINGS in your file — keep that mapping.)

**7. Limitations & Ethical Considerations**

* **Do not scan systems you do not own** or have explicit written permission to test. Unauthorized scanning can be illegal.
* The scanner is **non-exhaustive**: it uses simple pattern matching and reflection detection. Many vulnerabilities (e.g., DOM XSS, blind SQLi, auth bypass) require advanced techniques.
* JS-heavy single-page apps may require a browser automation tool (Selenium) to find inputs created dynamically.
* False positives are possible — each finding needs manual validation.

**8. Future Enhancements**

* Add a Flask UI (scan queues, scheduling, results dashboard).
* Add authentication handling (login session support, CSRF token handling).
* Add rate limiting, throttling, and scan scoping.
* Integrate with vulnerability trackers (Jira/CSV/JSON export).
* Add ML or heuristics for prioritization and deduplication.
* Add optional headless browser scanning for DOM-based XSS and JS-driven flows.

**9. Usage (CLI)**

Example:

python main.py -t https://target.example.com -o results.txt

This runs the tests and writes results to results.txt.

10. **References**

* OWASP Top 10 (2021) — guidelines and mapping
* Requests & BeautifulSoup docs
* General web security testing methodologies