# TASK 3: Secure Coding Review

## Secure Coding Review: Basic Python Network Sniffer

### 1. Application and Language Selected

- **Application:** sniffer.py, the basic network packet sniffer from the previous task.
- **Language:** Python 3
- **Dependencies:** scapy, colorama

### 2. Audit Methodology

We will perform a **manual code inspection** focusing on common security pitfalls relevant to a network application that runs with elevated privileges. The review will be guided by security principles such as:

- **Principle of Least Privilege:** Does the application use high privileges for longer than necessary?
- **Input Validation:** Is data received from the network (an untrusted source) properly handled?
- **Error Handling:** Are errors handled securely without leaking information or creating denial-of-service conditions?
- **Dependency Management:** Are the third-party libraries secure?

### 3. Security Vulnerability Findings

Here is a formal documentation of the findings.

**Security Audit Report for sniffer.py**

**Executive Summary:**

The sniffer.py script, while functional, contains several security vulnerabilities common in network tools. The most critical issue is the **prolonged use of root privileges**, which magnifies the impact of other vulnerabilities. The script is susceptible to a **Denial of Service (DoS)** attack via oversized packets and **improper output handling** that could lead to terminal manipulation. Recommendations focus on applying the principle of least privilege, validating input size, sanitizing output, and implementing specific error handling.

**Finding 1: Excessive Privileges / Failure to Drop Privileges**

- **Vulnerability:** The entire script runs with root/administrator privileges, but only the sniff() function requires them to bind to the network interface. The packet processing

logic (parsing, decoding, printing), which is the most complex part of the script, also runs with these elevated privileges.

- **Severity: High**
- **Impact:** Any vulnerability in the packet parsing or printing code (like a buffer overflow or a DoS flaw) can be exploited to gain full control over the system, as the code is running as root.
- **Recommendation:** For a simple script, the easiest remediation is to keep the code that runs as root **as simple and minimal as possible**. For more complex applications, the standard practice is to separate privileges: a small, privileged process captures packets and passes them to a larger, unprivileged process for analysis.

**Finding 2: Potential for Denial of Service (DoS) via Unbounded Data Processing**

- **Vulnerability:** The code handles the packet payload without any size checks.

  Generated python

  ```
  if Raw in packet:
      payload = packet[Raw].load
      # ...
      decoded_payload = payload.decode('utf-8', 'ignore')
      print(decoded_payload)
  ```

- **Severity: Medium**
- **Impact:** An attacker on the local network could send a specially crafted packet with an enormous payload (e.g., several gigabytes). The script would attempt to load this entire payload into memory, causing it to consume all available RAM and crash. This is a resource exhaustion DoS attack.
- **Recommendation: Always validate the size of untrusted input.** Before processing the payload, truncate it to a reasonable maximum length. This prevents memory exhaustion.

**Finding 3: Improper Output Sanitization Leading to Terminal Injection**

- **Vulnerability:** The script prints the decoded payload directly to the terminal.

  Generated python

  ```
  print(decoded_payload)
  ```

- **Severity: Low / Medium**
- **Impact:** A malicious payload could contain terminal escape sequences. These special character sequences can be used to:
  - Move the cursor and overwrite previous output, hiding malicious activity.
  - Change terminal colors to confuse the user.
  - In some (mostly older or misconfigured) terminals, potentially execute commands.
- **Recommendation: Sanitize all data before printing it to the console.** A simple way to do this is to replace non-printable characters with a placeholder (like .) or their hex representation.

**Finding 4: Overly Broad Exception Handling**

- **Vulnerability:** The try...except block uses except Exception:, which is too broad.

  Generated python

  ```
  try:
      decoded_payload = payload.decode('utf-8', 'ignore')
      print(decoded_payload)
  except Exception: # This is the issue
      print(payload)
  ```

- 
- **Severity: Low**
- **Impact:** This catches *all* exceptions, including KeyboardInterrupt (when the user presses Ctrl+C). This can make the program difficult to terminate cleanly. It also hides other potential RuntimeError or TypeError bugs that a developer should be aware of.
- **Recommendation: Catch specific exceptions.** The code is trying to handle a decoding error, so it should only catch UnicodeDecodeError.

## 4. Remediation Steps and Safer Code

Here is the remediated version of sniffer.py that addresses the findings above.

Generated python

```
# Import the necessary modules from Scapy
from scapy.all import *
from scapy.layers.inet import IP, TCP, UDP, ICMP
import sys
import re

# For coloring the output
IS_WINDOWS = sys.platform.startswith('win')
if IS_WINDOWS:
    from colorama import init
    init()

# Define color constants
R, G, Y, B, C, E = '\033[91m', '\033[92m', '\033[93m', '\033[94m', '\033[96m', '\033[0m'

# [REMEDIATION] Define a maximum payload size to prevent DoS
MAX_PAYLOAD_SIZE = 1024

def sanitize_payload(payload):
    """
    [REMEDIATION] Sanitize payload to prevent terminal injection.
    Replace non-printable characters with a '.'
    """
```

```
    if not payload:
        return ""
    # Regex to find non-printable characters (excluding common whitespace)
    # This keeps tabs, newlines, etc., but removes control characters.
    return re.sub(r'[^ -~ \t\n\r]', '.', payload)

def packet_callback(packet):
    """
    This function is called for each captured packet.
    It analyzes and prints key information about the packet.
    """
    if IP in packet:
        ip_src = packet[IP].src
        ip_dst = packet[IP].dst

        print(f"\n{B}[+] New Packet Captured{E}")
        print(f"{G}    Source IP: {ip_src}{E}")
        print(f"{Y}    Destination IP: {ip_dst}{E}")

        proto_name = ""
        if TCP in packet:
            proto_name = "TCP"
        elif UDP in packet:
            proto_name = "UDP"
        elif ICMP in packet:
            proto_name = "ICMP"
        else:
            proto_name = f"IP Protocol {packet[IP].proto}"

        print(f"{R}    Protocol: {proto_name}{E}")

        # Check for payload data
        if Raw in packet:
            payload_raw = packet[Raw].load

            # [REMEDIATION] Truncate payload to prevent DoS
            if len(payload_raw) > MAX_PAYLOAD_SIZE:
                print(f"{C}    Payload (truncated at {MAX_PAYLOAD_SIZE} bytes):{E}")
                payload_truncated = payload_raw[:MAX_PAYLOAD_SIZE]
            else:
                print(f"{C}    Payload:{E}")
                payload_truncated = payload_raw

            # [REMEDIATION] Use specific exception handling and sanitize output
            try:
                decoded_payload = payload_truncated.decode('utf-8', 'ignore')
```

```
            sanitized_payload = sanitize_payload(decoded_payload)
            print(sanitized_payload)
        except UnicodeDecodeError:
            # If it's not valid UTF-8, print the sanitized raw representation
            print(sanitize_payload(str(payload_truncated)))

def main():
    """
    Main function to start the sniffer.
    """
    print("Starting secure network sniffer...")
    # [RECOMMENDATION] Acknowledging the risk:
    # This script must be run with root/admin privileges.
    # The packet processing logic is kept simple to minimize the attack surface.
    sniff(prn=packet_callback, store=0, count=20)
    print("\nSniffing complete.")

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

## 5. General Secure Coding Best Practices

Based on this audit, here are some general recommendations for writing safer code:

1. **Follow the Principle of Least Privilege:** Never run code with more permissions than it absolutely needs. If a task requires root, isolate that task and drop privileges as soon as possible.
2. **Treat All Input as Untrusted:** Whether from a user, a file, a database, or the network, all input should be validated. Check for type, length, format, and range before processing it.
3. **Encode and Sanitize Output:** When displaying data, especially data that originated from an external source, always encode or sanitize it to prevent injection attacks (e.g., XSS in web apps, Terminal Injection in CLIs).
4. **Use Specific Exception Handling:** Avoid broad except clauses. Catching specific exceptions makes your code more robust, predictable, and secure.
5. **Keep Dependencies Updated:** Use tools like pip-audit (for Python) or GitHub's Dependabot to be notified of known vulnerabilities in the libraries you use. An attacker will always target the weakest link, which is often an old, vulnerable dependency.