```
# packet_capture.py
# Packet capture and attack detection module using Scapy

import time
import threading
from collections import defaultdict, deque
from datetime import datetime, timedelta
from typing import Dict, Set, Any
import logging
import socket
import uuid

try:
    from scapy.all import sniff, IP, TCP, UDP, ICMP, ARP, get_if_list
    from scapy.layers.inet import TCP
except ImportError:
    print("Scapy not installed. Install with: pip install scapy")
    exit(1)

from database import DatabaseManager

class AttackDetector:
    def __init__(self, time_window: int = 60):
        """Initialize attack detection with time window in seconds"""
        self.time_window = time_window
        self.port_scan_threshold = 10  # ports
```

```
        self.syn_flood_threshold = 100  # packets per window
        self.brute_force_threshold = 20  # attempts per window

        # Time-windowed counters
        self.port_scans = defaultdict(lambda: defaultdict(set))  # ip -> time -> ports
        self.syn_floods = defaultdict(lambda: defaultdict(int))  # ip -> time -> count
        self.brute_force = defaultdict(lambda: defaultdict(lambda: defaultdict(int)))  # ip -> time
-> port -> count
        self.arp_table = {}  # ip -> mac mapping

        # Thread lock for counter operations
        self.lock = threading.Lock()

        # Start cleanup thread
        self.cleanup_thread = threading.Thread(target=self._cleanup_counters, daemon=True)
        self.cleanup_thread.start()

        self.logger = logging.getLogger(__name__)

    def _get_time_window(self) -> int:
        """Get current time window identifier"""
        return int(time.time() // self.time_window)

    def _cleanup_counters(self):
        """Periodically clean up old counter data"""
        while True:
            time.sleep(self.time_window)
```

```
current_window = self._get_time_window()

with self.lock:
    # Clean port scan counters
    for ip in list(self.port_scans.keys()):
        for window in list(self.port_scans[ip].keys()):
            if window < current_window - 1:
                del self.port_scans[ip][window]
        if not self.port_scans[ip]:
            del self.port_scans[ip]

    # Clean SYN flood counters
    for ip in list(self.syn_floods.keys()):
        for window in list(self.syn_floods[ip].keys()):
            if window < current_window - 1:
                del self.syn_floods[ip][window]
        if not self.syn_floods[ip]:
            del self.syn_floods[ip]

    # Clean brute force counters
    for ip in list(self.brute_force.keys()):
        for window in list(self.brute_force[ip].keys()):
            if window < current_window - 1:
                del self.brute_force[ip][window]
        if not self.brute_force[ip]:
            del self.brute_force[ip]
```

```
def detect_port_scan(self, src_ip: str, dst_port: int) -> bool:
    """Detect port scanning activity"""
    current_window = self._get_time_window()

    with self.lock:
        self.port_scans[src_ip][current_window].add(dst_port)

        # Count unique ports accessed in current and previous window
        total_ports = len(self.port_scans[src_ip][current_window])
        if current_window - 1 in self.port_scans[src_ip]:
            total_ports += len(self.port_scans[src_ip][current_window - 1])

        return total_ports >= self.port_scan_threshold

def detect_syn_flood(self, src_ip: str) -> bool:
    """Detect SYN flood attack"""
    current_window = self._get_time_window()

    with self.lock:
        self.syn_floods[src_ip][current_window] += 1

        # Count SYN packets in current and previous window
        total_syns = self.syn_floods[src_ip][current_window]
        if current_window - 1 in self.syn_floods[src_ip]:
            total_syns += self.syn_floods[src_ip][current_window - 1]
```

```
        return total_syns >= self.syn_flood_threshold

    def detect_brute_force(self, src_ip: str, dst_port: int) -> bool:
        """Detect brute force attack"""
        current_window = self._get_time_window()

        with self.lock:
            self.brute_force[src_ip][current_window][dst_port] += 1

            # Count attempts to specific port in current and previous window
            total_attempts = self.brute_force[src_ip][current_window][dst_port]
            if current_window - 1 in self.brute_force[src_ip]:
                total_attempts += self.brute_force[src_ip][current_window - 1][dst_port]

            return total_attempts >= self.brute_force_threshold

    def detect_arp_spoofing(self, ip: str, mac: str) -> bool:
        """Detect ARP spoofing"""
        if ip in self.arp_table:
            if self.arp_table[ip] != mac:
                # Different MAC for same IP - potential spoofing
                return True
        else:
            self.arp_table[ip] = mac
        return False
```

```
class PacketCapture:
    def __init__(self, interface: str = None, node_id: str = None,
                 db_config: Dict = None):
        """Initialize packet capture system"""
        self.interface = interface or self._get_default_interface()
        self.node_id = node_id or f"node_{str(uuid.uuid4())[:8]}"
        self.running = False

        # Initialize database connection
        self.db = DatabaseManager(**(db_config or {}))

        # Initialize attack detector
        self.detector = AttackDetector()

        # Statistics
        self.stats = {
            'packets_captured': 0,
            'alerts_generated': 0,
            'start_time': None
        }

        self.logger = logging.getLogger(__name__)

    def _get_default_interface(self) -> str:
        """Get default network interface"""
```

```
    interfaces = get_if_list()
    # Filter out loopback and select first available interface
    for iface in interfaces:
        if iface != 'lo' and not iface.startswith('lo'):
            return iface
    return interfaces[0] if interfaces else None

def _extract_packet_info(self, packet) -> Dict[str, Any]:
    """Extract relevant information from packet"""
    info = {
        'source_ip': None,
        'destination_ip': None,
        'source_port': None,
        'destination_port': None,
        'protocol': 'Unknown',
        'packet_size': len(packet),
        'tcp_flags': None,
        'node_id': self.node_id,
        'raw_data': str(packet)[:1000]  # Truncate for storage
    }

    try:
        if IP in packet:
            info['source_ip'] = packet[IP].src
            info['destination_ip'] = packet[IP].dst
```

```
    if TCP in packet:
        info['protocol'] = 'TCP'
        info['source_port'] = packet[TCP].sport
        info['destination_port'] = packet[TCP].dport

        # Extract TCP flags
        flags = []
        if packet[TCP].flags.S: flags.append('SYN')
        if packet[TCP].flags.A: flags.append('ACK')
        if packet[TCP].flags.F: flags.append('FIN')
        if packet[TCP].flags.R: flags.append('RST')
        if packet[TCP].flags.P: flags.append('PSH')
        if packet[TCP].flags.U: flags.append('URG')
        info['tcp_flags'] = ','.join(flags)

    elif UDP in packet:
        info['protocol'] = 'UDP'
        info['source_port'] = packet[UDP].sport
        info['destination_port'] = packet[UDP].dport

    elif ICMP in packet:
        info['protocol'] = 'ICMP'

elif ARP in packet:
    info['protocol'] = 'ARP'
    info['source_ip'] = packet[ARP].psrc
```

```
            info['destination_ip'] = packet[ARP].pdst

    except Exception as e:
        self.logger.error(f"Error extracting packet info: {e}")

    return info

def _generate_alert(self, alert_type: str, source_ip: str,
                    destination_ip: str = None, severity: int = 1,
                    description: str = "", count: int = 1):
    """Generate and store security alert"""
    alert_data = {
        'alert_type': alert_type,
        'source_ip': source_ip,
        'destination_ip': destination_ip,
        'severity': severity,
        'description': description,
        'node_id': self.node_id,
        'count': count
    }

    if self.db.insert_alert(alert_data):
        self.stats['alerts_generated'] += 1
        self.logger.warning(f"ALERT: {alert_type} from {source_ip} - {description}")

def _analyze_packet(self, packet_info: Dict[str, Any]):
```

```
"""Analyze packet for suspicious activity"""
src_ip = packet_info['source_ip']
dst_ip = packet_info['destination_ip']
dst_port = packet_info['destination_port']
protocol = packet_info['protocol']

if not src_ip:
    return

# Detect port scanning
if dst_port and self.detector.detect_port_scan(src_ip, dst_port):
    self._generate_alert(
        'Port Scan',
        src_ip,
        dst_ip,
        severity=3,
        description=f"Port scanning detected from {src_ip}"
    )

# Detect SYN flood attacks
if protocol == 'TCP' and packet_info.get('tcp_flags', '').find('SYN') != -1:
    if self.detector.detect_syn_flood(src_ip):
        self._generate_alert(
            'SYN Flood',
            src_ip,
            dst_ip,
```

```
                severity=4,
                description=f"SYN flood attack detected from {src_ip}"
            )

    # Detect brute force attacks on common ports
    if dst_port in [22, 23, 21, 25, 110, 143, 993, 995, 3389]:
        if self.detector.detect_brute_force(src_ip, dst_port):
            self._generate_alert(
                'Brute Force',
                src_ip,
                dst_ip,
                severity=3,
                description=f"Brute force attack detected from {src_ip} on port {dst_port}"
            )

    # Detect ARP spoofing
    if protocol == 'ARP':
        # Extract MAC address from raw packet data
        try:
            if hasattr(packet_info, '_packet') and ARP in packet_info._packet:
                arp_packet = packet_info._packet[ARP]
                if self.detector.detect_arp_spoofing(arp_packet.psrc, arp_packet.hwsrc):
                    self._generate_alert(
                        'ARP Spoofing',
                        src_ip,
                        severity=2,
```

```
                    description=f"ARP spoofing detected for IP {src_ip}"
                )
        except Exception as e:
            self.logger.debug(f"ARP analysis error: {e}")

def _packet_handler(self, packet):
    """Handle captured packets"""
    try:
        # Extract packet information
        packet_info = self._extract_packet_info(packet)
        packet_info['_packet'] = packet  # Store original packet for analysis

        # Store packet in database
        self.db.insert_packet(packet_info)

        # Analyze for suspicious activity
        self._analyze_packet(packet_info)

        # Update statistics
        self.stats['packets_captured'] += 1

        # Log every 1000 packets
        if self.stats['packets_captured'] % 1000 == 0:
            self.logger.info(f"Captured {self.stats['packets_captured']} packets, "
                             f"Generated {self.stats['alerts_generated']} alerts")
```

```
    except Exception as e:
        self.logger.error(f"Error processing packet: {e}")

def start_capture(self, packet_filter: str = ""):
    """Start packet capture"""
    if self.running:
        self.logger.warning("Capture already running")
        return

    self.running = True
    self.stats['start_time'] = datetime.now()

    self.logger.info(f"Starting packet capture on interface {self.interface}")
    self.logger.info(f"Node ID: {self.node_id}")

    try:
        # Start packet capture
        sniff(
            iface=self.interface,
            prn=self._packet_handler,
            filter=packet_filter,
            store=0,  # Don't store packets in memory
            stop_filter=lambda x: not self.running
        )
    except Exception as e:
        self.logger.error(f"Capture error: {e}")
```

```
        finally:
            self.running = False

    def stop_capture(self):
        """Stop packet capture"""
        self.logger.info("Stopping packet capture")
        self.running = False

    def get_stats(self) -> Dict:
        """Get capture statistics"""
        stats = self.stats.copy()
        if stats['start_time']:
            stats['runtime'] = str(datetime.now() - stats['start_time'])
        return stats

def main():
    """Main function for standalone execution"""
    import argparse

    # Setup logging
    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
    )

    # Parse command line arguments
```

```
    parser = argparse.ArgumentParser(description='Network Traffic Analyzer - Packet
Capture Node')
    parser.add_argument('--interface', '-i', help='Network interface to monitor')
    parser.add_argument('--node-id', help='Unique identifier for this capture node')
    parser.add_argument('--db-host', default='localhost', help='Database host')
    parser.add_argument('--db-port', type=int, default=5432, help='Database port')
    parser.add_argument('--db-name', default='network_analyzer', help='Database name')
    parser.add_argument('--db-user', default='postgres', help='Database user')
    parser.add_argument('--db-password', default='password', help='Database password')
    parser.add_argument('--filter', help='BPF filter for packet capture')

    args = parser.parse_args()

    # Database configuration
    db_config = {
        'host': args.db_host,
        'port': args.db_port,
        'database': args.db_name,
        'user': args.db_user,
        'password': args.db_password
    }

    # Initialize and start packet capture
    try:
        capture = PacketCapture(
            interface=args.interface,
            node_id=args.node_id,
```

```
            db_config=db_config
        )

        print(f"Starting packet capture on {capture.interface}")
        print("Press Ctrl+C to stop...")

        capture.start_capture(args.filter or "")

    except KeyboardInterrupt:
        print("\nStopping packet capture...")
    except Exception as e:
        print(f"Error: {e}")

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