sniffer.py - Network Packet Capture Module

Overview

File: `client/sniffer.py`
Purpose: Capture network packets and save them to CSV files for later upload
Role: Producer in the producer-consumer architecture
Runs as: Standalone process (requires admin/root privileges)

What It Does

`sniffer.py` is responsible for capturing raw network traffic and converting it into structured data:

1. **Captures** packets from all network interfaces using Scapy
2. **Extracts** 30+ features from each packet (IPs, ports, protocols, etc.)
3. **Buffers** packets in memory
4. **Saves** to CSV files every 30 seconds using atomic writes
5. **Signals** completion with `.ready` marker files

Architecture

```
Network Interface(s)
       ↓
    Scapy
       ↓
packet_summary() → Extract features
       ↓
PacketBuffer → In-memory buffer
       ↓
(Every 30 seconds)
       ↓
save_to_csv_atomic() → Write to disk
       ↓
logs/pending_upload/packets_TIMESTAMP.csv
logs/pending_upload/packets_TIMESTAMP.csv.ready
```

Key Components

1. Configuration (Lines 24-31)

```
LOGS_DIR = os.path.join(SCRIPT_DIR, 'logs')
PENDING_DIR = os.path.join(LOGS_DIR, 'pending_upload')
```

```
SAVE_INTERVAL = 30  # Save CSV every 30 seconds
```

What you can change: - `SAVE_INTERVAL` - How often to save files (in seconds) - Lower = More frequent saves, more files, lower memory usage - Higher = Less frequent saves, fewer files, higher memory usage

---

2. Packet Feature Extraction (Lines 82-298)

Function: `packet_summary(pkt, interface)`

Purpose: Converts raw Scapy packet into dictionary with 30+ fields

Extracted Features:

Basic Fields (All Packets)

- `timestamp` - When packet was captured (Unix timestamp)
- `interface` - Network interface name (e.g., "Ethernet", "Wi-Fi")
- `length` - Packet size in bytes

IP Layer

- `src_ip` - Source IP address (IPv4 or IPv6)
- `dst_ip` - Destination IP address
- `protocol` - Protocol type (TCP, UDP, ICMP, ARP, etc.)

TCP Layer

- `src_port`, `dst_port` - Source/destination ports
- `tcp_flags` - TCP flags as string (e.g., "SYN,ACK")
- `tcp_syn`, `tcp_ack`, `tcp_fin`, `tcp_rst`, `tcp_psh` - Individual flags (boolean)
- `seq`, `ack` - Sequence and acknowledgment numbers

UDP Layer

- `src_port`, `dst_port` - Source/destination ports

ICMP Layer

- `icmp_type` - ICMP message type
- `icmp_code` - ICMP code

ARP Layer

- `arp_op` - Operation (1=request, 2=reply)
- `arp_psrc`, `arp_pdst` - Protocol (IP) addresses
- `arp_hwsrc`, `arp_hwdst` - Hardware (MAC) addresses

DNS Layer (UDP port 53)

- `dns_query` - Is this a DNS query? (boolean)
- `dns_response` - Is this a DNS response? (boolean)
- `dns_qname` - Queried domain name
- `dns_qtype` - Query type (1=A, 28=AAAA, etc.)
- `dns_answer_count` - Number of answers in response
- `dns_answer_size` - Total size of answer section

HTTP Layer (TCP ports 80, 8080, 8000, 3000)

- `http_method` - HTTP method (GET, POST, etc.)
- `http_path` - Request path
- `http_status_code` - Response status code
- `http_host` - Host header value

How it works:

```
if IP in pkt:                       # Check if IPv4 packet
    summary['src_ip'] = pkt[IP].src

    if TCP in pkt:                  # Check if TCP
        tcp = pkt[TCP]
        summary['src_port'] = tcp.sport
        summary['tcp_flags'] = decode_tcp_flags(tcp.flags.value)

        if Raw in pkt:              # Check for HTTP payload
            # Parse HTTP headers...
```

---

3. Atomic File Writing (Lines 38-63)

Function: `save_to_csv_atomic(packets, base_filename)`

Purpose: Save packets to CSV without risk of corruption

The Problem: If you write directly to a CSV file, another process might read it while it's incomplete:

```
# BAD: sender.py might read incomplete file!
with open('packets.csv', 'w') as f:
    writer.writerows(packets)  # Writing...
```

The Solution - Atomic Writes:

```
# Step 1: Write to temporary file
temp_file = base_filename + '.tmp'
with open(temp_file, 'w') as f:
    writer.writerows(packets)  # Write complete data
```

```
# Step 2: Atomic rename (instant, no partial states)
os.replace(temp_file, base_filename)  # This is atomic!

# Step 3: Create ready marker
Path(base_filename + '.ready').touch()  # Signal completion
```

Why this works: - `os.replace()` is an atomic operation (instant, no in-between state) - sender.py only processes files with `.ready` markers - No race conditions possible!

---

4. Packet Buffer Manager (Lines 316-361)

Class: `PacketBuffer`

Purpose: Manages in-memory packet storage and periodic saves

Attributes: - `buffer` - List of packet dictionaries - `save_interval` - How often to save (seconds) - `last_save_time` - Timestamp of last save - `lock` - Thread lock for safe concurrent access

Key Methods:

`add_packet(packet)`

```
def add_packet(self, packet):
    with self.lock:  # Thread-safe
        self.buffer.append(packet)
```

Called by packet capture threads to add new packets.

`save_buffer()`

```
def save_buffer(self):
    with self.lock:
        packets_to_save = self.buffer.copy()
        self.buffer.clear()  # Clear buffer

    # Save in background thread (non-blocking)
    threading.Thread(target=save_to_csv_atomic, ...).start()
```

Saves current buffer to file and clears it.

`periodic_save()`

```
def periodic_save(self):
    while running:
        time.sleep(1)  # Check every second
```

```
        if time_since_last_save >= self.save_interval:
            self.save_buffer()  # Auto-save!
```

Background thread that auto-saves every 30 seconds.

---

5. Multi-Interface Capture (Lines 363-387)

Function: `sniff_interface(interface, buffer)`

Purpose: Capture packets from a single network interface

```
def sniff_interface(interface, buffer):
    def handle_packet(pkt):
        summary = packet_summary(pkt, interface)
        buffer.add_packet(summary)

    while running:
        sniff(
            prn=handle_packet,    # Callback for each packet
            store=0,              # Don't store in memory (save RAM)
            iface=interface,      # Which interface to capture
            timeout=1             # Check 'running' flag every second
        )
```

Why `store=0`? - Scapy normally stores all packets in a list - `store=0` means: call callback, then discard packet - Saves memory on high-traffic networks

Why `timeout=1`? - Allows checking `running` flag every second - Enables graceful shutdown (Ctrl+C)

---

6. Main Execution (Lines 389-415)

Function: `main()`

Flow:

1. Create PacketBuffer
   ↓
2. Get all network interfaces (get_if_list())
   ↓
3. Start one thread per interface
   ↓
4. Wait for Ctrl+C
   ↓
5. Save final buffer
   ↓
6. Exit gracefully

Multi-threading:

```
threads = []
for iface in ['Ethernet', 'Wi-Fi', 'Loopback']:
    t = threading.Thread(
        target=sniff_interface,
        args=(iface, packet_buffer),
        daemon=True  # Die when main thread dies
    )
    t.start()
    threads.append(t)
```

Each interface gets its own capture thread, all writing to the same buffer.

Data Flow Example

Scenario: Capturing HTTP traffic

```
1. USER opens browser → google.com

2. NETWORK sends TCP SYN packet
   ↓
3. SCAPY captures packet
   ↓
4. packet_summary() extracts:
   {
     'timestamp': 1701234567.123,
     'interface': 'Wi-Fi',
     'src_ip': '192.168.1.100',
     'dst_ip': '142.250.185.46',
     'protocol': 'TCP',
     'src_port': 54321,
     'dst_port': 443,
     'tcp_flags': 'SYN',
     'tcp_syn': True,
     'length': 60,
     ...
   }
   ↓
5. PacketBuffer.add_packet() adds to buffer
   ↓
6. (After 30 seconds or at shutdown)
   ↓
7. save_to_csv_atomic() writes:
   - logs/pending_upload/packets_20251129_220000.csv.tmp
   - Rename to: packets_20251129_220000.csv
```

```
   - Create: packets_20251129_220000.csv.ready
   ↓
8. sender.py detects .ready file and uploads
```

Configuration Options

Change Save Interval

```
SAVE_INTERVAL = 60  # Save every 60 seconds instead of 30
```

Trade-off: Less frequent I/O, but higher memory usage

Capture Specific Interfaces Only

```
# In main():
interfaces = ['Ethernet']  # Only capture Ethernet
# interfaces = get_if_list()  # Comment out the original
```

Trade-off: Less CPU usage, but miss traffic on other interfaces

Add BPF Filter (Capture Specific Traffic Only)

```
# In sniff_interface():
sniff(
    prn=handle_packet,
    store=0,
    iface=interface,
    timeout=1,
    filter="tcp or udp"  # Only TCP/UDP, ignore ARP
)
```

Common filters: - `"tcp"` - Only TCP - `"port 80 or port 443"` - Only HTTP/HTTPS - `"not arp"` - Exclude ARP - `"host 192.168.1.1"` - Only traffic to/from specific IP

Monitoring & Debugging

Check if Running

```
ps aux | grep sniffer.py
```

View Live Output

```
python sniffer.py
# Output shows:
# - Interfaces detected
```

```
# - Auto-save notifications
# - File paths
```

Check Generated Files

```
ls -lh logs/pending_upload/
# Look for:
# - .csv files (actual data)
# - .csv.ready files (completion markers)
```

Count Packets in Buffer (Add Debug Logging)

```
# In periodic_save():
logger.info(f"Buffer size: {len(self.buffer)} packets")
```

Monitor Memory Usage

```
# Linux/Mac:
top -p $(pgrep -f sniffer.py)

# Windows:
# Task Manager → Details → python.exe
```

Troubleshooting

"Permission denied" Error

Cause: Packet capture requires admin/root privileges

Solution:

```
# Windows (run as Administrator):
python sniffer.py

# Linux/Mac:
sudo python sniffer.py
```

No Files Being Created

Check: 1. Is `PENDING_DIR` being created? `bash ls logs/pending_upload/`

2. Are packets being captured?
   - Add logging in `handle_packet()`:

```
def handle_packet(pkt):
    logger.info(f"Captured packet from {pkt.summary()}")
    # ...
```

3. Is 30 seconds passing?
   - Wait at least 30 seconds for first save

High Memory Usage

Cause: Buffer growing too large before saves

Solutions: 1. Reduce `SAVE_INTERVAL`: `python SAVE_INTERVAL = 15 # Save every 15 seconds`

2. Add BPF filter to capture less traffic:

```
filter="tcp or udp"  # Ignore ARP/broadcast
```

Missing Some Packet Fields

Cause: Not all packets have all fields (e.g., UDP packets don't have TCP flags)

This is normal: Fields are set to `None` if not applicable

```
timestamp,protocol,src_port,tcp_flags
1234567.123,UDP,53,None
1234567.456,TCP,80,SYN,ACK
```

Performance Tips

For High-Traffic Networks:

1. **Increase save interval** (less I/O):

   ```
   SAVE_INTERVAL = 60
   ```

2. **Use packet sampling** (capture 1 in N packets):

   ```
   packet_counter = 0

   def handle_packet(pkt):
       global packet_counter
       packet_counter += 1
       if packet_counter % 10 != 0:  # Only process 10% of packets
           return
       # ... rest of code
   ```

3. **Filter unnecessary protocols**:

   ```
   filter="tcp or udp"
   ```

Output File Format

File: `logs/pending_upload/packets_20251129_220000.csv`

Structure:

```
timestamp,interface,src_ip,dst_ip,protocol,length,src_port,dst_port,tcp_flags,...
1701234567.123,Wi-Fi,192.168.1.100,8.8.8.8,TCP,60,54321,443,SYN,...
1701234567.456,Wi-Fi,8.8.8.8,192.168.1.100,TCP,60,443,54321,SYN;ACK,...
```

Columns: 30+ fields (see packet_summary documentation above)

Summary

sniffer.py is designed to: Capture all network traffic safely
Extract rich features for ML analysis
Save data atomically (no corruption)
Run continuously without manual intervention
Handle multiple interfaces simultaneously
Use minimal memory (periodic flushes)
Enable graceful shutdown (Ctrl+C)

It does NOT: Upload to server (that's sender.py's job)
Aggregate data (that's aggregator.py's job)
Run ML predictions (that's aggregator.py's job)

Next step: Run `sender.py` to upload these CSV files to the server.