Network Intrusion Detection (IDS) System

What is an IDS?
IDS also known as Intrusion Detection Systems are security tools that monitor network traffic and systems for malicious activity which alerts administrators when detected.

However, there are many types of IDS:

NIDS - Network based IDS

This system monitors network traffic for suspicious activity

HIDS - Host based IDS

This system monitors system logs and file changes on individual hosts and is not directly deployed in the network

Signature based IDS

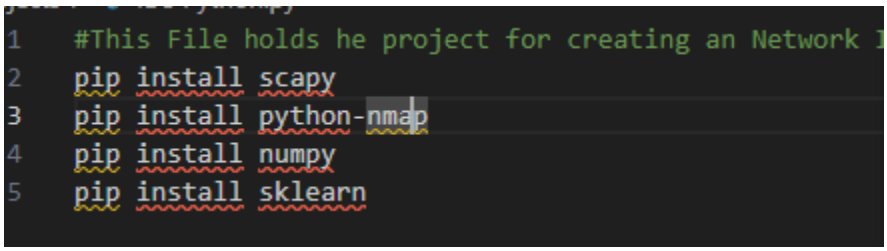
This system monitors network traffic and identifies malicious activity by comparing it against a database of known attack patterns.

Anomaly based IDS

This system identifies unusual behavior using prediction algorithms that are trained on previously seen attack patterns

Why do we need one?
There are many factors however and IDS is essential for monitoring network traffic to detect and alert on suspicious or malicious activities, helping to protect data and devices from cyberattacks. Think of it as a Car Scanner Tool. We use such devices to perform preventative maintenance on a nice new car. You would want to know what is going on with the car before something major happens.

How to:
Let's set up our environment first - We will first need a couple of open source libraries in python
- **scapy** (scapy is used for packet manipulation in computer networks. It allows users to create, send, capture and analyze network packets thus allowing for scanning, probing and network discovery)
- **python-nmap** (Allows users to interact with Nmap port scanner, enabling them to automate networking scanning and manipulate scanning results)
- **numpy** (In python, numpy is used for high level mathematical functions such as scientific computing)
- **sklearn** (This library is used for machine learning. It provides tools for various tasks like classification, regression, clustering, and dimensionality reduction, making it easier to build and evaluate predictive models



This project is built with help from freecodecamp and it will include:
Packet capture system
Traffic analysis module
A detection engine
An alert system

<u>Building the packet capture engine</u>:

```
pip install scapy
pip install python-nmap
pip install numpy
pip install sklearn

from scapy.all import sniff, IP, TCP
from collections import defaultdict
import threading
import queue


class PacketCapture:
    def __init__(self):
        self.packet_queue = queue.Queue()
        self.stop_capture = threading.Event()
    def packet_callback(self, packet):
        if IP in packet and TCP in packet:
            self.packet_queue.put(Packet)
    def start_capture(self, interface="eth0"):
        def capture_thread():
            sniff(iface=interface,
                  prn=self.packet_callback,
                  store=0,
                  stopfilter=lambda _: self.stop_capture.is_set())
        self.capture_thread = threading.Thread(target=capture_thread)
        self.capture_thread.start()
    def stop(self):
        self.stop_capture.set()
        self.capture_thread.join()
```

For this to work we will need to define `PacketCapture` class that will serve as the basis of our IDS. Booking at the image above, lets go over the functions and expand our understanding of why each line of code was added.

When defining the class PacketCapture we first defined `init` to initialize the class by creating a `queue.Queue` to store captured packets and a threading event to control when the capture should stop. For `self`, `self` refers to the instance of the class itself. It allows access to the attributes and methods of the class, ensuring that each object can maintain its own state and behavior. The `packet_callback` acts as a handler for each captured packet and checks if the packet contains both IP and TCP layers. If so, it will add it to the queue for further processing.

The `start_capture` beings capturing packets on a specified interface (much like Wireshark, a network traffic capture tool, we will default capture port to `eth0` to capture packets from the ethernet interface)

This will pass through to a separate thread to run Scapy's sniff function, which continuously monitors the interface for packets. The `stop_filter` parameter ensures the capture stops when the `stop_capture` event is triggered.

In addition, we have added `stop` which stops the capture by setting the `stop_capture` event and waits for the thread to finish execution. This will ensure the process terminates and allows for real-time packet capture without blocking the main thread.

Continue to next page

Building the Traffic Analysis Module:

```
class TrafficAnalyzer:
    def __init__(self):
        self.connections = defaultdict(list)
        self.flow_stats = defaultdict(lambda: {
            'packet_count': 0,
            'byte_count': 0,
            'start_time': None,
            'last_time': None
        })

    def analyze_packet(self, packet):
        if IP in packet and TCP in packet:
            ip_src = packet[IP].src
            ip_dst = packet[IP].dst
            port_src = packet[TCP].sport
            port_dst = packet[TCP].dport

            flow_key = (ip_src, ip_dst, port_src, port_dst)

            # Update flow statistics
            stats = self.flow_stats[flow_key]
            stats['packet_count'] += 1
            stats['byte_count'] += len(packet)
            current_time = packet.time

            if not stats['start_time']:
                stats['start_time'] = current_time
            stats['last_time'] = current_time

            return self.extract_features(packet, stats)

    def extract_features(self, packet, stats):
        return {
            'packet_size': len(packet),
            'flow_duration': stats['last_time'] - stats['start_time'],
            'packet_rate': stats['packet_count'] / (stats['last_time'] - stats['start_time']),
            'byte_rate': stats['byte_count'] / (stats['last_time'] - stats['start_time']),
            'tcp_flags': packet[TCP].flags,
            'window_size': packet[TCP].window
        }
```

In this section, from the image above, we must first define `TrafficAnalyzer` class to analyze the captured network traffic. Using this will allow us to track connection flows and calculate statistics for packets. Here we also used defaultdict data structure in python to manage connections and flows statistics by organizing data by unique flows.

The `__init__` initializes two attributes: `connections`, which stores the list of packets for each flow, and `flow_stats`, which stores statistical data for each flow. The data stored will include, but not limited to: packet count, byte count, start time and more.

The `analyze_packet` processes each packet by checking if a packet contains IP **AND** TCP layers. It will extract the source and destination IPs and port numbers forming a unique flowkey. This is to help identify the flow. It updates the statistics by incrementing the packet count, adding the packet's size to the byt count and updating the start and last time of the flow.

Finally, this section of code calls `extract_features` to calculate and return additional metrics.It will show detailed characteristics of the flow and the current packet. It will include packet size, flow duration, packet rate, byte rate, TCP flags and TCP window size.

Continue to next page

Building the Detection Engine

```python
class DetectionEngine:
    def __init__(self):
        self.anomaly_detector = IsolationForest(
            contamination=0.1,
            random_state=42
        )
        self.signature_rules = self.load_signature_rules()
        self.training_data = []

    def load_signature_rules(self):
        return {
            'syn_flood': {
                'condition': lambda features: (
                    features['tcp_flags'] == 2 and  # SYN flag
                    features['packet_rate'] > 100
                )
            },
            'port_scan': {
                'condition': lambda features: (
                    features['packet_size'] < 100 and
                    features['packet_rate'] > 50
                )
            }
        }

    def train_anomaly_detector(self, normal_traffic_data):
        self.anomaly_detector.fit(normal_traffic_data)

    def detect_threats(self, features):
        threats = []

        # Signature-based detection
        for rule_name, rule in self.signature_rules.items():
            if rule['condition'](features):
                threats.append({
                    'type': 'signature',
                    'rule': rule_name,
                    'confidence': 1.0
                })

        # Anomaly-based detection
        feature_vector = np.array([[
            features['packet_size'],
            features['packet_rate'],
            features['byte_rate']
        ]])

        anomaly_score = self.anomaly_detector.score_samples(feature_vector)[0]
        if anomaly_score < -0.5:  # Threshold for anomaly detection
            threats.append({
                'type': 'anomaly',
                'score': anomaly_score,
                'confidence': min(1.0, abs(anomaly_score))
            })

        return threats
```

From the image above, we are now defining the detection engine. Here this example will implement both signature and anomaly based detection methods. Because this is a hybrid based system we must use an

Isolation Forest model to detect anomalies. Using this will require pre-defined rules for identifying specific attack patterns.

Here, the `train_anamoly_detector` trains the Isolation Forest model using a dataset of normal traffic features. This enables the model to differentiate typical patterns from anomalies.

This code strip should be able to annotate the list of identified threats with either signature or anomaly.

Continue to next page

Building the Alert System

```python
import logging
import json
from datetime import datetime

class AlertSystem:
    def __init__(self, log_file="ids_alerts.log"):
        self.logger = logging.getLogger("IDS_Alerts")
        self.logger.setLevel(logging.INFO)

        handler = logging.FileHandler(log_file)
        formatter = logging.Formatter(
            '%(asctime)s - %(levelname)s - %(message)s'
        )
        handler.setFormatter(formatter)
        self.logger.addHandler(handler)

    def generate_alert(self, threat, packet_info):
        alert = {
            'timestamp': datetime.now().isoformat(),
            'threat_type': threat['type'],
            'source_ip': packet_info.get('source_ip'),
            'destination_ip': packet_info.get('destination_ip'),
            'confidence': threat.get('confidence', 0.0),
            'details': threat
        }

        self.logger.warning(json.dumps(alert))

        if threat['confidence'] > 0.8:
            self.logger.critical(
                f"High confidence threat detected: {json.dumps(alert)}"
            )
            # Implement additional notification methods here
            # (e.g., email, Slack, SIEM integration)
```

This part of the code is interesting. We continue to use `init` command which sets up a logger name `IDS_Alerts`. Additionally we are able to define INFO as a logging level to capture alert information. This will write logs to a specified file, `ids_alerts.log` by default. The `FileHandler` does exactly as it reads, it directs logs into the file while formatting the correspondent logs using `Formatter`. Moving down the code line, we see that the `generate_alert` command was used. This is responsible for creating structured alert entries. These alerts can consist of: Source and Destination IP addresses, type of threat detected, detection time, and any threat-specific details. These alerts are then logged as `WARNING` in json format.

Finally we have placed a code strip that will move any alert that has a confidence level higher than 0.8, it will be moved to `CRITICAL` level message which is also stored as a json file.

Continue to next page

Finalizing Code

In this portion we will be integrating all written code into a fully functional IDS.

```
class IntrusionDetectionSystem:
    def __init__(self, interface="eth0"):
        self.packet_capture = PacketCapture()
        self.traffic_analyzer = TrafficAnalyzer()
        self.detection_engine = DetectionEngine()
        self.alert_system = AlertSystem()

        self.interface = interface

    def start(self):
        print(f"Starting IDS on interface {self.interface}")
        self.packet_capture.start_capture(self.interface)

        while True:
            try:
                packet = self.packet_capture.packet_queue.get(timeout=1)
                features = self.traffic_analyzer.analyze_packet(packet)

                if features:
                    threats = self.detection_engine.detect_threats(features)

                    for threat in threats:
                        packet_info = {
                            'source_ip': packet[IP].src,
                            'destination_ip': packet[IP].dst,
                            'source_port': packet[TCP].sport,
                            'destination_port': packet[TCP].dport
                        }
                        self.alert_system.generate_alert(threat, packet_info)

            except queue.Empty:
                continue
            except KeyboardInterrupt:
                print("Stopping IDS...")
                self.packet_capture.stop()
                break

if __name__ == "__main__":
    ids = IntrusionDetectionSystem()
    ids.start()
```

Finalizing the IDS with the following code script will ensure everything written will be working homogeneously. Here, in the image above, IntrusionDetectionSystem class sets up the core components: PacketCapture, Traffic Analyzer, Detection Engine and Alert system into a full system. Remember in the beginning, self was mentioned to refer to the instance of the class itself. It allows access to the attributes and methods of the class, ensuring that each object can maintain its own state and behavior. It was also mentioned that the defaulted network interface to monitor was eth0.

The `start` function initiates the IDS and it will begin capturing packets. It will then extract and analyze each packet captured using `TrafficAnalyzer`. The output of this would be done by the `DetectionEngine` and if any threats emerge, the system will generate alerts from the `AlertSystem.`