Python Implementation of LDMC Algorithm for POX Multi-Controller

```
// LDMC Algorithm Implementation for POX Controller (Python)

from pox.core import core
import pox.openflow.libopenflow_01 as of
import pox.lib.packet as pkt
import threading
import time
import socket
import ssl
import json
import logging
import uuid

# Configuration (Replace with your actual values)
LC_ADDRESS = ('127.0.0.1', 8081)
PC_ADDRESSES = [('127.0.0.1', 8082), ('127.0.0.1', 8083)]
CERT_FILE = 'path/to/your/certificate.pem'
KEY_FILE = 'path/to/your/private_key.pem'
CA_CERT_FILE = 'path/to/ca_certificate.pem'
MONITORING_INTERVAL = 1  # Seconds
GLOBAL_MITIGATION_SEVERITY = 3
REGIONAL_MITIGATION_SEVERITY = 2

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

# Data Structures
class Alert:
    def __init__(self, src_ctrl_id, tgt_net_seg, atk_sev, ts, ext_feats):
        self.src_ctrl_id = src_ctrl_id
        self.tgt_net_seg = tgt_net_seg
        self.atk_sev = atk_sev
        self.ts = ts
        self.ext_feats = ext_feats

    def to_json(self):
        return json.dumps(self.__dict__)

    @classmethod
    def from_json(cls, json_str):
        data = json.loads(json_str)
        return cls(**data)

# Security Context Setup
context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
context.load_cert_chain(certfile=CERT_FILE, keyfile=KEY_FILE)
context.load_verify_locations(cafile=CA_CERT_FILE)
context.verify_mode = ssl.CERT_REQUIRED

# Communication Functions
def send_secure(sock, message):
    try:
        sock.sendall(message.encode())
    except Exception as e:
        logging.error(f"Error sending message: {e}")

def receive_secure(sock):
    try:
        data = sock.recv(4096).decode()
```

```
        return data
    except Exception as e:
        logging.error(f"Error receiving message: {e}")
        return None

# Placeholder Functions (Replace with actual implementations)
def anomaly_detected(switch):
    # Simulate anomaly detection logic
    return False

def detect_severity(anomaly):
    # Simulate severity detection logic
    return 1

def extract_features(switch):
    # Simulate feature extraction logic
    return []

def generate_global_mitigation_rules(alert):
    # Simulate global mitigation rule generation logic
    return ["global_rule"]

def generate_regional_mitigation_rules(alert):
    # Simulate regional mitigation rule generation logic
    return ["regional_rule"]

def generate_local_mitigation_rules(alert):
    # Simulate local mitigation rule generation logic
    return ["local_rule"]

def find_controllers(target_segment):
    # Simulate finding controllers for a segment
    return [('127.0.0.1', 8082)]

def find_controller(controller_id):
    # Simulate finding a controller by ID
    return ('127.0.0.1', 8082)

def apply_mitigation(controller_address, rules):
    # Simulate applying mitigation rules
    logging.info(f"Applying mitigation rules {rules} to {controller_address}")

def update_global_view(data):
    # Simulate updating global network view
    logging.info(f"Updating global view with data: {data}")

# Physical Controller (PC) Thread
def pc_thread(pc_address, managed_switches):
    pc_id = str(uuid.uuid4())
    try:
        with socket.create_connection(LC_ADDRESS) as sock:
            with context.wrap_socket(sock, server_hostname=LC_ADDRESS[0]) as ssock:
                while True:
                    for switch in managed_switches:
                        if anomaly_detected(switch):
                            severity = detect_severity(None) # replace None with anomaly
                            features = extract_features(switch)
                            alert = Alert(pc_id, "network_segment", severity, time.time(), features)
                            send_secure(ssock, alert.to_json())
                            send_secure(ssock, "local_data")  # Simulate sending local data
                            time.sleep(MONITORING_INTERVAL)
    except Exception as e:
```

```
        logging.error(f"PC thread error: {e}")

# Logical Controller (LC) Thread
def lc_thread():
    try:
        with socket.create_server(LC_ADDRESS) as server_socket:
            server_socket.listen(5)
            with context.wrap_socket(server_socket, server_side=True) as ssock:
                while True:
                    client_socket, addr = ssock.accept()
                    client_thread = threading.Thread(target=handle_client, args=(client_socket,))
                    client_thread.start()

    except Exception as e:
        logging.error(f"LC thread error: {e}")

def handle_client(client_socket):
    try:
        while True:
            data = receive_secure(client_socket)
            if data:
                try:
                    alert = Alert.from_json(data)
                    process_alert(alert)
                except json.JSONDecodeError:
                    update_global_view(data)
            else:
                break
    except Exception as e:
        logging.error(f"LC client handler error: {e}")

def process_alert(alert):
    if alert.atk_sev == GLOBAL_MITIGATION_SEVERITY:
        rules = generate_global_mitigation_rules(alert)
        controllers = find_controllers("Net")
        for controller in controllers:
            apply_mitigation(controller, rules)
    elif alert.atk_sev == REGIONAL_MITIGATION_SEVERITY:
        rules = generate_regional_mitigation_rules(alert)
        controllers = find_controllers(alert.tgt_net_seg)
        for controller in controllers:
            apply_mitigation(controller, rules)
    else:
        controller_address = find_controller(alert.src_ctrl_id)
        rules = generate_local_mitigation_rules(alert)
        apply_mitigation(controller_address, rules)

# POX Component
class LDMCComponent(object):
    def __init__(self):
        core.addListenerByName("UpEvent", self._handle_UpEvent)

    def _handle_UpEvent(self, event):
        logging.info("LDMC Component started")
        threading.Thread(target=lc_thread).start()
        for pc_address in PC_ADDRESSES:
            managed_switches = ["switch1", "switch2"]  # Replace with actual switches
            threading.Thread(target=pc_thread, args=(pc_address, managed_switches)).start()

def launch():
   core.registerNew(LDMCComponent)
```

Explanation of the POX Code:

1. **POX Component:**
   - The LDMCComponent class is a POX component that listens for the UpEvent to start the LDMC logic.
2. **Event Handling:**
   - The _handle_UpEvent() method is called when POX is ready, and it starts the LC and PC threads.
3. **Configuration, Data Structures, Security, Communication, and Placeholder Functions:**
   - These parts are identical to the code for other controllers and serve the same purpose.
4. **Physical Controller (PC) and Logical Controller (LC) Threads:**
   - These threads are also identical to the code for other controllers.
5. **POX Integration:**
   - The launch() function registers the LDMCComponent with the POX core.
   - You'll need to integrate the LDMC logic with POX's event handling and flow rule management mechanisms.
   - This involves using POX's APIs to listen for network events, retrieve switch information, and install flow rules.

Deployment Instructions (POX Controller):

1. **Prerequisites:**
   - Python: Ensure you have a compatible Python environment.
   - POX Controller: Download and install the POX controller.
   - SSL Certificates: Generate or obtain SSL certificates for secure communication.
   - Gson Library: Install the json python library.
2. **Code Integration (Continued):**
   - Create a new Python file (e.g., ldmc_pox.py) in your POX directory (typically in pox/ext).
   - Copy the provided code into it.
   - Update the configuration parameters with your environment's settings.
   - Place your certificate and truststore files in the specified paths.
3. **Run POX with the LDMC Component:**
   - Start POX from your terminal, including the LDMC component: ./pox.py ldmc_pox
   - If you have other components you want to run, include them as well: ./pox.py forwarding.l2_learning ldmc_pox
4. **Mininet-WiFi Deployment (Emulation):**
   - Mininet-WiFi Setup: Set up a Mininet-WiFi topology with OpenFlow-enabled and P4 switches (POX typically works with OpenFlow 1.0).
   - POX Integration: Configure the Mininet-WiFi switches to connect to the POX controller.
   - LDMC Deployment: Run POX with the ldmc_pox.py component as described above.
   - Testing: Simulate SYN flood attacks in Mininet-WiFi to test the LDMC framework's detection and mitigation capabilities.
5. **Real-World Deployment:**
   - Hardware Setup: Deploy P4-enabled switches and servers in your physical network.
   - POX Installation: Run the POX controller on a server.
   - LDMC Deployment: Run POX with the ldmc_pox.py component.
   - Network Configuration: Configure the P4 switches to connect to the POX controller.
   - Monitoring and Testing: Monitor the network for SYN flood attacks and test the LDMC framework's effectiveness.