Python Implementation of CWE Algorithm – (POX Controller)

```
# CWE Algorithm Implementation (Python)- (POX Controller)

from pox.core import core
import pox.openflow.libopenflow_01 as of
from pox.lib.packet import ethernet, ipv4, tcp
import weka.core.jvm as jvm
from weka.core.converters import ConverterUtils
from weka.core.instances import Instances
from weka.core.dense_instance import DenseInstance
from weka.classifiers.trees import J48, RandomForest
from weka.classifiers.lazy import IBk
from weka.classifiers.functions import SMO
from weka.classifiers.meta import AdaBoostM1
import threading
import time

log = core.getLogger()

class CWEApp(object):
    def __init__(self):
        core.openflow.addListenerByName("PacketIn", self._handle_packet_in)
        self.classifiers = {
            'KNN': IBk(),
            'DT': J48(),
            'RF': RandomForest(),
            'SVM': SMO(),
            'XGBoost': AdaBoostM1()
        }
        self.weights = {classifier: 1.0 for classifier in self.classifiers}
        self.threshold = 0.7
        self.update_interval = 60
        self.attributes = [
            'srcIP', 'dstIP', 'srcPort', 'dstPort', 'method',
            'payloadSize', 'errorCode', 'anomalyFlag', 'attack'
        ]
        self.training_data = Instances('MetaData', [
            (attr, 'NUMERIC' if attr != 'attack' else ['normal', 'attack'])
            for attr in self.attributes
        ], 0)
        self.training_data.class_index = len(self.attributes) - 1
        self.start_weight_updates()
        jvm.start()

    def start_weight_updates(self):
        def update_weights():
            while True:
                time.sleep(self.update_interval)
                self.update_weights_logic()

        thread = threading.Thread(target=update_weights)
        thread.daemon = True
        thread.start()

    def update_weights_logic(self):
        for classifier_name, classifier in self.classifiers.items():
            accuracy = self.calculate_classifier_accuracy(classifier_name)
            self.weights[classifier_name] *= (1 + accuracy)

    def calculate_classifier_accuracy(self, classifier_name):
```

```
        correct = 0
        total = self.training_data.num_instances
        if total == 0:
            return 0

        try:
            classifier = self.classifiers[classifier_name]
            for i in range(max(0, total - 10), total):
                prediction = classifier.classify_instance(self.training_data.instance(i))
                if prediction == self.training_data.instance(i).class_value():
                    correct += 1
        except Exception as e:
            log.error(f"Error calculating accuracy for {classifier_name}: {e}")

        return correct / min(10, total)

    def _handle_packet_in(self, event):
        packet = event.parsed
        if not packet.parsed:
            log.warning("Ignoring incomplete packet")
            return

        eth = packet.find('ethernet')
        if eth:
            if eth.type == ethernet.IP_TYPE:
                ipv4_pkt = packet.find('ipv4')
                if ipv4_pkt:
                    if ipv4_pkt.protocol == ipv4.TCP_PROTOCOL:
                        tcp_pkt = packet.find('tcp')
                        if tcp_pkt:
                            if tcp_pkt.dstport == 80 or tcp_pkt.dstport == 443:
                                payload = event.ofp.data[eth.hdr_len + ipv4_pkt.hdr_len +
tcp_pkt.hdr_len:]
                                self.process_metadata(payload)

    def process_metadata(self, metadata_bytes):
        metadata_str = metadata_bytes.decode('utf-8')
        metadata_values = metadata_str.split(',')

        if len(metadata_values) < 8:
            return

        try:
            metadata = [float(value) for value in metadata_values]
        except ValueError as e:
            log.error(f"Error parsing metadata: {e}")
            return

        instance = DenseInstance(len(self.attributes) - 1)
        for i, value in enumerate(metadata):
            instance.set_value(i, value)
        instance.dataset = self.training_data

        predictions = {}
        confidence_scores = {}
        for classifier_name, classifier in self.classifiers.items():
            try:
                prediction = classifier.distribution_for_instance(instance)[1]
                confidence = classifier.classify_instance(instance)
                predictions[classifier_name] = prediction
                confidence_scores[classifier_name] = confidence
            except Exception as e:
```

```
                log.error(f"Error with classifier {classifier_name}: {e}")

        weighted_score = self.calculate_weighted_score(predictions, confidence_scores)

        if weighted_score > self.threshold:
            self.send_alert("Coordinated attack detected", weighted_score)

        instance_with_class = DenseInstance(len(self.attributes))
        for i, value in enumerate(metadata):
            instance_with_class.set_value(i, value)
        instance_with_class.set_value(len(self.attributes) - 1, 'attack' if weighted_score >
self.threshold else 'normal')
        instance_with_class.dataset = self.training_data
        self.training_data.add(instance_with_class)

        for classifier in self.classifiers.values():
            classifier.build_classifier(self.training_data)

    def calculate_weighted_score(self, predictions, confidence_scores):
        weighted_score = 0.0
        total_weight = 0.0
        for classifier_name, prediction in predictions.items():
            weighted_score += prediction * self.weights[classifier_name] *
confidence_scores[classifier_name]
            total_weight += self.weights[classifier_name]
        return weighted_score / total_weight if total_weight > 0 else 0.0

    def send_alert(self, message, score):
        log.info(f"Alert: {message} (Score: {score})")
        # Implement logic to send alert to CRS module

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

Explanation of the Python Code:

1. **Imports:** Imports POX, packet, and Weka libraries.
2. **Class Definition:** CWEApp is a class that handles packet processing.
3. **Variables:** classifiers, weights, threshold, update_interval, training_data: CWE algorithm parameters.
4. **__init__() Method:**
    - Registers the _handle_packet_in method to handle PacketIn events.
    - Initializes classifiers (KNN, DT, RF, SVM, XGBoost) and sets initial weights.
    - Sets the attack detection threshold and weight update interval.
    - Defines the structure of metadata attributes and creates a Weka Instances object.
    - Starts a background thread for periodic weight updates.
    - Starts the JVM for Weka.
5. **start_weight_updates() Method:** Creates and starts a thread for periodic weight updates.
6. **update_weights_logic() Method:** Updates classifier weights based on their accuracy.
7. **calculate_classifier_accuracy() Method:** Calculates classifier accuracy by comparing predictions to training data.
8. **_handle_packet_in() Method:**
    - Handles packet-in events.
    - Parses Ethernet, IPv4, and TCP headers.
    - Calls process_metadata() for HTTP/HTTPS traffic (TCP ports 80 or 443).
9. **process_metadata() Method:**

- Extracts metadata from the packet payload.
- Creates a Weka DenseInstance from the metadata.
- Performs ensemble classification using the Weka classifiers.
- Calculates the weighted score based on classifier predictions and weights.
- Makes a decision based on the threshold and sends an alert if an attack is detected.
- Adds the processed instance to the training data and rebuilds the classifiers.

10. **calculate_weighted_score() Method:** Calculates the weighted average of classifier predictions.
11. **send_alert() Method:** Sends an alert message to the log (and can be modified to send to a CRS).
12. **launch() Function:** Registers the CWEApp with the POX core.

Deployment Instructions:

Emulated Environment (Mininet-WiFi):

1. **Install Prerequisites:**
   - Python 2.7 (POX is primarily designed for Python 2).
   - POX: Download POX from its repository (or clone it using Git).
   - Python Weka Wrapper: pip install python-weka-wrapper3.
   - Mininet-WiFi.
2. **Create POX Application:** Create a Python file (e.g., cwe_app.py) and paste the application code.
3. **Run POX:**
   - Navigate to the POX directory.
   - Start POX with your application: python pox.py cwe_app.
4. **Create Mininet-WiFi Topology:** Create a Mininet-WiFi topology with a P4 software switch.
5. **Configure P4 Switch:** Configure the P4 switch to send metadata packets to the POX controller.
6. **Generate Traffic:** Generate HTTP/HTTPS traffic using tools like hping3 or curl.
7. **Monitor POX Logs:** Monitor the POX console logs for alerts.

Real-World Environment:

1. **Install POX:** Install POX on a server.
2. **P4 Hardware:** Use a P4-programmable switch.
3. **Network Configuration:** Configure the network to forward traffic to the P4 switch.
4. **Run POX Application:** Run the POX application.
5. **Training:** Train the Weka classifiers with real network data.
6. **CRS Integration:** Implement send_alert() to send alerts to the CRS module.
7. **Monitoring:** Monitor POX logs and the CRS module for alerts.
8. **Performance and Security:** Adjust parameters, optimize classifiers, and secure the system.
9. **Maintenance:** Regularly update models and monitor the system