Python Implementation of CRS Algorithm

```
# CRS Algorithm Implementation (Python)

import threading
import time
import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
log = logging.getLogger(__name__)

class CRSApp:
    def __init__(self, p4_switch, web_firewall, cwe_module):
        self.p4_switch = p4_switch
        self.web_firewall = web_firewall
        self.cwe_module = cwe_module
        self.mitigation_strategies = {}  # target: strategy
        self.monitoring_thread = None
        self.running = False

    def start(self):
        self.running = True
        self.monitoring_thread = threading.Thread(target=self._monitor_attacks)
        self.monitoring_thread.daemon = True
        self.monitoring_thread.start()
        log.info("CRS Application started.")

    def stop(self):
        self.running = False
        if self.monitoring_thread:
            self.monitoring_thread.join()
        log.info("CRS Application stopped.")

    def _monitor_attacks(self):
        while self.running:
            signal = self.cwe_module.receive_signal()
            if signal:
                self._process_attack_signal(signal)

            self._monitor_existing_attacks()
            time.sleep(10)  # Monitor every 10 seconds

    def _process_attack_signal(self, signal):
        attack_type, target, confidence = signal
        severity = self._analyze_severity(attack_type, target, confidence)
        strategy = self._select_strategy(attack_type, severity)
        self._implement_mitigation(target, strategy)
        self.mitigation_strategies[target] = strategy

    def _analyze_severity(self, attack_type, target, confidence):
        if confidence > 0.8:
            return 2  # High severity
        elif confidence > 0.5:
            return 1  # Medium severity
        else:
            return 0  # Low severity

    def _select_strategy(self, attack_type, severity):
        if severity == 2:
            return "Blocking"
        elif severity == 1:
```

```
            return "Rate Limiting"
        else:
            return "Redirection"

    def _implement_mitigation(self, target, strategy):
        log.info(f"Implementing mitigation: {strategy} for {target}")
        if strategy == "Rate Limiting":
            self.p4_switch.rate_limit(target)
        elif strategy == "Blocking":
            self.p4_switch.block(target)
            self.web_firewall.block(target)
        elif strategy == "Redirection":
            self.p4_switch.redirect(target, "honeypot")  # Replace with actual honeypot address

    def _monitor_existing_attacks(self):
        for target, strategy in self.mitigation_strategies.items():
            if self._is_mitigation_ineffective(target, strategy):
                self._adjust_mitigation_strategy(target, strategy)

    def _is_mitigation_ineffective(self, target, strategy):
        # Implement logic to monitor mitigation effectiveness
        return False  # Placeholder

    def _adjust_mitigation_strategy(self, target, current_strategy):
        if current_strategy == "Rate Limiting":
            new_strategy = "Blocking"
        elif current_strategy == "Blocking":
            new_strategy = "Redirection"
        else:
            new_strategy = "Rate Limiting"
        log.info(f"Adjusting mitigation for {target} to {new_strategy}")
        self._implement_mitigation(target, new_strategy)
        self.mitigation_strategies[target] = new_strategy

# Interfaces (Replace with actual implementations)
class P4Switch:
    def rate_limit(self, target):
        log.info(f"P4 Switch: Rate limiting {target}")

    def block(self, target):
        log.info(f"P4 Switch: Blocking {target}")

    def redirect(self, target, honeypot):
        log.info(f"P4 Switch: Redirecting {target} to {honeypot}")

class WebFirewall:
    def block(self, target):
        log.info(f"Web Firewall: Blocking {target}")

class CWEModule:
    def receive_signal(self):
        # Simulate receiving signal (replace with actual logic)
        return None # or ('DDoS', '192.168.1.10', 0.9)

# Example usage
if __name__ == "__main__":
    p4_switch = P4Switch()
    web_firewall = WebFirewall()
    cwe_module = CWEModule()
    crs_app = CRSApp(p4_switch, web_firewall, cwe_module)
    crs_app.start()
    try:
```

```
        time.sleep(60) #run for 60 seconds.
    except KeyboardInterrupt:
        pass
    finally:
       crs_app.stop()
```

Explanation of the Code:

1. **Imports:** Imports threading, time, and logging libraries.
2. **Class Definition:** CRSApp is the main class.
3. **Variables:**
    - p4_switch, web_firewall, cwe_module: Interfaces to interact with network components.
    - mitigation_strategies: A dictionary to track mitigation strategies.
    - monitoring_thread: Thread for periodic monitoring.
    - running: Flag to control the monitoring loop.
4. **__init__() Method:** Initializes the application.
5. **start() Method:** Starts the monitoring thread.
6. **stop() Method:** Stops the application.
7. **_monitor_attacks() Method:**
    - Receives signals from the CWE module.
    - Processes signals and monitors existing mitigations.
8. **_process_attack_signal() Method:**
    - Analyzes severity and selects a strategy.
    - Implements the strategy.
9. **_analyze_severity() Method:** Determines attack severity.
10. **_select_strategy() Method:** Selects a mitigation strategy.
11. **_implement_mitigation() Method:** Executes mitigation actions.
12. **_monitor_existing_attacks() Method:** Checks mitigation effectiveness.
13. **_is_mitigation_ineffective() Method:** (Placeholder) Checks effectiveness.
14. **_adjust_mitigation_strategy() Method:** Adjusts mitigation.
15. **Interfaces:** Placeholder classes for P4 switch, web firewall, and CWE module.

Deployment Instructions:

Emulated Environment (Mininet-WiFi):

1. **Install Prerequisites:**
    - Python 3.
    - Mininet-WiFi.
2. **Create CRS Application:**
    - Create crs_app.py and paste the code.
    - Implement P4Switch, WebFirewall, and CWEModule to interact with Mininet-WiFi and your emulated P4 switch and web server.
3. **Run the Application:**
    - Run python crs_app.py.
4. **Generate Attack Traffic:**
    - Use tools in Mininet-WiFi to simulate attacks.
5. **Monitor Logs:**
    - Observe the application's logs for mitigation actions.

Real-World Environment:

1. **Install Python:**
    - Install Python 3 on a server.
2. **Create CRS Application:**
    - Create crs_app.py.
    - Implement interfaces to interact with your P4 switch, web firewall, and CWE module.
3. **Run the Application:** Run python crs_app.py.
4. **Test and Monitor:**
    - **Generate Test Traffic:** Begin by generating controlled test traffic that mimics various attack scenarios. This can involve using penetration testing tools or simulating attack patterns based on your threat model.
    - **Verify Signal Reception:** Ensure the CRS application is correctly receiving attack detection signals from the CWE module. Check application logs and any monitoring dashboards to confirm signal reception.
    - **Observe Mitigation Actions:** Closely monitor the behavior of your P4-programmable switch and web application firewall. Verify that the CRS application is successfully triggering the appropriate mitigation actions (rate limiting, blocking, redirection).
    - **Monitor Network Behavior:** Use network monitoring tools to observe traffic patterns and analyze the effectiveness of the mitigation strategies. Look for reductions in attack traffic, improved response times, and overall network stability.
    - **Verify Dynamic Adjustment:** Test the CRS application's ability to dynamically adjust mitigation strategies. For example, simulate a scenario where rate limiting is ineffective, and verify that the application escalates to blocking.
5. **Integrate:**
    - **SIEM Integration:** Integrate the CRS application with a Security Information and Event Management (SIEM) system. This allows you to centralize security alerts, correlate events, and gain a comprehensive view of your security posture. Implement mechanisms to forward CRS alerts to the SIEM, such as using syslog, API integrations, or message queues.
    - **Alerting and Escalation:** Implement robust alerting and escalation procedures. Configure the CRS application to send alerts to network security personnel via email, SMS, or other notification channels. Define clear escalation paths for different types of security incidents.
6. **Implement Security Measures:**
    - **Secure Communication:** Secure the communication channels between the CRS application and other network components (P4 switch, web firewall, CWE module). Use encryption protocols such as TLS/SSL to protect sensitive data in transit.
    - **Access Control:** Implement strong access control mechanisms to restrict access to the CRS application and its configuration. Use role-based access control (RBAC) to grant permissions based on user roles and responsibilities.
    - **Input Validation:** Implement input validation to prevent malicious input from being injected into the CRS application. Sanitize and validate all data received from external sources.
    - **Logging and Auditing:** Implement comprehensive logging and auditing to track all actions performed by the CRS application. This helps with incident response, forensic analysis, and compliance reporting.

7. **Performance Tuning and Optimization:**
    - **Resource Monitoring:** Monitor the CRS application's resource usage (CPU, memory, network). Identify any performance bottlenecks and optimize the application's code and configuration.
    - **Concurrency and Parallelism:** If necessary, implement concurrency and parallelism to improve the application's performance. Use techniques such as threading or asynchronous programming to handle multiple attack signals and mitigation actions simultaneously.
    - **Database Optimization:** If the CRS application uses a database, optimize database queries and indexing to improve performance.
8. **Maintenance:**
    - **Regular Updates:** Regularly update the CRS application and its dependencies to patch security vulnerabilities and improve performance.
    - **Testing and Validation:** Regularly test and validate the CRS application's functionality to ensure it is working correctly.
    - **Strategy Review:** Periodically review and update the mitigation strategies based on evolving threats and attack patterns.
    - **Performance Monitoring:** Continuously monitor the application's performance and resource usage.
    - **Log Analysis:** Regularly analyze the application's logs to identify any errors or anomalies.