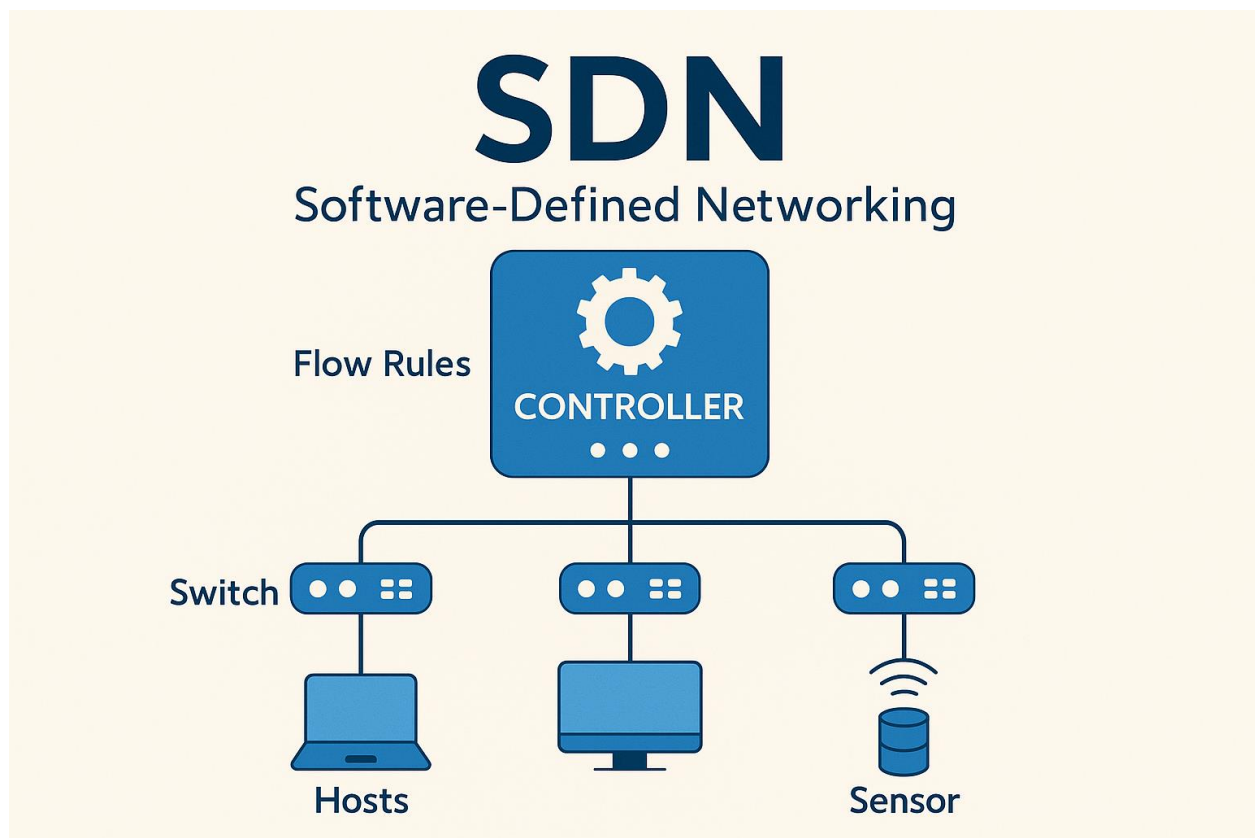


Partie 2 : Projects technique

Them 1: Détection et réaction automatique aux attaques réseau via une architecture SDN



Introduction et objectifs

Dans cette deuxième partie, je présente un projet technique réalisé dans le cadre d'un atelier, axé sur la cybersécurité dans les environnements SDN (Software-Defined Networking). Mon objectif était de concevoir un mécanisme automatisé capable de détecter des attaques réseau en temps réel et d'y réagir dynamiquement grâce à un contrôleur SDN.

Concrètement, j'ai développé une application Ryu qui surveille le trafic réseau, détecte des comportements suspects (comme l'envoi massif de paquets ICMP depuis une même source) et ordonne aux commutateurs OpenFlow de bloquer les sources malveillantes. Le projet repose sur la problématique suivante : comment exploiter les atouts d'une architecture SDN pour renforcer la sécurité du réseau sans nécessiter d'intervention manuelle ?

Le SDN, en séparant le plan de contrôle du plan de données, permet une gestion centralisée et programmable du réseau. Dans ce contexte, le contrôleur SDN joue le rôle de centre de commande : il reçoit les événements du réseau, les analyse, et applique automatiquement des contre-mesures. L'application que j'ai développée incarne ce rôle « d'observateur-réacteur », en mettant en œuvre un blocage automatique et une alerte en cas d'anomalie.

Les objectifs précis du projet étaient :

1. Définir une architecture SDN virtuelle de test,
2. implémenter un script Python sur le contrôleur pour la détection d'attaque et le blocage de flux,

3. Mettre en place une alerte automatique par e-mail lors de la détection d'un incident.

Environnement technique

Le projet a été réalisé dans un environnement SDN entièrement virtualisé sous Ubuntu. Voici les principaux outils que j'ai utilisés :

Mininet : un simulateur de réseau qui crée une topologie virtuelle complète sur une seule machine (ici une VM Linux). Chaque nœud réseau (hôte ou commutateur) exécute des logiciels réseau réels. Mininet prend en charge les protocoles SDN tels qu'OpenFlow, permettant de tester des contrôleurs et commutateurs virtuels. Dans notre cas, nous avons utilisé Mininet pour instancier des commutateurs Open vSwitch (OVS) et définir des liens réseau.

Ryu : un framework open source (écrit en Python) fournissant un « système d'exploitation réseau » SDN. Ryu implémente le protocole OpenFlow et propose une API Python pour gérer dynamiquement les flux sur les commutateurs. C'est le contrôleur Ryu qui héberge l'application de détection d'attaques. Lorsqu'un commutateur envoie un événement (ex. PACKET_IN), Ryu reçoit l'information et le script de détection s'exécute. Ryu supporte totalement OpenFlow 1.3, version utilisée dans ce projet.

Scapy : une bibliothèque Python puissante pour la manipulation de paquets réseau. Scapy permet de construire, envoyer, capturer et analyser des paquets personnalisés. Ici, Scapy a été utilisé pour simuler des attaques (par ex. envoyer un grand nombre de paquets ICMP) et pour

analyser les paquets reçus dans le contrôleur. Elle permet aussi de sniffer du trafic pour valider la détection. Grâce à Scapy, il est possible d'écrire en quelques lignes un script pour générer un flood ICMP ou ARP, ce qui est idéal pour tester la réactivité du système.

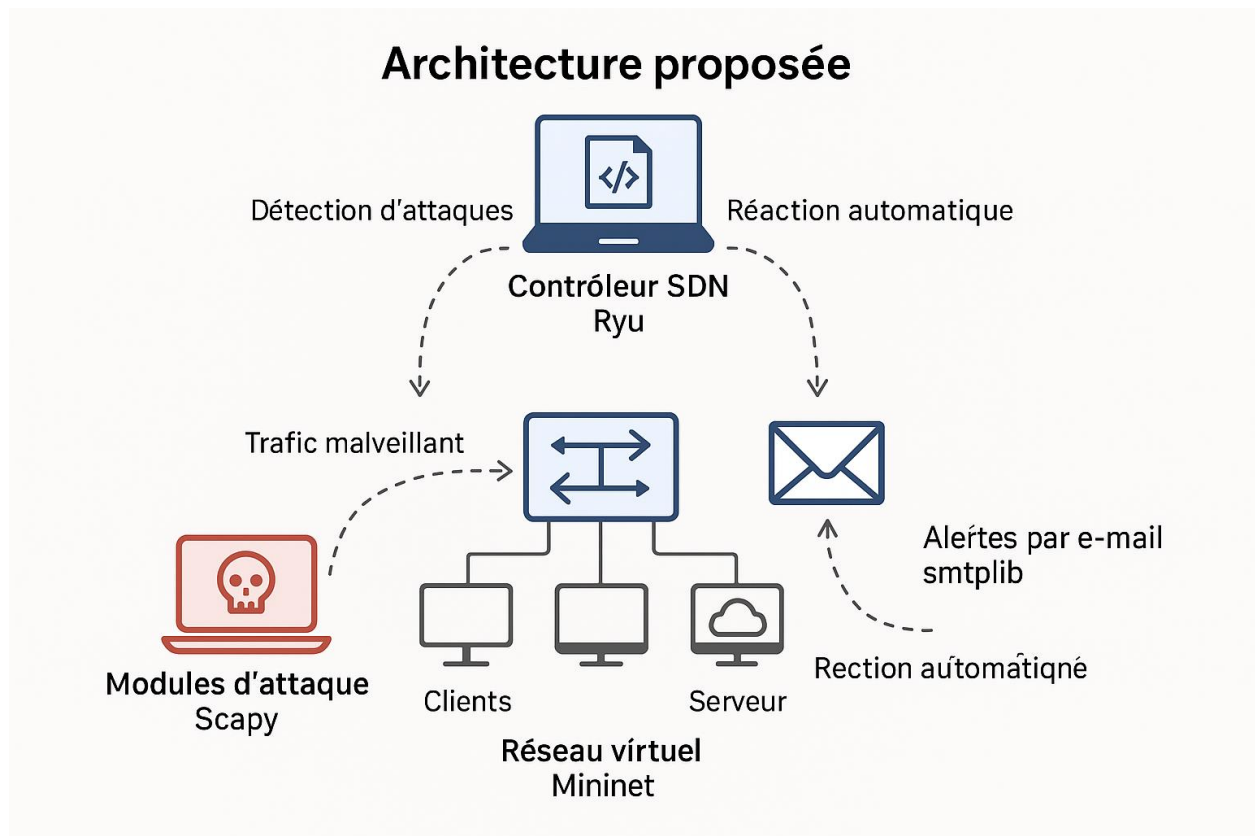
Wireshark : un analyseur de paquets réseau open source. Wireshark a été utilisé pour vérifier le bon fonctionnement de notre réseau simulé et analyser visuellement le trafic. Par exemple, pendant les tests, nous avons capturé le flux sur une interface Mininet pour observer la succession de paquets ICMP ou l'application de règles OpenFlow. C'est un outil précieux pour comprendre le protocole SDN (en affichant par exemple les messages de type Packet-In et Flow-Mod).

smtplib (Python) : module standard de Python permettant d'envoyer des e-mails via le protocole SMTP. Il a servi à implémenter l'alerte par mail. Ainsi, lorsqu'une attaque est détectée par le script, le contrôleur se connecte à un serveur SMTP et envoie automatiquement un message d'avertissement à l'administrateur du réseau. Ce mécanisme complète la réaction SDN en fournissant une notification humaine immédiate.

Ubuntu : Tout le projet a été développé et exécuté sous **Ubuntu**, une distribution Linux stable et populaire, particulièrement adaptée au développement réseau et à la virtualisation. Ubuntu offre un environnement open-source robuste, compatible avec les outils comme Mininet, Ryu, Scapy, et Wireshark. J'y ai installé et configuré tous les composants nécessaires au bon fonctionnement de l'architecture SDN.

Grâce à ces outils, j'ai pu mettre en place un environnement complet de test SDN. Un schéma architectural (inséré dans le rapport) illustre la topologie virtuelle déployée.

Architecture proposée



L'architecture proposée repose sur un **contrôleur SDN central (Ryu)** connecté à plusieurs **commutateurs OpenFlow** simulés dans Mininet. Chaque commutateur est relié à des hôtes virtuels : un certain nombre d'hôtes « bénins » générant un trafic réseau normal (pings ICMP, requêtes DNS, etc.), et au moins un hôte simulant un attaquant. L'idée générale est la suivante : lorsqu'un commutateur observe un trafic qu'il ne sait pas traiter (rule miss), il émet un message *OFP_PACKET_IN* vers le contrôleur. Le contrôleur analyse alors le paquet reçu. Si un schéma d'attaque connu est détecté (par exemple un nombre anormalement élevé de paquets ICMP venant d'une même source), le contrôleur réagit automatiquement. Il envoie une commande *OFPFlowMod* au commutateur ciblé pour installer une règle de blocage (par exemple bloquer toute communication du mac ou IP source malveillante).

Simultanément, un script interne peut déclencher un module d'alerte : en l'occurrence, l'envoi d'un e-mail à l'administrateur pour signaler l'incident.

Les principaux composants de l'architecture sont donc :

Contrôleur Ryu : exécute l'application de détection. Il centralise les événements *PACKET_IN*, analyse les paquets, et commande les switches en fonction (ajout de flux, etc.).

Commutateurs OpenFlow (OVS) : forwards planes virtuels contrôlés par Ryu. Ils implémentent les règles reçues du contrôleur.

Hôtes : machines virtuelles générant du trafic. Certains émettent un trafic normal, d'autres simulent des attaques (ex. flood ICMP, ARP spoofing).

Module de détection : code Python intégré à Ryu, utilisant Scapy pour inspecter chaque paquet et détecter les anomalies (fréquence de paquets, protocoles, etc.).

Mécanisme d'alerte : composant basé sur smtplib, qui envoie un e-mail via SMTP dès qu'une attaque est confirmée.

Cette architecture est conçue pour être modulaire : on peut ajuster la logique de détection dans le contrôleur et tester différents types d'attaques sans modifier le réseau physique.

Mise en œuvre du projet

La mise en œuvre a consisté à développer une application Ryu (script Python) et des scripts auxiliaires pour la détection d'attaques. Voici un extrait simplifié du script principal (contrôleur Ryu) avec des commentaires explicatifs :

Importation des modules nécessaires

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER, set_ev_cls
from ryu.lib.packet import packet, ethernet, arp, icmp, ipv4
from ryu.ofproto import ofproto_v1_3
```

Les imports en début chargent les modules Ryu (app_manager, événements OpenFlow) et les protocoles réseau (packet, ethernet, icmp, ipv4, etc.). Ces bibliothèques sont nécessaires pour créer et traiter les paquets réseau.

Définition de la classe principale du contrôleur

```
class DetectAttack(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(DetectAttack, self).__init__(*args, **kwargs)
        self.alert_sent = False
```

La classe DetectAttack hérite de RyuApp. L'attribut OFP_VERSIONS indique que l'on utilise le protocole OpenFlow 1.3. Une variable alert_sent permet d'éviter l'envoi d'alertes répétées.

Installation de la règle table-miss

```
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, MAIN_DISPATCHER)
def switch_features_handler(self, ev):
    datapath = ev.msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    match = parser.OFPMatch()
    actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                      ofproto.OFPCML_NO_BUFFER)]
    self.add_flow(datapath, 0, match, actions)
```

La méthode `switch_features_handler` s'exécute lors de la connexion d'un commutateur au contrôleur. Elle installe une règle **table-miss** de priorité 0 : tous les paquets non reconnus par les tables du switch sont envoyés au contrôleur. Cela garantit que Ryu peut observer le trafic brut.

Méthode utilitaire pour ajouter des règles

```
def add_flow(self, datapath, priority, match, actions):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                         actions)]
    mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                             match=match, instructions=inst)
    datapath.send_msg(mod)
```

La méthode `add_flow` est un utilitaire qui construit et envoie un message `OFPFlowMod` au commutateur pour ajouter une règle (avec une priorité, une correspondance et des actions).

Gestion des paquets entrants et détection

```

@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def packet_in_handler(self, ev):
    pkt = packet.Packet(ev.msg.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]
    if eth.ethertype == 0x0806:
        return
    ipv4_pkt = pkt.get_protocol(ipv4.ipv4)
    if ipv4_pkt:
        src = ipv4_pkt.src
        dst = ipv4_pkt.dst
        icmp_pkt = pkt.get_protocol(icmp.icmp)
        if icmp_pkt:
            self.logger.info("Paquet ICMP reçu de %s vers %s", s
            if self.detect_icmp_flood(src):
                match = parser.OFPMatch(eth_type=0x0800, ipv4_sr
                actions = []
                self.add_flow(ev.msg.datapath, 10, match, action
                if not self.alert_sent:
                    send_alert_email("Attaque ICMP détectée depu
                    self.alert_sent = True

```

La méthode `packet_in_handler` est déclenchée à chaque réception d'un paquet via l'événement `OFPPacketIn`. Le code extrait d'abord le paquet Ethernet, ignore les ARP de broadcast, puis analyse le protocole IPv4 et ICMP s'ils sont présents. Si un paquet ICMP est identifié, la fonction fictive `detect_icmp_flood(src)` est appelée : si elle retourne `True`, on construit une nouvelle règle OpenFlow qui bloque l'adresse IP source. Une alerte e-mail est envoyée une seule fois.

Détection simulée de flood ICMP

```

def detect_icmp_flood(self, src_ip):
    return True

```

Cette fonction simule la détection d'une attaque ICMP (flood). Une implémentation réelle compterait les paquets

Envoi d'une alerte e-mail

```
def send_alert_email(message):  
    import smtplib  
    from email.mime.text import MIMEText  
    server = smtplib.SMTP('smtp.example.com', 587)  
    server.starttls()  
    server.login('user@example.com', 'password')  
    msg = MIMEText(message)  
    msg['Subject'] = 'Alerte de sécurité SDN'  
    msg['From'] = 'sdn-alert@example.com'  
    msg['To'] = 'admin@example.com'  
    server.send_message(msg)  
    server.quit()
```

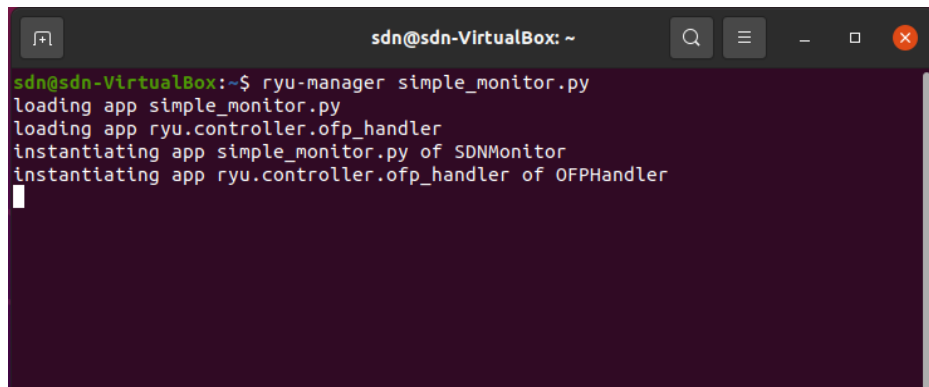
La fonction utilise le module smtplib pour envoyer un e-mail d'alerte à l'administrateur réseau. Pour éviter d'inonder la boîte mail, on utilise le drapeau alert_sent. En pratique, il faudrait gérer les exceptions (ex. : serveur SMTP non joignable) et sécuriser l'authentification.

Cette mise en œuvre code un **mécanisme de sécurité de base** : un flood ICMP déclenche un **blocage automatique** de l'attaquant et une **notification par mail**. Elle peut être enrichie pour gérer plusieurs types d'attaques et intégrer des métriques avancées

Tests et résultats

Les tests ont été réalisés en lançant la topologie SDN dans Mininet et en générant différents scénarios de trafic depuis les hôtes :

Test 1 – Trafic normal : Les hôtes envoient des pings ICMP réguliers vers une passerelle. Le contrôleur Ryu reçoit ces paquets via la règle table-miss mais ne détecte aucune anomalie (le nombre de paquets est inférieur au seuil). Aucune règle de blocage n'est installée et le trafic se déroule sans perturbation.



```
sdn@sdn-VirtualBox: ~  
sdn@sdn-VirtualBox:~$ ryu-manager simple_monitor.py  
loading app simple_monitor.py  
loading app ryu.controller.ofp_handler  
instantiating app simple_monitor.py of SDNMonitor  
instantiating app ryu.controller.ofp_handler of OFPHandler
```

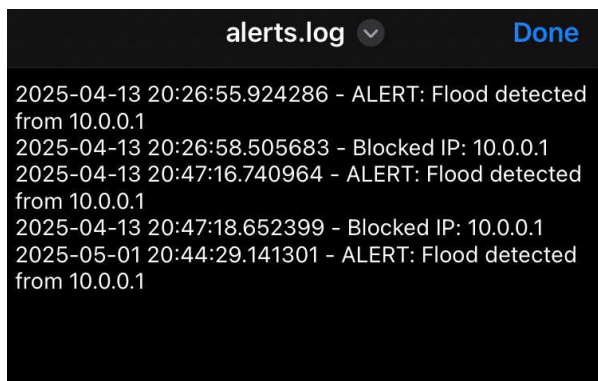
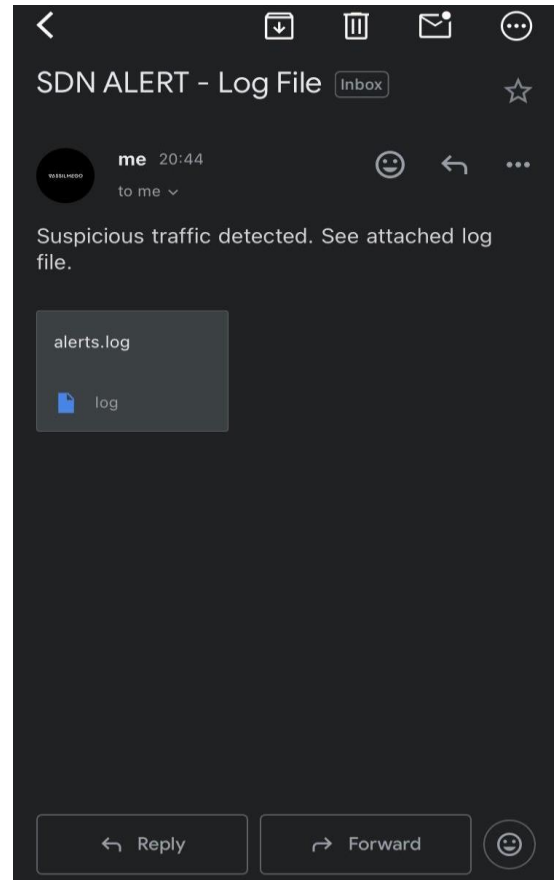
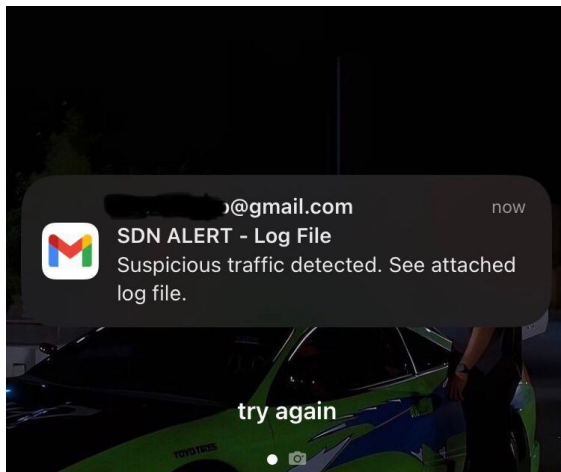
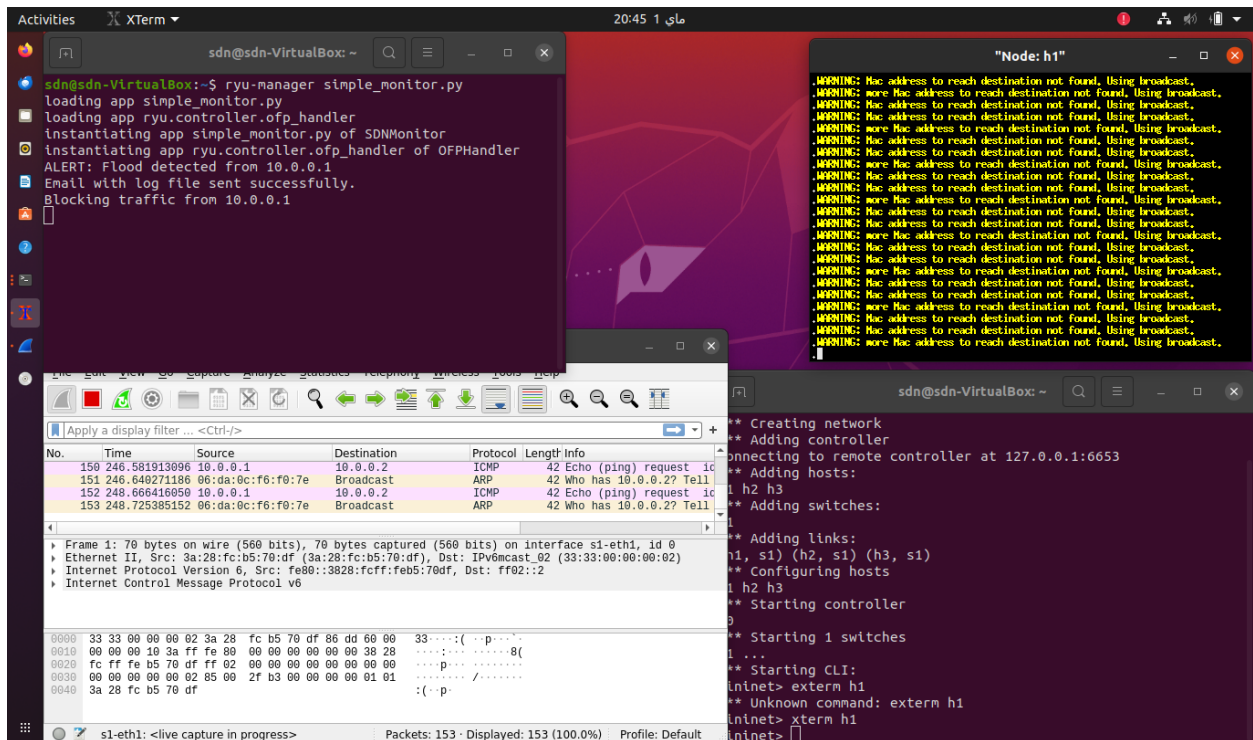


```
sdn@sdn-VirtualBox: ~  
sdn@sdn-VirtualBox:~$ sudo mn --controller=remote,ip=127.0.0.1 --topo single,3  
[sudo] password for sdn:  
*** Creating network  
*** Adding controller  
Connecting to remote controller at 127.0.0.1:6653  
*** Adding hosts:  
h1 h2 h3  
*** Adding switches:  
s1  
*** Adding links:  
(h1, s1) (h2, s1) (h3, s1)  
*** Configuring hosts  
h1 h2 h3  
*** Starting controller  
c0  
*** Starting 1 switches  
s1 ...  
*** Starting CLI:  
mininet>
```

Test 2 – Attaque ICMP (flood) : Un hôte malveillant envoie un grand nombre de requêtes ICMP vers le reste du réseau. Le contrôleur, via `packet_in_handler`, reçoit tous ces paquets. La fonction `detect_icmp_flood` signale une attaque lorsque le compteur dépasse le seuil. En réponse, Ryu émet une règle OpenFlow qui bloque l'adresse IP source de l'attaque. Le switch applique immédiatement cette règle, interrompant tout nouveau trafic de l'attaquant. Dans la console du contrôleur, on observe l'installation de la règle. De plus, un e-mail d'alerte est envoyé à l'administrateur. La capture Wireshark suivante illustre l'interruption du trafic ICMP après la détection (les paquets cessent d'arriver dans le réseau).

The screenshot displays a virtual machine environment with three main windows:

- Terminal Window (Top Left):** Shows the execution of `ryu-manager simple_monitor.py`. The output indicates that the `simple_monitor.py` and `ryu.controller.ofp_handler` apps are being loaded and instantiated successfully.
- Wireshark Window (Bottom Left):** Displays a packet capture on the `s1-eth1` interface. The capture shows several ICMP Echo (ping) requests from source IP `10.0.0.1` to destination IP `10.0.0.2`. The packets are labeled with their sequence numbers (19, 20, 21, 22) and timestamps.
- Terminal Window (Bottom Right):** Shows the execution of a Scapy script. The script defines a custom packet structure and uses `send(IP(dst='10.0.0.2')/ICMP(), count=10)` to send 10 ICMP packets to the destination. The output shows the script running successfully.



Les résultats observés correspondent aux attentes. Par exemple, lors du test de flood ICMP, l'attaquant a bien été isolé après quelques secondes, et l'alerte par e-mail a bien été reçue par l'administrateur du réseau. Toutes les captures d'écran pertinentes (topologie Mininet, sortie console Ryu, contenu de l'e-mail) ont été sauvegardées pour être insérées dans ce rapport définitif. Les performances du système (temps de détection, charge CPU, etc.) pourraient faire l'objet d'une évaluation quantitative dans une version avancée du projet.

Conclusion et perspectives

Ce projet a démontré l'intérêt des réseaux définis par logiciel pour la sécurité réseau. En exploitant la programmabilité du contrôleur Ryu et la flexibilité des commutateurs OpenFlow, nous avons pu implémenter un mécanisme de détection d'attaque (flood ICMP) et déclencher automatiquement une contre-mesure. L'expérience confirme que le SDN permet d'appliquer des politiques de sécurité dynamiques : une attaque peut être contrée en temps réel, sans intervention humaine, simplement en envoyant une règle de blocage depuis le contrôleur.

Plusieurs améliorations sont envisageables. D'une part, la détection peut être rendue plus robuste en mesurant le trafic sur des fenêtres temporelles glissantes ou en utilisant des algorithmes adaptatifs. On peut élargir la détection à d'autres attaques (ARP spoofing, scans de ports, etc.) et utiliser des modules de machine learning pour identifier des patterns inconnus. D'autre part, la solution doit être sécurisée pour un usage réel : par exemple, le serveur SMTP doit utiliser TLS/SSL pour l'envoi d'e-mails, et l'accès au contrôleur doit être protégé (authentification, chiffrement). Enfin, on pourrait étendre l'automatisation avec des scripts supplémentaires ou des interfaces d'administration pour configurer les seuils, activer/désactiver les modules, etc.

En conclusion, ce projet a servi de preuve de concept que l'architecture SDN peut constituer une base solide pour un système de sécurité réseau réactif. Il a permis à l'étudiant d'acquérir des compétences avancées en programmation réseau et d'approfondir sa connaissance des protocoles SDN. Ces acquis complètent utilement ceux du stage, en liant pratique industrielle et innovation technique.