Scapy

Introducción

Scapy (https://github.com/secdev/scapy ; https://scapy.net) es dos cosas: una utilidad y una librería (o "módulo"). Ambas están escritas en Python y ambas sirven para crear, manipular y enviar paquetes "a la carta". Para usar Scapy como módulo, este se deberá importar primero en los scripts desarrollados en lenguaje Python que necesiten de su funcionalidad (como cualquier otro módulo) y a continuación ya estarán disponibles los distintos métodos que proporciona. Pero por ahora solo vamos a usar Scapy como utilidad, ejecutándolo directamente desde un terminal (donde no será necesario hacerlo como *sudo* a no ser que se quiera hacer envíos).

Hay varios métodos para instalar Scapy: o bien descargándolo desde los repositorios oficiales de la distribución usada (en el caso de Fedora, por ejemplo, está disponible con el nombre de "python3-scapy"), o bien descargándolo del repositorio PyPi (que es el almacén oficial de los módulos Python, el cual ofrece gestión de dependencias y una versión actualizada de cada paquete) o bien descargando el código fuente directamente de GitHub y lidiar con sus dependencias. Emplearemos este último método porque Scapy no requiere de dependencias especiales (por lo que el proceso de instalación será sencillo) y lograremos disponer de la versión más actualizada posible de Scapy a día de hoy. Así pues, deberemos ejecutar:

git clone https://github.com/secdev/scapy
cd scapy && sudo python setup.py install

NOTA: En general, en qualsevol paquet Python que tingués dependències, abans d'executar la comanda *sudo python setup.py install* hauria calgut executar la comanda *pip -r requirements.txt* per instal.lar-les.

NOTA: En el caso de haber optado por usar el repositorio PyPi (el cual dispone de la última versión oficial del paquete), sería tan sencillo como haber ejecutado *sudo apt install python3-pip && pip install scapy*

A partir d'aquí, si executem la comanda *scapy* en un terminal, obtendremos un "shell" interno desde donde podremos escribir de forma interactiva los comandos propios de Scapy que deseemos (los cuales, de hecho, se corresponden con los objetos, propiedades y métodos de la librería). Para salir de él deberemos escribir *exit(), quit()* o *CTRL+D*

NOTA: Es posible que en arrancar el shell interno nos aparezcan varios mensajes de WARNING informando de que nos faltan paquetes instalados. Estos paquetes no son imprescindibles para la funcionalidad de Scapy y, de hecho, en este documento no los necesitaremos, pero si se desea que no aparezcan estos mensajes, se pueden instalar; en Ubuntu serían "tcpdump", "graphviz", "imagemagick", "python-matplotlib", "python-cryptography" y "python-pyx" i en Fedora "tcpdump", "graphviz", "imagemagick", "matplotlib", "python-cryptography", "PyX", "numpy" y "sox".

Tal como enseguida veremos, Scapy ofrece varios métodos/comandos para solo enviar, solo recibir, y enviar+recibir paquetes, tanto a nivel de capa 2 (enlace de datos) como de capa 3 (red). Además tiene varios métodos/comandos de alto nivel como *p0f()* -para el passive OS fingerprinting- o *arpcachepoison* -para el ARP Spoofing- (entre otras), las cuales pueden hacer lo mismo que la mayoría de las aplicaciones de seguridad. De hecho, con Scapy también se puede escanear hosts y puertos en la red ("à la" nmap), realizar funciones de sniffer ("à la" tcpdump/tshark), realizar gráficas estadísticas en 2D, 3D o pdf (mediante el empleo de librerías Python auxiliares), etc.

Una característica muy interesante del "shell" Scapy es que es, de hecho, un intérprete Python, por lo que además podremos escribir completos scripts en este lenguaje siguiendo sus reglas sintácticas y pudiendo emplear todas sus construcciones y palabras clave (*if, while*, variables, etc). En este sentido, hay algunas reglas básicas en Python que deberías saber para manejarte mejor en Scapy:

*Los bloques no van entre llaves: se distinguen por la indentación apropiada; 4 espacios y líneas vacías.
*No se necesita punto-y-coma como separador de sentencias (pero se puede usar si quieres poner varias sentencias en la misma línea)
*No se necesitan paréntesis en las sentencias condicionales como *if* o *while*, las sentencias condicionales acaban con ':' después de poner la expresión

Comandos básicos

Algunos comandos básicos de Scapy son...:

quit() Sale de Scapy

lsc() Lista todos los comandos Scapy y una breve descripción de cada uno

help(comando) Muestra una ayuda más extensa sobre el comando concreto indicado

ls() Lista los protocolos que Scapy es capaz de manejar

conf Muestra la configuración Scapy en forma de parejas variable<->valor.
Para ver el valor de una variable, se puede hacer *print(conf.nombrevariable)*
Para cambiarlo, *conf.nombrevariable=nuevovalor*
(o *conf.nombrevariable="nuevovalor"* si el valor es una cadena y no un número)
Por ejemplo, la variable "*iface*" indica la tarjeta de red sobre la que Scapy trabajará

...pero sobre todo...:

Ether(...) Construye una cabecera Ethernet (indicando entre paréntesis los valores deseados para los campos indicados). Podría servir para generar un paquete ARP, por ejemplo, si se indica el payload adecuado (con los campos pertinentes) mediante el comando *ARP(...)*

IP(...) Construye una cabecera paquete IP (indicando entre paréntesis los valores deseados para los campos indicados). Puede servir para generar un paquete ICMP si se indica el payload adecuado (con los campos pertinentes) mediante el comando *ICMP(...)*, o bien, para incluir cabeceras de niveles superiores.

TCP(...) Construye una cabecera TCP (indicando entre paréntesis los valores deseados para los campos indicados)

UDP(...) Construye una cabecera UDP (indicando entre paréntesis los valores deseados para los campos indicados)

Raw(...) Añade "a mano" un "payload" de nivel de aplicación al paquete. Normalmente este payload representa el contenido generado en el nivel de aplicación pero puede ser tan sencillo como una cadena indicada como parámetro manualmente (o bien mediante una orden como *RandString(size=nºcaracteres))* o similares)

...y muchos otros comandos relacionados cada uno con un determinado protocolo de red (de todos los niveles posibles, incluyendo algunos del nivel de aplicación que estén basados en UDP), los cuales permiten manipular fácilmente los campos concretos de cada uno de ellos: *DNS(...)*, *DHCP(...)* o *NTP(...)*

Los comandos anteriores se pueden usar junto con *ls()* para observar los diferentes campos de las cabeceras correspondientes al protocolo indicado, su tipo y el valor por defecto que tendrían en un paquete creado de cero por Scapy, así, por ejemplo, si hacemos ...:

ls(Ether) Se puede ver que sus campos más relevantes son *dst* (dirección MAC de destino) y *src* (dirección MAC de origen). El destino puede ser, si se desea, una MAC aleatoria generada mediante el comando *RandMAC()*

ls(ARP) Se puede ver que sus campos más relevantes son *op* (cuyo valor indica si es una petición -1- o una respuesta -2-) y *hwsrc*, *hwdst*, *psrc*, y *pdst* ,los cuales indican las direcciones MAC e IP de origen y destino, respectivamente.

ls(IP) Se puede ver que sus campos más relevantes son *dst* (dirección IP/nombre de destino), *src* (dirección IP/nombre de origen) y *ttl* (Time To Live). El destino puede ser, si se desea, la IP/máscara de una red o una IP aleatoria generada mediante el comando *RandIP()*

ls(ICMP) Se puede ver que su campo más relevante es *type*

ls(TCP) Se puede ver que sus campos más relevantes son *dport* (puerto de destino), *sport* (puerto de origen) y *flags* (SYN, ACK, RST, FIN…). Los puertos pueden ser, si se desea, un número aleatorio generado mediante el comando *RandShort()* .
Los flags se indicarán mediante su inicial ("S","A","R","F"…) y si hay más de uno, consecutivamente (por ejemplo, un paquete SYN/ACK se indicará así: "SA")

ls(UDP) Se puede ver que sus campos más relevantes son *dport* (puerto de destino) y *sport* (puerto de origen)

Y, obviamente *ls(DNS)*, *ls(DHCP)*, etc

Creación de paquetes

A partir de aquí, para crear un paquete básico (en este caso, un paquete IP con una dirección específica de destino -al no indicar ninguna cabecera Ethernet concreta, Scapy la rellenará con sus valores por defecto-), se puede hacer por ejemplo: `paqueteIP=IP(dst="www.google.com")`

Comandos interesantes para inspeccionar el contenido de un paquete son:

nombrepaquete.show() Muestra los valores actuales de los diferentes campos de las cabeceras del paquete asociado. También se puede indicar solo una determinada cabecera si nos interesa únicamente conocer los campos de ella, así por ejemplo: *nombrepaquete[TCP].show()*

ls(nombrePaquete) Muestra además de los valores actuales de los diferentes campos de las cabeceras del paquete asociado, su tipo y su valor por defecto. Igualmente se puede filtrar para una cabecera solo, así por ejemplo: *ls(nombrePaquete[IP])*

hexdump(nombrePaquete) Muestra el contenido del paquete (cabeceras y datos) de forma directa en hexadecimal.

NOTA: Scapy elige como valores por defecto los más "sensatos". Por ejemplo, la IP de origen del paquete la obtiene a partir de la IP de destino indicada y la tabla de rutas del sistema, la MAC de origen la elige al seleccionar la tarjeta de red de salida, los campos "type" Ethernet e IP se definen según el contenido de capas superiores, etc. Otros valores por defecto son: puerto TCP de origen= 20, puerto TCP de destino = 80, puertos UDP de origen y destino = 53, tipo de mensajes ICMP = "echo request".

Se pueden ir encapsulando cabeceras de niveles superiores dentro de otras de niveles inferiores con el símbolo "/", así:

```
# Partimos del mismo paquete que generamos en el ejemplo anterior
paqueteIP=IP(dst="www.google.com")
paqueteIP.show()
# Una vez generado el paquete aún podemos modificar los valores de los campos de su cabecera.
# También se pueden borrar con del(paquete.campo), así, por ejemplo: del(paqueteIP.ttl)
paqueteIP.ttl=23
paqueteTCP=TCP(dport=8080)
# La barra / sirve para la composición entre capas al construir nuestro paquete; en este caso IP/TCP.
paquete=paqueteIP/paqueteTCP
# Si el paquete dispone de varias cabeceras superpuestos, para modificar el valor de algún campo de una cabecera
#concreta simplemente bastará con indicar entre corchetes de qué cabecera estamos hablando, así:
paquete[IP].ttl=24
paquete.show()
```

En el ejemplo anterior hemos construido un paquete, capa por capa y los datos de campos no introducidos se rellenan por defecto. Podemos seguir manipulando añadiendo, por ejemplo, los flags TCP SYN+FIN….: `paquete=paqueteIP/TCP(flags="SF")`

Incluso podemos agregar datos al paquete ya construido (lo que se llama el "payload") añadiendo un nivel más en lo que sería el nivel de aplicación, así (suponiendo que seguimos con la contrucción de líneas anteriores): `paquete=paquete/Raw("Hola,esto son los datos encapsulados en el paquete")`

Tampoco es necesario nombrar cada nivel por separado: se pueden añadir directamente con los comandos *IP(), TCP()*, etc. Por ejemplo, para realizar de una sola vez todo lo hasta ahora visto se podría hacer: `paquete=IP(dst="www.google.com",ttl=23)/TCP(dport=8080,flags="SF")/Raw("Hola...")`

Envío de paquetes

Scapy puede enviar los paquetes contruidos por nosotros con el comando *send(nombrePaquete)*.

También se puede enviar el mismo paquete múltiples veces (por ejemplo,10, así: *send(paquete*10))*

O una lista de paquetes diferentes (sobre las listas hablaremos en próximos párrafos).

Parámetros interesantes del comando *send()* son:

inter=nº (indica el n.º de segundos de espera entre el envío de un paquete y el siguiente)
loop=1 (indica que se van a enviar los paquetes especificados e forma infinita)
verbose=nº (indica el nivel de verbosidad del comando; un valor 0 no muestra nada)

Por ejemplo: *send(paquete,inter=3,loop=1)* enviará el mismo paquete indefinidamente esperando 3 segundos entre envío y envío.

Para que Scapy reciba la respuesta (si es que hay) de un paquete enviado por él, debemos utilizar otros comandos. El más sencillo es *sr1(nombrePaquete)*, el cual captura un solo paquete de respuesta (el primero que se reciba: "nombrepaquete" puede ser un solo paquete o una lista). Por ejemplo, el siguiente ejemplo genera y envía un paquete "ping" (sin "payload") a Google, y obtiene -solo- el primer "pong".

```
paquete=IP(dst="www.google.es")/ICMP(type=8)
paqueteRecibido=sr1(paquete)
paqueteRecibido.show()
```

Lo mismo de podría haber hecho paso a paso:

```
ip = IP()
ip.dst="www.google.es"
icmp = ICMP()
icmp.type=8
paquete=ip/icmp
paqueteRecibido= sr1(paquete)
paqueteRecibido.show()
```

O todo en un solo paso: `sr1(IP(dst="www.google.es")/ICMP(type=8)).show()`

Otro ejemplo es esta petición DNS recursiva (rd=1) cuya respuesta deseada está en el campo "rdata":
`sr1(IP(dst="8.8.8.8")/UDP()/DNS(rd=1,qd=DNSQR(qname="www.slashdot.org",qtype="A"))).show()`

NOTA: A partir de aquí se pueden hacer varios tests: probar de falsificar la dirección IP de origen (se verá que no se recibe echo reply del destino), probar de poner un ttl diferente de por defecto, probar de escribir un type diferente para generar un paquete ICMP de otro tipo, etc

Parámetros interesantes del comando *sr1()* son, por ejemplo:

timeout=nº (indica el nº de segundos que el comando esperará a recibir una respuesta desde que se envió el paquete -o el último de ellos si se envían varios-; por defecto se espera un tiempo infinito por lo que el usuario debe pulsar CTRL+C para interrumpir la escucha)
retry=-nº (indica cuántos paquetes sin respuesta se deben detectar como máximo para dejarlo estar -notar que el valor indicado ha de ser negativo-; este parámetro se puede combinar con *timeout* o no)
iface=nombreTarjeta (indica la tarjeta de red por donde se escuchará la llegada de las respuestas -en el caso de querer que sea una diferente de la configurada por defecto mediante *conf.iface*-)
verbose=nº (ya conocido)
inter=nº (ya conocido)
multi=true (para aceptar múltiples respuestas de un mismo paquete enviado -por ejemplo, para obtener varias ofertas de servidores DHCP en una solicitud "discover"-)
filter="…" (indica un filtro a aplicar a las respuestas y así solo capturar aquellas que concuerden con él; la sintaxis de los filtros es igual a los "filtros de captura" de Wireshark, también llamados filtros BPF).

Otro comando útil para enviar paquetes y recibir, en este caso, todas las eventuales respuestas posibles, es *sr(nombrePaquete)* Este comando retorna una tupla[NOTA] de dos elementos: el primero es una lista[NOTA] de tuplas, cada una de también dos elementos: cada paquete enviado con respuesta y su correspondiente respuesta; el segundo es una lista de los paquetes enviados sin respuesta. Por otro lado, acepta los mismos parámetros que *sr1()*.

[NOTA] Para entender qué es una "tupla" y una "lista" en Python, hay que conocer cómo funcionan las agrupaciones de datos en este lenguaje:

* Una **tupla** es un conjunto de valores separados por comas y, opcionalmente, rodeado por paréntesis. Sería similar a los arrays conocidos en otros lenguajes salvo por dos diferencias: los elementos de una tupla pueden ser de diferentes tipos (enteros, decimales, cadenas, etc) y las tuplas por definición son inmutables (es decir, de solo lectura). Un ejemplo de definición de tupla: *mitupla=(1,"hola",3.5)*

*Una **lista** es un conjunto de valores separados por comas rodeado por corchetes. Es similar a una tupla en el sentido de que sus elementos también pueden ser de diferentes tipos pero, por el contrario, las listas por definición son mutables (es decir, sus elementos pueden variar tanto en número como de valor). Un ejemplo de definición de lista: *milista=[1,"hola",3.5]* . Por otro lado, es bueno saber que las listas como tales en Python implementan métodos propios (para, por ejemplo, añadir fácilmente un nuevo elemento, contarlos, etc); más información en http://enricbaltasar.com/python-summary-methods-lists

También es interesante conocer el concepto "diccionario":

*Un **diccionario** es un conjunto mutable de parejas clave<->valor separadas por comas y rodeado de llaves, escribiéndose las parejas mediante la sintaxis *nombreClave:valor* (donde "nombreClave" ha de ser única). En otros lenguajes reciben el nombre de "hashes" o "maps". Un ejemplo de definición de diccionario: *midict={1:"hola","adios":2}*

Para acceder a un elemento concreto de una tupla o lista (ya sea para leer su contenido o para modificarlo) se ha de escribir entre corchetes el número de índice de ese elemento (el primero es el número 0 siempre), así por ejemplo: *milista[3]* (en este caso se accedería al cuarto elemento). En el caso de los diccionarios la notación es similar pero en vez de indicar el número de índice, se ha de escribir el nombre de la clave correspondiente al elemento deseado.

Para recorrer todos los elementos de una tupla o lista se puede utilizar el bucle *for i in coleccion :* donde "i" es una variable que representará en cada iteración del bucle el índice del elemento actual de la tupla o lista recorrida (empezando por el primero hasta llegar al último) y "coleccion" es el nombre de esa tupla o lista. Para recorrer un diccionario podemos usar en cambio *for i in midict.keys()* -para recorrer las claves, lo que de forma conceptual sería similar a recorrer una tupla o lista- pero también *for i midict.values()* -para recorrer directamente los valores-, entre otras muchas opciones.

Para saber más sobre las diferentes agrupaciones de datos reconocidas por Python y sus diferentes características y usos, recomiendo leer la documentación oficial disponible en https://docs.python.org/3/tutorial/datastructures.html o, de forma más informal, leer el siguiente artículo: http://thomas-cokelaer.info/tutorials/python/data_structures.html

Por ejemplo, los siguientes comandos guardan en la lista de tuplas "ans" los paquetes enviados (elementos [0] de cada tupla) junto con sus respuestas (elementos [1] de cada tupla) y en la lista "unans" los paquetes no respondidos (el nombre de "ans" y "unans" es completamente arbitrario). Además, podemos hacer uso de el método *nombrelista.summary()* propio de las listas, el cual sirve para ver un resumen de los datos más relevantes de los distintos elementos de la lista correspondiente (IPs, puertos, flags). Notar que la primera línia del siguiente ejemplo envia tres paquets, cadascun dirigit a un port de destí diferent: 21, 22 i 23 (en realitat internament executa tres vegades la comanda *sr()* , una per cada port diferent indicat); això vol dir que, en principi, es rebran tres respostes i, per tant, *ans* serà una llista de tres tuples

```
ans,unans=sr(IP(dst="192.168.1.1")/TCP(dport=[21,22,23]),timeout=2)
ans.summary()
unans.summary()
```

Alternativamente podemos usar el símbolo "_" para tener el resultado de ejecutar *sr()* separado en la lista de paquetes respondidos y sus respuestas (elemento [0] de la tupla devuelta por *sr()*) y la lista de paquetes no respondidos (elemento [1] de esa tupla). El símbolo "_" significa "el resultado obtenido del comando anterior". Es decir, si solo queremos ver el resumen de los paquetes respondidos podríamos hacer:

```
sr(IP(dst="192.168.1.1")/TCP(dport=[21,22,23]),timeout=2)
ans=_[0]
ans.summary()
_[1].summary()
```

NOTA: Si en comptes d'indicar múltiples valors concrets com s'ha vist a l'exemple anterior es vol indicar un rang, es pot fer servir l'ordre de Python *range()*, la qual té com a primer paràmetre el primer valor del rang desitjat i com a segon paràmetre l´últim valor del rang desitjat + 1. Així, per exemple, *millista=range(1,4)* generarà una llista amb el contingut [1,2,3] .

También existe el comando *srloop(nombrePaquete,count=nº)*, el cual es muy similar a *sr()* pero repitiendo el envío del paquete (o lista de paquetes) un número determinado de veces indicado por el parámetro *count* y, además, (si *verbose* no es igual a 0), mostrando por pantalla una respuesta (la primera recibida) para cada vez. Parámetros extra interesantes de este comando son: *timeout=nº* (ya conocido), *inter=nº* (ya conocido) y *verbose=nº* (ya conocido).

Scapy puede realizar muchas más cosas. Para profundizar en su estudio recomiendo leer la siguiente serie de artículos:

https://seguridadyredes.wordpress.com/2009/12/03/scapy-manipulacion-avanzada-e-interactiva-de-paquetes-parte-1/
https://seguridadyredes.wordpress.com/2009/12/10/scapy-manipulacion-avanzada-e-interactiva-de-paquetes-parte-2/
https://seguridadyredes.wordpress.com/2010/01/18/scapy-manipulacion-avanzada-e-interactiva-de-paquetes-parte-3/
https://seguridadyredes.wordpress.com/2010/02/09/scapy-manipulacion-avanzada-e-interactiva-de-paquetes-parte-4/

EJERCICIOS:

1.-a) Construye y envía con *sr1()* un paquete ICMP de tipo "echo-request" al ordenador www.hola.com y justifica el valor del campo "type" de la cabecera ICMP y el del campo "src" de la cabecera IP del paquete recibido.

b) Construye y envía con *sr1()* un paquete TCP con puerto de destino 80 al ordenador www.hola.com y justifica el valor de los campos "dport" (ojo con este!) y "flags" de la cabecera TCP del paquete recibido

c) Executa les següents comandes i, a partir del resultat final que obtens, dedueix quins ports té oberts la màquina objectiu. **Pista:** fixa't en les flags TCP de les respostes (si són "SA" -de SYN i ACK- vol dir que el port remot vol seguir la connexió "3-way shake" començada per nosaltres i, per tant, està obert; si són"RA" -de RST i ACK- vol dir que el port remot està tancat i si no es rep cap resposta vol dir que hi ha un tallafocs entremig).

```
p=IP(dst="192.168.15.10")/TCP(sport=RandShort(),dport=[22,80,110,443],flags="S")
sr(p,timeout=3)
ans,unans = _
ans.summary()
```

cBIS) Sabiendo que el comando *print()* es capaz de mostrar el contenido completo de una tupla (o lista) indicada como parámetro. ¿qué diferencia ves entre lo que mostraba en pantalla la orden *ans.summary()* anterior y lo que se muestra al ejecutar el siguiente bucle?

```
for unatupla in ans:
    print(unatupla)
```

cTRIS) ¿Quina diferència hi ha entre escriure la darrera línia anterior (*print(unatupla)*) i aquesta: *unatupla[1].show()* ? ¿I si escrius en canvi *unatupla[1][IP].show()* ? ¿I *unatupla[1][TCP].show()* ?

d) Justifica el tipus de paquet rebut com a resposta a l'enviament fet per *sr1()* d'un paquet generat així: *p=IP(ttl=2,dst="www.hola.com")*

e) Obre un terminal i executa la comanda *ncat -u -l -p 5000* (el paràmetre *-u* és per indicar que el port 5000 és de tipus UDP). Obre un altre terminal, executa les següents línies i observa el que apareix a la consola del servidor Ncat:

```
conf.L3socket=L3RawSocket
p=IP(dst="127.0.0.1")/UDP(sport=45678,dport=5000)/Raw("Hola\n")
send(p)
```

NOTA: La primera línia és necessària per a què Scapy pugui enviar correctament paquets a la IP loopback, ja que aquesta IP és especial.
NOTA: El valor de *sport* pot ser qualsevol mentre sigui un número elevat menor de 65535
NOTA: El salt de línia al final del "payload" és necessari per a què el paquet estigui ben format

¿Per què si intentes fer el mateix però usant el protocol TCP en comptes d'UDP no funciona? Pista: canvia la darrera línia (*send(p)*) per *sr1(p)* i observa el valor del camp "flags" de la capçalera TCP de la resposta.

f) Sabent que el valor decimal de cada "flag" TCP és el següent: FIN=1, SYN=2, RST=4, PSH=8, ACK=16 i URG=32 i que si hi ha varis "flags" actius a la vegada el valor resultant és la suma, dedueix què és el que mostra el codi següent

```
p=IP(dst="www.google.com")/TCP()
ans,unans=sr(p*5,inter=2)
for a in ans:
    if a[1][TCP].flags == 18:
        print(a[1][IP].src)
```

Per enviar paquets directament des de la capa 2 (és a dir, manipulant les capçaleres Ethernet) cal utilitzar, en comptes de les comandes *send(), sr1(), sr()* o *srloop()*, les comandes "equivalents" *sendp(), srp1(), srp()* o *srploop()*, respectivament. El cas més habitual en el què ens interessarà usar aquestes comandes de capa 2 és quan necessitem modificar manualment la direcció MAC d'origen o de destí del/s paquet/s a generar. Afortunadament, aquestes comandes de capa 2 tenen els mateixos paràmetres que les seves corresponents comandes de capa 3.

2.-a) Executa les següents comandes i dedueix per a què serveixen. ¿Què n'obtens? ¿Quina comanda de terminal vista en exercicis anteriors seria equivalent?

```
ans,unans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst="192.168.15.0/24"),timeout=2)
ans.summary()
```

b) Obre el Wireshark i torna a executar la comanda *srp()* anterior. ¿Què veus?

c) Manté el Wireshark capturant i aplica el filtre *icmp*. Utilitza seguidament l'eina Scapy per ara crear i enviar a la màquina on tens el Wireshark funcionant un paquet ICMP de tipus 3. ¿El detecta el Wireshark? ¿La màquina de destí genera algun missatge de resposta?

Un altre exemple on s'envien paquets a la MAC "broadcast" és en el cas dels paquets DHCP DISCOVER. Recordem que el procés per demanar una IP a través del protocol DHCP és primer enviar un paquet d'aquest tipus a tota la xarxa i esperar a rebre algun paquet DHCP OFFER d'algun servidor DHCP candidat (per seguidament respondre a tothom confirmant l'oferta d'un determinat servidor amb un paquet DHCP REQUEST -també de tipus "broadcast"- i, finalment, rebre la confirmació del servidor concret escollit amb un paquet DHCP ACK).

3.-a) Executa les següents comandes i raona quin tipus de paquet és el que generes (pots tenir el Wireshark encés per rebre'l allà i observar-lo millor) i quin és el paquet que obtens com a resposta (recorda que el pots veure per pantalla amb *r.show()*). ¿Quina màquina et respon? ¿Quin valor tenen les opcions rebudes "router" i "name_server"?

```
conf.checkIPaddr = 0
versioMAC,MACbinaria = get_if_raw_hwaddr(conf.iface)
p=Ether(dst="ff:ff:ff:ff:ff:ff")/IP(src="0.0.0.0",dst="255.255.255.255")/UDP(sport=68,dport=67)
        /BOOTP(chaddr=MACbinaria)/DHCP(options=[("message-type","discover"),"end"])
r = srp1(p)
```

NOTA: Com es pot veure, un paquet DHCP està format, al nivell d'aplicació, per una "capçalera" compatible amb el protocol antic BOOTP que conté (un altre cop) la MAC de la màquina origen ¡en format binari! i després el "payload" pròpiament dit les opcions que defineixen el tipus de paquet ("Discover", "Offer", etc); sempre ha d'haver al final una opción anomenada "end" que indica el final del paquet.

NOTA: Scapy normalment s'assegura que la resposta provingui de la mateixa IP a la qual s'envia el primer paquet. Com que la petició DHCP és feta a la IP de destí de "broadcast" (255.255.255.255) però qualsevol resposta tindrà com a IP d'origen la del servidor DHCP que respon, hem de fer que Scapy accepti aquestes respostes on no coincideixen les IPs de destí i de resposta; això s'aconsegueix amb la directiva *conf..checkIPaddr=0* abans de començar.

NOTA: Si en comptes de la línia *r=srp1(p)* del codi anterior haguéssim escrit les següents línies...:

```
ans, unans = srp(p, multi=true,timeout=10)
for i in ans:
    print(i[1][IP].src)
```

...hauríem implementat un sistema de detecció de servidors DHCP "espuris" a la xarxa local. Aquí la clau està en el paràmetre "multi=true", el qual fa que Scapy s'esperi (dins del període marcat per "timeout") a obtenir més d'una resposta després d'haver rebut la primera. Si no el posem, Scapy acceptarà la primera resposta corresponent al paquet enviat i això és justament el que volem evitar per poder recopilar totes les eventuals múltiples respostes a la mateixa petició DISCOVER que provinguin de diferents servidors DHCP.

b) ¿Per a què serveixen les línies de codi sota el comentari "craft DHCP REQUEST from DHCP OFFER" que apareixen a https://github.com/mrizvic/scapy/blob/master/dhcp-discover-transcation.py ?

> NOTA: Per saber més sobre el protocol DHCP es pot consultar
> https://support.microsoft.com/en-us/help/169289/dhcp-dynamic-host-configuration-protocol-basics

Després d'afegir la capa de transport UDP (amb el camp *dport* igual a *53*), Scapy proporciona la comanda *DNS()* per poder especificar els camps adients per realitzar peticions fent servir aquest protocol (i també generar respostes, per què no?...en aquest cas llavors el que s'hauria d'establir a 53 és el camp *sport*) De totes els camps que apareixen a *ls(DNS)*, els més importants són:

*ID: És un identificador de 16 bits que el sistema operatiu utilitza per distingir entre consultes. D'aquesta manera es pot saber a quina consulta correspon cada resposta que arriba (l'ID de resposta és el mateix que el de la consulta). No caldrà especificar-lo a mà perquè el sistema operatiu ja li assigna un valor adient automàticament

*QR: Tipus de missatge (0 = consulta , 1 = resposta). Si no s'indica, per defecte és 0

*OPCODE: Tipus de consulta (0 = estàndar , 1 = inversa, 2 = petició d'estat de servidor). Si no s'indica, per defecte és 0

*RD: Indica si es vol una resposta recursiva (1) o no (0). Si no s'indica, per defecte és 0

*QDCOUNT: Número de consultes que conté el missatge. Si no s'indica, per defecte és 1

*QD : Camp de petició. Està format al seu torn per tres camps (fent de separador el caràcter *null*) :

*QNAME : Nom de domini/host (ha de començar amb \n i el punt s'ha d'escriure amb 0x03)
*QTYPE : Registre a consultar ("A" -1 en hexadecimal-,"CNAME","AAAA","NS",…)
*QCLASS : Sempre valdrà "IN" (d'Internet) o, equivalentment, 1 en hexadecimal

Aquest camp de petició es pot escriure directament "a pèl" així, per exemple:
qd="\nwww\x03hola\x03com\x00\x01\x00\x01"
o bé, de forma molt més convenient, amb la construcció DNSQR que ofereix Scapy, així:
qd= DNSQR(qname="www.hola.com",qtype="A")

4.-Realitza amb la comanda *sr1()* d'Scapy la consulta DNS adient (i observa els camps del paquet de resposta) per esbrinar la IP i els "àlies" associats al nom "www.hola.com". **Pista:** un exemple molt semblant es troba al final de la pàgina 4 de la teoria.

El "fuzzing" és un conjunt de tècniques que tenen com objectiu injectar aleatòria i automatitzadament dades contra un determinat servei o software per tal de generar errors i així poder corregir els defectes. En l'àmbit de les xarxes, el"fuzzing" consisteix concretament en generar i enviar paquets malformats per comprovar com gestiona aquest tipus de situacions els serveis o protocols estudiats. En aquest sentit, Scapy proporciona la comanda *p=fuzz(PROTOCOL())*, la qual genera un paquet "p" amb els camps de la capçalera corresponent al protocol indicat entre parèntesis (ARP, IP, ICMP, UDP, TCP, DNS, etc) plens de valors aleatoris.

Es pot escollir a quins camps concrets es vol aplicar el "fuzzing" i a quins no si entre parèntesis s'indiquen els valors concrets dels camps que no volem que siguin "aleatoritzats"; per exemple, la comanda `p=fuzz(IP(dst="192.168.18.10"))/fuzz(TCP(dport=80))` generarà un paquet amb la IP i el port de destí indicats i la resta de camps de les capçaleres IP i TCP aleatoris.

5.-a) ¿Per què tots els paquets enviats acaben dins de la llista "unans"?. **Pista:** dedueix la raó a partir dels valors que mostra *summary()*

```
p=fuzz(IP(dst="192.168.1.1"))/fuzz(TCP(dport=80))
ans,unans=srloop(p,count=6,verbose=0)
unans.summary()
```

b) Seguint repassant l'ús de la comanda *srloop()*, dedueix ara què fa aquest script, línia a línia (apareix una comanda no vista fins ara, *p.haslayer()*, però és força intuitiva):

```
ans, unans = srloop(IP(dst=["8.8.8.8", "8.8.4.4"])/ICMP(), inter=0.1, timeout=0.1, count=10, verbose=0)
for a in ans:
    if a[1].haslayer(ICMP) == True:
        print("IP:" + a[1][IP].src + " Codi:" + str(a[1][ICMP].code))
```

Scapy també es pot utilitzar com a "sniffer" mitjançant la comanda *sniff()* . Paràmetres interessants d'aquesta comanda són:

count=nº : indica la quantitat total de paquets que es voler "recol.lectar", els quals s'afegiran a una llista com a valor de retorn. Un cop arribat a aquest número la "recol.lecció" s'acabarà.
timeout=nº : alternativa al paràmetre anterior, indica el temps de "recol.lecció" passat el qual s'acabarà aquesta. Igualment, els paquets "recol.lectats" s'afegiran a la llista indicada com valor de retorn
filter="…" : indica un filtre BPF per tal de recollir només aquells paquets que coincideixin amb les característiques indicades al filtre (direccions MAC d'origen o de destí, direccions IP d'origen o de destí, ports d'origen o de destí, etc); la sintaxis d'aquests filtres és idèntica a la dels filtres de captura de Wireshark.
store=0 : descarta de la memòria tots els paquets un cop rebuts: si es vol monitoritzar la xarxa durant molta estona és recomanable indicar aquest valor
offline="fitxer.pcap" : llegeix els paquets d'un fitxer de captura "pcap" -com els que genera Wireshark- en comptes d'esnifar-los en temps real
prn=nomFuncio : indica quina funció Python pròpia s'executarà per cada paquet esnifat -sempre que concordi amb el filtre, si n'hi ha-, el qual pot passar-se com a paràmetre; el valor retornat d'aquesta funció es veurà a la pantalla

6.-a) Executa les següents comandes i digues què és el que veus per pantalla:

```
catched=sniff(timeout=5,count=20)
catched.summary()
catched[3].show()
```

b) Executa les següents comandes i digues, després d'obrir un navegador i anar a la pàgina web de Google, què és el que veus per pantalla:

```
catched=sniff(filter="tcp and host www.google.com",count=4)
catched.nsummary()
catched[3][TCP].show()
```

Para crear una función Python propia hay que seguir los siguientes pasos básicos:

1.-Empezar su definición con una línia tal como *def nombreFuncion(param1,param2,...)**:***
2.-Escribir el cuerpo de la función indentado respecto la línia anterior
3.-Opcionalmente, acabar la definición con una lína tal como *return(valor)*

Por ejemplo, la siguiente función devolverá el resultado de sumar los dos valores pasados como parámetros:

```
def suma(x,y):
    r=x+y
    return(r)
```

Para utilizar la función anterior se puede hacer lo siguiente, por ejemplo:

```
valor=suma(3,4)
print(valor)
```

c) Executa les següents comandes i seguidament obre un altre terminal per realitzar un "ping" a qualsevol màquina; digues què és el que veus per pantalla i quin és el contingut del fitxer que obtens (obre'l amb el Wireshark):

```
def veure(paquet):
        print("IP:" + paquet[IP].src + " Tipus:" + str(paquet[ICMP].type))
sniff(filter="icmp",prn=veure,count=6)
llista=_
llista.nsummary()
wrpcap("fitxer.pcap",llista)
```

cBIS) ¿Per a què creus que serveix el següent codi? ¿Quina utilitat té aquí l'operació mòdul tenint en compte que el fitxer "fitxer.pcap" és el generat a l'apartat anterior?

```
llista=rdpcap("fitxer.pcap")
i=0
while i < len(llista):
        if i % 2 == 0:
                send(llista[i])
        i=i+1
```

d) Pensa el que faria el codi següent (allà on posa "x.x.x.x" representa que hauria d'anar la teva direcció IP)

```
s = sniff(filter="dst host x.x.x.x and icmp[0]=8", count=1)
req = s[0]
while True:
        send(IP(src=req[IP].dst,dst=req[IP].src)/ICMP(type=0))
```

La gràcia de l'Scapy és que a més d'oferir un entorn interactiu, també és una llibreria Python que es pot fer servir com qualsevol altra llibreria dins d'scripts escrits en aquest llenguatge. Només cal importar-la al començament del nostre codi (amb la línia *from scapy.all import **) i ja està: si a partir d'aquí escrivim el nostre script i el gravem per exemple amb el nom de "script.py", per executar-lo haurem d'escriure en un terminal *python script.py* o bé, si la seva primera línia és *#!/usr/bin/python* i té permisos d'execució, simplement així: *./script.py*

En Python, els paràmetres passats a l'script des del terminal es poden recopilar important el mòdul "sys" (*import sys*) i fent servir llavors les variables *sys.argv[1]* (equivalent en Bash a $1), *sys.argv[2]* (equivalent a $2), etc També es podria fer servir directament *sys.argv* (que representa la llista de paràmetres passats, cadascun com un element d'ella) o *len(sys.argv)* , equivalent en Bash a $#+1. Una alternativa podria ser utilitzar el mòdul "argparse", molt més complet

D'altra banda, si el que es volgués és que el nostre script Python demanés els valors dels paràmetres de forma interactiva, es pot utilitzar la comanda *variable=input("Pregunta a mostrar")*, que seria l'equivalent a *read -p "Pregunta a mostrar" variable* en Bash. Cal tenir en compte, però, que qualsevol valor recollit per *input()* és reconegut automàticament com a "string"; si es necessita que sigui d'un altre tipus s'haurà de fer la conversió explícitament (el que se'n diu fer un "cast"); per exemple, per transformar el tipus del valor recollit en sencer, caldria fer *int(variable)*

7.-a) Prova el següent script i digues què fa

```
#!/usr/bin/python
import sys
```

```
from scapy.all import *
conf.verb=0
if len(sys.argv) != 3:
    print("Uso: " + sys.argv[0] + " dir.IP" + " puerto")
    sys.exit(1)
destino=sys.argv[1]
puerto=int(sys.argv[2])
r=sr1(IP(dst=destino)/TCP(dport=puerto))
if r[TCP].flags == 18:
    print(r[IP].src + " tiene el puerto " + str(r[TCP].sport) + " abierto")
else:
    print(r[IP].src + " tiene el puerto " + str(r[TCP].sport) + " cerrado")
```

b) Ídem

```
#!/usr/bin/python
import sys
if len(sys.argv) != 2:
    print("Uso: " + sys.argv[0] + " dir.IP.red/mascara" )
    sys.exit(1)
from scapy.all import *
conf.verb=0
ans,unans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst=sys.argv[1]),timeout=2)
for s,r in ans:
   print("El ordenador " + r[Ether].src + " tiene la IP solicitada (" + r[ARP].psrc + ")")
```

c) Ídem

```
#!/usr/bin/python
from scapy.all import *
def arp_display(pkt):
    if pkt[ARP].op == 1:  # who-has (request)
       return("Request:" + pkt[ARP].psrc + " is asking about " + pkt[ARP].pdst)
    if pkt[ARP].op == 2:  # is-at (response)
      return("Response:" + pkt[ARP].hwsrc + "has address " + pkt[ARP].psrc)
sniff(prn=arp_display, filter='arp', store=0, count=10)
```

d) Ídem. ¿Per què busquem que el valor de la propietat *reply.type* a l'*elif* sigui 3?

```
#!/usr/bin/python
from scapy.all import *
hostname = "www.google.com"
for i in range(1,28):
    p=IP(dst=hostname,ttl=i)/UDP(dport=33434)
    reply=sr1(p,verbose=0,timeout=5)
    if reply is None:
        break
    elif reply.type == 3:
        print("Fin!" + reply.src)
        break
    else:
        print(str(i) + " salto: " + reply.src)
```

e) Ídem

```
#!/usr/bin/python
from scapy.all import *
```

```
counter = 0
def custom_action(packet):
    global counter
    counter += 1
    return(counter, packet[0][1].src, packet[0][1].dst)
sniff(filter="ip", prn=custom_action)
```

Per realitzar una conexió TCP cal executar primer el "3-way handshake".Els passos són els següents:

1.- S'envia un paquet SYN amb un determinat número de seqüència (a escollir aleatori).Per exemple:

```
r=sr1(IP(dst="192.168.15.10")/TCP(sport=9876,dport=80,flags="S",seq=12345))
```

2.-Del paquet rebut (de tipus SYN/ACK) s'obté el seu número de seqüència (que haurà generat l'altre extrem pel seu compte) i s'hi suma 1. El valor resultant s'assigna com a valor del camp ACK del paquet que s'enviarà a l'altre extrem de nou, el qual ara com a número de seqüència tindrà el consecutiu al del paquet anterior (valor que coincideix amb el del camp ACK rebut de l'altre extrem):

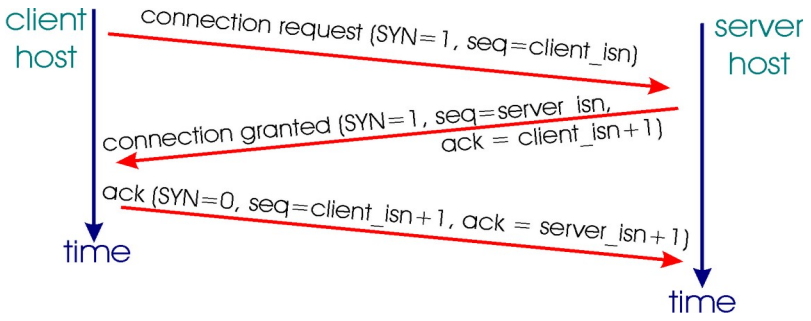
```
ack=r[TCP].seq + 1
send(IP(dst="192.168.15.10")/TCP(sport=9876,dport=80,flags="A",seq=12346,ack=ack))
```

NOTA: El valor de *sport* el podríem haver assignat a *r[TCP].dport* i el de *seq* a *r[TCP].ack*

3.-A partir d'aquí, ja està establerta la connexió TCP i, per tant, tot seguit ja es podrà enviar els paquets desitjats de nivells superior, com per exemple, una petició HTTP (fent servir el mateix número de seqüència i valor "ack" que el paquet anterior, ja que aquests números no han de canviar fins que es rebi algun altre paquet de l'altre extrem):

```
w=sr1(IP(dst="192.168.15.10")/TCP(sport=9876,dport=80,flags="A",seq=12346,ack=ack)
      /Raw("GET / HTTP/1.1\nHost:elpuig.xeill.net\n\n"))
```

El procés es pot veure més clarament en aquest diagrama:



Cal tenir en compte, però, que els passos anteriors poden no funcionar. La raó és que les sessions TCP generades amb Scapy són totalment "independents" de les que controla el sistema operatiu i això vol dir que en rebre el paquet SYN/ACK de l'altre extrem, el sistema operatiu no el reconeixerà com un paquet sol.licitat pertanyent a cap sessió oberta per ell i, per tant, es "prendrà la justícia pel seu compte" i tallarà la connexió enviant un paquet de tipus RST. Una solució a això seria utilitzar el tallafocs del sistema per bloquejar aquests paquets RST. En el nostre exemple, això equivaldria a executar, abans de realitzar la conexió, la següent comanda (sí, el valor "RST" ha d'estar repetit):

sudo iptables -A OUTPUT -d 192.168.15.10 -p tcp --dport 80 --tcp-flags RST RST -j DROP

NOTA: Si fem servir UDP passaria el mateix...en aquest cas el sistema operatiu generaria pel seu compte un paquet ICMP de tipus "Port unreachable"

8.-a) ¿Què fa el següent script Python?

```
#!/usr/bin/python
import os
from scapy.all import *

print("\n[*] Droping RST output packets to destiny")
os.system('iptables -A OUTPUT -p tcp --tcp-flags RST RST -d 8.43.85.67 -j DROP')

paqSYN=IP(dst="elpuig.xeill.net")/TCP(sport=RandNum(1024,65535),dport=80,flags="S",seq=42)
print("\n[*] Sending SYN packet")
respSYNACK=sr1(paqSYN,verbose=0)
print("\n[*] Receiving SYN,ACK packet")

paqACK=IP(dst="elpuig.xeill.net")/TCP(sport=respSYNACK.dport,dport=80,flags="A",seq=respSYNACK.ack,ack=respSYNACK.seq+1)
print("\n[*] Sending ACK packet")
send(paqACK,verbose=0)
print("\n[*] Done!")

paqGET=paqACK/Raw("GET /robots.txt HTTP/1.1\r\nHost:elpuig.xeill.net\r\n\r\n")
print("\n[*] Sending GET request")
#El paràmetre multi=1 s'ha de posar perquè la resposta del servidor pot estar formada per varis paquets
parelles,basura=sr(paqGET,timeout=2,multi=1,verbose=0)
print("\n[*] Receiving HTTP response\n")
for parella in parelles:
        if parella[1].haslayer(Raw):
                print(parella[1].load)

#Faltaria realitzar la seqüència de desconnexió (FIN/ACK, FIN/ACK, ACK) per fer-ho bé

print("\n[*] Restoring RST output packets to destiny")
os.system("iptables -D OUTPUT -p tcp --tcp-flags RST RST -d 8.43.85.67 -j DROP")
```

NOTA: Es poden fer servir variables dins de la comanda os.system() simplement concatenant-les en mig de la cadena que actúa com a paràmetre (és a dir, així per exemple: *os.system("echo " + variable + " >> fitxer.txt")* .

NOTA:Una manera alternativa (més segura) d'executar una comanda del shell des de Python és utilitzant el mètode *subprocess.run(["nomComanda","param1","param2",variable])* pertanyent a la llibreria *subprocess*

b) Modifica l'script anterior per a què rebi com a paràmetre el nom (o la IP) d'un servidor web qualsevol i per a què realitzi una petició HTTP (GET) dirigida a la seva pàgina principal ("/"), mostrant per pantalla el contingut de la resposta.

c) Obre un terminal i executa la comanda *ncat -l -p 5000*. Obre un altre terminal i executa-hi les comandes Scapy adients per, després d'haver establert el 3-way handshake amb aquest servidor Netcat , enviar-li la paraula "Hola" (hauràs d'aconseguir, per tant, què aquesta paraula es mostri a la pantalla del servidor Netcat). **Pista**: Fes servir com a referència l'script de l'apartat a) i simplement adaptar-lo.

Ja sabem que existeixen moltes tècniques diferents per esbrinar si un destí "víctima" té determinats ports oberts (unes més ràpides però més "sorolloses", unes altres més lentes però més "invisibles", unes més fiables, unes altres no tant, etc). Totes aquestes tècniques en realitat es poden posar en pràctica amb diferents paràmetres de la comanda *nmap* (-sS, -sT, -sA, -sW, -sM, -sU, -sN, -sF, -sX, etc) però amb Scapy es poden realitzar també d'una forma més "manual". A continuació es mostren alguns exemples (que es poden veure gràficament a https://resources.infosecinstitute.com/port-scanning-using-scapy) :

9.-a) Implementa l'escaneig -sT de Nmap ("TCP connect scan") executant el següent codi i digues què significa el valor 0x12 de l'"if"[NOTA] i el perquè de l'enviament del paquet "AR". Canvia el port de destí pel 81 i torna a provar l'script: ¿observes una resposta diferent?

```
#!/usr/bin/python
from scapy.all import *
conf.verb=0
dst_ip = "192.168.15.10"
src_port = RandShort()
dst_port=80
resp = sr1(IP(dst=dst_ip)/TCP(sport=src_port,dport=dst_port,flags="S"))
if resp.getlayer(TCP).flags == 0x12 :
    send(IP(dst=dst_ip)/TCP(sport=src_port,dport=dst_port,flags="AR"))
    print("Open")
else:
    print("Closed")
```

NOTA: Recorda que els valors numèrics dels flags són F=1 , S=2 , R=4 , P=8, A=16 i U=32 i que si un paquet té vàries flags activades el valor final del camp "flags" és la suma dels valors de cada flag activada. Per tant, si un paquet és SYN/ACK, el valor del seu camp "flags" serà 2+16=18, que en hexadecimal és 0x12. Scapy ens fa el favor de permetre escriure en comptes d'aquest valor un d'equivalent amb lletres ("SA", en aquest cas), però no està de més saber quins són els valors "reals".

b) Implementa l'escaneig -sS de Nmap ("TCP stealth scan") executant el següent codi i digues quina diferència hi ha amb el comportament de l'anterior escaneig en el cas de què el port estigui obert. ¿Què significa si la resposta és de tipus ICMP? Canvia el port de destí pel 81 i torna a provar l'script: ¿observes una resposta diferent?

```
#!/usr/bin/python
from scapy.all import *
conf.verb=0
dst_ip = "192.168.15.10"
src_port = RandShort()
dst_port=80
resp = sr1(IP(dst=dst_ip)/TCP(sport=src_port,dport=dst_port,flags="S"))
if resp.haslayer(TCP) :
    if resp.getlayer(TCP).flags == 0x12 :
        send(IP(dst=dst_ip)/TCP(sport=src_port,dport=dst_port,flags="R"))
        print("Open")
    else:
        print("Closed")
elif resp.haslayer(ICMP) :
    if int(resp.getlayer(ICMP).type)==3 and int(resp.getlayer(ICMP).code) in [1,2,3,9,10,13]:
        print("Filtered")
```

c) Implementa l'escaneig -sX de Nmap ("Xmas scan") executant el següent codi i digues per què necessitem el paràmetre "timeout" a l'enviament del paquet, quin és el significat de la condició del primer "if" i el del valor 0x04 del primer "elif". Canvia el port de destí pel 81 i torna a provar l'script: ¿observes una resposta diferent? (en teoria hauria de ser que sí però a la pràctica sembla que no...ai)

```
#!/usr/bin/python
from scapy.all import *
conf.verb=0
dst_ip = "192.168.15.10"
src_port = RandShort()
dst_port=80
resp = sr1(IP(dst=dst_ip)/TCP(sport=src_port,dport=dst_port,flags="FPU"), timeout=5)
if str(type(resp))=="<class 'NoneType'>":
    print("Open|Filtered")
    quit()
elif resp.haslayer(TCP) and resp.getlayer(TCP).flags == 0x04 :
    print("Closed")
elif resp.haslayer(ICMP) and int(resp.getlayer(ICMP).type)==3 and int(resp.getlayer(ICMP).code) in [1,2,3,9,10,13] :
    print("Filtered")
```

d) L'escaneig -sF de Nmap ("FIN scan") obté els mateixos resultats que el de tipus Xmas: si la màquina remota no contesta es dedueix que el port a investigar està obert, si contesta amb un paquet RST es dedueix que està tancat i si contesta amb un paquet ICMP es dedueix que està filtrat. Sabent això, quin únic canvi hauries de fer al codi anterior per convertir-lo en un script que faci l'escaneig FIN?

e) L'escaneig -sN de Nmap ("Null scan") obté els mateixos resultats que el de tipus Xmas o FIN: si la màquina remota no contesta es dedueix que el port a investigar està obert, si contesta amb un paquet RST es dedueix que està tancat i si contesta amb un paquet ICMP es dedueix que està filtrat. Sabent això, quin únic canvi hauries de fer al codi anterior per convertir-lo en un script que faci l'escaneig Null?

f) L'escaneig -sA de Nmap ("ACK scan") no s'usa per esbrinar si un port està obert o no sinó per saber si hi ha un tallafocs a "l'altra banda" o no perquè el que indica és si el port està filtrat o no. La idea és enviar un paquet ACK que no pertany a cap connexió coneguda (un paquet "espuri") i observar com es comporta la "víctima". Executa el següent codi i comprova què passa amb la màquina (i port) remots que vulguis:

```
#!/usr/bin/python
from scapy.all import *
conf.verb=0
dst_ip = "192.168.15.10"
src_port = RandShort()
dst_port=80
resp = sr1(IP(dst=dst_ip)/TCP(sport=src_port,dport=dst_port,flags="A"))
if resp.haslayer(TCP) and resp.getlayer(TCP).flags == 0x04:
      print("Unfiltered")
else:  #It can be a ICMP type=3 response or none at all
    print("Filtered")
```

g) Troba a l'enllaç indicat al paràgraf blau previa a aquest exercici quina és la diferència que hi ha entre el "ACK Scan" i l'escaneig -sW de Nmap ("TCP Window scan").

h) El següent codi implementa un escaneig -sU de Nmap ("UDP scan"); dedueix quin tipus de paquets rebràs segons l'estat del port de la màquina remota (tancat, obert, filtrat, no resposta).

```
#!/usr/bin/python
from scapy.all import *
conf.verb=0
dst_ip = "192.168.1.1"
dst_port=53
dst_timeout=5
resp = sr1(IP(dst=dst_ip)/UDP(dport=dst_port),timeout=dst_timeout)
if (str(type(resp))=="<class 'NoneType'>"):
        print("Open|Filtered")
        quit()
elif (resp.haslayer(UDP)):
        print("Open")
elif(resp.haslayer(ICMP)):
        if(int(resp.getlayer(ICMP).type)==3 and int(resp.getlayer(ICMP).code)==3):
            print("Closed")
        elif(int(resp.getlayer(ICMP).type)==3 and int(resp.getlayer(ICMP).code) in [1,2,9,10,13]):
            print("Filtered")
```

NOTA: Remember: TCP is a connection-oriented protocol and UDP is a connection-less protocol. A connection-oriented protocol is a protocol in which a communication channel should be available between the client and server and only then is a further packet transfer made. If there is no communication channel between the client and the server, then no further communication takes place. A Connection-less protocol is a protocol in which a packet transfer takes place without checking if there is a communication channel available between the client and the server. The data is just sent on to the destination, assuming that the destination is available.

i) ¿Què implementa el següent codi font: https://github.com/interference-security/Multiport/blob/master/multiport.py ?

10.- Otros programas/librerías complementarias:
https://github.com/invernizzi/scapy-http
https://github.com/tintinweb/scapy-ssl_tls
https://github.com/secdev/scapy/blob/master/doc/notebooks/tls/notebook1_x509.ipynb

https://github.com/secdev/scapy/blob/master/doc/notebooks/HTTP_2_Tuto.ipynb

Otros:
https://ostinato.org/
https://github.com/jemcek/packETH
https://github.com/appneta/tcpreplay

No funciona:
A partir de lo ya visto, podemos realizar acciones más avanzadas, como las siguientes:

Ejemplo avanzado nº1

Escaneador para una red concreta de clase C que permite enumerar todos los hosts de esa red que tengan abierto el puerto 80. El frag SYN se utiliza para iniciar una conexión. Una respuesta SA (SYN/ACK) significa que el puerto está a la escucha, un RA (RESET/ACK) significa que está cerrado, y una ausencia de respuesta significa que el host está apagado o un firewall está descartando los paquetes:

```
sr(IP(dst="192.168.1.0/24")/TCP(dport=80),timeout=3)
ans = _[0]
for pout, pin in ans:
        if pin.flags == 2:
                print pout.dst
```

---

conf.route
conf.route.add
conf.route.del

we can use Scapy to re-create a packet that has been sniffed or received. The method command() returns a string of the commands necessary for this task.
bind_layers
split_layers

Solución del ejercicio 8c):

```
#!/usr/bin/python3
import os
from scapy.all import *
#conf.L3socket=L3RawSocket
print("\n[*] Droping RST output packets to destiny")
os.system('iptables -A OUTPUT -p tcp --tcp-flags RST RST -d 192.168.1.36 -j DROP')
print("\n[*] Sending SYN packet and receiving SYN,ACK response")
paqSYN=IP(dst="192.168.1.36")/TCP(sport=RandNum(1024,65535),dport=5000,flags="S",seq=42)
respSYNACK=sr1(paqSYN,verbose=0)
```