Capítulo 19

Sockets raw

Los sockets más usados con diferencia son los TCP seguidos de los UDP. Pero hay muchos otros tipos de socket. Este capítulo es una introducción muy práctica a los «sockets raw»[1].

El término *raw* en informática se suele utilizar para indicar acceso directo a los datos que proporciona un dispositivo, o al menos con menor intervención del SO o librerías involucradas[2]. Este acceso directo tiene tres implicaciones principales:

- Mayor flexibilidad, al no estar limitado por las reglas o normas que impongan las capas de alto nivel que ofrece el sistema operativo.
- Acceso privilegiado, debido precisamente a que dichas posibilidades tienen un impacto directo sobre la seguridad del sistema y la privacidad de sus usuarios.
- Menos soporte, ya que son precisamente las capas del sistema operativo que se dejan a un lado las que simplifican el manejo del recurso. El «modo raw» conlleva un nivel de abstracción mucho menor y por tanto, más complejidad técnica.

Estas tres cuestiones se pueden aplicar casi a cualquier dispositivo que permita un acceso «raw», sea un periférico USB, una consola o, como en este caso, un socket.

Con los sockets `AF_INET:SOCK_STREAM` o `AF_INET:SOCK_DGRAM` no es posible acceder (para leer o escribir) a las cabeceras de ninguno de los protocolos de TCP/IP de la capa de transporte o inferior, ya sea IP, ICMP, ARP, TCP, etc. Esos sockets únicamente permiten indicar cuál será la carga útil de los segmentos TCP o UDP y solo indirectamente se puede influir en algunos de los campos de sus cabeceras: puerto origen y destino y poco más[3].

[1] A veces (dolorosamente) traducido como «conector directo».
[2] El adjetivo *raw* (crudo) se utiliza como contraposición a *cooked* (cocinado).
[3] a menos que acudamos a llamadas al sistema como `setsockopt()`[sc].

En raras situaciones se necesita ofrecer servicios que implican a protocolos de capas 2 y 3, o a las cabeceras de tramas, paquetes y segmentos, que normalmente quedan fuera de la vista del programador. Algunos programas de este tipo pueden ser `ping`, `traceroute`, `arping` o un *sniffer* cualquiera. Entonces ¿cómo se hacen estos programas? La respuesta, como habrás imaginado, pasa por los sockets raw.

19.1. Acceso privilegiado

En §18.1.1 hablamos de la necesidad de `sudo` o *capabilities* para ejecutar un sniffer y eso se debe precisamente a que utilizan sockets raw, de modo que todo eso es aplicable a cualquier programa que escribamos y también los utilice.

Lamentablemente, las *capabilities* solo se pueden aplicar a programas binarios, no a scripts de Python. Por eso, para ejecutar los programas que veremos en este capítulo, tendrás que utilizar `sudo`.

Aparte de la ejecución del programa, también necesitas configurar la interfaz de red en «modo promiscuo». Si tu interfaz de red es una tarjeta Ethernet o WiFi, únicamente las tramas broadcast, multicast o que vayan dirigidas específicamente a su dirección MAC serán capturadas y entregadas al subsistema de red. Sin embargo, si pretendes utilizar un socket raw, es muy probable que te interese recibir todo el tráfico que llegue a la interfaz de red de tu computador, y no sólo el mencionado.

Para lograrlo debes activar el «modo promiscuo» de la NIC, algo que puede hacer con el comando `ip`:

```
$ sudo ip link set eth0 promisc on
```

O con `ifconfig`:

```
$ sudo ifconfig eth0 promisc
```

De forma análoga puedes saber si la interfaz está en modo promiscuo con el siguiente comando (fíjate en el flag PROMISC).

```
$ ip link show eth0
2: eth0: <BROADCAST,MULTICAST,PROMISC,UP,LOWER_UP> mtu 1500 qdisc \
         pfifo_fast state UP qlen 1000
    link/ether 00:1b:c2:32:71:32 brd ff:ff:ff:ff:ff:ff
```

Y el equivalente con `ifconfig`:

```
$ /sbin/ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:1e:c9:34:7e:92
          inet addr:192.168.2.4  Bcast:192.168.2.255  Mask:255.255.255.0
```

```
          inet6 addr: fe80::21e:c9ff:fe34:7e92/64 Scope:Link
          UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1500  Metric:1
          RX packets:160791 errors:0 dropped:0 overruns:0 frame:0
          TX packets:121923 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:177101459 (168.8 MiB)  TX bytes:18361567 (17.5 MiB)
          Memory:fdfe0000-fe000000
```

19.2. Tipos de sockets raw

Lo primero a tener en cuenta es que hay dos tipos básicos de socket raw, y que la decisión de cuál utilizar depende totalmente del objetivo y requisitos de la aplicación que se desea:

Familia `AF_PACKET`
: Los sockets raw de la familia `AF_PACKET` son los de más bajo nivel y permiten leer y escribir cabeceras de protocolos de cualquier capa.

Familia `AF_INET`
: Los sockets raw `AF_INET` delegan al sistema operativo la construcción de las cabeceras de enlace y permiten una manipulación «compartida» de las cabeceras de red.

En las próximas secciones veremos en detalle la utilidad y funcionamiento de ambas familias.

19.3. Sockets `AF_PACKET:SOCK_RAW`

Son los sockets raw más flexibles y de más bajo nivel, y representan la elección obligada si el objetivo es crear un *sniffer* o algo parecido. Precisamente el siguiente listado es un *sniffer* extremadamente básico que imprime por consola las tramas Ethernet/WiFi completas recibidas por cualquier interfaz y portando cualquier protocolo.

```
import socket

ETH_P_ALL = 3

sock = socket.socket(socket.AF_PACKET, socket.SOCK_RAW,
                     socket.htons(ETH_P_ALL))

while 1:
    print("--\n{!r}".format(sock.recvfrom(1600)))
```

LISTADO 19.1: Sniffer básico con `AF_PACKET:SOCK_RAW`
/raw/sniff-all.py

Y a continuación el programa en funcionamiento:

```
$ sudo ./sniff-all.py
--
(b'\xff\xff\xff\xff\xff\xff\xa8\x92,
\xce\xcd\xb3\x08\x06\x00\x01\x08\x00\x06\x04\x00\x01\xa8\x92,
\xce\xcd\xb3\xac\x13\xb0O\x00\x00\x00\x00\x00\x00\xac\x13\xb0\x01',
('eth0', 2054, 1, 1, '\xa8\x92,\xce\xcd\xb3'))
```

Haciendo modificaciones mínimas a este programa es posible filtrar el tráfico en dos aspectos:

Tipo de trama
: Es decir, el código que identifica el protocolo encapsulado como carga útil.[4] Para ello se utiliza el tercer campo del constructor de `socket`.

La interfaz de red
: Se logra vinculando el socket a una interfaz de red concreta por medio del método `bind()`.

El uso de ambos «filtros» queda demostrado en el siguiente programa, llamado `sniff-arp.py`. Sólo muestra menajes ARP recibidos por la interfaz que se indique como argumento:

```
import sys
import socket

if len(sys.argv) != 2:
    print("usage: {} <iface>".format(sys.argv[0]))
    exit(1)

ETH_P_ARP = 0x0806

sock = socket.socket(socket.AF_PACKET, socket.SOCK_RAW,
                     socket.htons(ETH_P_ARP))
sock.bind((sys.argv[1], ETH_P_ARP))

while 1:
    print("--\n{!r}".format(sock.recv(1600)))
```

LISTADO 19.2: Sniffer `AF_PACKET:SOCK_RAW` filtrando ARP
`/raw/sniff-arp.py`

Y el programa en acción:

```
$ sudo ./sniff-arp.py wlan0
--
b'\xff\xff\xff\xff\xff\xffl>m\x84y\x1d\x08\x06\x00\x01\x08\x00\x06\x04\x00\x01l>m\x84y
\x1d\xa1C\x11<\x00\x00\x00\x00\x00\x00\xa1C\x11\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

[4] `http://www.iana.org/assignments/ethernet-numbers`

Si te fijas, es fácil identificar la cabecera Ethernet en esa secuencia de bytes. Aparecen 6 bytes `0xff`, que corresponden a la dirección broadcast de Ethernet; y más adelante `0x0806` qu,e como hemos visto en el programa, es el tipo para payload ARP.

De este modo tan sencillo es posible realizar un *sniffer* completamente a medida de las necesidades concretas. Pero todo esto sólo sirve para leer tramas. Ahora veremos cómo enviar, lo que abre un interesante mundo de posibilidades (y riesgos de seguridad).

Si quieres identificar el origen del paquete puedes utilizar el método `recvfrom()` en lugar de `recv()`. En ese caso el valor de retorno es una tupla que incluye, entre otras cosas, el nombre de la interfaz (*p. ej.* «eth0») y la dirección MAC origen como una secuencia de bytes.

19.3.1. Construir y enviar tramas

El mismo socket creado en los ejemplos anteriores se puede utilizar para enviar datos. Para sintetizar un paquete, es decir, construir cabeceras de acuerdo a las especificaciones, se utiliza normalmente el módulo `struct`[5].

El siguiente listado envía una cabecera Ethernet cuyos campos son:

Destino: `FF:FF:FF:FF:FF:FF`
Origen: `00:01:02:03:04:05`
Protocolo: 0x0806 (ARP)

```
import sys
import socket
import struct

if len(sys.argv) != 2:
    print("usage: {} <iface>".format(sys.argv[0]))
    exit(1)

ETH_P_ARP = 0x0806

sock = socket.socket(socket.AF_PACKET, socket.SOCK_RAW,
                     socket.htons(ETH_P_ARP))
sock.bind((sys.argv[1], ETH_P_ARP))

sock.send(struct.pack('!6s6sh', 6 * b'\xFF',
                      b'\x00\x01\x02\x03\x04\x05', ETH_P_ARP))
```

LISTADO 19.3: Enviando una trama Ethernet con AF_PACKET:SOCK_RAW
/raw/send-wrong-eth.py

[5]Consulte el capítulo 17 para más información sobre dicho módulo.

Si capturas esa trama con `wireshark` o `tshark` verás que aparece con un «malformed packet», y con razón: es sólo una cabecera ¡no tiene carga útil!

```
$ tshark -a duration:7 -n -f arp
Capturing on 'wlan0'
    1 0.000000000 00:01:02:03:04:05 → ff:ff:ff:ff:ff:ff ARP 14 [Malformed Packet]
```

Y eso lógicamente contradice todas las normas del protocolo Ethernet. Resumiendo, este programa no sirve para nada, sólo para que veas que se puede construir y enviar lo que quieras a la red, incluso aunque sea un completo sinsentido.

19.3.2. Implementando un `arping`

Aunque hay muchas variantes, el programa `arping` envía una petición ARP Request y espera la respuesta correspondiente. En esta sección veremos una implementación que sirve para ilustrar el uso de los sockets raw de la familia `AF_PACKET`.

19.3.2.1. Generando mensajes

El programa necesita enviar mensajes ARP Request, que irán encapsulados en tramas Ethernet. Una forma de implementar esta tarea (llamada a veces «sintetizar paquetes») y aprovechar la POO es escribir una clase por cada tipo de mensaje. Por tanto, la clase para generar el mensaje ARP Request es algo tan sencillo como esto:

```
class Ether:
    def __init__(self, hwsrc, hwdst):
        self.hwsrc = hwsrc
        self.hwdst = hwdst
        self.payload = None

    def set_payload(self, payload):
        self.payload = payload
        payload.frame = self

    def serialize(self):
        retval = struct.pack("!6s6sh", self.hwdst, self.hwsrc,
                             self.payload.proto) + self.payload.serialize()

        return retval + (60-len(retval)) * "\x00"
```

Lo único a destacar de la clase `Ether` es el método `serialize()` que se encarga de generar la representación binaria de los datos que corresponden a la cabecera, concretamente dirección MAC destino, MAC origen, protocolo (el que indique el payload) y a continuación el payload propiamente dicho.

Esos datos se empaquetan en binario gracias a `struct.pack()`[6] indicando que se trata de 2 secuencias de 6 bytes (`6s6s`) y un entero de 16 bits (`h`). La última línea de ese método calcula y concatena el relleno (*padding*) necesario para que la trama alcance el tamaño mínimo necesario de 60 bytes.

La clase para generar mensajes ARP Request es incluso más sencilla:

```
class ArpRequest:
    proto = ETH_P_ARP

    def __init__(self, psrc, pdst):
        self.psrc = socket.inet_aton(psrc)
        self.pdst = socket.inet_aton(pdst)
        self.frame = None

    def serialize(self):
        return struct.pack("!HHbbH6s4s6s4s", 0x1, 0x0800, 6, 4, 1,
                           self.frame.hwsrc, self.psrc, "\x00",  self.pdst)
```

19.3.2.2. Leyendo mensajes

La otra funcionalidad importante del programa es reconocer los mensajes que se obtendrán como respuesta si todo va bien. Se trata de discretizar el valor de cada campo representándolo en un formato adecuado. Esa tarea se suele llamar «disección de paquetes». Como en el caso anterior, una buena forma de hacer esto es delegar el reconocimiento (*parsing*) de cada tipo de mensaje en una clase específica. Hace falta una clase para reconocer tramas Ethernet y otra para reconocer mensajes ARP Reply.

La clase para reconocer tramas Ethernet puede ser algo tan sencillo como esto:

```
class EtherDissector:
    def __init__(self, frame):
        try:
            (self.hwdst,
            self.hwsrc,
            self.proto) = struct.unpack("!6s6sh", frame[:14])
        except struct.error:
            raise DissectionError

        self.payload = frame[14:]
```

El constructor acepta por parámetro una secuencia de bytes, es decir, la trama tal como se lee del socket. Los valores que «desempaqueta» con `struct` y que estarán accesibles como atributos públicos son: dirección MAC destino, MAC origen, protocolo y payload.

[6] Ver `http://docs.python.org/library/struct.html#format-characters`

El disector del mensaje ARP Reply, llamado `ArpReplyDissector`, es también muy sencillo:

```
class ArpReplyDissector:
    def __init__(self, msg):
        self.msg = msg

        if struct.unpack("!H", self.msg[6:8])[0] != ARP_REPLY:
            raise DissectionError

        try:
            (self.hwsrc, self.psrc,
            self.hwdst, self.pdst) = struct.unpack("!6s4s6s4s", msg[8:28])
        except struct.error:
            raise DissectionError
```

El constructor de la clase acepta por parámetro una secuencia de bytes, que corresponden con la carga útil de una trama. Como antes, los valores de todos los campos quedan disponibles como atributos de la instancia. Si algún campo o formato no corresponde, el constructor lanza la excepción `DissectionError`.

19.3.2.3. Programa principal

Solo queda escribir la función principal, la que realmente crea, lee y escribe en el socket. Aparece en el siguiente listado:

```
def main(ipsrc, ipdst, iface):
    print("Request: Who has {0}? Tell {1}".format(ipdst, ipsrc))

    sock = socket.socket(socket.AF_PACKET, socket.SOCK_RAW,
                         socket.htons(ETH_P_ARP))
    sock.bind((iface, ETH_P_ARP))

    frame = Ether(sock.getsockname()[-1], BROADCAST)
    frame.set_payload(ArpRequest(ipsrc, ipdst))

    sock.send(frame.serialize())

    while 1:
        eth = EtherDissector(sock.recv(2048))

        try:
            arp_reply = ArpReplyDissector(eth.payload)
            if arp_reply.hwdst == frame.hwsrc:
                print("Reply:   {0} is at {1}".format(
                      ipdst, display_mac(arp_reply.hwsrc)))
                break
        except DissectionError:
            print(".")
```

La función `main()` acepta las direcciones IP del host origen y destino, y la interfaz de red (línea 1). Primero crea y vincula el socket a la interfaz solicitada (lineas 4-6). A continuación crea una trama Ethernet con destino

broadcast y origen la MAC de la interfaz (línea 8), y le fija como payload una instancia de `ArpRequest`. El método `send()` envía la trama en su formato binario (línea 11).

El bucle `while` espera la respuesta. En cada iteración se lee y disecciona una trama (línea 14). Si esa trama contiene un mensaje ARP Reply, es decir, si `ArpReplyDissector` no lanza la excepción `DissectionError`, se comprueba además que esa sea la respuesta ARP que se espera y no otra (línea 18). Si es así se imprime la dirección IP del destino y la dirección MAC asociada a esa IP, que es el objetivo final del programa (líneas 19-20). Puedes encontrar una versión ampliada en el archivo `raw/arping.py`.

19.4. Sockets `AF_INET:SOCK_RAW`

A pesar de la flexibilidad y potencia de los sockets `AF_PACKET`, no siempre son la mejor elección ya que el programador debe parsear y generar el contenido de todas las cabeceras. Eso puede ser bastante engorroso cuando entra en juego el cálculo de checksums u otros datos no tan directos.

Los sockets `AF_INET:SOCK_RAW` pueden ser una buena alternativa si solo te interesa «tocar» las cabeceras de transporte, dejando al sistema operativo todo el trabajo relacionado con las de enlace, y opcionalmente las de red.

19.4.1. Capturando mensajes

El siguiente programa imprime por consola todos los paquetes IP que contengan un segmento UDP:

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_RAW,
                     socket.getprotobyname('udp'))
while 1:
    print("--\n{!r}".format(sock.recv(1600)))
```

LISTADO 19.4: Sniffer de mensajes UDP con `AF_INET:SOCK_RAW`
`/raw/sniff-udp.py`

La función `getprotobyname()` devuelve el número de protocolo[7] a partir de su nombre (linea 4). Es interesante destacar que el resultado del método `recv()` es el paquete IP completo, incluyendo cabecera (línea 7).

[7] `http://www.iana.org/assignments/protocol-numbers/protocol-numbers.xml`

Como en el caso de los socket `AF_PACKET` puedes identificar el origen del paquete (su dirección IP) sin tener que parsear la cabecera IP. Para lograrlo utiliza el método `recvfrom()` en lugar de `recv()`. En ese caso el valor de retorno es una tupla con la forma (datos, dirección), teniendo en cuenta que la dirección es su vez una tupla (IP, 0).

19.4.2. Enviando

Para enviar datos sobre este tipo de socket debes utilizar el método `sendto()` indicando la dirección destino. El Listado 19.5 envía envía un segmento UDP «sintético», pero válido, que contiene el texto «hello Inet».

```
import socket
import struct

sock = socket.socket(socket.AF_INET, socket.SOCK_RAW,
                     socket.getprotobyname('udp'))

payload = b'hello Inet'
udp_pkt = struct.pack('!4h', 0, 2000, 8+len(payload), 0) + payload
sock.sendto(udp_pkt, ('127.0.0.1', 0))
```

LISTADO 19.5: Sintetizando un mensaje UDP con `AF_INET:SOCK_RAW`
`/raw/send-udp.py`

Puedes comprobar su funcionamiento ejecutando un servidor UDP en el puerto 2000 gracias a `ncat`. En un terminal ejecuta:

```
$ ncat -l -p 2000
```

Y en otro terminal, pero en la misma máquina, ejecuta:

```
$ ./send-udp.py
```

Si todo ha ido bien, en el primer terminal debería aparecer el texto «Hello Inet».

19.4.2.1. `IP_HDRINCL`

Como has podido comprobar en el ejemplo anterior, es posible enviar un segmento sin tener que construir la cabecera IP, únicamente la UDP. Sin embargo, puede haber ocasiones en las que el programador necesite «tocar» también la cabecera IP. Eso se consigue con la opción `IP_HDRINCL`.

La ventaja respecto al socket `AF_PACKET` es doble: no hay que molestarse con la cabecera de enlace, y además el SO puede rellenar por nosotros algunos de los campos más latosos si así queremos (poniendo ceros en ellos). Esos campos son:

- El checksum.
- La dirección IP origen.
- El identificador del mensaje.
- El campo de longitud total.

Esta opción, como la gran mayoría, debe fijarse explícitamente después de crear el socket por medio del método `setsockopt()`, tal como se indica:

```
sock.setsockopt(socket.SOL_IP, socket.IP_HDRINCL, 1)
```

Esto resulta muy útil cuando quieres utilizar el socket para enviar distintos protocolos, y por tanto necesitas tener acceso al campo *proto*. Para poder hacer eso ha de crearse un socket de un *protocolo* especial identificado como `IPPROTO_RAW`:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_RAW)
```

Aunque tiene un pequeño inconveniente: no se puede leer de este tipo de socket, tendrás que crear un socket adicional para poder leer los mensajes entrantes.