
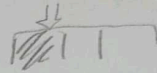
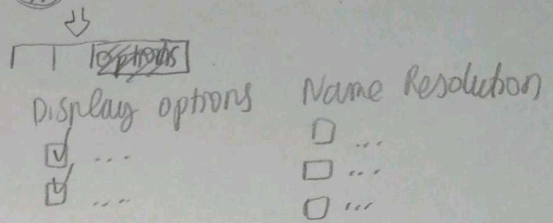


Comandos en la terminal:

- para instalar: aptk-install wireshark (algo así)
 sudo wireshark


 ← capture options



← dar click

(start)

(añadir una columna)

Edit → Appearance → Columns → 

④ File Edit View Go

↓
 Time display format → {

Framecap len (longitud que tenía los capturados)
 E52: frame len > 1000 and ip

Al exportar ☐ Exported...
☒ @ 0.5 play

②,3

Frame 1 - Total Length = (14) s, empro

E5: 92 bytes - 78B

③ Title

interval beta time (tiempo entre paquetes aunque ese no se muestre)
 displayed (tiempo entre paquetes solo los que se muestran)

BIBLIOTECA RC1 - LIBPCAP

← esta biblioteca permite capturar paquetes, filtrarlos, modificarlos y estudiarlos desde un programa C o un script de Python

- Abrir un archivo pcap:

para abrir un archivo (=traza) previamente capturado

`pcap_open_offline(filename, errbuf)`

una cadena con el nombre del archivo .pcap que se desea abrir

un bytearray donde se guardará un mensaje de error en caso de que algo haya fallado al abrir el fichero

la función devuelve un descriptor al archivo .pcap o None en caso de error

EJ: `p = pcap_open_offline('traza.pcap', errbuf)`

↑ Abre para lectura el archivo traza.pcap. En caso de error, guarda el mensaje en el bytearray errbuf

- Capturar de un interfaz:

para abrir un interfaz para captura (necesario permisos de superusuario) En VMware solo usar sudo

`pcap_open_live(device, snaplen, promisc, to_ms, errbuf)`

cadena con el nombre de la interfaz que se quiere abrir
EJ: eth0, ens33

cantidad de bytes que se quieren guardar por cada paquete. Es útil cuando no nos interesa la carga útil del paquete y así reduciríamos el tamaño de captura

• `promisc`: indica si queremos abrirlo en modo promiscuo = 1
no promiscuo = 0

• `to_ms`: duración del timeout de lectura. (polling)
Tiempo que se espera para leer varios paquetes en una misma transacción

• `errbuf`: un bytearray donde se guardará un mensaje de error en caso de que algo haya fallado al abrir la interfaz

→ return = un descriptor al archivo .pcap o None en caso de error

EJ: `p = pcap_open_live('ens33', BUFSIZ, 0, 100, errbuf)`

↑ Abre la interfaz ens33 en modo no promiscuo, capturando el paquete

BUFSIZ en su totalidad, con un timeout de lectura de 100ms. (puede que la VM le alerte sobre la imposibilidad de capturar tráfico de modo promiscuo en caso de modificar el 3º argumento, no es importante para la realización de las prácticas, acepte y continúe).

En caso de error, guarda el mensaje en el bytearray errbuf

Leer tráfico de archivo o interfaz:

`pcap_loop(descriptor, cnt, callback, user)`

• descriptor: descriptor PCAP del que queramos leer (que anteriormente hemos abierto con `open_pcap_live` o `open_pcap_offline`).

• cnt: n- paquetes a analizar (-1 para ilimitados)

• callback: una función de atención al paquete.

Esta función se ejecutará por cada paquete leído o capturado.

• user: variable auxiliar que sirve para pasar datos a la función de atención.

→ return =

- 0 si se leyó la traza entera o se superó el límite cnt
- -1 si hubo errores
- -2 si fue interrumpido por `pcap_breakloop()` (u otras)
- Otros valores si se capturó un paquete

EJ: `ret = pcap_loop(descriptor, -1, procesa-paquete, None)`

* callback(user, pkt_header, pkt_data)

Según el caso anterior, nombraremos a esta función `procesa-paquete`, con la signatura indicada.

• user: datos auxiliares de usuario que se han pasado a `pcap_loop`

• pkt_header: objeto de tipo `pcap_pkthdr` que contiene la cabecera pcap del paquete leído o capturado. Este objeto tiene tres campos:

- pkt_header.ts: objeto timestamp que contiene el tiempo de captura del paquete. A su vez, este objeto tiene dos campos:

- ts.tv-sec: timestamp del paquete en segundos

- ts.tv-sec: microsegundos dentro del segundo actual de timestamp.

OTO: este valor nunca debe ser ≥ 1000000 . Esta variable NO almacena el mismo tiempo que `tv-sec` pero en microsegundos, sino la parte fraccional del tiempo de captura.

- pkt_header.len: longitud real del paquete

- pkt_header.captlen: longitud capturada del paquete.

Esto es, pkt_data solo contendrá pkt_header.captlen bytes.

• pkt_data: un bytearray que contiene los datos del paquete en caso de éxito.

- Nota: no usar otras funciones para leer paquetes no buscadas en bucles.

EJ: `pcap_next_ex()`

`pcap_next`

- ¿Motivo? De tipo docente.

- ¿Remeetate? Evaluación nula en las prácticas.

Guardar archivo pcap ← para guardar un archivo pcap necesitamos primero crear el archivo donde vamos a ir volcando los paquetes.

pcap_open_dead(linktype, snaplen) Para ello se usan las funciones pcap_open_dead y pcap_dump_open

• linktype: tipo de enlace de los paquetes que vamos a guardar.

Típicam., para redes Ethernet: DLT_EN10MB

• snaplen: tamaño máx. que queramos guardar de cada paquete.

Típicam., para redes Ethernet: 1514 Bytes (ojo: puede haber jumboframes) ^{+ grandes} ↓

→ return = un descriptor de archivo pcap

EJ: `dexer2 = pcap_open_dead(DLT_EN10MB, 1514)`

↑ Abre un descriptor de archivo pcap para paquetes Ethernet, guardando como máx. 1514 Bytes de cada paquete.

pcap_dump_open(dexer, filename)

• dexer: descriptor de archivo pcap previam. abierto con pcap_open_dead

• filename: cadena con el nombre del archivo pcap en el que queramos guardar los paquetes.

→ return = objeto dumper que se usará para guardar paquetes

EJ: `pdumper = pcap_dump_open(dexer2, 'salida.pcap')`

↑ Crea un archivo llamado salida.pcap con las características (tipo de enlace, y tamaño máx. de paquete) de dexer2 (que indicamos en el pcap_open_dead).

Nos devuelve un objeto dumper.

Para guardar paquetes en el archivo creado con pcap_dump_open:

pcap_dump(dumper, h, sp)

• dumper: dumper devuelto por pcap_dump_open

• h: objeto de tipo pcap_pkthdr con la cabecera pcap del paquete que vamos a guardar.

• sp: bytearray con el contenido del paquete.

Se van a guardar tantos bytes como indiquemos en el campo caplen del parámetro h.

EJ: `pcap_dump(pdumper, h, pkt_data)`

↑ Guardamos en pdumper el paquete apuntado por packet con cabecera h.

German archive:

an archive:

`pcap_close(descriptor)` ← para { `pcap_open_live`
`pcap_open_offline`
`pcap_open_dead`

descriptor a cerrar

`pcap-dump-close (dumper)` ← para `pcap-dump-open`
 ↑
 dumper a cerrar

- Enviar tráfico:

```
pcap_inject(device, buf, size)
```

- descr: descriptor pcap abierto con pcap_open_live.
- buf: bytearray con el contenido de la trama a enviar.
- size: entero que representa el n-bytes que tiene la trama a enviar.

→ **return** = un entero que representa el n° bytes enviados. Para comprobar el correcto retorno de esta función, se aconseja comparar si **return** = **size**

```
EJ: ret = pcap_inject(dexer, trama, len(trama))
```

ifungig

```
sudo python3 o/practice1.py --if ens33 --debug
```

/alternativ: png nam.es