

Redes de Computadoras**Trabajo práctico N°4 - 2024****Capa de Transporte - Sockets TCP y UDP - Escaneo y reenvío de Puertos****Objetivo**

- Comunicar computadoras mediante sockets TCP y UDP.
- Analizar el intercambio de segmentos en el establecimiento y finalización de conexiones TCP, y en el intercambio de datos mediante TCP y UDP.
- Comprender el uso y utilidad de herramientas de escaneo de puertos.
- Implementar reenvío de puertos sobre un servidor NAT y comprender su utilidad.

Metodología

Trabajo individual o grupal. 2 estudiantes por grupo máximo.

Tiempo de realización: 2 clases.

Actividades obligatorias (Para aprobación con nota 7)

- Presentar en clase los programas se solicitan en las actividades 1 y 2 funcionando correctamente y subir a través de la plataforma Moodle los códigos fuente de los programas.
- Escribir y cargar a través de la plataforma Moodle un breve informe que incluya:
 - Descripción breve del funcionamiento de los programas escritos en las actividades 1 y 2, incluyendo instrucciones de ejecución.
 - Resultado de las actividades 4.4 y 4.5.
- Responder correctamente el cuestionario "Trabajo Práctico N°4 - Cuestionario".

Actividades opcionales (Para sumar puntos a la calificación)

- Realizar la actividad 5. Mostrar en clases el programa funcionando y subir el código fuente a través de la plataforma Moodle.

Materiales necesarios

- Computadoras con acceso a Internet, con sistema operativo Linux (preferentemente) o Windows.
- Algunos de los siguientes lenguajes de programación: Python versión 3 o C++.
- Escáner de puertos Nmap (usualmente instalado por defecto en Linux, también puede descargarse directamente de los repositorios de Linux. En Windows puede descargarse de su página oficial <https://nmap.org/>).

La facultad de Ingeniería provee computadoras con acceso a Internet y sistema operativo Linux Ubuntu.

Actividad 1: Sockets UDP

Escriba una aplicación que consista en un chat basado en UDP. La aplicación debe estar constituida por un solo proceso (no habrá proceso servidor ni proceso cliente). Deberán correrse en varias computadoras, al menos 2. Deberá poder comunicarse con las aplicaciones escritas por sus compañeros.

La aplicación deberá cumplir los siguientes requisitos:

- Al iniciar, los procesos deberán pedir que se ingrese un nombre de usuario.
- Los datos ingresados por teclado en cualquiera de los procesos deberán ser enviados a todas las direcciones IP de la red LAN, incluso la que los originó, siendo el puerto destino en todos los procesos el puerto UDP 60000 (60 mil). El mensaje debe enviarse con el formato:

Nombre_de_usuario:datos

De modo que los receptores puedan recuperar del mensaje recibido los datos enviados y el usuario que los envió.

- Los procesos deben esperar datos en el puerto UDP 60000 (60 mil). Los datos que lleguen deben mostrarse con el formato:

“usuario (dirección IP) dice: datos ”

Es decir, indicar el usuario que envió los datos, la dirección IP del usuario y los datos (deberá utilizar funciones para manejo de cadena de caracteres).

- Al escribir la palabra “exit” en alguno de los procesos, el proceso debe finalizar. Debe enviarse el mensaje “usuario:exit” a los demás procesos. Los demás procesos deben mostrar el mensaje:

“El usuario nombre_del_usuario (dirección IP) ha abandonado la conversación”

- Cuando un nuevo proceso se una al chat, debe enviar el mensaje usuario:nuevo, los demás procesos deben mostrar el mensaje:

“El usuario nombre_de_usuario se ha unido a la conversación”

- Cada proceso debe poder ejecutarse y poder finalizar de manera independiente de los demás, en cualquier orden de inicialización y finalización. Incluso debe poder correr un solo proceso que se comunicará solo con sí mismo.
- **Los procesos deben finalizar sin producir excepciones ni errores.**

Sugerencia: usar dos hilos, uno para el envío de datos, y otro para la recepción.

Comentarios de implementación:

Por defecto, un socket no puede enviar datos por broadcast. Debe habilitarlo para poder enviar datos a la IP de broadcast de la red. Vea el Anexo 1.

No podrá ejecutar dos procesos del chatUDP en una misma computadora, porque no podrán escuchar datos de un mismo puerto de forma simultánea. Para la etapa de implementación y debugging, puede hacer que el proceso chatUDP envíe datos a los puertos 60000 y 60001 y ejecutar otra instancia del programa escuchando en el puerto 60001.

Actividad 2: Sockets TCP

Escriba una aplicación cliente-servidor tipo chat formada por dos procesos que deben comunicarse a través de una conexión TCP.

La aplicación deberá cumplir los siguientes requisitos:

- Cuando se ingresen datos por teclado en la pantalla de uno de los procesos, los mismos se deben mostrar por pantalla, y deben ser enviados y mostrados en la pantalla del otro proceso.
- Ambos procesos deben finalizar al escribir la palabra “exit” en solo uno de los procesos (en cualquiera). **Ambos procesos deben cerrarse sin producir errores ni excepciones.**
- La aplicación deberá funcionar corriendo ambos procesos en computadoras diferentes.

Nota: la primitiva “recibir” es bloqueante, por lo que no podrá esperar datos desde un socket y escribir al mismo tiempo. Sugerencia: usar dos hilos, uno para el envío de datos, y otro para la recepción (el anexo 1 muestra como crear sockets y como trabajar con hilos en Python. El anexo 2 muestra como crear sockets y como trabajar con hilos en C++).

Actividad 3: Análisis de tráfico

Para esta actividad necesitará dos computadoras. Utilice la herramienta Wireshark para realizar análisis de tráfico. Utilice los filtros `tcp.port==numero_de_puerto` o `udp.port==numero_de_puerto` para filtrar el tráfico útil del resto del tráfico.

Verifique si con Wireshark puede ver los datos que los procesos se están enviando. ¿Un intruso podría ver los datos si logra capturar los paquetes TCP o UDP? (conteste a través de Moodle).

Actividad 4: Escaneo de puertos

Vea el apéndice 3 para detalles de uso de Nmap.

4.1 Realice un escaneo de puertos con la herramienta nmap para buscar todas las computadoras que hay en su LAN. Pruebe un escaneo ICMP y un escaneo SYN al puerto 80.

Escaneo ICMP: ***nmap -sP 192.168.0.1-254*** (Nota: utilice las IP de su red).

Escaneo SYN al puerto 80: ***sudo nmap -sS -p 80 192.168.0.1-254*** (Nota: utilice las IP de su red).

Nota: Un escaneo SYN busca todos los puertos TCP menores que 1000 (u otro valor según la implementación) de la máquina o máquinas objetivos, por lo que un escaneo de este tipo a todas las máquinas de una red podría tomar un tiempo grande. Como el objetivo de este punto del trabajo práctico es solo buscar todas las computadoras que hay en su LAN, se prueba solo con un puerto.

4.2 Elija una de las computadoras que ve en su red y analice todos los puertos TCP desde el 1 al 60000 (***sudo nmap -p 1-60000 -sS dirección_IP***).

4.3 Elija una de las computadoras que ve en su red y analice todos los puertos UDP desde el 1 al 60000 (***sudo nmap -p 1-60000 -sU dirección_IP***).

4.4 (*actividad con resultados a agregar en el informe*) Realice un escaneo buscando host al azar usando ***nmap -sS -sU --top-ports 5 -iR 100*** (siendo el número entre paréntesis la cantidad de IPs a generar aleatoriamente). Busque algún host con el puerto ssh o http (no https) abierto (debe decir “open”, no debe decir “filtered” o “closed”). Si no encuentra ningún hosts, intente nuevamente con un número de IP aleatorias mayor. Seleccione un objetivo y detecte su ubicación geográfica (en el trabajo práctico N°3 se presentaron varios mecanismos para localizar geográficamente una IP).

4.5 (*actividad con resultados a agregar en el informe*) Realice un escaneo a todas las IPs de Corea del Norte. Indique:

- El número de IPs encontradas.
- Alguna IP con servicio ssh.
- Alguna IP con un servicio web (http o https).

En sistemas Linux, con “cat /etc/services | grep servicio” puede saber el número de puerto de algún servicio (por ejemplo, “cat /etc/services | grep ssh” indicará el puerto utilizado por el servicio ssh).

Para que una máquina ofrezca algún servicio, debe tener un proceso asociado al puerto esperando peticiones. Los procesos usualmente se llaman como el servicio ofrecido más la letra “d” (por ejemplo, un router ospf ejecutará un proceso llamado

ospfd o demonio ospf asociado a algún puerto). Nota: este escaneo puede demorar varios minutos.

Actividad 5: Transferencia de archivos sobre sockets TCP (Opcional)

Escriba una aplicación cliente-servidor formada por dos procesos que deben comunicarse a través de una conexión TCP. El proceso servidor, luego de recibir la conexión del proceso cliente, debe transferirle un archivo. El proceso cliente debe recibir el archivo y almacenarlo localmente.

El usuario de la aplicación servidor debe ingresar por teclado el nombre del archivo a transferir.

Podría ser de ayuda el uso de librerías para manejo de archivos como “os” en Python o “filesystem” en C++. Investigue su uso.

Anexo 1: Implementación de Sockets e hilos en Python 3

Sockets en Python 3

Librerías necesarias para implementar sockets:

import socket

socket.socket(familia, tipo)

Funcionamiento: Crea un objeto de la clase socket.

Argumentos:

- familia: dominio de operación. Puede valer:
 - socket.AF_INET: Internet sobre IPv4.
 - socket.AF_INET6: Internet sobre IPv6.
 - socket.AF_UNIX: Dentro de un mismo sistema de archivos.
- type: tipo de socket:
 - socket.SOCK_STREAM: Lee o escribe datos desde o hacia flujos o buffers. Los procesos leen o escriben en esos flujos, los socket toman los datos escritos en esos flujos y los envían de manera independiente a la forma en que los procesos leen o escriben. Compatible con TCP.
 - socket.SOCK_DGRAM: Lee o escribe mensajes. Compatible con UDP.

Valor de retorno: objeto tipo socket

Ejemplo de uso:

```
socket_server=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

socket_server.bind((string dirección IP,int puerto))

Enlaza el socket a una dirección IP y un número de puerto. Necesario para el extremo que espera conexiones en sockets TCP. También necesario en extremos que esperan peticiones UDP (servidores UDP).

Si necesita que el sistema operativo asigne una dirección IP automáticamente, indique 0.0.0.0

Ejemplo de uso: socket_server.bind(('192.168.1.39',1501))

Nota 1: para saber su IP, ejecute "ifconfig" o "ip a", según versión de Linux.

Nota 2: asigne puertos libres a sus sockets. Para verificar qué puertos están siendo utilizados, ejecute "nmap dirección-IP -p 1-60000".

socket_server.listen(5)

Indica al proceso que escuche conexiones entrantes en el puerto indicado. Su argumento es el número máximo de conexiones que pueden estar pendientes de ser aceptadas.

(socket_cliente, address) = socket_server.accept()

Bloquea el proceso hasta que se recibe una conexión remota. Si previamente se ejecutó listen, la conexión remota puede haberse producido y estar en espera de ser aceptada. Cuando se recibe una conexión remota, crea un nuevo socket.

Argumentos: ninguno.

Valor de retorno: tupla que contiene el nuevo objeto tipo socket que se mantiene conectado con el socket que pidió la conexión y dirección (IP y puerto) del socket que pidió la conexión.

socket_id.connect((string direccion_IP,int puerto))

Funcionamiento: Conecta con un socket en una computadora remota (puede ser la misma computadora). Asigna una IP y puerto disponible en el socket local.

Argumentos: dirección IP y puerto al que se desea conectar.

Ejemplo de uso: socket_id.connect(("192.168.1.39",1500))

socket_cliente.recv(tamaño)

Funcionamiento: Lee datos desde el buffer de recepción del socket TCP.

Argumentos: cantidad máxima de bytes a recibir.

Valor de retorno: cadena de bytes recibidos. Si se desea convertir a una cadena de caracteres, es necesario codificar al formato adecuado.

Ejemplo de uso: buffer_entrada = socket_cliente.recv(100).decode("utf-8")

socket_id.send(buffer_salida)

Funcionamiento: Envía datos a través de un socket TCP conectado a un socket remoto.

Argumentos: cadena de bytes a enviar. Si se desea enviar una cadena de caracteres, se debe codificar la misma en bytes.

Ejemplo de uso: socket_Id.send(buffer_salida.encode())

socket.recvfrom(tamaño)

Recibe datos en un socket UDP.

Argumento: cantidad máxima de caracteres a recibir.

Valor de retorno: tupla que contiene los datos recibidos y las direcciones IP y de puerto del proceso que envió los datos.

Ejemplo, si se ejecuta `buffer=nombre_socket.recvfrom(1024)`, entonces, con `buffer[0].decode()` podrá acceder a los datos, y con `buffer[1]` a la IP y puerto origen.

`socket.sendto(buffer_envio,(IP destino,Puerto destino))`

Envía datos sobre un socket UDP.

Argumentos: `buffer` donde están los datos a ser enviados, y las direcciones IP y puerto a los cuales se enviarán los datos.

`socket.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)`

Habilita el envío de datos por broadcast.

Manejo de hilos en Python 3

`import threading`

Importa el módulo `threading`, que incluye las funciones indicadas abajo.

`hilo1 = threading.Thread(target=function, args=(argumento1,argumento2,))`

Crea un objeto de la clase `Thread`, que permitirá manejar un hilo. La función “function” define el código del hilo. La misma debe ser escrita por el programador. El hilo no comienza a correr al ser creado. Debe llamarse el método `start()`.

`hilo1 = threading.Thread(target=function)`

Igual que la anterior, pero la función “function” no tiene argumentos.

`hilo1 = threading.Thread(target=function, args=(argumento1,), daemon=True)`

Igual que las anteriores, pero declara el hilo como demonio.

`hilo1.start()`

Comienza la ejecución del hilo.

`hilo1.join()`

Bloquea el proceso o hilo desde donde es disparada la ejecución del `hilo1`, hasta que el hilo “hilo1” termina su funcionamiento.

Otras funciones útiles en Python 3

print("texto ejemplo", variable, "texto ejemplo",)

Imprime por pantalla el texto entre paréntesis. Pueden incluirse variables y expresiones matemáticas.

input()

Toma una cadena de datos desde teclado. Devuelve como valor de retorno la cadena leída. Es una función bloqueante.

Apéndice 2: Implementación de Sockets e hilos en C y C++

Sockets en C++

Librerías necesarias:

```
#include <sys/socket.h>
```

```
#include <netinet/in.h>
```

```
#include <unistd.h>
```

```
#include <stdlib.h>
```

```
#include <arpa/inet.h>
```

En Windows necesita la librería winsock.h.

int socket(int domain, int type, int protocol);

Crea un socket. Sus argumentos son:

- domain: dominio de operación. Puede valer:
 - AF_INET: Internet sobre IPv4.
 - AF_INET6: Internet sobre IPv6.
 - AF_UNIX: Dentro de un mismo sistema de archivos.
- type: tipo de socket:
 - SOCK_STREAM: Lee o escribe datos desde o hacia flujos o buffers. Los procesos leen o escriben en esos flujos, los socket toman los datos escritos en esos flujos y los envían de manera independiente a la forma en que los procesos leen o escriben. Compatible con TCP.
 - SOCK_DGRAM: Lee o escribe mensajes. Compatible con UDP.
- protocol: tipo de protocolo.
 - 0: permite que el sistema seleccione el protocolo por defecto (recomendado).

Retorna el identificador del socket creado o -1 si hubo un error.

Ejemplo de uso:

```
socket_servidor = socket(AF_INET, SOCK_STREAM, 0);  
if(socket_servidor<0){  
    cout<<"nError creando socket";  
    exit(1);  
}
```

int bind(int identificador, struct sockaddr *local_addr, socklen_t addr_length)

Enlaza el socket a una dirección IP y un número de puerto. Sus argumentos son:

- **identificador:** identificador del socket al cual se desea asignar una dirección IP y un puerto.
- ***local_addr:** puntero a una estructura donde se almacena la información sobre el IP y puerto a asignar al socket.
- **addr_length:** tamaño de la estructura mencionada arriba (No se pasa la estructura como argumento, sino la dirección de memoria donde comienza la estructura, por lo que debe indicarse el tamaño).

Valor de retorno: 0 si la función se ejecutó correctamente. -1 si hubo algún error.

Ejemplo de uso:

```
if(bind(socket_servidor, (struct sockaddr *) &servidor_addr, sizeof(servidor_addr))<0){  
    cout<<"nError .....";  
}
```

Estructura sockaddr

Antes de ejecutar *bind*, es necesario llenar los campos de la estructura *servidor_addr*.

La misma puede llenarse de la siguiente manera:

```
struct sockaddr_in servidor_addr;  
servidor_addr.sin_family = AF_INET;  
servidor_addr.sin_addr.s_addr = inet_addr("129.5.24.1")  
servidor_addr.sin_port = htons(1500);
```

int listen(int identificador_socket, int queue_size);

Indica al proceso que escuche conexiones entrantes en el puerto indicado. Sus argumentos son:

- **identificador_socket:** Identificador del socket.
- **queue_size:** Número máximo de conexiones que pueden estar pendientes de ser aceptadas.

Retorna 0 si la ejecución fue exitosa, y -1 si hubo error.

Ejemplo de uso:

```
if(listen(socket_servidor,5)<0){  
    cout<<"nError en funcion bind";  
    .....  
}
```

int accept(int identificador_socket, struct sockaddr *remote_host, socklen_t addr_length)

Bloquea el proceso hasta que se recibe una conexión remota (la conexión remota puede haberse producido y estar en espera de ser aceptada). Cuando se recibe una conexión remota, crea un nuevo socket. Sus argumentos son:

- **identificador_socket**: socket en el que se esperará una conexión entrante.
- ***remote_host**: puntero a una estructura en la cual se almacenará la IP y puerto del proceso cliente que pide conectarse.
- **addr_length**: tamaño de la estructura mencionada anteriormente.

Retorna el identificador del nuevo socket, el mismo permanecerá conectado con el cliente hasta que se ejecute la primitiva **close(identificador_socket)**;

Ejemplo de uso:

```
cliente_len = sizeof(cliente_addr);  
new_socket= accept(socket_servidor, (struct sockaddr *) &cliente_addr, (socklen_t *)  
&cliente_len);
```

La estructura *cliente_addr* será escrita por la función *accept* con los datos del cliente que pidió la conexión. Tiene el mismo formato (*sockaddr*) que la estructura empleada por la función *bind()*. Sus campos pueden consultarse para conocer quien se conectó de la siguiente manera:

```
cout<<"nIP: "<<inet_ntoa(cliente_addr.sin_addr);  
cout<<"nPort: "<<ntohs(cliente_addr.sin_port);
```

int connect(int identificador_socket, struct sockaddr *remote_host, socklen_t addr_length)

Conecta un socket con un puerto en una computadora remota (puede ser la misma computadora). Sus argumentos son:

- **identificador_socket**: socket a través del cual se realizará el pedido de conexión.
- ***remote_host**: puntero a una estructura en la cual se encuentra almacenada la IP y puerto del proceso servidor con el cual se desea conectar.
- **addr_length**: tamaño de la estructura mencionada anteriormente.

Retorna 0 si la conexión fue exitosa, y -1 si hubo un error.

Ejemplo de uso:

```
if(connect(socket_id, (struct sockaddr *) &remote_server_addr, (socklen_t)
sizeof(remote_server_addr))<0){
    cout<<"Error funcion Connect";
    exit(1);
}
```

La estructura *remote_host* debe ser previamente escrita de la misma manera que en la función bind.

int write(identificador_socket, void *buffer, size_t tamaño);

Envía datos a través de un socket. Sus argumentos son:

- *identificador_socket*: socket a través del cual se enviarán datos.
- *buffer*: puntero del buffer donde están los datos a enviar (el buffer debe ser tipo *char*).
- *tamaño*: tamaño del buffer donde están los datos a enviar.

Retorna el número de bytes enviados o -1 si hubo un error.

Ejemplo de uso:

```
if(write(identificador_socket, "Hello, world!\n", 13)<0){
    cout<<"Error enviando datos\n";
}
```

int read(identificador_socket, void *buffer, size_t tamaño);

Lee datos desde el buffer de recepción. Sus argumentos son:

- *identificador_socket*: socket desde el cual se leen datos.
- *buffer*: puntero del buffer donde se escribirán los datos leídos.
- *tamaño*: tamaño del buffer donde están los datos a enviar.

Retorna el número de bytes leídos o -1 si hubo un error.

Ejemplo de uso:

Manejo de hilos en C++

Debe incluir la librería ***thread*** e incluir el flag ***-pthread*** cuando compile.

Crear un hilo. Debe incluir el código a ejecutar por el hilo en una función que no devuelva ningún valor de retorno (tipo ***void***), y reciba todos los argumentos que sean necesarios. Luego, para crear y ejecutar el hilo (Note también que se está utilizando el constructor de la clase ***thread***):

```
thread nombre_hilo;  
nombre_hilo=thread(nombre_funcion, argumento1, argumento2);
```

argumento1 y argumento2 son argumentos que se pasan a la función que define el hilo.

Sincronización de hilos: Existen muchos métodos para sincronizar hilos. En este práctico será de gran utilidad (siempre necesario) el método **join()**.

```
if(hilo_1.joinable()==true){  
    hilo_1.join();  
}
```

El hilo en el cual se ejecute la instrucción **join()** esperará a que el **hilo_1** finalice su ejecución. Note que no se puede ejecutar **join()** si **hilo_1** ya ha finalizado su ejecución, por eso primero se debe verificar si **hilo_1.joinable()** es igual a **true**.

Apéndice 3: Nmap

Nmap o network map (mapeador de redes) es una poderosa herramienta (incluso controvertida, ya que un uso indebido puede considerarse una actividad ilegal) para escaneo de puertos que permite detectar equipos conectados a una red, detectar puertos abiertos, características de las máquinas escaneadas, y otras opciones. Es muy útil como herramienta de diagnóstico o en auditorías de seguridad, ya que un puerto abierto es un riesgo de seguridad, debido a que un usuario malintencionado, o incluso un malware (usualmente una botnet) podría usar esos puertos para intentar conectarse o loguearse a la máquina, o simplemente recopilar información de una máquina o una red. Un puerto cerrado también puede proveer información útil a un atacante, ya que la forma de responder puede brindar información acerca del sistema operativo o características del sistema. Por otro lado, el simple hecho de que un puerto responda permite saber que el puerto puede no estar protegido por un firewall u otro tipo de barrera de seguridad. Por este motivo, el escaneo de puertos suele ser la primera etapa en un ataque.

Nmap envía distintos tipos de paquetes de prueba o sondas a los puertos indicados y espera una respuesta para verificar su estado. El estado de los puertos puede ser:

- Abierto: es un puerto en el cual hay una aplicación escuchando (esperando conexiones o paquetes).
- Cerrado: no posee ninguna aplicación asociada esperando datos.
- Filtrado: Indica que el puerto no responde. Esto usualmente ocurre porque alguna barrera de seguridad (firewall) impide que a dicho puerto puedan llegar paquetes, o que el puerto responda.

Instalación

Muchas versiones de Linux ya lo tienen instalado por defecto. Si no la posee, puede instalarse con “sudo apt-get install nmap”. Para instalar nmap en Windows, descárguelo desde <https://nmap.org/book/inst-windows.html> (seleccione instalar también Nping, lo usaremos más adelante) Deberá realizar algunas modificaciones al registro del sistema de Windows, tal como se indica en la página indicada.

Escaneo de puertos simple

comando: **nmap rango_de_IPs**

Dicho comando escaneará si están abiertos los puertos de 0 a 1000 del rango de IPs indicado.

El rango de direcciones IPs a verificar (**rango_de_IPs**) se expresa como: $x_2-x_1.x_2-x_1.x_2-x_1.x_2-x_1$ donde x_2 y x_1 representan los valores máximos y mínimos de cada dígito de las direcciones IP en las cuales desea buscar host y puertos. Por ejemplo, para buscar si hay host con IPs en el rango desde 192.168.1.1 hasta 192.168.1.40, deberá usar 192.168.1.1-40, o para buscar si hay hosts con IPs en el rango desde 192.168.0.1 hasta 192.168.30.254 deberá usar 192.168.0-30.1-254. (Nota: hay más opciones para especificar los rangos de host, consulte la ayuda de nmap para más información).

La opción **-v** (*nmap -v rango_de_IPs*) significa verbosity level, es decir, “explicar con más palabras lo que hace”. La opción **-vv** incrementa el nivel de verbosidad (si es la primera vez que usa nmap, pruebe todas opciones y vea la diferencia).

Si indica una sola IP en lugar de un rango, escaneará solo los puertos de ese host.

Especificación de puertos a escanear

Para escanear un rango específico de puertos de un host agregue a nmap el siguiente parámetro: **-p puerto_inicial-puerto_final**

Si no se indica, nmap solo escanea puertos en el rango 0-1000. Rangos de uso común van desde 0 a 60000. Puede escanear los puertos de una sola IP o de un rango de IPs (la última opción puede requerir mucho tiempo).

Ejemplo: *nmap -p 0-60000 192.168.1.1-254* escaneará los puertos desde el 0 al 60000 en los hosts indicados.

La opción **--top-ports N** indica a nmap que busque los N puertos más comunes. Por ejemplo *--top-ports 10*, buscará en los 10 puertos más comunes (80, 21, 22, 110, etc.).

Diferentes técnicas de escaneo

nmap utiliza por defecto paquetes de sondeo SYN, y espera una respuesta SYN-ACK, pero no envía el tercer paquete ACK necesario para establecer una conexión TCP (repase protocolo TCP si no recuerda estos conceptos).

Debido a que el escaneo de puertos suele ser el primer paso antes de un ataque, muchos firewall e IDSs (Intrusion Detection System) utilizan filtros para bloquear (no permitir el tránsito) de paquetes usualmente empleados en el escaneo de puertos. Incluso pueden devolver paquetes ICMP como “host unreachable” o “time to live exceeded” para encubrir la acción del firewall o IDS y engañar al atacante. Por tal motivo, nmap emplea diferentes tipos de técnicas de escaneo para intentar sortear estas barreras de protección. Se resumen a continuación algunos.

Escaneo con paquetes SYN

Envía paquetes SYN (ver protocolo TCP) y espera la respuesta ACK-SYN de puertos abiertos, o RST de puertos cerrados. En caso de detectar un puerto abierto, no envía el paquete ACK que permitiría completar una conexión TCP, en su lugar enviará el paquete RST. Para realizar este sondeo, debe agregar el parámetro **-sS**.

Ejemplo: `nmap -sS -p rango_de_puertos ip_host` (o rango de IPs)

Puede requerir permisos de superusuario (advertencia QUITTING!).

Escaneo con intentos de conexión con primitiva connect() de TCP (parámetro -sT)

Utiliza la primitiva connect() de la interfaz socket. Se debe usar el parámetro -sT. Es fácil de detectar por firewalls.

Ejemplo: `nmap -v -sT ip_host` (o rango de IPs)

Sondeo UDP (parámetro -sU)

Intenta recibir respuesta desde puertos a los que usualmente hay servicios que utilizan protocolo UDP, como DNS o DHCP.

Sondeo de puertos filtrados y no filtrados o sondeo ACK (-sA)

Envía paquetes TCP ACK. Todo puerto alcanzable, abierto o cerrado, responderá con un RST indicando que el mensaje ACK no era esperado. Solo los puertos no alcanzables no responderán, como aquellos protegidos por un firewall. Este sondeo no tiene el propósito de detectar puertos, sino de analizar si algún posible firewall no permite paquetes dirigidos a algún puerto determinado. La respuesta será cuantos puertos están filtrados y cuantos no.

Detección del sistema operativo

-O: habilita detección del sistema operativo enviando varias sondas y analizando las respuestas (cada sistema operativo responde de manera diferente). --osscan-guess: Habilita la detección agresiva del sistema operativo.

Nivel de intensidad

Opción --version-intensity, asigna el tipo de sondas a emplear. Varía de 0 a 9. Mientras más alto el número, mayor cantidad de sondas usadas, por lo tanto, mayor probabilidad de encontrar servicios.