# PRÁCTICA CALIFICADA Nº 3
## Administración de Redes

Fecha: 9 de noviembre de 2024
Tiempo de duración: 3:00 horas

Realice la siguiente actividad "**Programando sockets**" paso a paso y escriba sus conclusiones y responda las preguntas del documento. De ser necesario incluya imágenes que Ud. considere necesarias para validar su práctica.

**Notas:**

***pip3 install python-socketio***
***pip3 install aiohttp***

WebSockets es una tecnología que proporciona comunicación en tiempo real entre un cliente y un servidor a través de una conexión TCP, eliminando la necesidad de que los clientes verifiquen continuamente si los puntos finales API tienen actualizaciones o contenido nuevo. Los clientes crean una única conexión a un servidor WebSocket y esperan escuchar nuevos eventos o mensajes del servidor.

La principal ventaja de WebSockets es que son más eficientes porque reducen la carga de la red y envían información en forma de mensajes a una gran cantidad de clientes. Entre las principales características de WebSockets, podemos destacar las siguientes:

- Proporcionan comunicación bidireccional (dúplex completo) a través de una únicaconexión TCP.
- Proporcionan comunicación en tiempo real entre un servidor y sus clientes conectados. Esto permite la aparición de nuevas aplicaciones orientadas a la gestión de eventos de forma asincrónica.
- Proporcionan simultaneidad y mejoran el rendimiento, optimizando los tiempos de respuesta y dando como resultado aplicaciones web más confiables.

**Implementación de un servidor con socket.io**

**Para implementar un servidor basado en socket.io, necesitamos introducir otros módulos como asyncio y aiohttp:**

- asyncio es un módulo de Python que nos ayuda a realizar la programación concurrente de un solo hilo en Python. Está disponible en Python 3.7; la documentación se puede encontrar en https://docs.python.org/3/library/asyncio.html.
- aiohttp es una biblioteca para crear aplicaciones cliente y servidor integrados en asyncio. El módulo utiliza las ventajas de WebSockets de forma nativa para comunicarse entre diferentes partes de la aplicación de forma asincrónica. La documentación está disponible en http://aiohttp.readthedocs.io/en/stable.

Revisa el código **web_socket_server.py** donde implementamos un servidor socket.io usando el framework aiohttp, el cual, en un nivel bajo, usa asyncio.

En el código anterior, implementamos un servidor basado en socket.io que usa el módulo aiohttp. Como puedes ver en el código, hemos definido dos métodos: el método index (), que devolverá un mensaje de respuesta basado en la solicitud del punto final raíz "/", y el método print message (), que imprime el identificador del socket y los datos emitidos por el evento. Este método está anotado con @ socketio.on ("message"). Esta anotación hace que la función escuche eventos de tipo mensaje, y cuando estos eventos ocurren, actuará sobre esos eventos. En nuestro ejemplo, el mensaje es el tipo de evento que hará que se llame a la función print_message ().

Escribimos el siguiente código para el cliente en **web_socket_client.py.**

Si revisamos el código anterior, estamos usando el método connect() del socketio. La claseClient () para conectarse al servidor que está escuchando en el puerto 8080. Definimos dosmétodos, uno para conectar y otro para desconectar.

Para llamar a la función print_message () en el servidor, necesitamos emitir el evento de mensaje y pasar los datos como un diccionario de objetos. Para ejecutar los dos scripts anteriores, necesitamos ejecutar dos terminales por separado, uno para el cliente y otro para el servidor. Primero, debe ejecutar el servidor y luego ejecutar el cliente para verificar la información enviada como mensaje.

**Programación en Socket**

Los sockets son los componentes principales que nos permiten explotar las capacidades del sistema operativo para interactuar con la red. Puedes considerar los sockets como un canalde comunicación punto a punto entre un cliente y un servidor. Los sockets de red son una forma sencilla de establecer contacto entre procesos en

las mismas máquinas o en diferentes. El concepto de socket es muy similar al uso de descriptores de archivos para sistemas operativos UNIX. Los comandos como read () y write () para trabajar con archivos tienen un comportamiento similar al manejo de sockets. Una dirección de socket para una red consta de una dirección IP y un número de puerto. El objetivo de un socket es comunicar procesos a través de la red.

**Sockets de red en Python**

La comunicación entre diferentes entidades en una red se basa en el concepto de socket clásico desarrollado por Python. Un socket se especifica mediante la dirección IP de la máquina, el puerto que está escuchando y el protocolo que utiliza. La creación de un socket en Python se realiza mediante el método socket.socket ().

La sintaxis general del método de socket es la siguiente:

*s = socket.socket (socket_family, socket_type, protocolo = 0)*

La sintaxis anterior representa las familias de direcciones y el protocolo de la capa de transporte. Según el tipo de comunicación, los sockets se clasifican de la siguiente manera:

- Sockets TCP (socket. SOCK STREAM)
- Sockets UDP (socket. SOCK DGRAM).

La principal diferencia entre TCP y UDP es que TCP está orientado a la conexión, mientras que UDP no está orientado a la conexión.

Los sockets también se pueden clasificar por familia. Las siguientes opciones están disponibles:

- Sockets UNIX (socket. AF UNIX), que se crearon antes de la definición de red y sebasan en datos
- Socket. AF INET para trabajar con el protocolo IPv4
- Socket.AF INET6 socket para trabajar con el protocolo IPv6

Los tipos y funciones necesarios para trabajar con sockets se pueden encontrar en Python en el módulo socket. El módulo de socket proporciona todas las funcionalidades necesarias para escribir rápidamente clientes y servidores TCP y UDP.

Revisa los métodos de cliente y servidor,

**Cliente básico con el módulo socket**

Podemos comenzar a probar cómo enviar y recibir datos desde un sitio web. Una vez establecida la conexión, podemos enviar y recibir datos utilizando los métodos send () y recv () para comunicaciones TCP. Para la comunicación UDP, podríamos usar los métodossendto () y recvfrom () en su lugar. Veamos cómo funciona esto con el código **socket_data**.py.

Hasta ahora, hemos analizado los métodos disponibles en el módulo de socket para el lado del cliente y del servidor e implementamos un cliente básico. Ahora nos movemos para aprender cómo podemos implementar un servidor basado en el protocolo HTTP.

**Implementando un servidor HTTP en Python**

Conociendo los métodos que hemos revisado anteriormente, podríamosimplementar nuestro propio servidor HTTP. Para esta tarea, podríamos usar el método bind (), que acepta la dirección IP y el puerto como parámetros. El módulo desocket proporciona el método listen (), que te permite poner en cola hasta un máximode n solicitudes. Por ejemplo, podríamos establecer el número máximo de solicitudes en 5 con la declaración mysocket.listen (5).

Comprueba que en el código http_server.py, usamos localhost para aceptar conexiones desde la misma máquina. El puerto podría ser 80, pero como necesita privilegios de root, usaremos uno mayor o igual a 8080.

Si queremos probar el servidor HTTP, podríamos crear otro script que nos permita obtener la respuesta enviada por el servidor que hemos creado. Puede encontrar el siguiente código en el archivo **testing_http_server.py**.

**¿Qué sucede?**

Implementación de una shell inverso con sockets

Un shell inverso es una acción mediante la cual un usuario obtiene acceso al shell de un servidor externo. Por ejemplo, si estás trabajando en una fase de pentesting (https://www.cisco.com/c/en/us/products/security/what-is-pen-testing.html) posterior a la explotación y te gustaría crear un script que se invoca en ciertos escenarios y obtendrás un shell para acceder al sistema de archivos de otra máquina, podríamos construir nuestro propio shell inverso en Python.

Hacemos esto con el código **reverse_shell.py**.

En el código anterior, usamos módulos de subprocess y os. Desde el módulo de

socket, estamos usando el método sock.connect () para conectarnos a un host correspondiente a una determinada dirección IP y puerto especificados (en nuestro caso es localhost). Una vez que hayamos obtenido el shell, podríamos obtener un listado de directorios usando el comando /bin / ls, pero primero necesitamos establecer la conexión a nuestro socket a través de la salida del comando. Logramos esto con la instrucción os.dup2 (sock.fileno ()). Para ejecutar el script y obtener un shell inverso con éxito, necesitamos lanzar un programa que esté escuchando la dirección y el puerto anteriores.

Ejercicio: ejecuta la aplicación llamada netcat con el comando ncat -l -v -p 45679, indicando el puerto que declaramos en el script, para obtener un shell inverso en la dirección localhostusando el puerto 45679.

**Resolución de dominios IPS, direcciones y administración de excepciones**

La mayoría de las aplicaciones cliente-servidor actuales, como los navegadores, implementan la resolución de nombres de dominio (DNS) para convertir un dominio en una dirección IP. El sistema de nombres de dominio fue diseñado para almacenar una base dedatos descentralizada y estructurada jerárquicamente, donde se almacenan las relacionesentre un nombre y su dirección IP.

**Recopilación de información con sockets**

El módulo socket nos proporciona una serie de métodos que pueden sernos útiles en el caso de que necesitemos convertir un nombre de host en una dirección IP y viceversa.

El siguiente script es un ejemplo de cómo podemos utilizar estos métodos para obtener información de los servidores DNS de Google. Puedes encontrar el siguiente código en el archivo socket_methods.py:

En el código anterior, usamos el módulo de socket para obtener información sobre los servidores DNS de un dominio y una dirección IP específicos.

En el resultado, podemos ver cómo estamos obteniendo servidores DNS, un nombre completo y direcciones IPv4 e IPv6 para un dominio específico. Es un proceso sencillo para obtener información sobre el servidor que está trabajando detrás de un dominio.

**Usando el comando de búsqueda inversa**

Las conexiones a Internet entre computadoras conectadas a una red se realizan usando direcciones IP. Por lo tanto, antes de que comience la conexión, se realiza una traducción del nombre de la máquina a su dirección IP. Este proceso se denomina Resolución DNS Directa y nos permite asociar una dirección IP con un nombre de dominio. Para hacer esto, podemos usar el socket. Método gethostbyname (hostname).

La resolución inversa es la que nos permite asociar un nombre de dominio con una dirección IP específica. El comando reverse lookup obtiene el nombre de host de la dirección IP. Para esta tarea, podemos usar el método gethostbyaddr (). En el script socket_reverse_lookup.py obtenemos el nombre de host de la dirección IP de 8.8.8.8.

Si la dirección IP es incorrecta, la llamada al método gethostbyaddr () arrojará una excepción con el mensaje "Error for resolving ip address: [Errno -2]
Name or service not known".

**Administrar excepciones de socket**

Cuando estamos trabajando con el módulo de sockets, es importante tener en cuenta que puede ocurrir un error al intentar establecer una conexión con un host remoto porque el servidor no está funcionando o se está reiniciando. Se definen diferentes tipos de excepciones en la biblioteca sockets de Python para diferentes errores. Para manejar estas excepciones, podemos usar los bloques try y accept.

El siguiente ejemplo te muestra cómo manejar las excepciones. Puede encontrar el siguiente código en el archivo **manage_socket_errors.py**:

Explica los resultados que aparecen cuando se ejecuta el archivo anterior.

**Escaneo de puertos con sockets**

De la misma forma que contamos con herramientas como Nmap (https://nmap.org/ ) para analizar los puertos que tiene abiertos una máquina, con el módulo de socket podríamos implementar una funcionalidad similar para detectar puertos abiertos para luego detectar vulnerabilidades en un servicio que se encuentre abierto en dicho servidor.

**Implementación de un escáner de puerto básico**

Los sockets son el bloque de construcción fundamental para la comunicación de red, y al llamar al método connect_ex (), podemos probar fácilmente si un puerto en particular está abierto, cerrado o filtrado. Por ejemplo, podríamos implementar una función que acepte como parámetros una dirección IP y una lista de puertos, y devuelve para cada puerto si está abierto o cerrado.

En el siguiente ejemplo, estamos implementando un escáner de puertos usando módulos socket y sys. Usamos el módulo sys para salir del script con la instrucción sys.exit () y devolver el control al intérprete en caso de un error de conexión.

Utilizamos el siguiente código en el archivo **check_ports_socket.py**.

Si ejecutamos el script anterior, podemos ver cómo comprueba cada puerto en localhost y devuelve una dirección IP o dominio específico, independientemente de si está abierto o cerrado. El primer parámetro puede ser una dirección IP o un nombre de dominio, porque el módulo de socket puede resolver una dirección IP de un dominio y un dominio de una dirección IP.

¿Qué sucede si ejecutamos la función con una dirección IP o nombre de dominio que no existe?

Explica el funcionamiento de sock.settimeout().

El siguiente código de Python te permite buscar puertos abiertos en un host local o remoto. El script busca puertos seleccionados en una determinada dirección IP ingresada por el usuario y refleja los puertos abiertos al usuario. Si el puerto está bloqueado, también revela el motivo, por ejemplo, como resultado de una conexión de tiempo de espera.

```
import socket import sys
from datetime import datetime import errno

remoteServer = input("Ingresa un host remoto para escanear: ")
remoteServerIP = socket.gethostbyname(remoteServer)

print("Ingresa el rango de puertos que te gustaria escanear en la

maquina")startPort = input("Ingresa un puerto de inicio: ")

endPort = input("Ingresa un puerto final: ")

print("Espera, escaneando host remoto", remoteServerIP)
```

*time_init = datetime.now()*

….

El código anterior que se guarda en **socket_port_scanner.py** se realizará un escaneo en cada uno de los puertos indicados. Para hacer esto, estamos usando el método connect_ex () para determinar si está abierto o cerrado. Si ese método devuelve un 0 como respuesta, el puerto se clasifica como Open. Si devuelve otro valor de respuesta, el puerto se clasifica como Closed y se muestra el código de error devuelto.

**Escáner de puertos avanzado**

El siguiente script de Python nos permitirá escanear una dirección IP con las funciones portScanning y socketScan. El programa busca puertos seleccionados en un dominio específico resuelto a partir de la dirección IP ingresada por el usuario por parámetro. En el siguiente script, el usuario debe introducir como parámetros obligatorios el host y un puerto, separados por una coma:

*python3 socket_advanced_port_scanner.py -h.*

En el código anterior, podemos ver el programa principal donde obtenemos los parámetrosde host obligatorios y los puertos para ejecutar el script. Cuando se han recopilado estos parámetros, llamamos al método portScanning, que resuelve la dirección IP y el nombre de host. Luego llamamos al método socketScan, que usa el módulo socket para evaluar el estado del puerto.

La principal ventaja de implementar un escáner de puertos es que podemos realizar solicitudes a un rango de direcciones de puertos de servidor en un host para determinar los servicios disponibles en una máquina remota.

**Implementación de un cliente TCP simple y un servidor TCP**

Aquí el concepto detrás del desarrollo de esta aplicación es que el servidor de socket es responsable de aceptar conexiones de clientes desde una dirección IP y un puerto específicos. Implementando un servidor y un cliente con sockets En Python, se puede crear un socket que actúa como cliente o servidor.

Los sockets de cliente son responsables de conectarse contra un host, puerto y protocolo en particular. Los sockets del servidor son responsables de recibir las conexiones del cliente en un puerto y protocolo en particular.

**Implementando el servidor TCP**

Presentamos un servidor (tcp_server.py) TCP multihilos. El socket del servidor abre un socket TCP en localhost 9998 y escucha las solicitudes en un bucle infinito. Cuando el servidor recibe una solicitud del socket del cliente, devolverá un mensaje que indica que se ha establecido una conexión desde otra máquina.

**Implementando el cliente TCP**

El socket del cliente (**tcp_client.py**) abre el mismo tipo de socket que ha creado el servidor y envía un mensaje al servidor. El servidor responde y finaliza su ejecución, cerrando el cliente de socket. Aquí configuramos un servidor HTTP en la dirección 127.0.0.1 a través del puerto estándar 9998. El cliente se conectará a la misma dirección IP y puerto para recibir 1024 bytes de datos en la respuesta y los almacenará en una variable llamada buffer, para luego mostrar esa variable al usuario.

Si revisas el código anterior, la instrucción s.connect ((host, puerto)) conecta al cliente con el servidor, y el método s.recv (1024) recibe los mensajes enviados por el servidor.

**Implementación de un cliente UDP simple y un servidor UDP**

Escribamos una aplicación donde un servidor escucha todas las conexiones y mensajes a través de un puerto específico e imprime en la consola cualquier mensaje que se haya intercambiado entre el cliente y el servidor.

UDP es un protocolo que está al mismo nivel que TCP, es decir, por encima de la capa IP. Ofrece un servicio en modo desconectado a las aplicaciones que lo utilizan. Este protocolo es adecuado para aplicaciones que requieren una comunicación eficiente que no tiene que preocuparse por la pérdida de paquetes.

Las aplicaciones típicas de UDP son la telefonía por Internet y la transmisión de video. El

encabezado de una trama UDP se compone de cuatro campos:

- El puerto de origen UDP
- El puerto de destino UDP.
- La longitud del mensaje UDP
- checkSum contiene información relacionada con el campo de control de errores.

La única diferencia entre trabajar con TCP y UDP en Python es que al crear el socket en UDP, debe usar SOCK_DGRAM en lugar de SOCK_STREAM. La principal diferencia entreTCP

y UDP es que UDP no está orientado a la conexión, y esto significa que no hay garantía de que nuestros paquetes lleguen a sus destinos y no hay notificación de error si falla una entrega.

Ahora vamos a implementar la misma aplicación que hemos visto antes para pasar mensajes entre el cliente y el servidor. La única diferencia es que ahora vamos a utilizar el protocolo UDP en lugar de TCP. Vamos a crear un servidor UDP síncrono, lo que significa que cada solicitud debe esperar hasta el final del proceso de la solicitud anterior. El método bind () se utilizará para asociar el puerto con la dirección IP. Para recibir el mensaje, usamos el recvfrom () y sendto () métodos para enviar.

**Implementando el servidor UDP**

La principal diferencia con la versión TCP es que UDP no tiene control sobre los errores en los paquetes que se envían entre el cliente y el servidor. Otra diferencia entre un socket TCP y un socket UDP es que debe especificar SOCK_DGRAM en lugar de SOCK_ STREAM al crear el objeto socket.

Revisa el código **udp_server.py**.

En el código anterior, vemos que socket.SOCK_DGRAM crea un socket UDP, y la instrucción data, addr = s.recvfrom (buffer) devuelve los datos y la dirección de la fuente.

**Implementando el cliente UDP**

Para comenzar a implementar el cliente, necesitaremos declarar la dirección IP y el puerto donde está escuchando el servidor. Este número de puerto es arbitrario, pero debe asegurarse de que está usando el mismo puerto que el servidor y que no está usando un puerto que ya haya sido tomado por otro proceso o aplicación:

*SERVER_IP = "127.0.0.1"*
*SERVER_PORT = 6789*

Una vez establecidas las constantes anteriores para la dirección IP y el puerto, llega el momento de crear el socket a través del cual estaremos enviando nuestro mensaje UDP al servidor:

*clientSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)*

Y finalmente, una vez que hemos construido un nuevo socket, es hora de escribir el código que enviará nuestro mensaje UDP:

```
address = (SERVER_IP ,SERVER_PORT)
socket_client.sendto(bytes(message,encoding='utf8'),address)
```

Revisa el código **udp_client.py**.

En el código anterior, estamos creando una aplicación cliente basada en el protocolo UDP.

Para enviar un mensaje a una dirección específica, usamos el método sendto () y para recibir un mensaje de la aplicación del servidor, usamos el método recvfrom ().

Finalmente, es importante considerar que si intentamos usar SOCK_STREAM con el socket UDP, probablemente tenemos un error: socket.error.

Por lo tanto, es importante recordar que tenemos que usar el mismo tipo de socket para el cliente y el servidor cuando estamos construyendo aplicaciones orientadas a pasar mensajes con sockets.