UNIVERSIDAD PERUANA CAYETANO HEREDIA

Actividad 13 : TCP/IP

Objetivos

- Comprender los principios básicos de las redes de computadoras, incluyendo las capas OSI y TCP/IP.
- Familiarizarse con los diferentes protocolos de red, como ARP, ICMP, IGMP y NDP.
- Entender los protocolos de aplicación comunes, como HTTP, FTP, SMTP, POP, IMAP, Telnet y SSH.
- Conocer el funcionamiento de los servicios de correo electrónico y los agentes de entrega y recuperación de correo
- Comprender el papel de TCP y UDP en la capa de transporte y el proceso de handshake de tres vías en TCP.
- Aprender sobre el control de flujo y la multiplexación en la capa de transporte para una comunicación eficiente.
- Familiarizarse con los protocolos de seguridad como SSL/TLS y SSH, así como con la autenticación y el cifrado de datos.
- Entender los conceptos de certificados SSL/TLS, certificados autofirmados y X.509.
- Conocer el funcionamiento de DHCP para la asignación dinámica de direcciones IP.
- Aprender sobre la configuración de direcciones IP estáticas y dinámicas, así como sobre NAT y enrutamiento de red.
- Entender el funcionamiento del protocolo de internet (IP) y la asignación de prefijos de direcciones IP.
- Conocer los protocolos de túneles y la capa de internet para la transferencia de datos entre redes.

Aspectos básicos de la actividad

Sean los siguientes conceptos dados en clase:

5-4-3 rule Anonymous FTP Application layer ARP ARP table Connection DHCP Dynamic IP address Email-delivery agent Email-retrieval agent Flow control FTP HTTP Header Header extension ICMP IGMP IMAP Internet layer Interoperability IP address prefix Keep alive LDAP Link layer Multiplexing MIME MTU NAT NDP Netmask POP Port R-utilities Self-signed certificate Sequence number SFTP SSH SMTP SSL/TLS Stateless Address Autoconfiguration Static IP address TCP three-way handshake Telnet TTL Transport layer Tunnel X.509 certificate [1].

Problema 1: Diseño de un sistema de entrega y recuperación de correo electrónico

Contexto: Una empresa necesita diseñar un sistema de correo electrónico robusto que utilice SMTP, IMAP, y SSL/TLS para la entrega y recuperación segura de correo electrónico.

Requisitos:

[1] Puedes utilizar el internet o tus notas para verificar los conceptos dados

- Desarrollar un diagrama de red que muestre cómo se integrarían SMTP, IMAP, y SSL/TLS en la infraestructura existente.
- Escribir un pseudocódigo para la configuración del servidor SMTP y IMAP que también maneje conexiones SSL/TLS.
- Explicar cómo se gestionarán los certificados X.509 en este sistema y la importancia de estos para SSL/TLS.
- Discutir cómo se manejarán las direcciones IP dinámicas y estáticas dentro de esta red, especialmente en relación con DHCP y NAT.

Para tu presentación y código a presentar puedes utilizar:

Requisitos técnicos y diseño:

Configuración del Servidor SMTP y IMAP:

1. Implementaremos servidores SMTP y IMAP usando librerías en Python que simulen el comportamiento de estos protocolos. Manejo de SSL/TLS:

2. Utilizaremos SSL/TLS para encriptar las conexiones SMTP e IMAP, garantizando la confidencialidad y la integridad de los datos transmitidos. Integración con DHCP y NAT:

3. Discutiremos cómo las direcciones IP dinámicas asignadas por DHCP y la traducción de direcciones realizada por NAT pueden afectar la configuración y el funcionamiento de los servidores de correo.

Paso 1: Configuración del servidor SMTP y IMAP en Python

Vamos a usar la biblioteca smtplib para SMTP y imaplib para IMAP. Aquí hay un ejemplo de cómo se podrían configurar estos servidores:

```
import smtplib from email.mime.text

import MIMEText



def setup_smtp_server():

    server = smtplib.SMTP_SSL('smtp.example.com', 465)

server.login('user@example.com', 'password') return

server def send_email(server):

    msg = MIMEText('This is a test email.')
    msg['Subject'] = 'SMTP SSL Test'

    msg['From'] = 'user@example.com'
```

```
    msg['To'] = 'recipient@example.com'
    server.send_message(msg)
    server.quit() smtp_server =
setup_smtp_server()
send_email(smtp_server) import
imaplib


def setup_imap_server():
    mail = imaplib.IMAP4_SSL('imap.example.com')
    mail.login('user@example.com', 'password')
    return mail


def fetch_emails(mail): mail.select('inbox') result, data
    = mail.search(None, 'ALL') mail_ids = data[0] id_list
    = mail_ids.split() latest_email_id = id_list[-1] result,
    data = mail.fetch(latest_email_id, '(RFC822)')
    raw_email = data[0][1] print(raw_email.decode('utf-
    8')) mail.logout()
imap_server = setup_imap_server()
fetch_emails(imap_server)
```

Paso 2: Implementación de SSL/TLS

Para implementar SSL/TLS en estos servidores, utilizamos la opción de conexión segura en smtplib (SMTP_SSL) y imaplib (IMAP4_SSL). Esto garantiza que todas las comunicaciones entre el cliente y el servidor están encriptadas.

Paso 3: Manejo de Certificados X.509

Los certificados X.509 se utilizan para validar la identidad del servidor. Los detalles sobre cómo se manejan estos certificados, especialmente en relación con la creación y el almacenamiento, serían cruciales.

import ssl context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)

context.load_cert_chain(certfile="server_cert.pem",

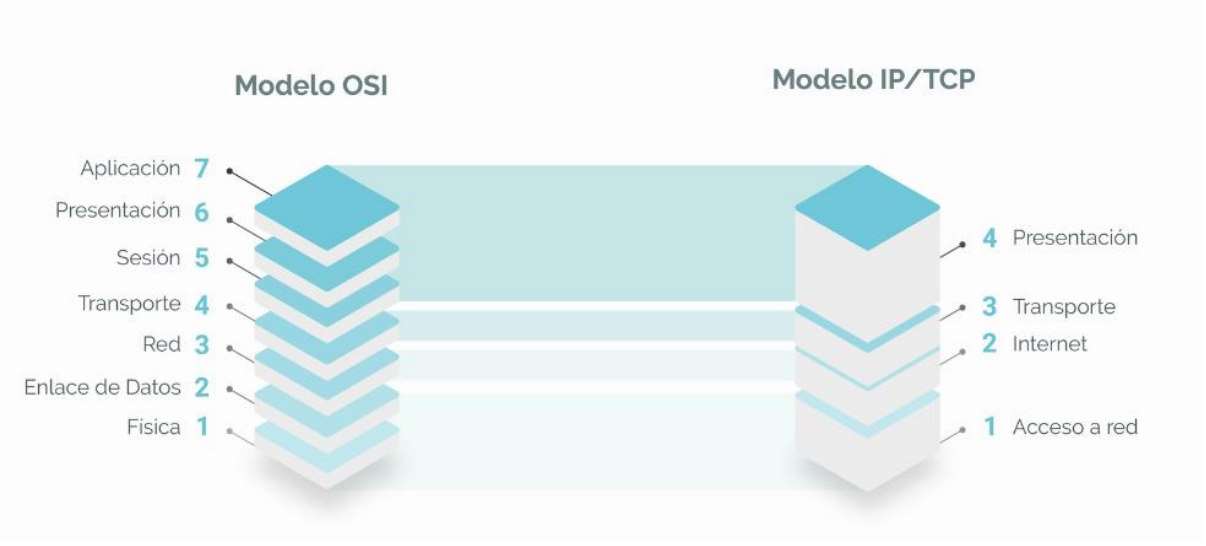
keyfile="server_key.pem")

Paso 4: Discusión sobre DHCP y NAT

Es importante discutir cómo la asignación de direcciones IP dinámicas a través de DHCP y la traducción de direcciones IP realizada por NAT pueden afectar el acceso y la visibilidad de los servidores de correo desde el exterior de la red local. Es posible que se necesiten configuraciones de NAT estático o el uso de servicios DNS dinámicos para garantizar que los servidores sean accesibles consistentemente.

Problema 2: Implementación de un protocolo de red personalizado sobre TCP

Contexto: Diseñar un protocolo de aplicación personalizado para un sistema de archivos distribuido que se ejecutará sobre TCP, utilizando técnicas como multiplexación y control de flujo.



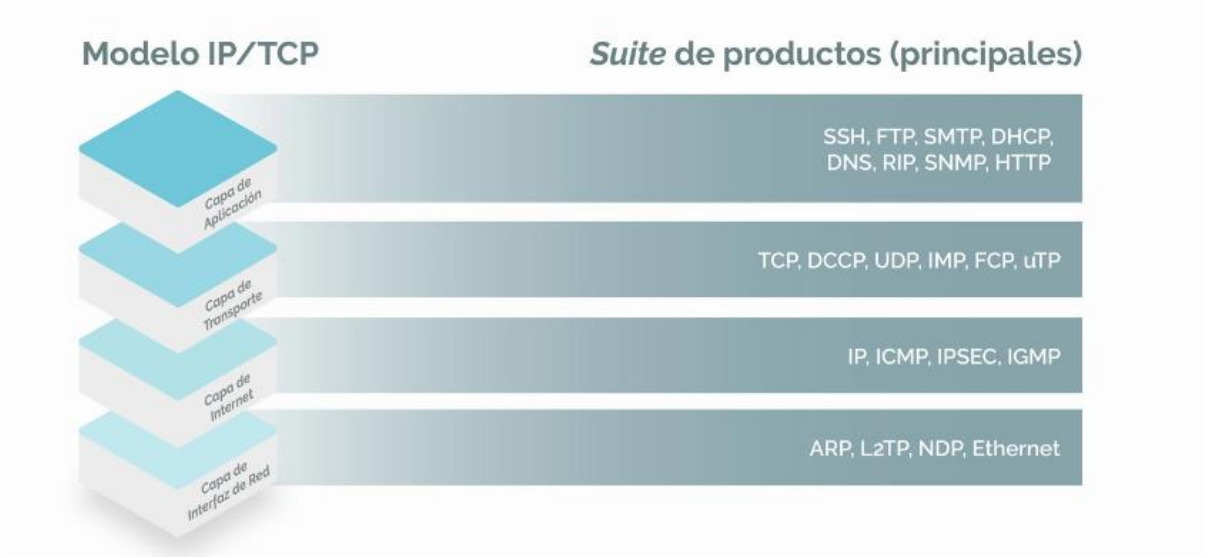
Requisitos:

- Definir el formato del mensaje, incluyendo cabeceras y extensiones de cabecera.
- Desarrollar un esquema de control de flujo que gestione eficazmente la transferencia de archivos grandes a través de redes con alta latencia.
- Escribir un pseudocódigo para las funciones de conexión, como el handshake de tres vías de TCP, y cómo se manejará la retransmisión.

- Evaluar el uso de NAT y su impacto en las conexiones de red en este protocolo, particularmente cuando se utilizan direcciones IP dinámicas.

Para tu presentación y código a presentar puedes utilizar:
Objetivos:

1. Diseño del protocolo: Definir el formato del mensaje, incluidas las cabeceras y las extensiones de cabecera para manejar funcionalidades específicas como control de flujo y recuperación de errores.
2. Implementación de control de flujo: Desarrollar un esquema de control de flujo para manejar eficazmente la transferencia de datos sobre TCP.
3. Simulación con Python: Simular el protocolo utilizando Python para evaluar su rendimiento y robustez en escenarios de red simulados.



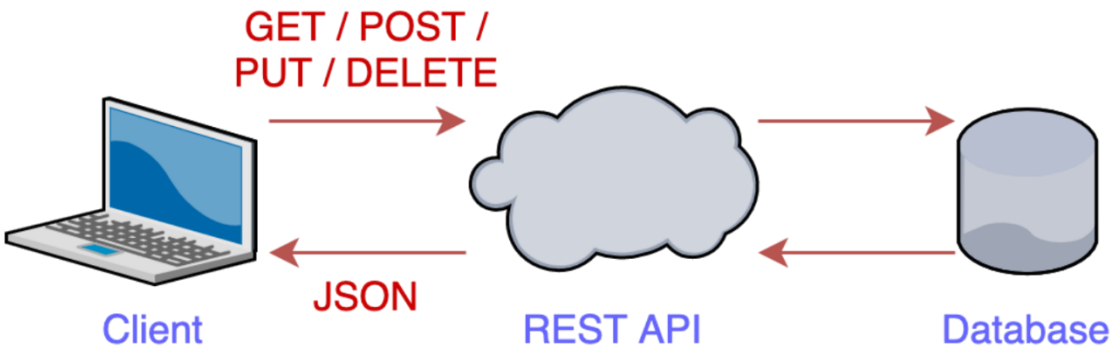
Paso 1: Diseño del protocolo

Vamos a definir un protocolo simple que incluya operaciones básicas como PUT, GET, y DELETE para interactuar con archivos en el sistema distribuido.

Ejemplo de Especificación del Protocolo:

- PUT: Enviar un archivo al sistema.
- GET: Recuperar un archivo del sistema.
- DELETE: Eliminar un archivo del sistema.

Cada mensaje tendrá una cabecera que incluye el tipo de operación, el tamaño del mensaje, y un número de secuencia para el control de flujo y la recuperación de errores.



Paso 2: Implementación de control de flujo

Usaremos un mecanismo de control de flujo basado en ventana deslizante para asegurar la entrega fiable y eficiente de los archivos, especialmente en redes con alta latencia.

Código Python para simulación del protocolo:

```
import socket

import struct


def send_message(sock, msg_type, seq_num, data):

    header = struct.pack('!I I', msg_type, seq_num)

    message = header + data.encode()

    sock.sendall(message)


def receive_message(sock):

    header = sock.recv(8)

    msg_type, seq_num =

    struct.unpack('!I I', header) data

    = sock.recv(1024) # ajustar

    según el tamaño esperado del
```

```
    mensaje return msg_type,

    seq_num, data.decode()


def main(): host =

    'localhost' port =

    12345


    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    server.bind((host, port))

    server.listen(1) print("Server

    listening on port", port)


    client_sock, addr = server.accept()

    print("Connected by", addr)


    # Simulación de recepción de un mensaje msg_type,

    seq_num, data = receive_message(client_sock)

    print("Received:", msg_type, seq_num, data)


    # Envío de una respuesta

    send_message(client_sock, 1, seq_num + 1, "Ack")


    client_sock.close()
    server.close()
```

```
if __name__ == '__main__':
    main()
```

CODIGO MEJORADO

```
import socket
import struct

# Definición de constantes para tipos de mensaje
MSG_PUT = 1
MSG_GET = 2
MSG_DELETE = 3

# Definición de constantes para estados del control de flujo
STATE_WAITING_ACK = 0
STATE_WAITING_DATA = 1

def send_message(sock, msg_type, seq_num, data):
    # Empaqueta el mensaje con la cabecera y lo envía
    header = struct.pack('!I I', msg_type, seq_num)
    message = header + data.encode()
    sock.sendall(message)

def receive_message(sock):
    # Recibe un mensaje y lo desempaqueta
```

```
    header = sock.recv(8)
    msg_type, seq_num = struct.unpack('!I I', header)
    data = sock.recv(1024)  # ajustar según el tamaño esperado del mensaje
    return msg_type, seq_num, data.decode()

def main():
    host = 'localhost'
    port = 12345

    # Configuración del servidor TCP
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind((host, port))
    server.listen(1)
    print("Server listening on port", port)

    # Acepta la conexión entrante
    client_sock, addr = server.accept()
    print("Connected by", addr)

    # Inicialización del control de flujo
    state = STATE_WAITING_DATA
    expected_seq_num = 0

    # Bucle principal para la comunicación
```

```
while True:
    if state == STATE_WAITING_DATA:
        # Espera recibir un mensaje GET
        msg_type, seq_num, data = receive_message(client_sock)
        if msg_type == MSG_GET and seq_num == expected_seq_num:
            print("Received GET:", seq_num, data)
            # Simulación de procesamiento de la solicitud GET
            response_data = f"File content for {data}"
            send_message(client_sock, MSG_GET, seq_num + 1, response_data)
            expected_seq_num += 1
        else:
            print("Error: Unexpected message or sequence number")

    elif state == STATE_WAITING_ACK:
        # Espera recibir un mensaje ACK
        msg_type, seq_num, data = receive_message(client_sock)
        if msg_type == MSG_ACK and seq_num == expected_seq_num: # type: ignore
            print("Received ACK:", seq_num)
            state = STATE_WAITING_DATA
        else:
            print("Error: Unexpected message or sequence number")

# Cierra la conexión
client_sock.close()
```

```
    server.close()


if _name_ == 'main':
    main()
```

Paso 3: Evaluación del protocolo

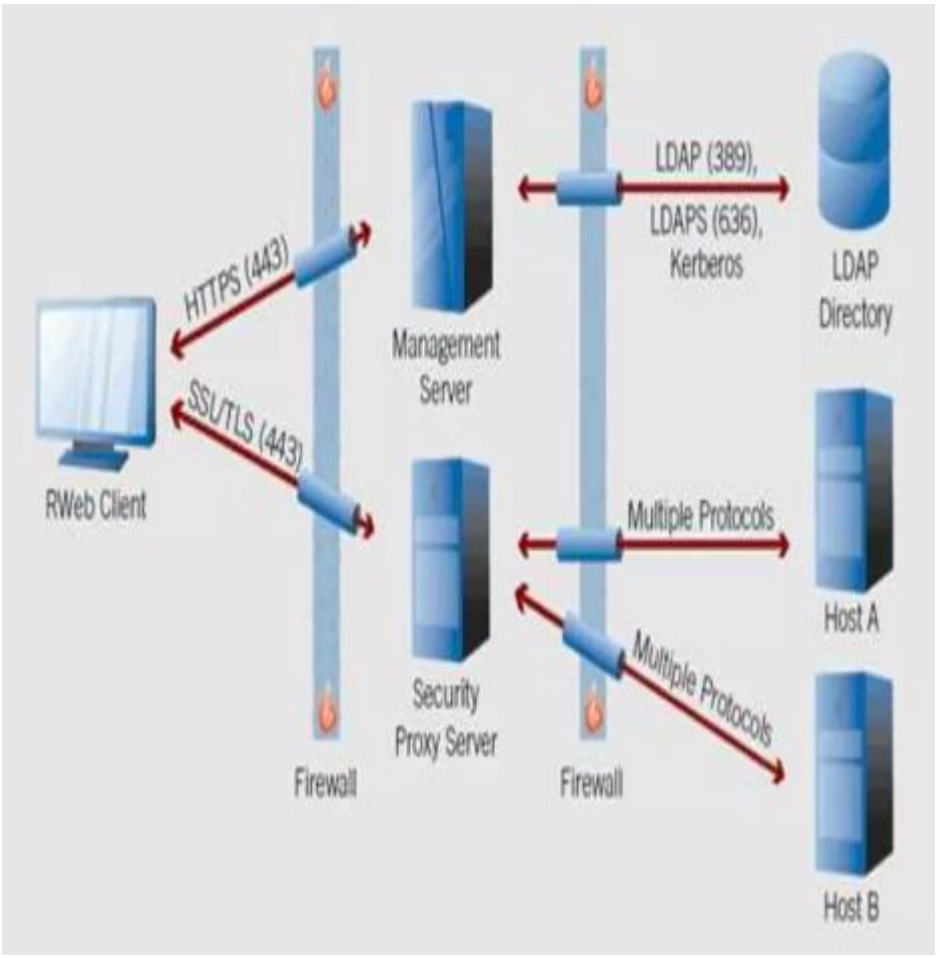
Después de implementar el protocolo, usaríamos herramientas como Wireshark para monitorear la eficacia del control de flujo y el manejo de errores durante la transferencia de archivos. Esto podría involucrar la simulación de condiciones de red adversas, como alta latencia y pérdida de paquetes, para ver cómo el protocolo se comporta y se recupera de estos problemas.

Problema 3: Creación de un sistema de autenticación segura con LDAP y SSH

Contexto: Una organización requiere un sistema de autenticación segura que utilice LDAP para la gestión de identidades y SSH para el acceso remoto.

Requisitos:

- Diagramar cómo LDAP y SSH pueden integrarse para proporcionar un sistema de autenticación y autorización.
- Discutir la implementación de un túnel SSH que pueda encapsular la comunicación LDAP, incluyendo detalles sobre la configuración de port forwarding.
- Escribir un pseudocódigo que ilustre cómo se puede implementar un certificado autofirmado y cómo LDAP maneja la autenticación.
- Analizar el papel de la capa de transporte en este sistema, con énfasis en los detalles de SSL/TLS para la protección de datos.



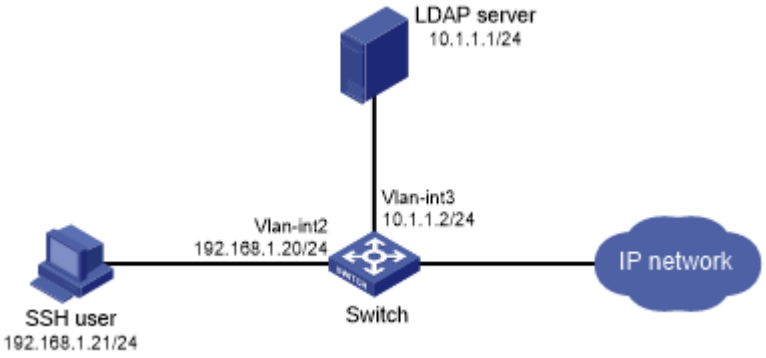
Para tu presentación y código a presentar puedes utilizar:

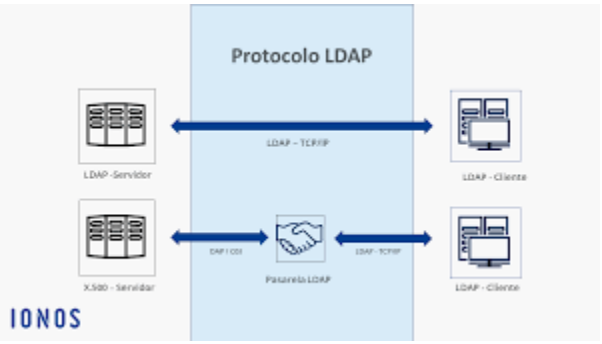
Objetivos:

1. Integración de LDAP y SSH: Configurar un servidor LDAP y permitir el acceso seguro a través de SSH, incluyendo el reenvío de puertos y la encapsulación.
2. Implementación de Seguridad: Utilizar técnicas de cifrado y autenticación, incluyendo el manejo de certificados autofirmados.
3. Simulación con Python: Implementar scripts en Python que simulan la configuración y operación de este sistema de autenticación.
4. Evaluación de Seguridad: Discutir cómo evaluar la seguridad del sistema y manejar posibles vulnerabilidades.

Paso 1: Configuración de LDAP y SSH

Vamos a configurar un servidor LDAP y proporcionar acceso a él a través de un túnel SSH seguro. Usaremos Python para simular el establecimiento de una conexión SSH con reenvío de puertos.





Código python para la configuración de SSH y LDAP:

```
import paramiko


def create_ssh_tunnel(user, password, host, remote_host, local_port, remote_port):
    client = paramiko.SSHClient()
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    client.connect(host, username=user, password=password)


    # Establecer un reenvío de puertos para LDAP
    tunnel = client.get_transport().open_channel('direct-tcpip', (remote_host, remote_port), ('localhost', local_port)) return client, tunnel


def main():
```

```
    ssh_user = 'admin' ssh_password

    = 'securepassword' ssh_host =

    'example.com' ldap_host =

    'ldap.example.com'

    local_ldap_port = 389

    remote_ldap_port = 389



    client, tunnel = create_ssh_tunnel(ssh_user, ssh_password, ssh_host, ldap_host,
local_ldap_port, remote_ldap_port) print(f"SSH tunnel established for

    LDAP on port {local_ldap_port}")

    # Aquí se simularían operaciones LDAP a través del túnel #

    Por ejemplo, consultas LDAP, autenticación de usuarios, etc.



    tunnel.close()

    client.close()



if __name__ == '__main__':

    main()
```

Paso 2: Implementación de seguridad

Usaremos SSL/TLS para cifrar la comunicación LDAP. Además, configuraremos certificados autofirmados para asegurar la comunicación entre el cliente y el servidor LDAP a través del túnel SSH.

Código Python para la generación de certificados autofirmados:

```
from OpenSSL import crypto
```


```
def create_self_signed_cert(cert_file, key_file):
    k = crypto.PKey()
    k.generate_key(crypto.TYPE_RSA, 2048)

    cert = crypto.X509()
    cert.get_subject().C = "US"
    cert.get_subject().ST = "California"
    cert.get_subject().L = "San Francisco"
    cert.get_subject().O = "My Company"
    cert.get_subject().OU = "My
    Organizational Unit"
    cert.get_subject().CN =
    "mydomain.com"
    cert.set_serial_number(1000)
    cert.gmtime_adj_notBefore(0)
    cert.gmtime_adj_notAfter(10*365*24*6
    0*60)
    cert.set_issuer(cert.get_subject())
    cert.set_pubkey(k) cert.sign(k,
    'sha256')

    open(cert_file, "wt").write(crypto.dump_certificate(crypto.FILETYPE_PEM,
cert).decode('utf-8'))
```

```
    open(key_file, "wt").write(crypto.dump_privatekey(crypto.FILETYPE_PEM, k).decode('utf-
8'))


create_self_signed_cert('ldap_cert.pem', 'ldap_key.pem')
```

Paso 3: Evaluación de seguridad

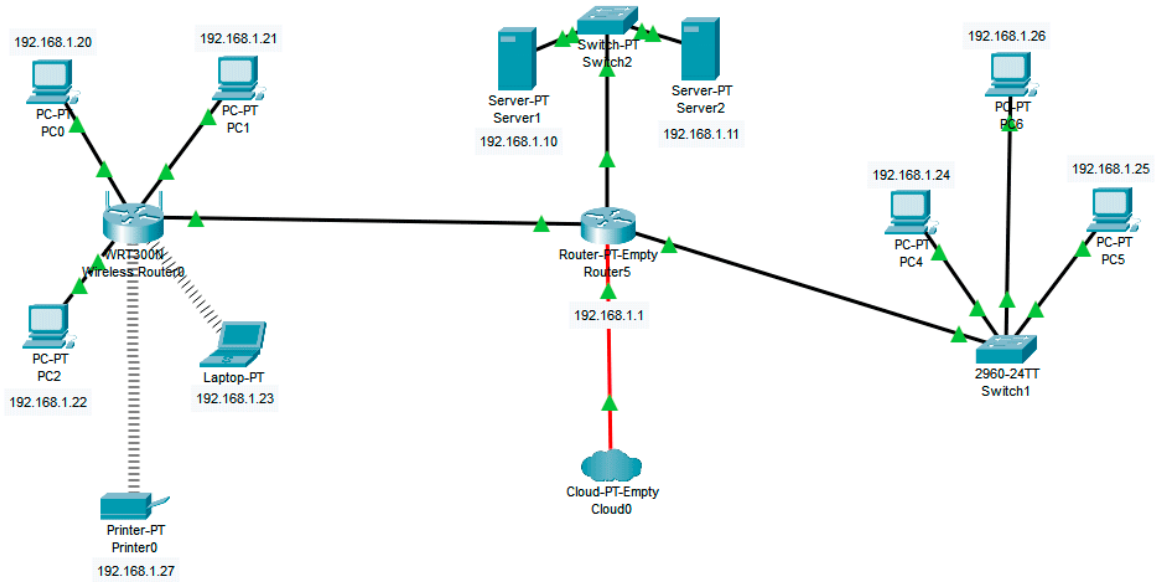
Discutir cómo evaluar la seguridad del sistema implementado, abordando potenciales vulnerabilidades como ataques de intermediario y configuraciones erróneas de certificados. La evaluación incluirá pruebas de penetración y revisión de configuraciones.

Problema 4: Simulación de interoperabilidad de red con múltiples protocolos

Contexto: Simular un entorno de red que utilice múltiples protocolos de la pila TCP/IP, asegurando la interoperabilidad entre dispositivos que utilizan diferentes configuraciones de red.

Requisitos:

- Crear un escenario que implique IP, ICMP, IGMP, y ARP, describiendo cómo cada uno afecta la operación y gestión de la red.
- Desarrollar un plan para la simulación de esta red, incluyendo la configuración de la tabla ARP, el manejo de ICMP para la detección de errores, y la utilización de IGMP para la gestión del tráfico multicast.
- Proponer métodos para probar y validar la interoperabilidad de estos protocolos en diferentes segmentos de la red.
- Discutir cómo se podrían usar las utilidades de red (R-utilities) para monitorear y resolver problemas en esta red.



Para tu presentación y código a presentar puedes utilizar:

Objetivos:

1. Simulación de protocolos de Red: Crear un entorno simulado que incluya IP, ICMP, IGMP, y ARP para observar y gestionar la comunicación entre dispositivos.
2. Implementación de la simulación en Python: Utilizar Python para crear scripts que simulen el envío y recepción de paquetes utilizando estos protocolos.
3. Evaluación de interoperabilidad: Analizar cómo estos protocolos interactúan y gestionan la conectividad, especialmente en escenarios de alta densidad de red y tráfico multicast.
4. Uso de R-utilities para monitoreo y resolución de problemas: Implementar y discutir cómo herramientas como traceroute, ping y otras utilidades pueden ayudar a diagnosticar y resolver problemas en la red.

Paso 1: Simulación de protocolos de red en Python

Usaremos Python para simular la generación y el manejo de paquetes de red para IP, ICMP, IGMP y ARP. Utilizaremos la biblioteca scapy, que es muy potente para manipular y enviar paquetes de red.

Código Python para la simulación de protocolos:

```
from scapy.all import *



def simulate_ip(): packet = IP(dst="192.168.1.1") / ICMP() / "Hello, this

    is an IP packet" send(packet)



def simulate_icmp(): icmp_echo = IP(dst="192.168.1.1") /

    ICMP(type=8, code=0) / "Ping" send(icmp_echo)



def simulate_igmp():
    igmp_packet = IP(dst="224.0.0.1") / IGMP(type=0x16, gaddr="224.0.0.1")

    send(igmp_packet)
```

```
def simulate_arp(): arp_request =
    ARP(pdst='192.168.1.2')
    send(arp_request)


simulate_ip()
simulate_icmp()
simulate_igmp()
simulate_arp()
```

Paso 2: Evaluación de interoperabilidad

Después de simular cada protocolo, observaremos cómo los paquetes son gestionados y dirigidos a través de la red. Esto ayudará a identificar problemas de interoperabilidad y eficiencia en la comunicación entre dispositivos.

Paso 3: Uso de R-utilities para diagnóstico

Implementaremos scripts para utilizar R-utilities como traceroute y ping, y simular cómo se pueden usar para monitorear y diagnosticar problemas en la red.

Código Python para Utilizar R-utilities:

```
import os


def run_traceroute(target):
    response = os.system(f"traceroute {target}")
    print(response)
def run_ping(target): response =
    os.system(f"ping -c 4 {target}")
    print(response)
```


```
run_traceroute('192.168.1.1')

run_ping('192.168.1.1')
```

Presentación:

Parte 1: Prepara una presentación para resolver los anteriores problemas:

Estructura de la exposición:

1. Introducción:

- Saludo y presentación del presentador/es.
- Breve descripción del objetivo de la exposición y los temas que se cubrirán.
- Importancia de entender los conceptos de redes de computadoras en el contexto actual de la tecnología de la información.

2. Desarrollo de los temas:

- Fundamentos de Redes: Explicación de los conceptos básicos como ARPANET, backbone, Bluetooth, broadcast, cache memory, checksum, etc.
- Seguridad y fiabilidad de redes: Discusión sobre temas como encriptación, firewalls, TCP/IP, etc.
- Infraestructura de redes: Descripción de los componentes de una red, topologías, direccionamiento MAC, dispositivos de red, etc.
- Protocolos y comunicación: Explicación de protocolos como Ethernet, TCP/IP, OSI, así como los diferentes tipos de redes y protocolos de enrutamiento.

3. Estudios de caso:

- Análisis de estudios de caso reales relacionados con problemas de redes de computadoras y cómo se resolvieron.
- Discusión sobre lecciones aprendidas y mejores prácticas identificadas en cada caso.

4. Preguntas y respuestas:

- Sesión interactiva donde los asistentes pueden hacer preguntas sobre los temas tratados.
- Responder preguntas y proporcionar aclaraciones adicionales sobre los conceptos presentados.

6. Conclusión:

- Resumen de los puntos clave cubiertos durante la exposición.
- Reflexión sobre la importancia de comprender los conceptos de redes de computadoras en la era digital.
- Agradecimiento a los asistentes y cierre de la exposición.

Consejos para una exposición exitosa:

1. Conocimiento profundo: Asegúrate de comprender completamente los temas que vas a presentar y estar preparado para responder preguntas.

2. Claridad y concisión: Explica los conceptos de manera clara y sencilla, evitando tecnicismos innecesarios.
3. Ejemplos prácticos: Utiliza ejemplos prácticos y casos de uso para ilustrar los conceptos teóricos.
4. Interacción con la audiencia: Fomenta la participación de la audiencia haciendo preguntas o invitándolos a compartir sus experiencias.
5. Material visual atractivo: Utiliza diapositivas, gráficos o demostraciones visuales para mantener el interés de la audiencia.

Parte 2: Entrega de la actividad completa

Elementos a entregar:

1. Código fuente de la implementación del protocolo de red en un repositorio público en GitHub.
2. Presentación detallada en formato PDF, que incluya los objetivos, la metodología, los resultados y las conclusiones de la actividad y la presentación corregida.
3. En la presentación y en el repositorio, indicar claramente que se ha trabajado en equipo en el desarrollo del proyecto.

Fecha de entrega: 28 de abril hasta las 23:59

La actividad completa debe ser entregada en los repositorios individuales de los estudiantes en la fecha indicada.