Computer Network

AmirReza Azari
99101087

گزارش پروژه درس شبکه‌های کامپیوتری

مقدمات

در این گزارش قصد داریم نحوه انجام پروژه درس شبکه‌های کامپیوتری را بیان نماییم. ابتدا توضیحات و تعریف پروژه را داریم و سپس مراحل را ذکر خواهیم کرد.

هدف در این پروژه پیاده سازی سرور و کلاینتی است که از طریق DNS بتوانند محدودیت های اعمال شده بر شبکه را دور بزنند. برای شروع پروژه VPN مبتنی بر DNS که شامل پیاده‌سازی یک کلاینت و سرور است، می‌توانیم از زبان Python استفاده کنیم.

همانطور که در صورت پروژه ذکر شده است، خواسته اصلی این پروژه این است که ما کد کلاینت و سروری پیاده سازی کنیم که بتواند محدودیت این شبکه را دور بزند. به طور دقیق‌تر کد کلاینت باید روی هر کدام از کامپیوترهای داخلی این شبکه قابلیت اجرا داشته باشد و وظیفه آن اتصال به کامپیوتری است که در این شبکه امکان ارتباط با بیرون را دارد. کد سرور روی این کامپیوتر خاص اجرا می شود و ظیفه دارد درخواست‌های کلاینت ها را به بیرون از شبکه منتقل کرده و جواب آن را به کلاینت برگرداند. برای این موضوع، باید به نکات مختلفی توجه کنید. همان طور که گفته شد، امکان ارتباط با این کامپیوتر تنها از طریق پروتکل DNS امکان پذیر است. به طور کلی DNS تنها امکان انتقال ۵۱۲ بایت داده را دارد در حالی که پیام های TCP می توانند تا ۶۴ کیلوبایت هم باشند. برای رفع این مشکل می توان در قسمت Additional درDNS ، یک Record OPT از نوع Unknown اضافه کرد و در RData آن بسته‌های TCP مربوطه را قرار دارد. بدین ترتیب و از طریق استفاده از مکانیزم EDNS امکان انتقال ۴ کیلوبایت پیام در یک بسته DNS امکان پذیر خواهد شد. به بیان دیگر ما باید عملا درخواست های دسترسی به اینترنت که عموما درخواست‌هایی از جنس TCP هستند را در قسمت RDATA در بسته‌های DNS به سرور ارسال کنید و سرور نیز بعد از بررسی پیام با پروتکل TCP با بیرون ارتباط برقرار کرده و جواب آن را به صورت DNS به کلاینت بازگرداند.

توجه کنید برای این که بسته های لایه Transport را بتوان به صورت مجزا از اینترفیس عادی شبکه بررسی کرد، احتمالا باید یک Interface جدید تعریف کنیم. این کار در لینوکس از طریق TUN/TAP API امکان پذیر است. همچنین برای هندل کردن ارتباط اینترفیس جدید ایجاد شده با اینترفیس عادی اینترنت موجود، یک NAT در سرور تعریف می‌کنیم.

حال گام به گام مسائل را پیش می‌بریم:

اوبونتو و ماشین مجازی

ابتدا اوبونتو را در ماشین مجازی VMWare راه انداخته و virtual studio code را بر روی آن نصب می‌نماییم.

برای این کار ابتدا نسخه deb. را دانلود کرده و با دستورات زیر مراحل نصب را طی می‌کنیم.

```
sudo apt install ./<file>.deb


# If you're on an older Linux distribution, you will need to run this instead:

# sudo dpkg -i <file>.deb

# sudo apt-get install -f # Install dependencies

sudo apt-get install wget gpg

wget -qO- https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor > packages.microsoft.gpg

sudo install -D -o root -g root -m 644 packages.microsoft.gpg /etc/apt/keyrings/packages.microsoft.gpg

echo "deb [arch=amd64,arm64,armhf signed-by=/etc/apt/keyrings/packages.microsoft.gpg] https://packages.microsoft.com/repos/code stable main" |sudo tee /etc/apt/sources.list.d/vscode.list > /dev/null

rm -f packages.microsoft.gpg
```

حال پس از راه اندازی آن ادامه می‌دهیم.

سرور

از برنامه سرور شروع می‌کنیم:

```python
import socket
import logging
from scapy.all import IP, TCP, send, sr1

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')


# تابع تبدیل دامنه به فرمت DNS
def dns_format_conversion(domain_name: str) -> bytes:
    segments = domain_name.split('.')
    dns_representation = b''
    for segment in segments:
        segment_length = len(segment)
        dns_representation += segment_length.to_bytes(1, 'big') + segment.encode('utf-8')
    return dns_representation + b'\x00'


# تابع تبدیل فرمت DNS به دامنه
def reverse_dns_format(dns_bytes: bytes) -> str:
    domain_segments = []
    position = 0
    while position < len(dns_bytes) and dns_bytes[position] != 0:
        segment_length = dns_bytes[position]
        position += 1
        domain_segments.append(dns_bytes[position:position + segment_length].decode('utf-8'))
        position += segment_length
    return '.'.join(domain_segments)
```

```python
# ارسال و دریافت بسته‌ها
def process_packet(packet_data: bytes) -> bytes:
    ip_packet = IP(packet_data)
    if ip_packet.haslayer(TCP):
        logging.info(f"Processing packet for {ip_packet[IP].dst}:{ip_packet[TCP].dport}")
        answer = sr1(ip_packet, timeout=5, verbose=0)
        if answer:
            return bytes(answer)
        else:
            logging.warning("No response received.")
            return b''
    else:
        logging.warning("Received a non-TCP packet. Ignoring it.")
        return b''


# مدیریت اتصال کلاینت
def client_interaction(client_sock):
    incoming_data = client_sock.recv(2048)
    if incoming_data:
        outgoing_data = process_packet(incoming_data)
        client_sock.sendall(outgoing_data)
    client_sock.close()
```

```python
# اجرای سرور
def launch_server():
    bind_address = '127.0.0.1'
    bind_port = 8888
    sock_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock_server.bind((bind_address, bind_port))
    sock_server.listen(10)
    logging.info("Server is running on port 8888")

    while True:
        conn, client_address = sock_server.accept()
        logging.info(f"New connection from {client_address}")
        client_interaction(conn)


if __name__ == "__main__":
    launch_server()
```

تابع dns_format_conversion:

این تابع یک دامنه مثلاً (example.com) رو به فرمت بایت‌های DNS تبدیل می‌کنه. در فرمت DNS، هر بخش از دامنه (مانند example و com) باید به صورت طول + داده‌ها (label) کدگذاری بشه.

ابتدا رشته‌ی دامنه رو بر اساس نقطه (.) به بخش‌های جداگانه تقسیم می‌کنه و در لیستی به نام segments ذخیره می‌کنه. برای هر بخش، ابتدا طول اون بخش (به صورت یک عدد صحیح) به بایت تبدیل می‌شه و سپس خود بخش (رشته) به بایت‌ها تبدیل می‌شه. این دو بایت (طول + داده) به بایت‌های نهایی اضافه می‌شه تا فرمت DNS ساخته بشه. در پایان، بایت \x00اضافه می‌شه که نشان‌دهنده پایان دامنه است. این فرمت بایت در نهایت برگردونده می‌شه و می‌تونه برای ارسال در شبکه استفاده بشه.

تابع reverse_dns_format:

این تابع برعکس تابع dns_format_conversion عمل می‌کنه. یعنی یک رشته بایت DNS رو گرفته و اون رو به یک دامنه قابل خواندن (رشته) تبدیل می‌کنه. ابتدا یک لیست خالی به نام domain_segments ایجاد می‌شه که برای نگهداری بخش‌های دامنه استفاده می‌شه. با استفاده از یک حلقه، هر بخش از دامنه از داده‌های بایت خارج می‌شه:

- ابتدا طول هر بخش رو از بایت می‌خونه.
- سپس با توجه به طول مشخص شده، خود بخش (label) از بایت‌ها استخراج می‌شه و به رشته تبدیل می‌شه.

این بخش به لیست domain_segments اضافه می‌شه. حلقه تا زمانی ادامه پیدا می‌کنه که به بایت صفر (\x00) برسیم، که نشانه پایان دامنه است. در نهایت، بخش‌های دامنه با استفاده از نقطه (.) به هم متصل می‌شن تا دامنه نهایی ایجاد بشه و به صورت رشته بازگردونده بشه.

تابع process_packet:

این تابع مسئول پردازش بسته‌های TCP ورودی و ارسال اون‌ها به مقصد هست. در واقع، این تابع بسته رو دریافت کرده، بررسی می‌کنه که آیا بسته TCP است یا نه، و در نهایت اون رو به مقصد می‌فرسته و پاسخ رو دریافت

می‌کنه. ابتدا بسته ورودی رو به یک شیء از نوع IP تبدیل می‌کنه تا بتونه به لایه‌های مختلف بسته دسترسی داشته باشه.

سپس بررسی می‌کنه که آیا بسته شامل لایه TCP هست یا نه. اگر بسته TCP باشه، به این معنیه که اطلاعات مقصد بسته در لایه TCP قرار داره و باید به اونجا ارسال بشه.

اگر بسته TCP باشه:

- اطلاعات مقصد بسته (آدرس IP و پورت مقصد) رو با استفاده از لاگینگ چاپ می‌کنه.
- بسته رو با استفاده از تابع (sr1که از کتابخانه Scapy است) ارسال می‌کنه و منتظر دریافت پاسخ می‌مونه. این تابع بسته رو ارسال کرده و اولین پاسخی که از مقصد می‌رسه رو دریافت می‌کنه.
- اگر پاسخی دریافت بشه، اون رو به صورت بایت‌ها برمی‌گردونه.
- اگر پاسخی دریافت نشه، یک پیام هشدار ثبت می‌کنه و یک بایت خالی (b'') برمی‌گردونه.

اگر بسته TCP نباشه، اون رو نادیده می‌گیره و یک پیام هشدار در لاگ ثبت می‌کنه.

تابع client_interaction:

این تابع مسئول مدیریت ارتباط با کلاینت‌ها هست. زمانی که یک کلاینت به سرور متصل می‌شه، این تابع بسته‌های ارسالی کلاینت رو دریافت کرده، اون‌ها رو پردازش می‌کنه و سپس پاسخ مناسب رو به کلاینت برمی‌گردونه. ابتدا یک بسته ورودی از کلاینت دریافت می‌کنه. این دریافت با استفاده از متد recv صورت می‌گیره که تا حداکثر 2048 بایت داده رو از کلاینت می‌خونه. سپس بررسی می‌کنه که آیا داده‌ای از کلاینت دریافت شده یا خیر (با استفاده از شرط :if incoming_data) .اگر داده‌ای وجود داشته باشه:

داده رو به تابع process_packet ارسال می‌کنه تا بسته پردازش بشه و پاسخ دریافت بشه.

سپس پاسخی که از تابع process_packet دریافت کرده رو به کلاینت برمی‌گردونه (با استفاده از sendall). در پایان، ارتباط با کلاینت بسته می‌شه.(()client_sock.close)

تابع launch_server:

این تابع مسئول راه‌اندازی و اجرای سرور هست. سرور به طور مداوم در حال گوش دادن برای اتصالات ورودی کلاینت‌هاست و هر اتصال جدید رو پردازش می‌کنه.

کلاینت

حالا راجع به کد TUN.py صحبت می‌کنیم.

```python
import os
import time
import socket
import struct
import logging
import subprocess
from fcntl import ioctl
from scapy.all import *
from ipaddress import IPv4Address
from concurrent.futures import ThreadPoolExecutor

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# ثابت‌های مرتبط با تنظیمات TUN
_TUNSETIFF = 0x400454ca
_IFF_TUN = 0x0001
_IFF_NO_PI = 0x1000

class VirtualTunnel:
    def __init__(self, interface_name: str, ip_address: IPv4Address, subnet_mask: IPv4Address):
        self.interface_name = interface_name
        self.ip_address = ip_address
        self.subnet_mask = subnet_mask

        self.fd = os.open('/dev/net/tun', os.O_RDWR)
        ifr = struct.pack('16sH', interface_name.encode('utf-8'), _IFF_TUN | _IFF_NO_PI)
        ioctl(self.fd, _TUNSETIFF, ifr)
        logging.info('Virtual interface %s created successfully', self.interface_name)

        subprocess.check_call(['ip', 'addr', 'add', f'{self.ip_address}/{self.subnet_mask}', 'dev', self.interface_name])
```

```python
class VirtualTunnel:
    def __init__(self, interface_name: str, ip_address: IPv4Address, subnet_mask: IPv4Address):
        self.interface_name = interface_name
        self.ip_address = ip_address
        self.subnet_mask = subnet_mask

        self.fd = os.open('/dev/net/tun', os.O_RDWR)
        ifr = struct.pack('16sH', interface_name.encode('utf-8'), _IFF_TUN | _IFF_NO_PI)
        ioctl(self.fd, _TUNSETIFF, ifr)
        logging.info('Virtual interface %s created successfully', self.interface_name)

        subprocess.check_call(['ip', 'addr', 'add', f'{self.ip_address}/{self.subnet_mask}', 'dev', self.interface_name])
        logging.info('IP %s/%s assigned to %s', self.ip_address, self.subnet_mask, self.interface_name)

    def activate(self):
        subprocess.check_call(['ip', 'link', 'set', 'dev', self.interface_name, 'up'])
        logging.info('Interface %s is active now', self.interface_name)

    def read_packet(self, buffer_size: int = 8096) -> bytes:
        return os.read(self.fd, buffer_size)

    def send_packet(self, data: bytes) -> None:
        os.write(self.fd, data)


def is_valid_tcp(packet: bytes) -> bool:
    return packet[9] == 0x06
```

```python
def transmit_packet(packet_data: bytes) -> bytes:
    ip_packet = IP(packet_data)
    reply = sr1(ip_packet, timeout=1, verbose=0)

    if reply:
        return bytes(reply)

    return b''


def dns_to_bytes(domain_name: str) -> bytes:
    labels = domain_name.split('.')
    dns_data = b''.join(len(label).to_bytes(1, 'big') + label.encode('utf-8') for label in labels)
    return dns_data + b'\x00'


def bytes_to_dns(dns_data: bytes) -> str:
    labels = []
    idx = 0
    while idx < len(dns_data) and dns_data[idx] != 0:
        label_length = dns_data[idx]
        idx += 1
        labels.append(dns_data[idx:idx + label_length].decode('utf-8'))
        idx += label_length
    return '.'.join(labels)
```

```
def transmit_to_server(data_packet: bytes, server_ip: str, server_port: int) -> bytes:
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        client_socket.connect((server_ip, server_port))
        client_socket.sendall(data_packet)
        server_response = client_socket.recv(2048)
        return server_response
    finally:
        client_socket.close()


def fetch_web_content(url: str) -> bytes:
    try:
        response = subprocess.run(['curl', '-s', url], capture_output=True, text=True)
        return response.stdout.encode('utf-8')
    except Exception as e:
        logging.error(f"Failed to fetch content from {url}: {e}")
        return b''


def packet_processing_loop(interface: VirtualTunnel, target_ip: str, target_port: int):
    logging.info('Listening for packets on %s', interface.interface_name)
    while True:
        try:
            packet = interface.read_packet()
            if packet and is_valid_tcp(packet):
                logging.debug('TCP packet detected, forwarding to server')
                server_response = transmit_to_server(packet, target_ip, target_port)
```

```python
def packet_processing_loop(interface: VirtualTunnel, target_ip: str, target_port: int):
    logging.info('Listening for packets on %s', interface.interface_name)
    while True:
        try:
            packet = interface.read_packet()
            if packet and is_valid_tcp(packet):
                logging.debug('TCP packet detected, forwarding to server')
                server_response = transmit_to_server(packet, target_ip, target_port)
                if server_response:
                    interface.send_packet(server_response)
                response = transmit_packet(packet)
                interface.send_packet(response)
        except Exception as e:
            logging.error(f"Error occurred while processing packet: {e}")
        time.sleep(0.02)


def start_tunnel():
    tunnel_interface = VirtualTunnel(
        interface_name='my-tunnel',
        ip_address=IPv4Address('10.2.0.1'),
        subnet_mask=IPv4Address('255.255.255.0')
    )
    tunnel_interface.activate()

    server_address = "127.0.0.1"
    server_port = 8080

    with ThreadPoolExecutor(max_workers=2) as pool:
        pool.submit(packet_processing_loop, tunnel_interface, server_address, server_port)
```

کلاس VirtualTunnel:

این کلاس برای ایجاد و مدیریت یک اینترفیس مجازی TUN استفاده می‌شود.

- Init: این تابع سازنده اینترفیس مجازی را ایجاد کرده و IP و subnet mask را تنظیم می‌کند. همچنین دستگاه /dev/net/tun را باز کرده و یک اینترفیس مجازی با استفاده از ioctl ایجاد می‌کند.
- Activate: این تابع اینترفیس TUN را فعال می‌کند و آن را به حالت up می‌برد.
- read_packet: این تابع برای خواندن بسته‌ها از اینترفیس استفاده می‌شود.

- send_packet: این تابع برای نوشتن و ارسال بسته‌ها به اینترفیس استفاده می‌شود.

تابع is_valid_tcp:

این تابع بررسی می‌کند که آیا بسته ورودی از نوع TCP است یا خیر.

تابع transmit_packet:

این تابع بسته TCP را با استفاده از Scapy به مقصد ارسال می‌کند و پاسخ را به صورت بایت‌ها باز می‌گرداند.

تابع dns_to_bytes:

این توابع برای تبدیل فرمت‌های DNS به بایت و بالعکس استفاده می‌شوند.

تابع bytes_to_dns:

این توابع برای تبدیل فرمت‌های DNS به بایت و بالعکس استفاده می‌شوند.

تابع transmit_to_server:

این تابع بسته TCP را به سرور مشخص‌شده ارسال کرده و پاسخ سرور را دریافت می‌کند.

تابع fetch_web_content:

این تابع محتوای یک URL را با استفاده از ابزار curlدریافت می‌کند و در صورت موفقیت، محتوای آن را بازمی‌گرداند.

تابع packet_processing_loop:

این تابع حلقه اصلی برای خواندن بسته‌ها از اینترفیس TUN و پردازش آن‌ها است. بسته‌های TCP را به سرور ارسال کرده و پاسخ را به اینترفیس بازمی‌گرداند.

خب تا اینجا این دو برنامه توضیح داده شدند. کدها را به طور کامل‌تر در فایل‌های موجود می‌توانید مشاهده کنید.

اجرای آن‌ها در ویدیو موجود می‌باشد. مشکلاتی نیز خوردم که در ابتدا سرور کاملا بی دلیل کانکت نمی‌شد.

NAT

برای NAT بسیار مشکل داشتم که از https://github.com/ArshiAAkhavan/ark هم کمک گرفتم. اما در اصل از کد آقای اخوان و همان کدی که در کارگاه می‌زنند کمک گرفتم. توضیحات کد در همان لینک گیتهاب وجود دارد. مشابه همین کد پیاده سازی شده است.

برای اجازه NAT شدن نیز دستور زیر را حتما باید وارد کنیم.

```
sudo sysctl -w net.ipv4.ip_forward=1
```

دستورات کلی شامل همه بخش‌ها

```
# Disable the network interface net0
sudo ip link set dev net0 down

# Delete the network interface net0
sudo ip link delete net0

# Remove the default route for the net0 interface
sudo ip route del default dev net0

# Add a network route to IP 192.168.100.10 via gateway 192.168.1.1 using net0 interface
sudo ip route add 192.168.100.10 via 192.168.1.1 dev net0

# Delete the network route to IP 192.168.100.10 via gateway 192.168.1.1
sudo ip route del 192.168.100.10 via 192.168.1.1

# Add a new routing table named custom_route_table with ID 200 to the rt_tables file
sudo echo "200 custom_route_table" | sudo tee -a /etc/iproute2/rt_tables

# Mark traffic destined for IP 192.168.100.10 using iptables in the mangle table and PREROUTING chain
sudo iptables -t mangle -A PREROUTING -d 192.168.100.10 -j MARK --set-mark 2
```

```
# Add a routing rule to direct traffic marked with mark 2 to the routing table 200
sudo ip rule add fwmark 2 table 200

# Add a default route in table 200 that routes through the net0 interface
sudo ip route add default dev net0 table 200

# List the rules in the mangle table and PREROUTING chain with details
sudo iptables -t mangle -L PREROUTING -v

# Display all routing rules defined in the system
ip rule show

# Show the routes in the routing table 200
ip route show table 200

# Display the contents of the rt_tables file, which contains routing tables
sudo cat /etc/iproute2/rt_tables

# Flush all routes in the routing table 200
sudo ip route flush table 200

# Open the rt_tables file with the nano text editor for editing
sudo nano /etc/iproute2/rt_tables

# Install the iptables-persistent package to save iptables rules permanently
sudo apt-get install iptables-persistent

# Save the current iptables rules to persist after a system restart
sudo netfilter-persistent save
```

توضیحات کلی هر کدام بالای آن‌ها آمده است.

در نهایت، توضیحات کدهای کلاینت و سرور را نوشتیم. بخش NAT را توضیح دادیم که از آقای اخوان کمک گرفتیم و دستواتش در بخش آخر هم آوردیم. در نهایت هم یک سری دستور برای این آزمایش با کارکردهایشان به طور اضافی آوردیم. در ویدیو هم به کمک curl neverssl.com صحت کدها را تست کرده‌ایم.