Project 2-A: NAT

CS 436

Shay Ramirez

Kenneth Campbell

Files Included:

1. nat_table.py
2. scapy_tcp.py

How to use

1. Check that python is installed and up to date on your computer:
   Run the following command to check if / what version of python is installed:

   Python3 –version

2. Switch over to the directory containing the files:

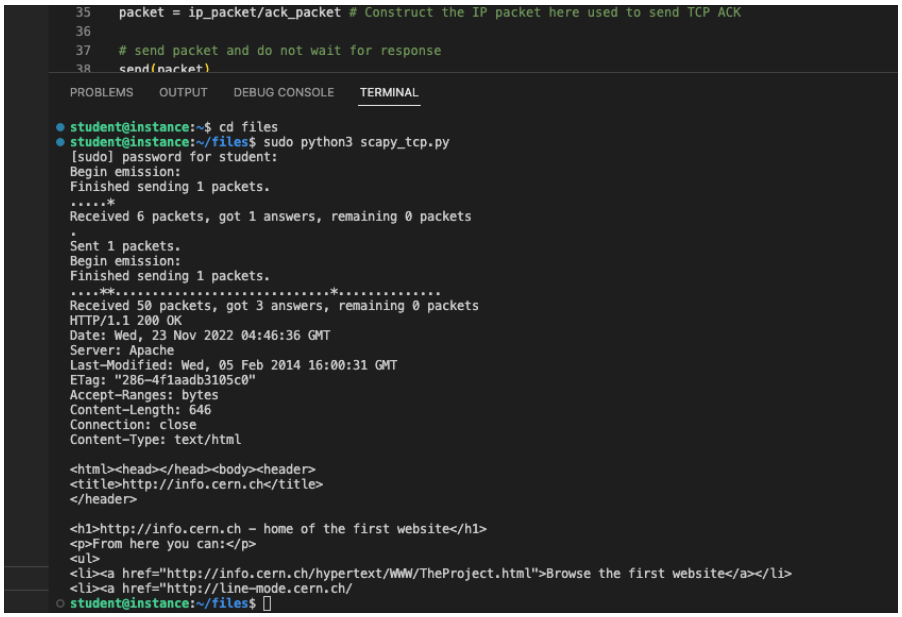
   NAT_TABLE.PY
   SCAPY_TCP.PY

3. You should use *sudo* command to run this program as root privilege is required for packet receiving.

4. Type in the following command to run the scapy_tcp.py file:

   sudo python3 scapy_tcp.py

   Expected Output:

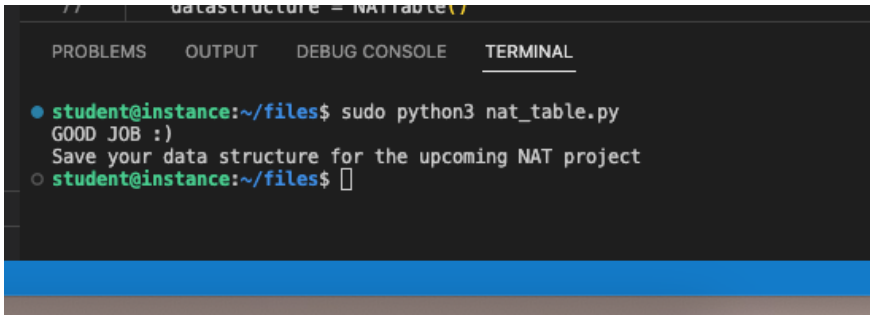


(If SCAPY_TCP.PY runs successfully, expect to see the above results)

5. Type in the following command to run the nat_table.py file:

   sudo python3 nat_table.py

   Expected Output:



   (If NAT_TABLE.PY runs successfully, expect to see the above results)

Explanation

1. nat_table.py

   Purpose:

   To serve as a NAT Table, which uses a translation table to manage the mapping between a (private IPv4 address, TCP port) and a (public IPv4 address, TCP port). This map / Nat Table will be accessed every time you receive a packet (from a client destined to a server, and also when a packet is received from a server, destined for a client).

   Imports

   i. IMPORT random

      Purpose

      Allows use of random number generator / utility

   ii. FROM typing IMPORT Tuple

      Purpose

      Allows use of the Tuple data structure

   Classes:

   iii. NATTable

Defines the class used for constructing and interacting with the NAT Table Object. As well as providing a tester class to ensure the code's functionality.

```
# This Class will support both ICMP and TCP packets
## icmp_mapping = NATTable()
## tcp_mapping = NATTable()
## and expose only set(x,y) and get(x,y) functions
class NATTable:
```

Functions:

i. __init__

Purpose:

This is the class constructor (function) that is responsible for initializing / declaring the data structures that will house the information provided to and by the class.

```
def __init__(self):
    # NAT translation table
    # ========================== WORK WHERE ===================================
    # IMPLEMENT THIS
    # creates an empty list to store registered port num and LAN IP
    self.data = []
    # creates a list to store port num and LAN IP (initialized with 0-63)
    self.lanList = list(range(64))
    # creates a list to store port num and WAN IP (initialized with 0-63)
    self.wanList = list(range(64))
```

ii. test_datastructure

Purpose:

This is a test function that simulates practical use of the NATTable class, ensuring that the code operates as expected.

```python
# DO NOT MODIFY TEST FUNCTION
def test_datastructure():
    datastructure = NATTable()

    used_ports = []

    computer1_ip = "10.0.0.1"
    computer1_port1 = 33450
    computer1_port2 = 39999

    computer2_ip = "10.0.0.50"
    computer2_port1 = 33450
    computer2_port2 = 34898

    computer3_ip = "10.0.0.120"
    computer3_port1 = 33450
    computer3_port2 = 35255
    computer3_port3 = 36878

    ip_src, port_src = datastructure.set(computer1_ip, computer1_port1)
    used_ports.append(port_src)
    assert ip_src == PUBLIC_IP

    ip_src, port_src = datastructure.get(ip_src, port_src)
    assert ip_src == computer1_ip
    assert port_src == computer1_port1

    ip_src, port_src = datastructure.set(computer2_ip, computer2_port1)
    assert ip_src == PUBLIC_IP
    assert port_src not in used_ports
    used_ports.append(port_src)

    ip_src, port_src = datastructure.set(computer2_ip, computer2_port1)
    assert ip_src == PUBLIC_IP
    assert port_src in used_ports

    ip_src, port_src = datastructure.get(ip_src, port_src)
    assert ip_src == computer2_ip
    assert port_src == computer2_port1

    ip_src, port_src = datastructure.set(computer3_ip, computer3_port1)
    assert ip_src == PUBLIC_IP
    assert port_src not in used_ports
    used_ports.append(port_src)

    ip_src, port_src = datastructure.set(computer2_ip, computer2_port2)
    assert ip_src == PUBLIC_IP
    assert port_src not in used_ports
    used_ports.append(port_src)

    for port in used_ports:
        assert port > 30000

    ip_src, port_src = datastructure.get(*datastructure.set(computer1_ip, computer1_port2))
    assert ip_src == computer1_ip
    assert port_src == computer1_port2

    ip_src, port_src = datastructure.get(*datastructure.set(computer1_ip, computer1_port2))
    assert ip_src == computer1_ip
    assert port_src == computer1_port2

    ip_src, port_src = datastructure.get(*datastructure.set(computer3_ip, computer3_port2))
    assert ip_src == computer3_ip
    assert port_src == computer3_port2

    ip_src, port_src = datastructure.get(*datastructure.set(computer3_ip, computer3_port3))
    assert ip_src == computer3_ip
    assert port_src == computer3_port3
```

iii. _random_id

Purpose:

This function is responsible for generating a random number that falls between the valid ranges of potential port numbers (falling between 30000, 65535, i.e. all port numbers available, that are greater than 30000).

```
# generate random number within valid port range (between 30000, 65535, all availbe if greater than
def _random_id(self):
    return random.randint(30001, 65535)
```

iv. Get

Purpose:

This function will retrieve the corresponding IP and port number that is mapped to a given source IP and port number.

```
# get function
# retrieves the LAN side mapping ip_src and id_src
def get(self, ip_dst, id_dst) -> Tuple[str, int]:

    #set source ip equal to destination ip
    ip_src = ip_dst
    # set source port equal to destination port
    id_src = id_dst

    # hash lanPort
    lanPort = id_src % 64
    # get value (tuple) from Lan List
    lanValue = self.lanList[lanPort]

    # seperate tuple into individual variables/objects
    ip_src = lanValue[0]
    id_src = lanValue[1]

    # ISSUE: not needed
    self.data

    # return the corresponding ip and port numbers
    return ip_src, id_src
```

v. Set

Purpose:

This function establishes a new random port for each new connection to use when interfacing with servers/clients that are outside/inside your current network. This function will also check if the connection has already been mapped, if the connection already exists, the respective IP and port number will be returned; otherwise, implies that the connection does not exist, and thus, this function will establish the new connection and add it to the NAT table.

```
# set function
# Creates a new random port for each NEW connection
# otherwise, returns saved data if source ip and port numbers are found
def set(self, ip_src, id_src) -> Tuple[str, int]:

    # set ip equal to public ip
    new_ip_src = PUBLIC_IP

    # get new random port
    new_id_src = self._random_id()

    # get LAN from WAN port num
    lanPort = new_id_src % 64

    # get WAN from LAN port num
    wanPort = id_src % 64

    # if IP and Port numbers are already mapped
    if (ip_src, id_src) in self.data:
        # get value from the WAN List
        wanValue = self.wanList[wanPort]
        # seperate tuple into individual variables/objects
        ip_src = wanValue[0]
        id_src = wanValue[1]

        # returns port num and WAN IP
        return ip_src, id_src

    else:
        # create new connection and add info to lists
        self.lanList[lanPort] = (ip_src, id_src)
        self.wanList[wanPort] = (new_ip_src, new_id_src)
        # add to table
        self.data.append((ip_src, id_src))
        #return new ip and port numbers
        return new_ip_src, new_id_src
```

2. scapy_tcp.py

Purpose:

To use packet encapsulation and Scapy knowledge to construct an IP packet containing an HTTP GET request to this host info.cern.ch.

scapy_tcp.py

```python
#!/usr/bin/env python3

import random
from scapy.sendrecv import send, sr1, sr
from scapy.layers.inet import TCP, IP

###### TCP
# https://github.com/secdev/scapy/blob/v2.4.5/scapy/layers/inet.py#L678
# pkt[TCP].sport  # accessing a field in the TCP Layer

# create ip packet
ip_packet = IP(dst="info.cern.ch")

# Random initial seq number
seq = random.randint(10000, 19999)

syn_packet = TCP(flags='S', seq=seq)

# Making a HTTP request
sport = 6805 # TCP
dport = 80 # HTTP

syn_packet[TCP].sport = sport
syn_packet[TCP].dport = dport

# Encapsulate TCP syn packet inside IP packet
packet = ip_packet/syn_packet   # Construct the IP packet here used to send TCP SYN

# send and receive 1 packet (sr1)
synack_response = sr1(packet)

# TCP necessary steps
next_seq = synack_response.ack
my_ack = synack_response.seq + 1

ack_packet = TCP(sport=sport, dport=dport, flags='A', seq=next_seq, ack=my_ack)

# Encapsulate TCP ack packet inside IP packet
packet = ip_packet/ack_packet # Construct the IP packet here used to send TCP ACK

# send packet and do not wait for response
send(packet)

# create tcp payload
payload_packet = TCP(sport=sport, dport=dport, flags='A', seq=next_seq, ack=my_ack) / "GET / HTTP/1.1\r\nHost: info.cern.ch\r\n\r\n"

# Encapsulate TCP payload packet inside IP packet
packet = ip_packet/payload_packet # Construct the IP packet here used to send HTTP Get request

# send and receive multiple packets (sr)
reply, error = sr(packet, multi=1, timeout=1)

# get results / payload / data
result = bytes(reply[-1][1][TCP].payload)

# display the results
print(result.decode('utf-8'))
```