A logo of a university of windsor

Description automatically generated with low confidence

|  |
| --- |
| **Course Name** |
| Networking and Data Security (COMP-8677) |

|  |
| --- |
| **Document Type** |
| Lab 4 |

|  |
| --- |
| **Professor** |
| Dr. Shaoquan Jiang |

|  |
| --- |
| **Team - Members                               Student ID** |
| Manjinder Singh                                 110097177 |

***NOTES :*** For simplicity questions and content from lab manual 4 are mentioned in a box.

|  |
| --- |
| **1. Packet Construction with Scapy**  In this exercise, you will practice to construct several packets using scapy. To start, run $sudo python3 (if you work on VM root, sudo is not needed) and then import scapy package:  from scapy.all import \*  Then, you are ready to practice constructing various packets. In each question, show your screen shots as the evidence of your work.  1) IP() is the function to construct a default IP header. You can use ls(IP()) to view the content. The first column is the field of IP header and the third column is the example format for the value of each field. |

**My Implementation of above:-**

A screenshot of a computer

Description automatically generated

**Screenshot 1:** Enlisting the fields and their default values for the IP layer

|  |
| --- |
| You can assign the value to create the IP header you want. Please construct an ip header **iph** with source 10.0.2.4 and destination 10.10.10.10. Use ls to show packet header. |

**My Implementation of above:-**

A screenshot of a computer program

Description automatically generated

**Screenshot 2:** Constructing an IP Header with Ips for source and destination

|  |
| --- |
| 2) Create a UDP **segment** with source port number 5000 and destination port number 5300 and data=”hello”. Use show2 to show your result. |

**My Implementation of above:-**

A screenshot of a computer code

Description automatically generated

**Screenshot 3:** Creating a UDP segment

|  |
| --- |
| 3) You can create ping packet by stacking IP header over ICMP(). Create a ping packet with your VM as source IP and 10.10.10.10 as your destination IP. Create an ip packet with the same source and destination IP (as in the ping packet) but with UDP segment in item 2 as its data field. Use show2() function to show the packet content. |

**My Implementation of above:-**

A screenshot of a computer

Description automatically generated

**Screenshot 4:** ifconfig displays network interface configuration information

A screenshot of a computer code

Description automatically generated

**Screenshot 5:** Displaying the details of the IP header and ICMP header

A screenshot of a computer

Description automatically generated

**Screenshot 6:** UDP Packet with Source Port 5000 to Destination Port 5300 and Payload 'hello'

|  |
| --- |
| 4) For a packet pkt, pkt[IP] is IP datagram and pkt[UDP] is the UDP segment of pkt. For ip datagram in item 3, use show2 to show the UDP segment. |

**My Implementation of above:-**

A screenshot of a computer code

Description automatically generated

**Screenshot 7:** Extracting and display the UDP segment from the udp\_pkt packet.

|  |
| --- |
| **2. Sniffing Packets**  Wireshark is the most popular sniffing tool, and it is easy to use. We will use it throughout the entire lab. The objective of the current task is to learn how to use Scapy to do packet sniffing in Python programs. A sample code is the following:  -----------------------------------------------------------  #!/usr/bin/python  from scapy.all import \*  def print\_pkt(pkt):  pkt.show2()  pkt = sniff(filter=’icmp’,prn=print\_pkt, iface=”br-xxx”) # br-xxx is the interface on VM you want to sniff  ------------------------------------------------------ |

|  |
| --- |
| **Task A**. The above program sniffs packets. For each captured packet, the callback function print pkt() will be invoked; this function will print out some of the information about the packet. Run the program with the root privilege and demonstrate that you can indeed capture packets. After that, run the program again, but without using the root privilege; describe and explain your observations.  // Run the program with the root privilege  **$ sudo python sniffer.py**  // Run the program without the root privilege  **$ python sniffer.py** |

***Solution of Task A:-***

**To get Interface details (from separate Terminal)**

A screenshot of a computer

Description automatically generated

**Screenshot 8:** Displaying network interface information

A screenshot of a computer screen

Description automatically generated

**Screenshot 9:** sniff.py Program

The Python program in **Screenshot 9** uses Scapy to sniff ICMP packets on a specified network interface and prints details of the first 5 captured packets, then shows the content.

The below command execution in **Screenshot 10** is equivalent to “sudo python3 sniff.py” as first we ran sudo su then python3 sniff.py

A screenshot of a computer

Description automatically generated

**Screenshot 10:** Running Sniff.py with root privileges | Beginning of O/P

A screen shot of a computer

Description automatically generated

**Screenshot 11:** Output of running Sniff.py with root privileges | Last Part of Output

A screenshot of a computer

Description automatically generated

**Screenshot 12:** Copying sniff.py file to volumes folder with root privileges

"**docker-compose build**" builds Docker images, while

"**docker-compose up**" starts and runs containers based on those images as specified in a Docker Compose file.

A screen shot of a computer

Description automatically generated

**Screenshot 13:** Docker commands execution for building, starting and running.

"**dockps**" is used to list the running Docker containers, and

"**docksh**" is used to open a shell session within a running Docker container.

A computer screen shot

Description automatically generated

**Screenshot 14:** Executing dockps

A screenshot of a computer screen

Description automatically generated

**Screenshot 15:** Executing dockps and docksh(for seed attacker shell session) command

A screenshot of a computer

Description automatically generated

**Screenshot 16:** Executing dockps and docksh(for victim shell session) command

A screen shot of a computer code

Description automatically generated

**Screenshot 17:** Running Sniff.py without root privileges

From **screenshot 10 until 17**, it is clearly visible that when we try to run the packet-sniffing program without utilizing the root privileges, we get permission errors. This main reason is because of capturing network packets which mainly demands elevated permissions. However, when we tried to run the program within a Docker container, it seems to work fine as Docker containers generally have more permissive network permissions by default.

To summarize, if we tries to capture packets using the "sniff.py" program, we'll either need to run it with root privileges (using "sudo python3 sniff.py") or we need to make sure that the user running the program must have the required network capture permissions. Docker containers may allow packet capture without requiring root privileges due to their specific network configuration.

|  |
| --- |
| **Task B**. In this task, you need to modify the **program** to *simultaneously* achieve two goals:  1. When we sniff packets, we are only interested certain types of packets. Your program only sniffs the ICMP packet with source IP address 10.10.10.10.  2. For each captured ICMP packet, reverse the source and destination IP address and modify the ICMP data field as “COMP8677-yourname”. Finally, send the modified packet.  Run your Wireshark to check if 10.10.10.10 replied to your packet sent by item 2. If yes, give a screenshot for one such packet. In this task, provide your program and the said screenshot. |

***Solution of Task B:-***

In the below **screenshot 18**, the Python program uses Scapy to sniff and capture ICMP packets on a specific interface ("br-3cb06be4a02f"). When it captures an ICMP packet with the source IP address "10.10.10.10," it prints the packet details, which constructs a new ICMP packet with modified content, and sends the new packet. The modification mainly involves reversing the source and destination IP addresses and changing the ICMP data field to "COMP8677-Manjinder Singh."

A screenshot of a computer

Description automatically generated

**Screenshot 18:** sniffer.py Program



A screen shot of a computer program

Description automatically generated

**Screenshot 19:** Copying the sniffer program to volumes and Docker commands execution for building, starting and running

A screenshot of a computer

Description automatically generated

**Screenshot 20:** Running sniffer.py program for docker container of seed-attacker and showing O/P(Part - 1)

A white screen with black text

Description automatically generated

**Screenshot 21:** Showing O/P(Part - 2)

(of running sniffer.py program for docker container of seed-attacker)

(Continuation of Screenshot 20 O/P)

A screenshot of a computer

Description automatically generated

**Screenshot 22:** Showing O/P(Part - 3)

Displaying Output to signify packet is sent successfully with the load

(Continuation of Screenshot 20 O/P)

A screenshot of a computer

Description automatically generated

**Screenshot 23:** Pinging 10.10.10.10 under docker container of victim(Output Part -1)

A screenshot of a computer

Description automatically generated

**Screenshot 24:** Pinging 10.10.10.10 under docker container of victim(Output Part -2)

A screenshot of a computer

Description automatically generated

**Screenshot 25:** Wireshark Screen to show the Load(**COMP8697-Manjinder Singh**)

(10.10.10.10 replied to our packet)

|  |
| --- |
| **Task C**. In this task, you will practice more for BPF filter. You have studied one in your Task B. If necessary, check the reference file BPF.pdf. Test your solution using the sniff function on the command line of python and show one packet content. Here is my example for the test.  A screenshot of a computer  Description automatically generated  **Screenshot 26 :** From Lab Manual  a) Capture any TCP packet that comes from www.example.com with destination network being your VM subnet (mostly 10.0.2.0/24). We remind you that in order to capture packets from example.com, you of course need to visit the web site.  b) Capture packets that come from source port 53 and a **particular** network such as 10.10.10.0/24. In the test, run $dig @10.10.10.10 www.mit.edu. (note: if you do your assignment at home, you can change 10.10.10.10 to 8.8.8.8 and the subnet 8.8.0.0/16). |

***Solution of Task C:-***

***Solution to Task C Part (a)***

A screenshot of a computer screen

Description automatically generated

**Screenshot 27:** task3.py Program

With reference to **Screenshot 27** program, it will capture a single TCP packet that meets our criteria and display its details using the show() method. For this, we are utilizing the Scapy library of python and having the necessary permissions in place to capture packets on our network interface when running this script.

A screenshot of a computer

Description automatically generated

**Screenshot 28:** Running [www.example.com](http://www.example.com) on Mozilla Firefox(in left side) and

(right side) displaying that we are running the program showed in **Screenshot 27** using root privileges.

When we run the program and run [www.eample.com](http://www.eample.com) website then we get results on the right side screen.

A computer code with numbers and symbols

Description automatically generated

**Screenshot 29:** Output continuation of Screenshot 28 while running the program.

***Solution to Task C Part (b)***

A screenshot of a computer

Description automatically generated

**Screenshot 32:** Program updated as per task C Part (b)

(explanation is below)

**Explanation of Program shown in Screenshot 32:** The Python script using Scapy library of python to capture and display the details of a single packet that meets the following criteria: it originates from source port 53 and belongs to a specific network range (e.g., 10.10.10.0/24). The captured packet's details are printed using the show() method.

A screenshot of a computer code

Description automatically generated

**Screenshot 33:** Output on running python program(of Screenshot 32) with root privileges

A screenshot of a computer

Description automatically generated

**Screenshot 34:** Output Continuation of Screenshot 33(while running python program(of Screenshot 32) with root privileges)

A screenshot of a computer code

Description automatically generated

**Screenshot 35:** Output Continuation of Screenshot 34(while running python program(of Screenshot 32) with root privileges)

A screenshot of a computer

Description automatically generated

**Screenshot 36:** Output Continuation of Screenshot 35(while running python program(of Screenshot 32) with root privileges)

A screenshot of a computer

Description automatically generated

**Screenshot 37:** Output on running [dig@10.10.10.10](mailto:dig@10.10.10.10) [www.mit.edu](http://www.mit.edu)

**With Reference to Screenshot 37** - The output of the dig command for querying the DNS server at IP address 10.10.10.10 for the hostname www.mit.edu would provide DNS-related information.

A screenshot of a computer

Description automatically generated

**Screenshot 38:** Wireshark Screenshot for reference wrt the query response

|  |
| --- |
| **Screenshot 39:** Question 3 |

***Solution of Question 3 Part 1***

A screenshot of a computer

Description automatically generated

**Screenshot 40:** Updated Program for Question 3 Part 1 with better readability

**Explanation of Program in Screenshot 40:** The Python script uses the Scapy library of python to create a **fake ARP** (Address Resolution Protocol) request. It **pretends** to be one machine on a network (with IP address "10.9.0.5") and **sends** this request to another machine (with IP address "10.9.0.6").

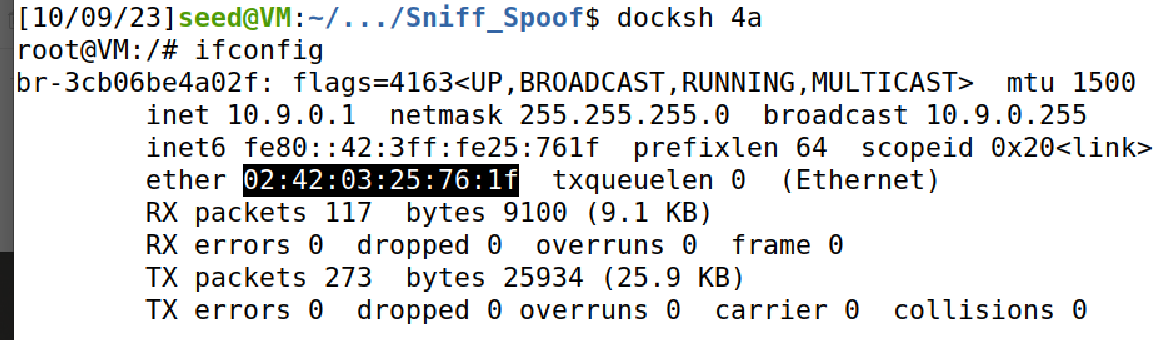
The **goal** is to trick the second machine into associating the attacker's MAC address with the IP address "10.9.0.5" in its ARP table. This technique is mainly used in **malicious attacks to intercept network traffic**.

***Solution of Question 3 Part 2 (Extension of Question 3 Part 1):-***

A close-up of a number

Description automatically generated

**Screenshot 41:** Executing dockps to display the running docker containers



**Screenshot 42:** Executing docksh to display the information docker container(specifically of seed-attacker)

A screenshot of a computer

Description automatically generated

**Screenshot 43:** Updated Program for Question 3



**Screenshot 44:** Output on running the program displayed above in Screenshot 43 with root privileges

A screenshot of a computer

Description automatically generated

**Screenshot 45:** Output showing entry of 10.9.0.5 in arp table on running for 10.9.0.6

***Solution of Question 3 Part 3 (Extension of Question 3 Part 1 and Part 2):-***

In this scenario, the primary objective is to manipulate the system at IP address **10.9.0.6** into believing that IP address **10.9.0.5** corresponds to the attacker's MAC address (**02:42:03:25:76:1f**).

Consequently, any data intended for 10.9.0.5 will be inadvertently routed to the attacker's device. This deceptive tactic is a crucial component of various "**man-in-the-middle**" attacks, enabling the attacker to intercept, modify, or obstruct the victim's communications without their knowledge.

To carry out this attack, the attacker first configures their MAC address (**mac\_attacker = "02:42:03:25:76:1f**") and identifies another MAC address they want to impersonate (**mac\_impersonate = "02:42:0a:09:00:05**").

Then, a crafted packet is generated to appear as though it originates from the impersonated IP address (**psrc ="10.9.0.5"),** but it uses the attacker's MAC address as the source (**hwsrc=mac\_attacker**). This packet is broadcasted across the entire local network using a broadcast MAC address **("ff:ff:ff:ff:ff:ff").**

Devices on the network that receive this broadcast request will update their ARP tables. When the legitimate device with IP address **10.9.0.6** responds to the request, its response is ignored because the ARP table has already been modified to associate the attacker's MAC address with this IP address.

As a result, any traffic originally intended for IP address **10.9.0.5** from **10.9.0.6** is now directed to the attacker's machine. This gives the **attacker** the ability **to intercept and manipulate the traffic** as desired.

It's crucial to emphasize that ARP cache poisoning should only be conducted in a controlled, ethical, and legal environment. Unauthorized or malicious use of this technique is both illegal and unethical, and can lead to legal repercussions if performed without proper permissions.

**References: -**

1. Lab Manual for Lab 4 from Brightspace
2. Lecture Notes for Lab 4 from Brightspace
3. Programs for Lab 4 from Brightspace
4. Scapy Python Library

**One Drive Link for Python Program, Lab 4 Solution(Word File and PDF Document) for Lab 4 Work:-**

[Networking and Data Security - Lab 4 - Submitted to Doc](https://uwin365-my.sharepoint.com/:f:/g/personal/lnu8_uwindsor_ca/EvneiTLQL5tKtvuXe6o9s8oBzPDA6CXIVKD16X84MVq2DQ?e=fIPxq0)