Packet Sniffing and Spoofing Lab

Modified 2022 for COMP 435: Computer Security Concepts at UNC

Overview

Packet sniffing and spoofing are two important concepts in network security; they are two major threats in network communication. Being able to understand these two threats is essential for understanding security measures in networking. There are many packet sniffing and spoofing tools, such as **`Wireshark, Tcpdump, Netwox, Scapy`**, etc. Some of these tools are widely used by security experts, as well as by attackers. Being able to use these tools is important for students, but what is more important for students in a network security course is to understand how these tools work, i.e., how packet sniffing and spoofing are implemented in software.

The objective of this lab is two-fold: learning to use the tools and understanding the technologies underlying these tools. For the second object, students will write simple sniffer and spoofing programs, and gain an in-depth understanding of the technical aspects of these programs. This lab covers the following topics:

- How the sniffing and spoofing work
- Packet sniffing using the **`Scapy`** library and Wireshark software

Submission

You will submit your final lab report to Gradescope. The final lab report should be a PDF where you complete all of the activities under the smaller “Submission” headings within each task. It will include the screenshots, descriptions of what you have done and what you have observed, and explanations of interesting observations and code snippets. Simply attaching code without any explanation will not receive credit.

Your screenshots must include a unique identifier. The simplest way to do this is to take a screenshot that encapsulates the VM interface itself (which should already happen before cropping), as it includes your onyen/name on the left and top-right.

Environment Setup

Please complete this lab in your class virtual machine! Instructions for logging into your VM are here: https://help.cs.unc.edu/en/blog/classvm. Note that you must be either on campus or logging on through a VPN. Instructions for connecting to the campus VPN are available for Mac, Windows, and Linux.

In this lab, we will use three machines that are connected to the same LAN. We can either use three VMs or three containers. Figure 1 depicts the lab environment setup using containers. We will do all the attacks on the attacker container, while using the other containers as the user machines.

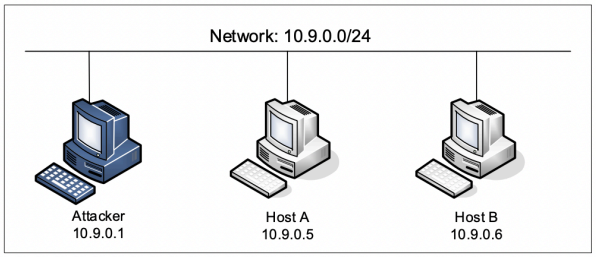


Figure 1. Lab Environment Setup

Container Setup and Commands

Please pull the new starter code (**`$ git pull starter main --rebase`** should do it). The docker-compose.yml file defines the containers you will use in this lab. If you'd like a more detailed explanation of the content in this file and the involved Dockerfile you can view the user manual here. We will provide you with any Docker commands you need to run to get started using the container.

In the following, we list some of the commonly used commands related to Docker and Compose. Since we are going to use these commands very frequently, we have created aliases for them in the .bashrc file.

`$ docker-compose build` # Build the container image
`$ docker-compose up` # Start the container
`$ docker-compose down` # Shut down the container

Aliases for the Compose commands above
`$ dcbuild` # Alias for: **`docker-compose build`**
`$ dcup` # Alias for: **`docker-compose up`**
`$ dcdown` # Alias for: **`docker-compose down`**

Within the lab4 directory use **`dcbuild`** and then **`dcup`** to start the containers defined in docker-compose.yml. All the containers will be running in the background.

If you run into an error that says a host is already in use when trying to run **`dcup`**, you can try rerunning the command with the **`--remove-orphans`** flag.

About the Attacker Container

In this lab, we use the attacker container as the attacker machine. If you look at the Docker file, you will see that the attacker container is configured differently from the other containers. Here are the differences:

Shared folder:

When we use the attacker container to launch attacks, we need to put the attacking code inside the attacker container. Code editing is more convenient inside the VM than in containers, because we can use our favorite editors. In order for the VM and container to share files, we have created a shared folder between the VM and the container using the Docker **`volumes`**. If you look at the Docker file, you will find out that we have added the following entry to some of the containers. It indicates mounting the **`./volumes`** folder on the host machine (i.e., the VM) to the **`./volumes`** folder inside the container. We will

write our code in the **./volumes** folder (on the VM), so they can be used inside the containers

```
volumes:
   - ./volumes:/volumes
```

Host Mode:

In this lab, the attacker needs to be able to sniff packets, but running sniffer programs inside a container has problems, because a container is effectively attached to a virtual switch, so it can only see its own traffic, and it is never going to see the packets among other containers. To solve this problem, we use the host mode for the attacker container. This allows the attacker container to see all the traffic. The following entry used on the attacker container:

```
network_mode: host
```

When a container is in the host mode, it sees all the host's network interfaces, and it even has the same IP addresses as the host. Basically, it is put in the same network namespace as the host VM. However, the container is still a separate machine, because its other namespaces are still different from the host.

Getting the network interface name

When we use the provided Compose file to create containers for this lab, a new network is created to connect the VM and the containers. The IP prefix for this network is **10.9.0.0/24**, which is specified in the **docker-compose.yml** file. The IP address assigned to our VM is **10.9.0.1**. We need to find the name of the corresponding network interface on our VM, because we need to use it in our programs. The interface name is the concatenation of **br-** and the ID of the network created by Docker. When we use **ifconfig** to list network interfaces, we will see quite a few. Look for the IP address **10.9.0.1**.

```
$ ifconfig
br-c93733e9f913: flags=4163 mtu 1500 inet 10.9.0.1 netmask
255.255.255.0 broadcast 10.9.0.255
 ...
```

Another way to get the interface name is to use the "**docker network**" command to find out the network ID ourselves (the name of the network is **seed-net**):

```
$ docker network ls
NETWORK ID          NAME              DRIVER              SCOPE
a82477ae4e6b        bridge            bridge              local
e99b370eb525        host              host                local
df62c6635eae        none              null                local
c93733e9f913        seed-net          bridge              local
```

Using Wireshark

To open Wireshark in the VM, Open it from the desktop of your VM. You should see this screen prompting you to select a capture filter.

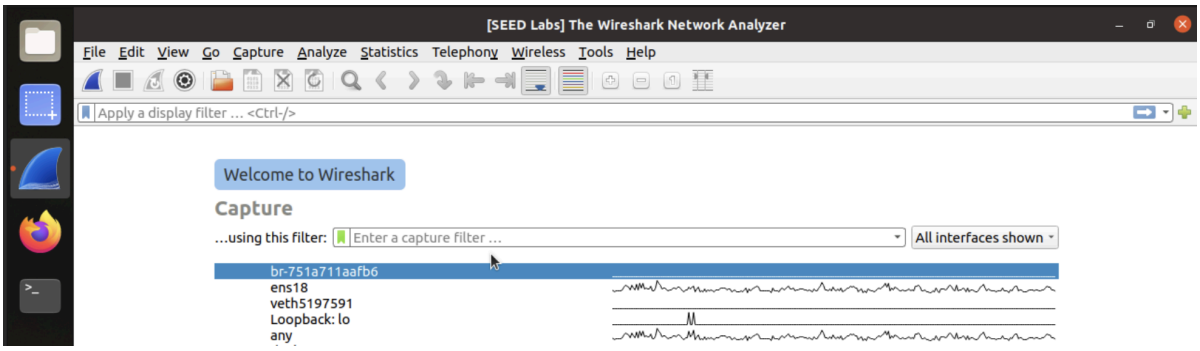


Figure 2. Selecting a Wireshark Capture Filter

Double click on the interface starting with br-(the VM's network interface created by Docker). Now Wireshark will start listening to and capturing network traffic and displaying it in the pane at the top of your screen. In the figure below you can see the live capture has begun, but no packets have arrived yet.

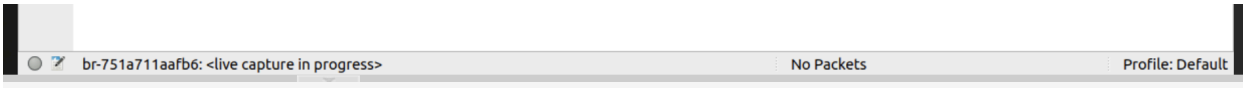
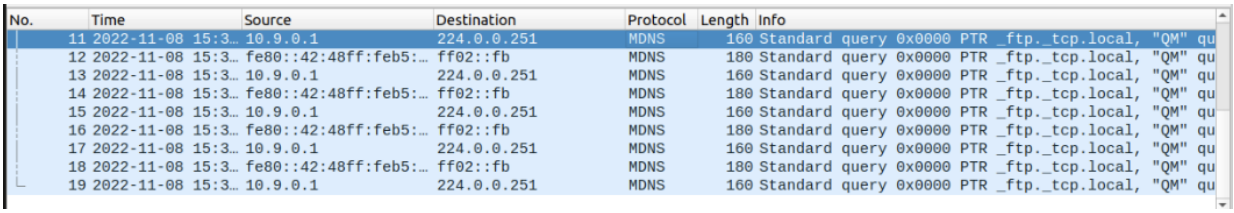


Figure 3. Verifying the Live Capture is in Progress

Once packets have been captured, we will be able to see the Time, Source, Destination, Protocol and some other information about the contents as denoted by the columns, and pictured below. You are encouraged to click on each packet when it arrives to explore and see what you observe!



No.	Time	Source	Destination	Protocol	Length	Info
11	2022-11-08 15:3…	10.9.0.1	224.0.0.251	MDNS	160	Standard query 0x0000 PTR _ftp._tcp.local, "QM" qu
12	2022-11-08 15:3…	fe80::42:48ff:feb5:…	ff02::fb	MDNS	180	Standard query 0x0000 PTR _ftp._tcp.local, "QM" qu
13	2022-11-08 15:3…	10.9.0.1	224.0.0.251	MDNS	160	Standard query 0x0000 PTR _ftp._tcp.local, "QM" qu
14	2022-11-08 15:3…	fe80::42:48ff:feb5:…	ff02::fb	MDNS	180	Standard query 0x0000 PTR _ftp._tcp.local, "QM" qu
15	2022-11-08 15:3…	10.9.0.1	224.0.0.251	MDNS	160	Standard query 0x0000 PTR _ftp._tcp.local, "QM" qu
16	2022-11-08 15:3…	fe80::42:48ff:feb5:…	ff02::fb	MDNS	180	Standard query 0x0000 PTR _ftp._tcp.local, "QM" qu
17	2022-11-08 15:3…	10.9.0.1	224.0.0.251	MDNS	160	Standard query 0x0000 PTR _ftp._tcp.local, "QM" qu
18	2022-11-08 15:3…	fe80::42:48ff:feb5:…	ff02::fb	MDNS	180	Standard query 0x0000 PTR _ftp._tcp.local, "QM" qu
19	2022-11-08 15:3…	10.9.0.1	224.0.0.251	MDNS	160	Standard query 0x0000 PTR _ftp._tcp.local, "QM" qu

Figure 4. Example of Captured Packets

Task 1: Sniffing Packets

Many tools can be used to do sniffing and spoofing, but most of them only provide fixed functionalities. Scapy is different: it can be used not only as a tool, but also as a building block to construct other sniffing and spoofing tools, i.e., we can integrate the Scapy functionalities into our own program. In this set of tasks, we will use Scapy for each task.

To use Scapy, we can write a Python program, and then execute this program using Python. See the following example. We should run Python using root privilege because the privilege is required for spoofing packets. At the beginning of the program (Line ①), we should import all Scapy's modules.

Listing 1: Python Script for Loading in Scapy and IP modules

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

from scapy.all import * ①

a = IP()
a.show()
```

You can run the above code by running the following commands:

```
chmod a+x mycode.py
./mycode.py
```

We can also get into the interactive mode of Python and then run our program one line at a time at the Python prompt. This is more convenient if we need to change our code frequently in an experiment.

Listing 2: Using the Python interactive prompt

```
$ python3
>>> from scapy.all import *
>>> a = IP()
>>> a.show()
###[ IP ]###
    version = 4
    ihl = None
    ...
```

Wireshark is the most popular sniffing tool, and it is easy to use. We will use it throughout the entire lab. However, it is difficult to use Wireshark for automation tasks. We will use **Scapy** for that purpose. The objective of this task is to learn how to use Scapy to do packet sniffing in Python programs. A sample code listing is provided in the following:

Listing 3: Example of packet sniffing code.

```
#!/usr/bin/env python3
from scapy.all import *

def print_pkt(pkt):
    pkt.show()

pkt = sniff(iface='br-c93733e9f913', filter='icmp', prn=print_pkt)
```

The code above will sniff the packets on the **br-c93733e9f913** interface. Please read the instructions in the lab setup section regarding how to get the interface name. If we want to sniff on multiple interfaces, we can put all the interfaces in a list, and assign it to **iface**. See the following example:

```
iface=['br-c93733e9f913', 'enp0s3']
```

Task 1A. In the above program, 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 **`(sudo python3 sniffer.py)`** 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.

Hints:

- In order to "capture packets" you will need to run your code for this task in one terminal and open a second terminal window and ping a device that is operating within the interface we set up earlier. If you are unfamiliar with how to use the **`ping`** command visit this page for more information.

Submission

1. Include all code written to complete the task.
2. Show a screenshot of the program being run with root privilege and the packets successfully being captured, including an ICMP packet displaying the 'src' and 'dst' fields.
3. Show a screenshot of the **`ping`** request(s) that caused the output in your first terminal
4. Show a screenshot of the program being run without root privilege.
5. Describe and explain your observations after running the program the second time.

Task 1B. Usually, when we sniff packets, we are only interested in certain types of packets. We can do that by setting filters in sniffing. Scapy's filter uses the BPF (Berkeley Packet Filter) syntax. The documentation for this can be found here and more examples can be found here. BPF has a very simple syntax: state each filter you want, and connect them with the `and` or `or` keyword. For example if you want to only listen for UDP traffic originating from a machine with IP address 168.192.0.1, the corresponding BPF filter would be `udp and src host 168.192.0.1.`

Please set the following filters and demonstrate your sniffer program again (each filter should be set separately):

- Capture only the ICMP packet
- Capture any TCP packet that comes from 10.9.0.1 (i.e YOU) and with a destination port number 23
    - In order to specifically capture TCP packets, you will need to use the **`telnet`** command in the new window (instead of **`ping`**). Your usage of the command should be identical (i.e. **`telnet <dst ip or hostname>`**) The destination should not be yourself. More information about telnet can be found at: https://en.wikipedia.org/wiki/Telnet
- Capture packets coming from or going to the subnet 10.9.0/24.
    - Run the **`ping`** command again to test this.

Hints:

- Everytime you restart the containers (AKA cycle through the **`dcdown`** and **`dcup`** commands) you will have to refresh your iface value
- When you run **`telnet`** properly, you will be asked to sign in. Use your current credentials (the login credentials for your VM.

Submission

1. Show a screenshot of the program capturing any TCP packet coming from our IP and with a destination port 23. Make sure the screenshot shows the src and dst fields.
2. Show a screenshot of you using the **`telnet`** command with the ip address you're connecting to visible.
3. Show what you changed about the code for the second filtering task.
4. Show a screenshot of the program capturing packets coming or going from the particular subnet given.
5. Show what you changed about the code for the last filtering task.
6. You should notice a difference in the packets displayed in this last filtering task vs in task 1.1A. What is the difference?

Task 2: Spoofing ICMP Packets

As a packet spoofing tool, Scapy allows us to set the fields of IP packets to arbitrary values. The objective of this task is to spoof IP packets with an arbitrary source IP address. We will spoof ICMP echo request packets, and send them to another VM on the same network. You can use Wireshark to observe whether our request will be accepted by the receiver. If it is accepted, an echo reply packet will be sent to the spoofed IP address. The following code shows an example of how to spoof an ICMP packet.

Listing 5: Example of how to spoof an ICMP packet

```
>>> from scapy.all import *
>>> a = IP() ①
>>> a.dst = '10.0.2.3' ②
>>> b = ICMP() ③
>>> p = a/b ④
>>> send(p) ⑤
. Sent 1 packets.
```

In the code above, Line ① creates an IP object from the IP class; a class attribute is defined for each IP header field. We can use **`ls(a)`** or **`ls(IP)`** to see all the attribute names/values. We can also use **`a.show()`** and **`IP.show()`** to do the same. Line ② shows how to set the destination IP address field. If a field is not set, a default value will be used.

Listing 5: List of IP attributes/values

```
>>> ls(a)
version : BitField (4 bits) = 4 (4)
ihl : BitField (4 bits) = None (None)
tos : XByteField = 0 (0)
len : ShortField = None (None)
id : ShortField = 1 (1)
flags : FlagsField (3 bits) = ()
frag : BitField (13 bits) = 0 (0)
```

```
ttl : ByteField = 64 (64)
proto : ByteEnumField = 0 (0)
chksum : XShortField = None (None)
src : SourceIPField = '127.0.0.1' (None)
dst : DestIPField = '127.0.0.1' (None)
options : PacketListField = [] ([])
```

Line ③ creates an ICMP object. The default type is echo request. In Line ④, we stack **a** and **b** together to form a new object. The **/** operator is overloaded by the IP class, so it no longer represents division; instead, it means adding **b** as the payload field of **a** and modifying the fields of a accordingly. As a result, we get a new object that represents an ICMP packet. We can now send out this packet using **send**() in Line ⑤. Please make any necessary changes to the sample code, and then demonstrate that you can spoof an ICMP echo request packet with an arbitrary source IP address.

Hints:

- In order to "prove" you are spoofing ICMP packets you have two options:
    - You need to run your code from task 1A in a separate terminal (in interactive mode) to capture packets. Make the necessary changes to capture an ICMP echo-reply packet.
    - Use Wireshark to capture packets.

Submission

1. A screenshot of the completed code. This can simply be a screenshot of an interactive python session. Be sure you include all the relevant lines, you may need to redo it so it is neater for the screenshot.
2. Capture a screenshot of the original terminal window running your code from task 1A with an ICMP echo-reply packet. Be sure to include the spoofed src and dst fields in the screenshot.
3. A few sentences explaining your code changes and strategy.

Task 3: Traceroute

The objective of this task is to use **Scapy** to estimate the distance, in terms of number of routers, between your VM and a selected destination. This is basically what is implemented by the **traceroute** tool. In this task, we will write our own tool. The idea is quite straightforward: just send a packet (any type) to the destination, with its Time-To-Live (TTL) field set to 1 first. This packet will be dropped by the first router, which will send us an ICMP error message, telling us that the time-to-live has exceeded. That is how we get the IP address of the first router. We then increase our TTL field to 2, send out another packet, and get the IP address of the second router. We will repeat this procedure until our packet finally reaches the destination. It should be noted that this experiment only gets an estimated result, because in theory, not all these packets take the same route (but in practice, they may within a short period of time). The code in the following shows one round in the procedure.

Listing 6: Snippet of procedure for estimating distance between routers

```
a = IP()
a.dst = ‘1.2.3.4’
a.ttl = 3
b = ICMP()
send(a/b)
```

If you are an experienced Python programmer, you can write your tool to perform the entire procedure automatically. If you are new to Python programming, you can do it by manually changing the TTL field in each round, and record the IP address based on your observation from Wireshark. Either way is acceptable, as long as you get the result.

Submission

1. Show screenshot of the completed code.
2. A screenshot of your code running and computing the distance between the VM and google.com.
3. A brief explanation of your approach.

Task 4: Sniffing-and-then-Spoofing

In this task, you will combine the sniffing and spoofing techniques to implement the following sniff-then-spoof program. You need two machines on the same LAN: the VM and the user container. From the user container, you **`ping`** an IP X. This will generate an ICMP echo request packet. If X is alive, the **`ping`** program will receive an echo reply, and print out the response. Your sniff-and-then-spoof program runs on the VM, which monitors the LAN through packet sniffing. Whenever it sees an ICMP echo request, regardless of what the target IP address is, your program should immediately send out an echo reply using the packet spoofing technique. Therefore, regardless of whether machine X is alive or not, the **`ping`** program will always receive a reply, indicating that X is alive. You need to use **`scapy`** to do this task. In your report, you need to provide evidence to demonstrate that your technique works.

Hints:

- If you have a packet p using multiple protocols (i.e. IP and ICMP), you can access the fields for a particular protocol with **`p[protocol].field`**. For example, to change the IP address **`src`** field, you would write **`p[IP].src = '...'.`**
- To correctly spoof a reply, it may be helpful to observe some real ICMP traffic with **`ping`** in a terminal
- You can use **`scapy`**'s **`show`** function to print out parts of the packet when debugging to reduce clutter, for example **`p[ICMP].show()`**
- Note that you need to use a different interface for monitoring traffic on the Internet (not just the traffic between docker containers). Recall you can specify an array of interfaces to **`scapy`**'s **`sniff`** function, instead of just using one.
- To capture packets indefinitely, we've been using Scapy's **`sniff`** function. You can change this to only capture n packets by passing **`sniff`** the **`count=n`** parameter.

Verifying Success:
If you run your program and **`ping 1.2.3.4`** at the same time (concurrently in separate terminals), your program should be able to spoof acceptable responses. That means that the **`ping`** command should behave as if a machine with IP address **`1.2.3.4`** is responding. If you get "Destination Host Unreachable" for **`ping 1.2.3.4`**, your code is not correctly spoofing yet.

Submission

1. A screenshot of the code written to complete the task above that successfully combines sniffing and spoofing.
2. Provide an explanation of your approach and how you confirmed that the attack was successful.