Sniffing/Spoofing Attacks (CS 915) Post-Lab Assignment Report

Hanzhi Zhang - 5525549

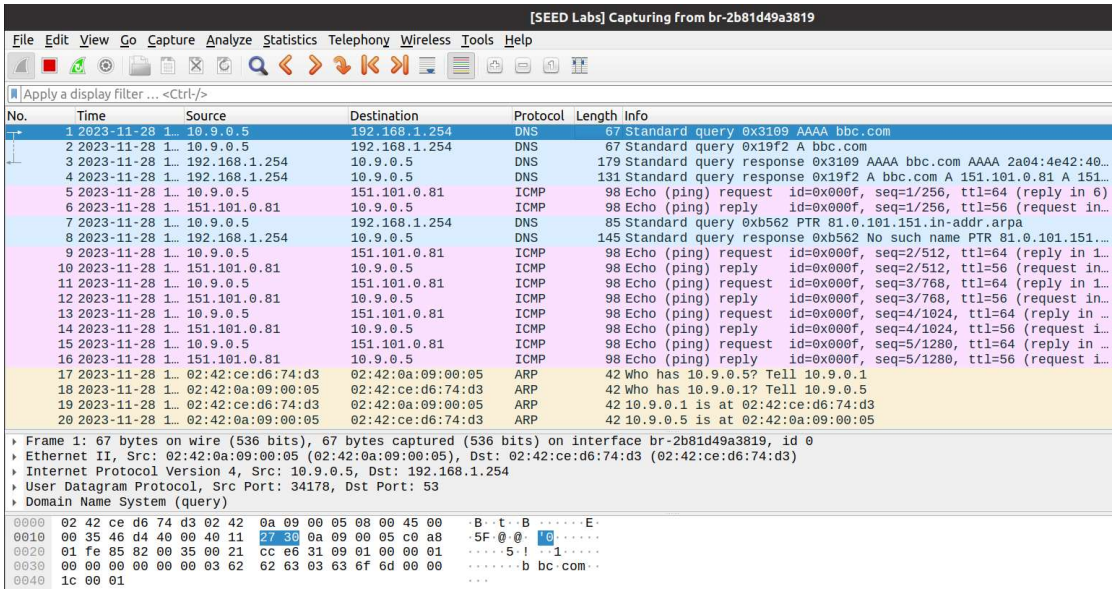
1. Task 1: Explain how you use Scapy, Wireshark and tcpdump to sniff packets.

```
[11/28/23]seed@VM:~/.../Lab 5$ sudo ./sniffer.py
###[ Ethernet ]###
  dst       = 02:42:ce:d6:74:d3
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 60999
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xaa9d
     src       = 10.9.0.5
     dst       = 151.101.0.81
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xc874
        id        = 0xe
        seq       = 0x1
###[ Raw ]###
           load      = '\xa6\\fe\x00\x00\x00\x00\
f !"#$%&\'()*+,-./01234567'
```

Scapy: set iface='br-2b81d49a3819' in sniffer.py and execute the code.

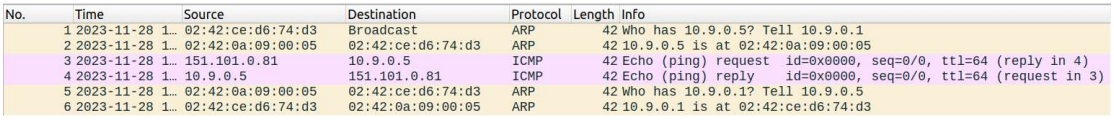
Wireshark: click 'br-2b81d49a3819', the pink entries are ICMP packets.

tcpdump: use 'icmp' in the command to filter. In all the cases we get pairs of icmp ping request and replies, the number of them consistent with how many packets victim machine sends and which ones are received.




```
[11/28/23]seed@VM:~/.../Lab 5$ sudo tcpdump icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on enp0s3, link-type EN10MB (Ethernet), capture size 262144 bytes
16:58:18.320050 IP VM > 151.101.192.81: ICMP echo request, id 26, seq 1, length 64
16:58:18.372853 IP 151.101.192.81 > VM: ICMP echo reply, id 26, seq 1, length 64
16:58:19.447752 IP VM > 151.101.192.81: ICMP echo request, id 26, seq 2, length 64
16:58:19.496118 IP 151.101.192.81 > VM: ICMP echo reply, id 26, seq 2, length 64
16:58:20.450037 IP VM > 151.101.192.81: ICMP echo request, id 26, seq 3, length 64
16:58:20.467524 IP 151.101.192.81 > VM: ICMP echo reply, id 26, seq 3, length 64
```

2. Task 2: Explain how you modify the provided code to spoof ICMP packets and the rationale for the modification. Present spoofing attack results with screenshots.①



No.	Time	Source	Destination	Protocol	Length	Info
1	2023-11-28 1…	02:42:ce:d6:74:d3	Broadcast	ARP	42	Who has 10.9.0.5? Tell 10.9.0.1
2	2023-11-28 1…	02:42:0a:09:00:05	02:42:ce:d6:74:d3	ARP	42	10.9.0.5 is at 02:42:0a:09:00:05
3	2023-11-28 1…	151.101.0.81	10.9.0.5	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=64 (reply in 4)
4	2023-11-28 1…	10.9.0.5	151.101.0.81	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=64 (request in 3)
5	2023-11-28 1…	02:42:0a:09:00:05	02:42:ce:d6:74:d3	ARP	42	Who has 10.9.0.1? Tell 10.9.0.5
6	2023-11-28 1…	02:42:ce:d6:74:d3	02:42:0a:09:00:05	ARP	42	10.9.0.1 is at 02:42:ce:d6:74:d3

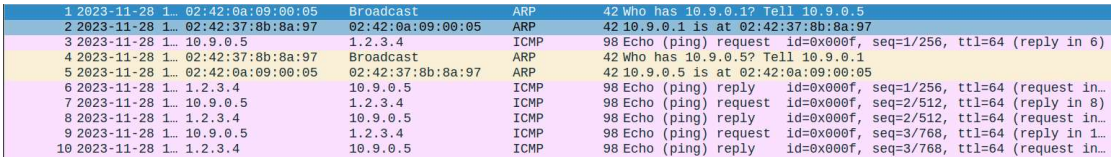
We add a.src = '151.101.0.81' to the provided code (between Line ① and Line ②), so a forged ping request from BBC.com is sent to the victim, the victim replied to it.

3. Task 3: Explain how you do the sniff-then-spoof attack in Task 3.

/* During this phase my Virtual Machine shut down and I had to restart it so the network interface name of the victim machine is different from that in Task 1 & 2 */

```
root@646b81e88f7a:/# ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
64 bytes from 1.2.3.4: icmp_seq=1 ttl=64 time=69.8 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=30.4 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=33.1 ms
^C
--- 1.2.3.4 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 30.435/44.441/69.823/17.979 ms
```

```
[11/28/23]seed@VM:~/.../Lab 5$ sudo ./sniff_spoof.py
Original Packet.........
Source IP :  10.9.0.5
Destination IP : 1.2.3.4
Spoofed Packet.........
Source IP :  1.2.3.4
Destination IP : 10.9.0.5
```



1	2023-11-28 1…	02:42:0a:09:00:05	Broadcast	ARP	42	Who has 10.9.0.1? Tell 10.9.0.5
2	2023-11-28 1…	02:42:37:8b:8a:97	02:42:0a:09:00:05	ARP	42	10.9.0.1 is at 02:42:37:8b:8a:97
3	2023-11-28 1…	10.9.0.5	1.2.3.4	ICMP	98	Echo (ping) request id=0x000f, seq=1/256, ttl=64 (reply in 6)
4	2023-11-28 1…	02:42:37:8b:8a:97	Broadcast	ARP	42	Who has 10.9.0.5? Tell 10.9.0.1
5	2023-11-28 1…	02:42:0a:09:00:05	02:42:37:8b:8a:97	ARP	42	10.9.0.5 is at 02:42:0a:09:00:05
6	2023-11-28 1…	1.2.3.4	10.9.0.5	ICMP	98	Echo (ping) reply id=0x000f, seq=1/256, ttl=64 (request in…
7	2023-11-28 1…	10.9.0.5	1.2.3.4	ICMP	98	Echo (ping) request id=0x000f, seq=2/512, ttl=64 (reply in 8)
8	2023-11-28 1…	1.2.3.4	10.9.0.5	ICMP	98	Echo (ping) reply id=0x000f, seq=2/512, ttl=64 (request in…
9	2023-11-28 1…	10.9.0.5	1.2.3.4	ICMP	98	Echo (ping) request id=0x000f, seq=3/768, ttl=64 (reply in 1…
10	2023-11-28 1…	1.2.3.4	10.9.0.5	ICMP	98	Echo (ping) reply id=0x000f, seq=3/768, ttl=64 (request in…

The sniff-and-then-spoof attack was successful, the victim was tricked to "believe" that it has received ICMP echo replies from the non-existing host 1.2.3.4.

```
root@646b81e88f7a:/# ping 10.9.0.99
PING 10.9.0.99 (10.9.0.99) 56(84) bytes of data.
From 10.9.0.5 icmp_seq=1 Destination Host Unreachable
From 10.9.0.5 icmp_seq=2 Destination Host Unreachable
From 10.9.0.5 icmp_seq=3 Destination Host Unreachable
^C
--- 10.9.0.99 ping statistics ---
5 packets transmitted, 0 received, +3 errors, 100% packet loss, time 4109ms
pipe 4
```

```
[11/28/23]seed@VM:~/.../Lab 5$ sudo ./sniff_spoof.py
```

1	2023-11-28 1…	02:42:0a:09:00:05	Broadcast	ARP	42	Who has 10.9.0.99? Tell 10.9.0.5
2	2023-11-28 1…	02:42:0a:09:00:05	Broadcast	ARP	42	Who has 10.9.0.99? Tell 10.9.0.5
3	2023-11-28 1…	02:42:0a:09:00:05	Broadcast	ARP	42	Who has 10.9.0.99? Tell 10.9.0.5
4	2023-11-28 1…	02:42:0a:09:00:05	Broadcast	ARP	42	Who has 10.9.0.99? Tell 10.9.0.5
5	2023-11-28 1…	02:42:0a:09:00:05	Broadcast	ARP	42	Who has 10.9.0.99? Tell 10.9.0.5
6	2023-11-28 1…	02:42:0a:09:00:05	Broadcast	ARP	42	Who has 10.9.0.99? Tell 10.9.0.5

The sniff-and-then-spoof was unsuccessful, the victim kept sending ARP broadcasts but did not send any ICMP requests, the attacker did not sniff any echo request, and had no chance to forge a reply. The victim decides that destination host is unreachable.

```
root@646b81e88f7a:/# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=56 time=51.2 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=64 time=70.1 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=2 ttl=64 time=22.9 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=56 time=30.2 ms (DUP!)
^C
--- 8.8.8.8 ping statistics ---
2 packets transmitted, 2 received, +2 duplicates, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 22.929/43.586/70.072/18.483 ms
```

```
[11/28/23]seed@VM:~/.../Lab 5$ sudo ./sniff_spoof.py
Original Packet.........
Source IP :  10.9.0.5
Destination IP : 8.8.8.8
Spoofed Packet.........
Source IP :  8.8.8.8
Destination IP : 10.9.0.5
```

7	2023-11-28 1…	10.9.0.5	8.8.8.8	ICMP	98	Echo (ping) request id=0x0011, seq=1/256, ttl=64 (reply in 1…
8	2023-11-28 1…	02:42:37:8b:8a:97	Broadcast	ARP	42	Who has 10.9.0.5? Tell 10.9.0.1
9	2023-11-28 1…	02:42:0a:09:00:05	02:42:37:8b:8a:97	ARP	42	10.9.0.5 is at 02:42:0a:09:00:05
10	2023-11-28 1…	8.8.8.8	10.9.0.5	ICMP	98	Echo (ping) reply id=0x0011, seq=1/256, ttl=56 (request in…
11	2023-11-28 1…	8.8.8.8	10.9.0.5	ICMP	98	Echo (ping) reply id=0x0011, seq=1/256, ttl=64
12	2023-11-28 1…	10.9.0.5	8.8.8.8	ICMP	98	Echo (ping) request id=0x0011, seq=2/512, ttl=64 (reply in 1…
13	2023-11-28 1…	8.8.8.8	10.9.0.5	ICMP	98	Echo (ping) reply id=0x0011, seq=2/512, ttl=64 (request in…
14	2023-11-28 1…	8.8.8.8	10.9.0.5	ICMP	98	Echo (ping) reply id=0x0011, seq=2/512, ttl=56
15	2023-11-28 1…	02:42:0a:09:00:05	02:42:37:8b:8a:97	ARP	42	Who has 10.9.0.1? Tell 10.9.0.5
16	2023-11-28 1…	02:42:37:8b:8a:97	02:42:0a:09:00:05	ARP	42	10.9.0.1 is at 02:42:37:8b:8a:97

The attacker did sniff the ICMP echo requests and sent back forged replies, but the real host also sent its replies, so the victim received duplicate replies (DUP!).

4. Answer the following two questions in Task 3.

   In the sniffing and then spoofing experiment in the lab, why can you get the echo reply from 1.2.3.4 which does not exist on the Internet?

Type 'ip route get 1.2.3.4' in the victim's terminal we see '1.2.3.4 via 10.9.0.1…', so the router our victim uses for 1.2.3.4 is actually the attacker! When the victim attempts to pin '1.2.3.4' from the Internet, an ICMP packet is sent using the attacker's MAC address as destination. (If this MAC is not in its ARP cache yet, it also sends its ARP broadcast to find out the MAC of the "router", as this "router" does exist, ICMP request will be sent eventually). The attacker sniffs the packet and spoofs a forged echo reply.

   In the above experiment, why can’t you get the echo reply from 10.9.0.99?

Type 'ip route get 10.9.0.99' in the victim's terminal we see '10.9.0.99…', that is to say as 10.9.0.99 and our victim are (supposed to be) in the same LAN, the victim does not use a router to forward packets to this destination. When it attempts to ping the IP, the ARP broadcasts are sent to find 10.9.0.99's MAC address, which it will never know for the host simply does not exist. As the victim is always broadcasting and waiting for the ARP reply, an ICMP request was never sent; we cannot reply to an "unsent" request.

An investigation to explain why Wireshark is still able to sniff packets:

```
[11/28/23]seed@VM:~/.../Lab 5$ pgrep wireshark
3743
[11/28/23]seed@VM:~/.../Lab 5$ ps -fp 3743
UID          PID    PPID  C STIME TTY          TIME CMD
seed        3743    2130  0 19:58 ?        00:00:12 wireshark
[11/28/23]seed@VM:~/.../Lab 5$ pstree -p 3743
wireshark(3743)─┬─dumpcap(3795)
                ├─{wireshark}(3747)
                ├─{wireshark}(3748)
                ├─{wireshark}(3749)
                ├─{wireshark}(3750)
                └─{wireshark}(3754)
[11/28/23]seed@VM:~/.../Lab 5$ ps -fp 3795
UID          PID    PPID  C STIME TTY          TIME CMD
seed        3795    3743  0 19:59 ?        00:00:00 /usr/bin/dumpcap -n -i br-48f7a03f1612
```

First, we use "pgrep" to find the process ID of the running program Wireshark.
The UID (effective user ID) of the Wireshark process is indeed "seed".
Using "pstree -p", we see a child process dumpcap is launced by Wireshark.
But the UID of the dumpcap process is also "seed"!

```
[11/28/23]seed@VM:~/.../Lab 5$ getcap /usr/bin/wireshark
[11/28/23]seed@VM:~/.../Lab 5$ getcap /usr/bin/dumpcap
/usr/bin/dumpcap = cap_net_admin,cap_net_raw+eip
```

Does Wireshark have any special Linux capabilities? No.
What about dumpcap? Yes, it has CAP_NET_ADMIN and CAP_NET_RAW, etc.

Judging from the name we guess that CAP_NET_RAW might be the capability we are more interested in – we want to know why the sniffer is able to use a raw socket without the root privilege. See Linux manual for capabilities, under the CAP_NET_RAW entry there is this line "Use RAW and PACKET sockets", just what we are looking for.

Now we see having root privilege is sufficient, not necessary, for a program to be able to use a raw socket (and to sniff packets). Here the UID of dumpcap is still not root, but the program is granted the capability to use raw sockets separately. Note that even this capability is given only to the capture utility, dumpcap, which Wireshark calls as a child process, not the entire program. This corresponds with the principle of least privilege.