数据包嗅探和伪造实验

1 概述

数据包嗅探和伪造是网络安全中两个重要的概念，它们是网络通信中的两大威胁。能够理解这些威胁对于了解网络中的安全措施至关重要。有许多嗅探和伪造工具，如 `Wireshark`, `Tcpdump`, `Scapy` 等等。这些工具被安全专家以及攻击者广泛使用。能够使用这些工具固然重要，但更为重要的是了解这些工具是如何工作的，即在软件中如何实现嗅探和伪造。

这个实验的目的是两方面的：学习使用这些工具并理解这些工具背后的技术。对于第二个目的，学生们将编写简单的嗅探和伪造程序，并深入理解这些程序的技术方面。本实验涵盖以下主题：

- 数据包嗅探和伪造的工作原理
- 使用 `pcap` 库和 Scapy 进行包嗅探
- 使用 raw socket 和 Scapy 进行数据包伪造
- 使用 Scapy

阅读资料和视频。 有关嗅探和伪造的详细讨论可以参见以下内容：

- SEED 教科书 *Computer & Internet Security: A Hands-on Approach*, 3rd Edition, by Wenliang Du. 详情请见 `https://www.handsonsecurity.net`.

实验环境。 本实验在 SEED Ubuntu 20.04 VM 中测试可行。您可以从 SEED 网站上下载我们预先构建好的镜像并在您自己的电脑上运行 SEED VM。然而，大多数 SEED 实验可以在云端进行，您可以按照我们的说明在云端创建 SEED VM。

教师注意事项。 这个实验包含两组任务。第一组任务侧重于使用工具进行包嗅探和伪造，这只需要一点 Python 编程（通常几行代码），学生不需要有太多的 Python 编程背景。

第二组任务主要针对计算机专业的学生。学生需要从头开始编写 C 程序来进行嗅探和伪造。这种方式可以让他们更深入地了解这些工具是如何工作的。学生们需要有一定的编程背景才能完成这项任务。这两组任务是独立的，老师可以根据学生的编程背景选择给学生布置一组或多组。

2 使用容器设置实验环境

在本实验中，我们将使用三台连接在同一局域网中的机器。我们可以使用三个虚拟机（VM）或三个容器。图 1展示了如何使用容器设置实验环境。我们将所有攻击都在攻击者容器上执行，同时将其他容器用作用户机器。

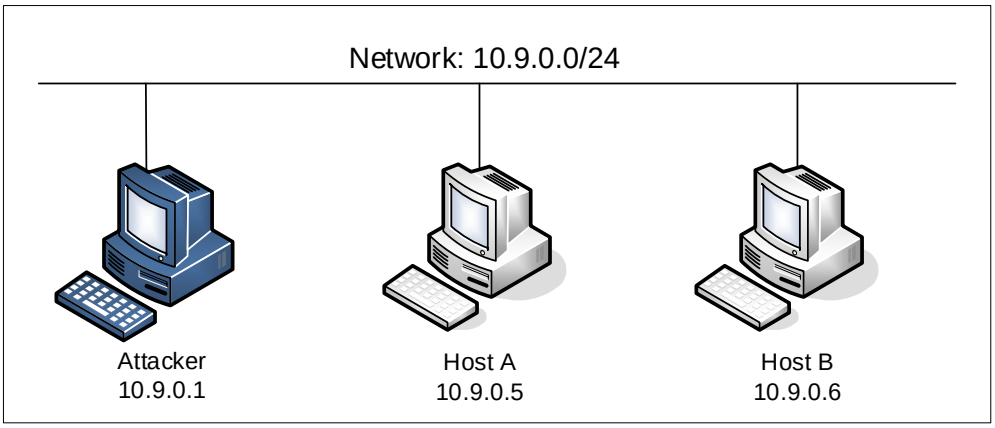

图 1: 实验环境设置

2.1 容器配置与命令

请从实验的网站下载 Labsetup.zip 文件到你的 VM 中，解压它，进入 Labsetup 文件夹，然后用 docker-compose.yml 文件安装实验环境。对这个文件及其包含的所有 Dockerfile 文件中的内容的详细解释都可以在链接到本实验网站的用户手册 [1] 中找到。如果这是您第一次使用容器设置 SEED 实验环境，那么阅读用户手册非常重要。

在下面，我们列出了一些与 Docker 和 Compose 相关的常用命令。由于我们将非常频繁地使用这些命令，因此我们在 .bashrc 文件（在我们提供的 SEED Ubuntu 20.04 虚拟机中）中为它们创建了别名。

```
$ docker-compose build  # 建立容器镜像
$ docker-compose up     # 启动容器
$ docker-compose down   # 关闭容器

// 上述 Compose 命令的别名
$ dcbuild       # docker-compose build 的别名
$ dcup          # docker-compose up 的别名
$ dcdown        # docker-compose down 的别名
```

所有容器都在后台运行。要在容器上运行命令，我们通常需要获得容器里的 Shell 。首先需要使用 docker ps 命令找出容器的 ID ，然后使用 docker exec 在该容器上启动 Shell 。我们已经在 .bashrc 文件中为这两个命令创建了别名。

```
$ dockps        // docker ps --format "{{.ID}}  {{.Names}}" 的别名
$ docksh <id>   // docker exec -it <id> /bin/bash 的别名

// 下面的例子展示了如何在主机 C 内部得到 Shell
$ dockps
```

[1] 如果你在部署容器的过程中发现从官方源下载容器镜像非常慢，可以参考手册中的说明使用当地的镜像服务器

```
b1004832e275  hostA-10.9.0.5
0af4ea7a3e2e  hostB-10.9.0.6
9652715c8e0a  hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#

// 注: 如果一条 docker 命令需要容器 ID, 你不需要
//     输入整个 ID 字符串。只要它们在所有容器当中
//     是独一无二的, 那只输入前几个字符就足够了。
```

如果你在设置实验环境时遇到问题，可以尝试从手册的 “Miscellaneous Problems” 部分中寻找解决方案。

2.2 攻击者容器介绍

在本实验中，我们可以使用虚拟机或攻击者容器作为攻击机器。如果你查看 Docker Compose 文件，你会看到攻击者容器的配置与其他容器不同。以下是这些差异：

- *共享文件夹*。当我们使用攻击者容器执行攻击时，需要将攻击代码放在攻击者容器内部。在虚拟机中进行代码编辑比在容器中更为方便，因为我们可以使用我们喜欢的编辑器。为了使虚拟机和容器共享文件，我们使用 Docker `volumes` 在虚拟机和容器之间创建了一个共享文件夹。如果你查看 Docker Compose 文件，就会发现我们已经在某些容器中添加了以下条目。它表示将主机（即 VM）上的 `./volumes` 文件夹挂载到容器内的 `/volumes` 文件夹。我们在虚拟机上将代码写入 `./volumes` 文件夹，就可以在容器内使用它们。

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

- *主机模式*。在本实验中，攻击者需要能够嗅探数据包。但在容器内运行嗅探程序存在问题，因为每个容器实际上是连接到一个虚拟交换机上，所以它只能看到自己的流量，而无法看到其他容器间的数据包。为了解决这个问题，我们将攻击者容器的网络模式设置为 `host` 模式，这允许攻击者容器看到所有的流量。以下是用于配置攻击者容器的条目：

  ```
  network_mode: host
  ```

  当容器的网络处于 `host` 模式，它可以看到主机的所有网络接口，且甚至拥有与主机相同的 IP 地址。它大体上与主机处于同一网络命名空间。然而，这个容器仍然是一台独立的机器，因为其他命名空间与主机不同。

获取网络接口名称。 当我们使用提供的 Compose 文件为实验创建容器时，Docker 会自动创建一个新的网络来连接 VM 和容器。该网络的 IP 前缀为 `10.9.0.0/24`，这在 `docker-compose.yml` 文件中指明。分配给 VM 的 IP 地址是 `10.9.0.1`。我们需要找到宿主机上对应网络接口的名称，因为我们在程

序中需要使用它。接口名称是由 br- 和 Docker 创建的网络 ID 拼接而成的。当我们使用 ifconfig 列出所有网络接口时，可以看到许多条目。寻找具有 IP 地址 10.9.0.1 的那一个。

```
$ ifconfig
br-c93733e9f913: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.1  netmask 255.255.255.0  broadcast 10.9.0.255
        ...
```

另一种获取接口名称的方法是使用 docker network 命令来查找网络 ID（网络名为 seed-net）：

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

3 实验任务集 1：使用 Scapy 嗅探和伪造数据包

许多工具都可以用于进行嗅探和伪造，但大多数只能提供固定功能。而 Scapy 不同之处在于它不仅可以作为一个工具来使用，还可以作为构建其他嗅探和伪造工具的模块。也就是说，我们可以将 Scapy 的功能集成到我们自己的程序中。在这个任务集中，我们将使用 Scapy。

要使用 Scapy，可以编写一个 Python 程序。以下是一个例子。为了嗅探包，我们需要使用 root 权限运行 Python。在程序开始处（第①行），应导入 Scapy 模块。

```
# view mycode.py
#!/usr/bin/env python3

from scapy.all import *    ①

a = IP()
a.show()

# python3 mycode.py
###[ IP ]###
  version   = 4
  ihl       = None
  ...

// 将程序设置为可执行文件（另一种运行 Python 程序的方式）
# chmod a+x mycode.py
# mycode.py
```

我们也可以进入 Python 的交互模式，在 Python 提示符下一行一行地运行我们的代码。

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

3.1 任务 1.1：嗅探包

Wireshark 是最常用的嗅探工具，且易于使用。在整个实验中我们将使用它。但是作为构建其他工具的基础模块，Wireshark 却很难胜任这一点。我们将使用 Scapy 来完成这一任务。这个任务的目标是学习如何在 Python 程序中使用 Scapy 进行包嗅探。以下提供了一个示例代码：

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

def print_pkt(pkt):
  pkt.show()

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

上述代码将在 `br-c93733e9f913` 接口上嗅探数据包。请参阅实验环境设置部分的说明来获取正确的接口名称。如果要同时在多个接口上嗅探包，可以将这些接口放入一个列表，并将其分配给 `iface` 变量。以下是一个例子：

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

任务 1.1A。 在上述程序中，对于捕获的每个包，回调函数 `print_pkt()` 将被调用，这个函数将打印一些关于该包的信息。请使用 root 权限运行程序并演示你确实可以捕获到包。然后再次运行程序，但不使用 root 权限，描述和解释你的观察结果。

```
// 使程序可执行
# chmod a+x sniffer.py

// 使用 root 权限运行程序
# sniffer.py

// 切换到 seed 账号，
// 不使用 root 权限再次运行程序
# su seed
$ sniffer.py
```

任务 1.1B。 通常，当我们嗅探包时，我们只对某些类型的包感兴趣。可以通过设置嗅探过滤器来实现这一点。Scapy 的过滤器使用 Berkeley Packet Filter (BPF) 语法，可以从互联网上找到 BPF 手册。请设置以下过滤器并再次演示你的嗅探程序（每个过滤器应单独设置）。

- 只捕获 ICMP 包
- 捕获来自某个特定 IP 地址并且目标端口号为 23 的 TCP 包。
- 捕获来自或去往某个特定网络的包。可以选择任意网络，例如 `128.230.0.0/16`，但不应选择与你的虚拟机连接的同一子网。

3.2 任务 1.2：伪造 ICMP 包

作为包伪造工具，Scapy 允许我们在 IP 包各字段设置任意值。这个任务的目标在伪造的数据包里设置任意源 IP 地址。我们伪造 ICMP echo 请求包，并将它们发送到同一网络中的一台主机。我们将使用 Wireshark 来观察接收者是否接受了我们的请求。以下代码展示了如何伪造 ICMP 数据包。

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

在上述代码中，第①行创建了一个 IP 对象，每个 IP 头部字段都有一个类属性。我们可以使用 `ls(a)` 或 `ls(IP)` 来查看所有属性名称和它们的值。我们也可以用 a.show() 和 IP.show() 来做同样的事情。第②行显示了如何设置目标 IP 地址字段。如果某个字段未被设置，则会使用默认值。

```
>>> ls(a)
version    : BitField (4 bits)       = 4               (4)
ihl        : BitField (4 bits)       = None            (None)
tos        : XByteField              = 0               (0)
len        : ShortField              = None            (None)
id         : ShortField              = 1               (1)
flags      : FlagsField (3 bits)     = <Flag 0 ()>     (<Flag 0 ()>)
frag       : BitField (13 bits)      = 0               (0)
ttl        : ByteField               = 64              (64)
proto      : ByteEnumField           = 0               (0)
chksum     : XShortField             = None            (None)
src        : SourceIPField           = '127.0.0.1'     (None)
dst        : DestIPField             = '127.0.0.1'     (None)
options    : PacketListField         = []              ([])
```

第③行创建了一个 ICMP 对象。默认类型是 echo 请求。在第④行中，我们将 `a` 和 `b` 相结合以形成一个新的对象。`/` 操作符被 IP 类重载，因此它不再表示除法，而是意味着将 `b` 作为 `a` 的负载字段并相

应地修改 a 的字段。结果我们得到了一个代表 ICMP 包的新对象。我们现在可以使用第⑤行的 send() 发送这个包。请根据示例代码做出必要的更改，并演示你能够伪造 ICMP echo 请求包。

3.3 任务 1.3：Traceroute

本任务的目标是使用 Scapy 来估计从你的虚拟机到目标的距离，即这之间隔着多少个路由器，这其实就是 traceroute 工具所做的事情。在本任务中，我们将编写自己的工具，实现的办法也较简单。首先，我们向目的地发送一个 IP 数据包（可以是任何类型），将它的生存时间（TTL）字段设置为 1。这个包将在第一个路由器处被丢弃，并返回一个 ICMP 错误消息，告诉我们生存时间已经超时。这就是我们获得第一个路由器的 IP 地址的方式。然后我们将 TTL 字段增加到 2，再次发送数据包，这次这个包可以到达第二个路由器，才会被丢弃，我们因此可以获取第二个路由器的 IP 地址。我们将重复此过程直到最终我们的包到达目的地。需要注意的是，这个实验仅能获得估计结果，因为在理论上，这些包不一定沿着相同的路径行进（但在实践中，在短时间内包走的路径大概率是相同的）。以下代码展示了该过程的一个回合。

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

如果你是一个经验较丰富的 Python 程序员，你可以编写一个工具自动完成整个过程。如果你对 Python 编程还是是新手，可以通过手动更改每个回合的 TTL 字段并用 Wireshark 获得返回的 ICMP 包。无论哪一种方式都可以接受，只要你能得到结果即可。

3.4 任务 1.4: 窃听和伪造结合

在本任务中，我们将结合使用窃听和伪造技术。我们需要在同一局域网上的两台机器，虚拟机（VM）和用户容器。从用户容器上，ping 一个 IP 地址 X，这会生成一个 ICMP echo 请求数据包。如果目标 X 是在线的，那么 ping 程序将接收到 echo 回复，并打印出响应。你的程序在虚拟机上运行，通过数据包嗅探监控局域网。每当看到一个 ICMP echo 请求时（不论目标 IP 地址是什么），它立即使用数据包伪造技术发送 echo 回复。因此，无论机器 X 是否是在线的，ping 程序都会接收到回复。你需要使用 Scapy 来完成此任务。在报告中，请提供证据以证明你的程序是成功的。

在实验过程中，在用户容器上 ping 以下三个 IP，报告并解释观察到的结果。

```
ping 1.2.3.4     # 互联网上的一个不存在的主机
ping 10.9.0.99   # 局域网上的一个不存在的主机
ping 8.8.8.8     # 互联网上的一个存在的主机
```

提示： 为了正确解释观察结果，需要理解 ARP 协议的工作原理。此外，也需要知道一些路由方面的知识。以下命令可以帮助你找到指定目标的路由器：

```
ip route get 1.2.3.4
```

4 实验任务集 2：编写程序以窃听和欺骗数据包

在这个任务集中，我们在主机虚拟机上编译 C 代码，并在容器中运行该代码。可以使用 "docker cp" 命令将文件从主机虚拟机复制到一个容器中。以下是一个示例：

```
$ dockps
f4501a488a69  hostA-10.9.0.5
85058cbdee62  hostB-10.9.0.6
24cbc879e371  seed-attacker

// 将 a.out 复制到容器 seed-attacker 的 /tmp 文件夹中
$ docker cp a.out 24cbc879e371:/tmp
```

4.1 任务 2.1：编写数据包嗅探程序

使用 pcap 库很容易编写嗅探程序。有了 pcap ，嗅探器的任务就是调用 pcap 库中的的一系列函数。一旦捕获到数据包，这些数据包就会被放入缓冲区以供进一步处理，捕获数据包的所有细节都由 pcap 库处理。SEED 书籍提供了一个样本代码，展示如何使用 pcap 编写一个简单的嗅探程序。以下为样本代码（请参阅书籍中的详细解释）。

```
#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>

/* 当捕获到数据包时此函数会被调用，可以在函数内处理每个数据包 */
void got_packet(u_char *args, const struct pcap_pkthdr *header,
        const u_char *packet)
{
   printf("Got a packet\n");
}

int main()
{
  pcap_t *handle;
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program fp;
  char filter_exp[] = "icmp";
  bpf_u_int32 net;

  // 第一步：在 eth3 网卡上打开一个 pcap session。
  // 需要根据自己机器的情况更改 "eth3" (使用 ifconfig 查找)。
  handle = pcap_open_live("eth3", BUFSIZ, 1, 1000, errbuf);

  // 第二步：将 filter_exp 编译成 BPF 代码
  pcap_compile(handle, &fp, filter_exp, 0, net);
```

```
  if (pcap_setfilter(handle, &fp) !=0) {
      pcap_perror(handle, "Error:");
      exit(EXIT_FAILURE);
  }

  // 第三步: 捕获数据包
  pcap_loop(handle, -1, got_packet, NULL);

  pcap_close(handle);   // 关闭会话句柄
  return 0;
}

// 注意: 在编译命令中不要忘记添加 "-lpcap"。
// 例如: gcc -o sniff sniff.c -lpcap
```

Tim Carstens 撰写了一篇关于如何使用 pcap 库编写嗅探程序的教程。教程可以在以下网址找到: http://www.tcpdump.org/pcap.htm。

任务 2.1A: 理解窃听器的工作原理。 在这个任务中, 学生需要编写一个能够打印捕获到的数据包的信息的程序。您可以输入上面的代码或从 SEED 书籍网站下载样本代码 (https://www.handsonsecurity.net/figurecode.html)。您需要提供截图来证明您的程序可以成功运行并产生预期的结果。此外, 请回答以下问题:

- **问题 1**。请用自己的话描述对于窃听程序而言必不可少的库调用序列 (总结性的描述就可以, 不需要做详细的解释)。
- **问题 2**。为什么需要 root 权限来运行嗅探程序? 如果在没有 root 权限的情况下执行, 程序的哪一行会出问题?
- **问题 3**。请在嗅探程序中开启和关闭混杂模式 (promiscuous mode)。第三参数 pcap_open_live() 中的值为 1 表示开启混杂模式 (0 表示关闭)。当这种模式开启和关闭时程序有何差异? 请展示。可以使用以下命令检查接口的混杂模式是开启还是关闭 (查看 promiscuity 的值)。

```
# ip -d link show dev br-f2478ef59744
1249: br-f2478ef59744: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 ...
    link/ether 02:42:ac:99:d1:88 brd ff:ff:ff:ff:ff:ff promiscuity 1 ...
```

任务 2.1B: 编写筛选器。 请为你的嗅探程序编写筛选表达式以捕获以下内容。可以从在线手册中找到 pcap 筛选器说明书。在实验报告中, 需要用截图来展示应用这些筛选器后的结果。

- 捕捉两个主机之间的 ICMP 数据包。
- 捕捉目的地端口号是 10 到 100 范围内的 TCP 数据包。

任务 2.1C: 窃听密码。 请展示如何使用嗅探程序 `telnet` 连接中传输的密码。您可能需要修改窃听代码来打印出捕获的 TCP 数据包的数据部分（`telnet` 使用 TCP）。你可以打印整个数据部分，并手动标注出密码部分的位置。

4.2 任务 2.2: 伪造数据包

当普通用户发送数据包时，操作系统通常不允许用户设置协议头部的所有字段（例如 TCP、UDP 和 IP 头部）。系统会自动设置大多数字段，只允许用户设置少数字段，如目的地 IP 地址等。然而，如果用户具有 root 权限，则可以设置数据包头中的任何字段。这被称为数据包伪造，并可以通过 *raw socket* 来实现。

Raw socket 允许程序员控制数据包的构造，从而使程序员能够构建任意的数据包，包括设置头部字段和负载部分。使用 raw socket 涉及四个步骤：（1）创建 raw socket，（2）设置 socket 选项，（3）构造数据包，以及（4）通过 Raw socket 发送数据包。有许多在线教程可以教您如何在 C 语言中使用 raw socket，我们将一些教程链接到了实验网页上。下面我们展示一个简单框架。

```
int sd;
struct sockaddr_in sin;
char buffer[1024]; // 可以更改缓冲区大小

/* 创建一个具有 IP 协议的 raw socket。IPPROTO_RAW 参数表明
 * IP 头部已经提供，让操作系统不要添加新的 IP 头部 */
sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
if(sd < 0) {
    perror("socket() error"); exit(-1);
}

/* 这个数据结构用于发送数据包时使用。通常我们需要填充多个字段，
 * 但对于 raw sockets，我们只需要填充这个字段 */
sin.sin_family = AF_INET;

// 在这里可以使用 buffer[] 构造 IP 数据包
//    - 构建 IP 头部 ...
//    - 构建 TCP/UDP/ICMP 头部 ...
//    - 如果需要的话，填写数据部分 ...
// 需要注意字节顺序。


/* 发送 IP 数据包。 ip_len 是数据包的实际大小 */
if(sendto(sd, buffer, ip_len, 0, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
      perror("sendto() error"); exit(-1);
}
```

任务 2.2A: 编写伪造程序。 请用 C 语言编写自己的数据包伪造程序。需要提供证据（例如，使用 Wireshark 数据包跟踪）来证明你的程序成功发送了欺骗性的 IP 数据包。

任务 2.2B: 伪造 ICMP echo 请求数据包。 以另一台机器的名义上发起一个 ICMP echo 请求数据包（使用该机器的 IP 地址作为源 IP 地址），这个数据包应发送到互联网上的一台在线机器。用 Wireshark 观察远程机器是否有回复。

问题。 请回答以下问题：

- **问题 4**。可以将 IP 数据包长度字段设置为任意值吗，不管数据包的真正大小是多少？
- **问题 5**。使用 raw socket 编程时，需要计算 IP 头部的校验和吗？
- **问题 6**。为什么需要 root 权限才能使用 raw socket？如果在没有 root 权限的情况下执行会发生什么？

4.3 任务 2.3: 嗅探和伪造结合

在本任务中，我们将结合使用窃听和伪造技术。我们需要在同一局域网上的两台机器，虚拟机（VM）和用户容器。从用户容器上，`ping` 一个 IP 地址 X，这会生成一个 ICMP echo 请求数据包。如果目标 X 是在线的，那么 `ping` 程序将接收到 echo 回复，并打印出响应。你的程序在虚拟机上运行，通过数据包嗅探监控局域网。每当看到一个 ICMP echo 请求时（不论目标 IP 地址是什么），它立即使用数据包伪造技术发送 echo 回复。因此，无论机器 X 是否是在线的，`ping` 程序都会接收到回复。你需要使用 C 来完成此任务。在报告中，请提供证据以证明你的程序是成功的。

5 指南

5.1 Raw 数据包中的数据填充

当使用 raw socket 发送数据包时，本质上是将数据包构造在一个缓冲区中。因此，要发送它时，只需向操作系统提供该缓冲区和数据包大小即可。直接在缓冲区上操作并不容易，所以一种常用的方法是将缓冲区（或其一部分）转换为结构体，例如 IP 头部结构体，这样就可以通过那些结构体的字段来引用缓冲区的内容。可以在程序中定义 IP、ICMP、TCP 和 UDP 等头部结构。以下示例说明了如何构造一个 UDP 数据包：

```
struct ipheader {
   type  field;
   .....
}

struct udpheader {
   type field;
   ......
}
```

```
// 这个缓冲区将用于构建原始数据包。
char buffer[1024];

// 将缓冲区转换为 IP 头部结构体
struct ipheader *ip = (struct ipheader *) buffer;

// 将缓冲区转换为 UDP 头部结构体
struct udpheader *udp = (struct udpheader *) (buffer + sizeof(struct ipheader));

// 为 IP 和 UDP 头部字段赋值。
ip->field = ...;
udp->field = ...;
```

5.2 网络/主机字节序与转换

需要关注网络和主机的字节序。如果使用的是 x86 CPU，主机字节序采用小端格式，而网络字节序则采用大端格式。无论数据是放入包缓冲区还是其他地方，都必须使用网络字节序，否则，构建的包是不正确的。

我们倒不必去知道自己的机器是使用哪种字节序，只要记住在将数据放入数据包缓冲区时将其转换为网络字节序，在从数据包缓冲区复制到计算机的数据结构中时再转换回主机字节序。如果数据是一个单独的字节，则不需要考虑顺序，但如果是 `short`，`int`，`long` 或由多个字节组成的数据类型，则需要调用以下函数之一来转换数据：

```
htonl(): 将主机字节序的无符号整数转换为网络字节序。
ntohl(): htonl() 的逆操作
htons(): 将主机字节序的无符号短整数转换为网络字节序。
ntohs(): htons() 的逆操作。
```

可能还需要使用 `inet_addr()`，`inet_network()`，`inet_ntoa()`，`inet_aton()` 将点分十进制形式（字符串）的 IP 地址转换为网络/主机字节序下的 32 位整数。可以在互联网上获取这些函数的手册。

6 提交

你需要提交一份带有截图的详细实验报告来描述你所做的工作和你观察到的现象。你还需要对一些有趣或令人惊讶的观察结果进行解释。请同时列出重要的代码段并附上解释。只是简单地附上代码不加以解释不会获得学分。