Scapy的安装及其使用

目录

- 目录
- 【 Scapy简介】
- 【 Scapy的安装】
  - 1.在Windows平台下安装
    - 1)安装Python2.6
    - 2)下载并安装scapy及其它软件
  - 2.在Linux（ Ubuntu）平台下安装
- 【 Scapy的使用】
  - 开始使用
  - 交互应用
    - 1.构造包
    - 2.叠加包
    - 3.读取PCAP文件
    - 4.收发包
    - 5.嗅探(抓包)
  - 高级应用
    - 1.TCP traceroute
    - 2.经典攻击
- 【 函数手册】
  - 有关收发包及读写文件的函数
  - 对单个包的操作的函数
  - 对一组包操作的函数
- 【 Automata及其在Scapy中的应用】
  - 1.Automata简介
  - 2.Scapy中的automata
  - 3.automata中的修饰符（ decorators）
    - 3.1 状态（states）修饰符
    - 3.2 转化（transitions）修饰符
    - 3.3 行为（actions）修饰符
  - 4. Automata举例
    - 4.1 一个小例子
    - 4.2 一个有网络包参与的例子
    - 4.3 Scapy源码中TFTP client的实现
- 【 Scapy面向测试的应用举例】
  - 例一
  - 例二

【Scapy简介】

Scapy是一个Python组件，它可以用来发送、嗅探、析构和构造网络包。通俗地讲，我们可以把它理解为类似wireshark和omnipeek的网络收发包和分析工具。它支持有线和无线两种模式，而且在无线模式下，并不像omnipeek需要两块无线网卡，也不需要特定的驱动--只是它的无线模式不能工作在Windows系统中，需要工作在如Linux的系统环境中。

Scapy是一个强大的交互式包处理程序。它可以构造或者析构很多协议、通过有线(或无线?)的方式发送出去、捕捉包、匹配请求和回复等等。Scapy可以轻易地处理绝大部分经典的任务比如：扫描（scanning)、路由追踪(tracerouting)、探测(probing)、单元测试 （unit tests)、攻击（attacks)或者网络发现(network discovery)。它可以替代hping、arpspoof、arp-sk、arping、p0f甚至部分替代Nmap、tcpdump和 tshark等程序。

Scapy在一些其他绝大部分工具不能处理的某些工作中表现得也相当出色，像发送非法包、注入自己的802.11帧、组合技术（VLAN hopping+ARP cache poisoning、WEp加密信道上的VOIP 解码)

总而言之，Scapy是一个强大的收发包和网络分析工具。

【Scapy的安装】

1.在Windows平台下安装

1)安装Python2.6

可以参考以下安装指南：

Python安装指南 http://172.31.88.91:9090/pages/viewpage.action?pageId=7799691
Python安装指南及一些小技巧 http://172.31.88.91:9090/pages/viewpage.action?pageId=7801919

2)下载并安装scapy及其它软件

> 注意：安装scapy前请确认已安装Python。以下安装包中的scapy为在scapy2.1.1基础上修改和增添了PPTP，L2TP，802.11协议，bug。

（1）所需安装文件的路径为 \\172.31.88.5\TestPublic\Wireless\TestTool\scapy_setup.rar(最新版可以在SVN/PySAT/branches/scapy里得到，此目录下除scapy无其他辅助软件)，将该压缩包保存到本地并解压缩。（解压缩后的文件夹中会有scapy和其它一些所需安装的软件，以及一个 README.TXT文本文件。）

（2）打开cmd，切换到解压缩之后的文件夹所在的分区。比如将scapy_setup.rar解压缩到了D盘，则在cmd中输入D:后回车。

（3）切换到安装文件所在的文件夹，比如将scapy_setup.rar解压缩到了D盘下scapy_setup文件夹，则在cmd中输入cd scapy_setup后回车。

（4）在cmd中输入easy_install scapy-2.1.1CX.zip后回车，即开始安装scapy。

（5）直接运行pywin32-214.win32-py2.6.exe

（6）直接运行WinPcap_4_1_1.exe

（7）直接运行pcap-1.1-scapy-20090720.win32-py2.6.exe

（8）直接运行dnet-1.12.win32-py2.6.exe

（9）直接运行pyreadline-1.5-win32-setup.exe

按步骤执行以上操作后，安装过程即可完成。

2.在Linux（Ubuntu）平台下安装

本文安装scapy的Linux平台为Ubuntu 9.10。由于Ubuntu 9.10中已经内置了Python，则我们无需如Windows平台中先安装Python，直接安装scapy即可。

（1）到路径\\172.31.88.5\TestPublic\Wireless\Test Tool下载scapy2.1.1CX.zip到本地，并将其解压缩。

（2）在终端Terminal（Linux中的命令行）中切换到刚才解压缩后的文件夹的路径。

（3）在终端中输入命令:

```
$sudo python setup.py install
```

即可完成安装。

【Scapy的使用】

• 开始使用

Unix-like系统下（本文的平台为Ubuntu 9.10），在终端输入scapy后回车即可：

```
xzf@ubuntu:~$ scapy
INFO: Can't import python gnuplot wrapper . Won't be able to plot.
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump().
WARNING: No route found for IPv6 destination :: (no default route?)
/usr/local/lib/python2.6/dist-packages/scapy/crypto/cert.py:6: DeprecationWarning: the sha module
is deprecated; use the hashlib module instead
import os, sys, math, socket, struct, sha, hmac, string, time
/usr/local/lib/python2.6/dist-packages/scapy/crypto/cert.py:7: DeprecationWarning: The popen2
module is deprecated. Use the subprocess module.
import random, popen2, tempfile
Welcome to Scapy (2.1.1CX)
>>>
```

出现以上信息说明scapy安装成功，并可正常使用。

在Windows下，在控制台输入scapy后回车即可：

```
C:\Documents and Settings\Administrator>scapy
INFO: "Can't import python gnuplot wrapper . Won't be able to plot."
INFO: "Can't import PyX. Won't be able to use psdump() or pdfdump()."
INFO: No IPv6 support in kernel
WARNING: No route found for IPv6 destination :: (no default route?)
INFO: "Can't import python Crypto lib. Won't be able to decrypt WEP."
INFO: Can't import python Crypto lib. Disabled certificate manipulation tools
Welcome to Scapy (2.1.1CX)
>>>
```

出现以上信息和最后的提示符，说明scapy已经安装完成可以使用。

在以上的提示信息中，scapy有5个INFO和1个WARNING，这些不会影响我们的一般使用（提示的信息为缺少一些画图、pdf输出、

下面介绍具体的使用方法。

- 交互应用

1. 构造包

Scapy中，网络协议用类（class）来定义，比如Ethernet，IP，TCP等等。构造包，相当于对类（class）的实例化。比如我们要构造一个ttl=10的IP包：

```
>>> a = IP(ttl=10)
>>> a
< IP ttl=10 \|>
{newcode}Pythonclass.attributepkt.attribute:
{newcode}>>> a.src
'127.0.0.1'
```

如何知道我构造的包里都有哪些元素呢？用Scapy中内置的ls()函数，用法是ls(pkt)：

```
>>> ls(a)
version : BitField = 4 (4)
ihl : BitField = None (None)
tos : XByteField = 0 (0)
len : ShortField = None (None)
id : ShortField = 1 (1)
flags : FlagsField = 0 (0)
frag : BitField = 0 (0)
ttl : ByteField = 10 (64)
proto : ByteEnumField = 0 (0)
chksum : XShortField = None (None)
src : Emph = '127.0.0.1' (None)
dst : Emph = '127.0.0.1' ('127.0.0.1')
options : PacketListField = \[\] (\[\])
```

通过观察可以发现，除了ttl的值是与我们的赋值相符合的外，其它我们没有赋值的域，都自动赋为默认值。这极大的简化了构

给构造完成的包中的域赋值：

```
>>> a.dst = "192.168.1.1"
>>> a
< IP ttl=10 dst=192.168.1.1 \|>
```

删除构造完成的包中的某个域的赋值：

```
>>> del(a.ttl)
>>> a
< IP dst=192.168.1.1 \|>
>>> a.ttl
64
```

我们可以发现，被删除的域的值不是0，而是自动恢复为默认值。

2. 叠加包

使用'/'可以把两层的包组叠加起来：

```
>>>IP()
<IP  |>
>>> IP()/TCP()
<IP  frag=0 proto=tcp |<TCP  |>>
>>> Ether()/IP()/TCP()
<Ether  type=0x800 |<IP  frag=0 proto=tcp |<TCP  |>>>
>>> IP()/TCP()/"GET / HTTP/1.0\r\n\r\n"
<IP frag=0 proto=TCP |<TCP |<Raw load='GET / HTTP/1.0\r\n\r\n' |>>>
>>> Ether()/IP()/IP()/UDP()
<Ether type=0x800 |<IP frag=0 proto=IP |<IP frag=0 proto=UDP |<UDP |>>>>
>>> IP(proto=55)/TCP()
<IP frag=0 proto=55 |<TCP |>>
```

叠加包的时候我们可以发现，单独一个Ether（）包，里面的type为0x0，叠加一个IP（）包后，Ether（）里面的type值自动改为0x800。再如IP（）上面可以叠加TCP（），UDP（）等包，叠加后IP（）中的proto的值也会自动做出相应的变化。

```
>>> p = IP()/TCP()
>>> p[IP].proto
6
>>> p = IP()/UDP()
>>> p[IP].proto
17
```

这些改变是scapy自动做出的，不必我们手动修改，为我们叠加包提供了很大的便捷。

3. 读取PCAP文件

scapy可以读取wireshark等工具抓到的pcap文件，show()函数可以显示包。

```
>>> p = rdpcap(r'E:\workstation\ScapyRewriteProj\frameCatch\l2tp.pcap')
>>> p
<l2tp.pcap: TCP:0 UDP:80 ICMP:0 Other:0>
>>>p0=p[0]
>>> p0.show()
###[ Ethernet ]###
dst= 00:24:1d:d1:83:1a
src= 00:0a:eb:ce:1e:32
type= 0x800
###[ IP ]###
version= 4L
ihl= 5L
tos= 0x0
len= 134
id= 0
flags= DF
frag= 0L
ttl= 64
proto= udp
chksum= 0xa3fe
src= 192.168.10.156
dst= 192.168.10.124
\options\
###[ UDP ]###
sport= l2tp
dport= l2tp
len= 114
chksum= 0x35b6
###[ L2TP ]###
type= Control message
len_present= Present
reserved0= 0L
seq_present= Present
reserved1= 0L
offset_present= Not present
priorty_present= Not present
reserved2= 0L
vers= 2L
len= 106
tunnel_id= 0
session_id= 0
Ns= 0
Nr= 0
\AVPs\
| ###[ L2TP Control Message AVP ]### |
| mandatory= True |
| hidden= False |
| rsvd= 0L |
| len= 8L |
| vendor_id= 0 |
| attr_type= Control message |
| ctrlmsg_type= SCCRQ |
| . |
| . |
| . |
| ###[ L2TP Window Size AVP ]### |
| mandatory= True |
| hidden= False |
| rsvd= 0L |
| len= 8L |
| vendor_id= 0 |
| attr_type= Receive window size |
| winsize= 4
```

每个###[xxx]###是一个类，可以直接通过"[]"访问,例如###[L2TP Window Size AVP]###元组可通过如下访问：

```
>>> l=p0[L2TP_WINSIZE_AVP|L2TP_WINSIZE_AVP]
>>> l.show()
###[ L2TP Window Size AVP ]###
 mandatory= True
 hidden= False
 rsvd= 0L
 len= 8L
 vendor_id= 0
 attr_type= Receive window size
 winsize= 4
```

注：L2TP_WINSIZE_AVP是L2TP Window Size AVP的类，可以通过p0.payload查看
hidden,mandatory是类L2TP_WINSIZE_AVP的属性，可以通过"."访问 ，例如：

```
>>> l.len
 8L
>>> l.att_type
 10
```


注：用show()函数显示的是解析后的数据，通过"."访问显示后的是原始数据。比如上例中的att_type

4.收发包

sr()和srp()函数均可以收发包，所不同的是sr是所收发的包必须为3层，而srp则为2层

```
>>> sr(IP(dst="192.168.1.1")/TCP(dport=[21,22,23]))
 Begin emission:
 Finished to send 3 packets.
 ...*.*.*
 Received 8 packets, got 3 answers, remaining 0 packets
 (<Results: TCP:3 UDP:0 ICMP:0 Other:0>, <Unanswered: TCP:0 UDP:0 ICMP:0 Other:0>)
 >>> ans,unans = _
 >>> ans.summary()
 IP / TCP 192.168.1.100:ftp_data > 192.168.1.1:ftp S ==> IP / TCP 192.168.1.1:ftp >
192.168.1.100:ftp_data RA / Padding
 IP / TCP 192.168.1.100:ftp_data > 192.168.1.1:22 S ==> IP / TCP 192.168.1.1:22 >
192.168.1.100:ftp_data RA / Padding
 IP / TCP 192.168.1.100:ftp_data > 192.168.1.1:telnet S ==> IP / TCP 192.168.1.1:telnet >
192.168.1.100:ftp_data RA / Padding
```

5.嗅探(抓包)

sniff()函数可以完成嗅探（抓包）功能

```
 >>>  sniff(filter="icmp and host 192.168.1.1", count=2)
 <Sniffed: UDP:0 TCP:0 ICMP:2 Other:0>
 >>>  a = _
 >>>  a.nsummary()
 0000 Ether / IP / ICMP 192.168.1.1 echo-request 0 / Raw
 0001 Ether / IP / ICMP 192.168.1.1 echo-request 0 / Raw
```

这里用到的过滤规则为BPF过滤规则，与tupdump的过滤规则一致，具体可以参考附件中tcpdump的过滤规则。

• 高级应用

1.TCP traceroute

TCP traceroute可以用sr函数来模拟:

```
>>> sr(IP(dst="www.tplink.net",ttl=(1,4))/TCP(flags=0x2))
 Begin emission:
 Finished to send 4 packets.
 ...*..*..*.............
 Received 24 packets, got 3 answers, remaining 1 packets
 (<Results: TCP:2 UDP:0 ICMP:1 Other:0>, <Unanswered: TCP:1 UDP:0 ICMP:0 Other:0>
 )
 >>>
 >>>
 >>> ans,unans = _
 >>> for snd,rcv in ans:
 ...     print snd.ttl ,rcv.src,isinstance(rcv.payload,TCP)
 1 192.168.1.1 False
 2 172.31.89.40 True
 3 172.31.89.40 True
```

也可以采用其内建的traceroute函数：

```
>>> traceroute(["www.tplink.net","testftp.tplink.net"],maxttl=4)
 Begin emission:
 Finished to send 8 packets.
 *******
 Received 7 packets, got 7 answers, remaining 1 packets
 172.31.81.69:tcp80 172.31.89.40:tcp80
 1 192.168.1.1     11 192.168.1.1     11
 2 172.31.81.69    SA -
 3 172.31.81.69    SA 172.31.89.40    SA
 4 172.31.81.69    SA 172.31.89.40    SA
 (<Traceroute: TCP:5 UDP:0 ICMP:2 Other:0>, <Unanswered: TCP:1 UDP:0 ICMP:0 Other:0>)
 >>>
```

2. 经典攻击

畸形包：

```
>>> send(IP(dst="10.1.1.5", ihl=2, version=3)/ICMP())
```

死亡之ping：

```
>>> send( fragment(IP(dst="10.0.0.5")/ICMP()/("X"*60000)) )
```

ARP毒性缓存

```
>>> send( Ether(dst=clientMAC)/ARP(op="who-has", psrc=gateway, pdst=client),
 inter=RandNum(10,40), loop=1 )
```

【函数手册】

- 有关收发包及读写文件的函数

1. sniff(count=0, store=1, offline=None, prn = None, lfilter=None, L2socket=None, timeout=None,

opened_socket=None, stop_filter=None, *arg, **karg)

嗅探包，返回嗅探到的包的列表。
filter：规定一个BPF过滤器。
count：要抓取的包的数量，0为无限多。
store：是否保存嗅探到的包。
prn：应用到每一个包的函数。如果有返回的东西，就显示出来。例如：prn = lambda x: x.summary()
sniff(prn=lambda x: x.summary()，filter='tcp')
lfilter：应用到每一个包的python 函数，决定是否完成下一步动作。例如：lfilter = lambda x: x.haslayer(Padding)
offline：从pcap 文件中读取包，而不是嗅探包。
timeout：在给定时间后停止嗅探（默认：None）。
L2socket：使用提供的二层接口。
opened_socket：提供一个可以使用.recv()的客体。

stop_filter: 应用到每一个包的python 函数，决定是否在一个特定的包后结束捕捉。例如：stop_filter = lambda x: x.haslayer(TCP)
e.g.

```
>>> p = sniff(filter="tcp",count=3)
>>> p
<Sniffed: TCP:3 UDP:0 ICMP:0 Other:0>
>>> p.show()
0000 Ether / IP / TCP 172.31.81.40:1330 > 172.31.88.91:9090 S
0001 Ether / IP / TCP 172.31.88.91:9090 > 172.31.81.40:1330 SA
0002 Ether / IP / TCP 172.31.81.40:1330 > 172.31.88.91:9090 A
```

2.wrpcap(filename, pkt, *args, **kargs)

将一组包写为pcap文件。

filename为要保存的文件名，包含路径。如"D:\pcap\ICMP.pcap"

pkt为要保存为pcap文件的一组包。

e.g.

```
>>> p = sniff(filter="icmp", count=10)
>>> wrpcap("D:\pcap\snficmp.pcap", p)
```

3.rdpcap(filename, count=-1)

读取pcap文件并返回包的列表。
filename为要打开的文件名，包含路径。如"D:\802.11\dot11.pcap"
count为读取的包的个数。默认为count=-1,读取所有包。
e.g.

```
>>> rdpcap("D:\pcap\snficmp.pcap")
>>> rdpcap("E:\pptp.pcap", count=50)
```

4.send(x, inter=0, loop=0, count=None, verbose=None, realtime=None, *args, **kargs)

发送第三层的包。
e.g.

```
>>> send(IP(dst="1.2.3.4")/ICMP())
```

5.sendp(x, inter=0, loop=0, iface=None, iface_hint=None, count=None, verbose=None, realtime=None, *args, **kargs)

发送第二层的包。
e.g.

```
>>> sendp(Ether()/IP(dst="1.2.3.4",ttl=(1,4)), iface="eth1")
>>> sendp("I'm travelling on Ethernet", iface="eth1", loop=1, inter=0.2)
```

6.sr(x,filter=None, iface=None, nofilter=0, args,*kargs)

发和收第三层的包。返回两个列表对。第一个是（发送的包，回应的包）的元组的列表，第二个是未回应的包的列表。可以把这两个列表分别赋给ans和uans。
filter: 规定一个BPF过滤器。
iface: 仅在给定的接口(网卡)侦听回应（listen answers）。
nofilter: 置1来避免使用BPF过滤器。
retry: 如果为正，表示重新发送未收到回应的包的次数。如果为负，表示当收完所有回应包后重新尝试的次数。
timeout: 在最后一个包发出之后等待的时间。
e.g.

```
>>> ans,uans = sr(IP(dst="172.31.88.5")/TCP(dport=[21,23,80,8080]))
Begin emission:
Finished to send 4 packets.
...*.*.*.*
Received 10 packets, got 4 answers, remaining 0 packets
>>> ans.summary()
IP / TCP 172.31.81.40:ftp_data > 172.31.88.5:ftp S ==> IP / TCP 172.31.88.5:ftp
> 172.31.81.40:ftp_data RA / Padding
IP / TCP 172.31.81.40:ftp_data > 172.31.88.5:telnet S ==> IP / TCP 172.31.88.5:t
elnet > 172.31.81.40:ftp_data RA / Padding
IP / TCP 172.31.81.40:ftp_data > 172.31.88.5:http S ==> IP / TCP 172.31.88.5:htt
p > 172.31.81.40:ftp_data RA / Padding
IP / TCP 172.31.81.40:ftp_data > 172.31.88.5:8080 S ==> IP / TCP 172.31.88.5:808
0 > 172.31.81.40:ftp_data RA / Padding
```

可以用以下命令来效仿常规的ICMP Ping。

```
>>> ans,unans=sr(IP(dst="192.168.1.1-254")/ICMP())
```

可由如下请求来收集可达的主机的信息。

```
>>> ans.summary(lambda (s,r): r.sprintf("%IP.src% is alive"))
```

当ICMP echo requests 被禁用的时候，我们仍可以使用不同的TCP Pings，如下面的TCP SYN Ping:

```
>>> ans,unans=sr( IP(dst="192.168.1.*")/TCP(dport=80,flags="S"))
```

任何返回给我们请求的响应都指出了一个可达的主机。我们可以通过如下的命令来收集可达主机的结果：

```
>>> ans.summary( lambda(s,r) : r.sprintf("%IP.src% is alive"))
```

如果其它方法都失败了我们还可以使用UDP Ping，它会使可达的主机产生ICMP Port unreachable errors。这里应该尽可能选择最可能关闭的端口，比如端口0。

```
>>> ans,unans=sr( IP(dst="192.168.*.1-10")/UDP(dport=0))
```

仍可由这个命令收集结果:

```
>>> ans.summary( lambda(s,r) : r.sprintf("%IP.src% is alive"))
```

7. sr1(x,filter=None,iface=None, nofilter=0, *args,**kargs)

发和收第三层的包，并只返回第一个应答包。
filter: 规定一个BPF过滤器。
iface: 仅在给定的接口(网卡)侦听回应（listen answers）。
nofilter: 置1来避免使用BPF过滤器。
e.g.

```
>>> p=sr1(IP(dst="www.slashdot.org")/ICMP()/"XXXXXXXXXXX"
```

8. srp(x,iface=None, iface_hint=None, filter=None, nofilter=0, type=ETH_P_ALL, *args,**kargs)

发和收第二层的包。
iface: 仅在给定的接口（网卡）工作。
filter: 规定一个BPF过滤器。
nofilter: 置1来避免使用BPF过滤器。
retry: 如果为正，表示重新发送未收到回应的包的次数。如果为负，表示当收完所有回应包后重新尝试的次数。
timeout: 在最后一个包发出之后等待的时间。
verbose: 设置赘言（verbosity）等级。
multi: 是否接收对同一激励（stimulus）的多个回应（multiple answers）。
e.g.

```
>>> ans,uans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst="172.31.81.0/24"),timeo
Begin emission:
Finished to send 256 packets.
..*.....................................*.*..**.*.*.*.*.*.*..*..*.*.*.*.*.*.*...*.*
.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*..*.*...*.........*.......*.*.*.*.........
.....................*.....................................................*....
......*..............*.....................*.*....*.....................
Received 316 packets, got 53 answers, remaining 203 packets
>>>
```

以上代码相当于ARP ping，IP地址范围为172.31.81.0-172.31.81.255
可以用以下代码来查看以上ARP ping的结果

```
>>> ans.summary(lambda (s,r): r.sprintf("%Ether.src% %ARP.psrc%"))
00:0f:e2:80:30:90 172.31.81.1
40:61:86:fc:71:99 172.31.81.37
40:61:86:fc:65:d9 172.31.81.38
40:61:86:fc:73:6e 172.31.81.40
40:61:86:fc:75:ad 172.31.81.39
40:61:86:fc:70:fd 172.31.81.41
40:61:86:fc:69:0b 172.31.81.42
40:61:86:fc:26:cf 172.31.81.43
40:61:86:fc:71:d3 172.31.81.44
40:61:86:fc:23:df 172.31.81.45
40:61:86:e5:b2:cf 172.31.81.46
40:61:86:fc:62:6f 172.31.81.48
00:19:66:35:e1:cb 172.31.81.50
40:61:86:fc:74:28 172.31.81.51
40:61:86:fc:6f:af 172.31.81.52
40:61:86:fc:72:40 172.31.81.53
40:61:86:e5:b3:96 172.31.81.54
00:22:68:50:07:25 172.31.81.55
40:61:86:fc:75:03 172.31.81.58
40:61:86:fc:72:55 172.31.81.59
40:61:86:e5:b2:00 172.31.81.60
......
```

9.srp1(args,*kargs)

发和收第二层的包，并只返回第一个应答。
nofilter：置1来避免使用BPF过滤器。
retry：如果为正，表示重新发送未收到回应的包的次数。如果为负，表示当收完所有回应包后重新尝试的次数。
timeout：在最后一个包发出之后等待的时间。
verbose：设置赘言（verbosity）等级。
multi：是否接收对同一激励（stimulus）的多个回应（multiple answers）。
filter：规定一个BPF过滤器。
iface：仅在给定的接口（网卡）工作。

10.arping(net, timeout=2, cache=0, verbose=None, **kargs)

发送ARP who-has 请求来检查哪些主机可达。
如果你想让arping修改内部ARP-Cache，设置cache=True。
以下代码即可实现如8.srp中ARP ping例子相同的效果。

```
>>> arping("172.31.81.*")
Begin emission:
Finished to send 256 packets.
\*********************************************\*
Received 49 packets, got 49 answers, remaining 207 packets
00:0f:e2:80:30:90 172.31.81.1
40:61:86:fc:71:99 172.31.81.37
40:61:86:fc:65:d9 172.31.81.38
40:61:86:fc:73:6e 172.31.81.40
40:61:86:fc:75:ad 172.31.81.39
40:61:86:fc:70:fd 172.31.81.41
40:61:86:fc:69:0b 172.31.81.42
40:61:86:fc:26:cf 172.31.81.43
40:61:86:fc:71:d3 172.31.81.44
40:61:86:fc:23:df 172.31.81.45
40:61:86:e5:b2:cf 172.31.81.46
40:61:86:fc:62:6f 172.31.81.48
......
```

11.traceroute(target, dport=80, minttl=1, maxttl=30, sport=RandShort(), l4 = None, filter=None, timeout=2, verbose=None, **kargs)

target: 目标地址，可以是ip地址，也可以是域名。如"www.baidu.com"。
minttl: 最小ttl。
maxttl: 最大ttl。
filter: 规定一个BPF过滤器。
timeout: 在最后一个包发出之后等待的时间。
verbose: 设置赘言（verbosity）等级。
例子请见上文"高级应用"中的traceroute的例子。

12.wireshark(pktlist)

将一组包用wireshark打开。

pktlist为一组要用wireshark软件打开的包。

e.g.

```
>>> p = sniff(filter="ip",count=5)
>>> p.show()
0000 Ether / IP / TCP 172.31.81.40:1580 > 222.186.189.237:http S
0001 Ether / IP / TCP 172.31.81.40:1580 > 222.186.189.237:http S
0002 Ether / IP / TCP 172.31.88.91:9090 > 172.31.81.40:1579 FA / Padding
0003 Ether / IP / TCP 172.31.81.40:1579 > 172.31.88.91:9090 A
0004 Ether / IP / UDP 172.31.81.210:netbios_ns > 172.31.81.255:netbios_ns / NBNS
QueryRequest
>>> wireshark(p)
```

如果电脑中有安装好的wireshark软件的话，这组包将会由wireshark软件打开。

13.tshark(args,*kargs)
""Sniff packets and print them calling pkt.show(), a bit like text wireshark""
抓包，每抓到一个包就显示出来，由于调用的pkt.show(),所以显示的是包的完全的信息。
这个函数其实是调用的sniff。
sniff(prn=lambda x: x.display(),args,*kargs)

- 对单个包的操作的函数

Command	Effect
str(pkt)	组装包
hexdump(pkt)	包的十六进制输出
ls(pkt)	得到包的域的值的表
pkt.summary()	单行的包的信息的摘要
pkt.show()	包的高级显示模式（for a developed view of the packet）
pkt.show2()	与show函数相同，但是显示的是组装后的包(比如，校验和是经过计算之后的)
pkt.sprintf()	显示包的域的值的格式化字符串（fills a format string with fields values of the packet）
pkt.decode_payload_as()	changes the way the payload is decoded
pkt.psdump()	draws a PostScript diagram with explained dissection（由于没安装Pyx所以没法使用）
pkt.pdfdump()	draws a PDF with explained dissection（由于没安装Pyx所以没法使用）
pkt.command()	返回能生成这个包的 Scapy命令

e.g.

```
>>> a = IP()/TCP()
>>> ls(a)
version    : BitField             = 4               (4)
ihl        : BitField             = None            (None)
tos        : XByteField           = 0               (0)
len        : ShortField           = None            (None)
id         : ShortField           = 1               (1)
flags      : FlagsField           = 0               (0)
frag       : BitField             = 0               (0)
ttl        : ByteField            = 64              (64)
proto      : ByteEnumField        = 6               (0)
chksum     : XShortField          = None            (None)
src        : Emph                 = '127.0.0.1'     (None)
dst        : Emph                 = '127.0.0.1'     ('127.0.0.1')
options    : PacketListField      = []              ([])
--
sport      : ShortEnumField       = 20              (20)
dport      : ShortEnumField       = 80              (80)
seq        : IntField             = 0               (0)
ack        : IntField             = 0               (0)
dataofs    : BitField             = None            (None)
reserved   : BitField             = 0               (0)
flags      : FlagsField           = 2               (2)
window     : ShortField           = 8192            (8192)
chksum     : XShortField          = None            (None)
urgptr     : ShortField           = 0               (0)
options    : TCPOptionsField      = {}              ({})
>>>
```

注意函数中pkt的位置，有的在括号中，有的在"."后面。

- 对一组包操作的函数

Command	Effect
summary()	显示每个包的摘要
nsummary()	显示每个包的摘要，并带包的序号
conversations()	displays a graph of conversations
show()	显示优先的（preferred）表现方式（通常是nsummary()）
filter()	返回用lambda函数过滤的一组包
hexdump()	返回所有包的十六进制显示
hexraw()	返回十六进制的所有包的未加工的层（ Raw layer）
padding()	返回十六进制的包的填充
nzpadding()	返回十六进制的包的非零填充
plot()	plots a lambda function applied to the packet list（由于没有gnuplot wrapper，此函数应该用不了）
make table()	显示与lambda函数相符的表格

e.g.

```
>>> p = rdpcap("D:\sniff.pcap")
>>> p.summary()
Ether / ARP who has 172.31.81.234 says 172.31.81.234 / Padding
Ether / IP / TCP 172.31.81.40:3417 > 220.181.126.18:http S
Ether / IP / TCP 172.31.81.40:3418 > 220.181.126.18:http S
Ether / IP / TCP 172.31.81.40:3419 > 220.181.126.17:http S
Ether / ARP who has 172.31.81.1 says 172.31.81.65 / Padding
Ether / IP / TCP 172.31.81.40:3421 > 220.181.126.17:http S
Ether / IP / TCP 172.31.81.40:3422 > 220.181.126.18:http S
Ether / IP / TCP 172.31.81.40:3423 > 220.181.126.18:http S
Ether / IP / TCP 172.31.81.40:3421 > 220.181.126.17:http S
Ether / IP / TCP 172.31.81.40:3422 > 220.181.126.18:http S
>>> p.show()
0000 Ether / ARP who has 172.31.81.234 says 172.31.81.234 / Padding
0001 Ether / IP / TCP 172.31.81.40:3417 > 220.181.126.18:http S
0002 Ether / IP / TCP 172.31.81.40:3418 > 220.181.126.18:http S
0003 Ether / IP / TCP 172.31.81.40:3419 > 220.181.126.17:http S
0004 Ether / ARP who has 172.31.81.1 says 172.31.81.65 / Padding
0005 Ether / IP / TCP 172.31.81.40:3421 > 220.181.126.17:http S
0006 Ether / IP / TCP 172.31.81.40:3422 > 220.181.126.18:http S
0007 Ether / IP / TCP 172.31.81.40:3423 > 220.181.126.18:http S
0008 Ether / IP / TCP 172.31.81.40:3421 > 220.181.126.17:http S
0009 Ether / IP / TCP 172.31.81.40:3422 > 220.181.126.18:http S
```

【Automata及其在Scapy中的应用】

1. Automata简介

Automata是英文单词Automaton的复数形式。Automaton意为自动机，机器人，源自希腊语automatos，原意self-acting，即自我行动。

Automata Theory理论，将Automata分为很多种类。如可以分为deterministic automata （确定性自动机），fuzzy automata （模糊自动机），finite state automata（有限状态自动机），infinite state automata （无限状态自动机），discrete automata （离散自动机），probabillistic automata （随机自动机），array automata （阵列自动机），picture automata （图片自动机）等等。

Automata 有两个典型的模型，Moore automata 和 Mealy automata。

2. Scapy中的automata

Scapy中的自动机为有限型自动机。自动机有不同的状态（states）：一个开始状态（start state），一个或多个结束状态(end state)和不定量的错误状态(error state)。从一个状态到另一个状态，中间为转化（transtions）过程。转化可以为特定条件（specific condition）触发的，可以为收到特定包（the reception of a specific packet）触发的，也可以为超时（timeout）触发的。当转化发生，就可以进行一个或多个行为（actions）。
一种行为可以与许多转化过程绑定在一起。参数可以由状态（states）传递到转化过程（transitions），也可以由转化过程传递给状态（states）和行为（actions）。

从程序员的角度来看，状态（states），转化过程（transitions），行为(actions)都是Automaton类(class)的子类(subclass)中的方法(methods)。这些方法由修饰符修饰，来为自动机的运转提供不同的信息（meta-information）。

3. automata中的修饰符（decorators）

3.1 状态（states）修饰符

状态（states）是由ATMT.state函数的结果修饰的方法（methods）。状态有三个可选的参数，初始（initial）,结束（final）和错误（error）。当被设为真（True）的时候，意味着状态（state）是初始（initial），结束（final）或者错误（error）。下面是一些状态的例子：

```
class Example(Automaton):
    @ATMT.state(initial=1)
    def BEGIN(self):
        print"This is the beginning state."


    @ATMT.state()
    def SOME_STATE(self):
        pass

    @ATMT.state(final=1)
    def END(self):
        return "Result of the automaton: 42"

    @ATMT.state(error=1)
    def ERROR(self):
        return "Partial result, or explanation"
```

当自动机转变到一个给定的状态(state)后，就会执行该状态（state）的方法（method）。定义（def）状态（state）中的方

3.2 转化（transitions）修饰符

转化(transitions)是由ATMT.condition，ATMT.receive_condition，ATMT.timeout三者之一的结果修饰的方法(methods)。转化把与他们相关联的状态方法(state method)取为参数。 **ATMT.timeout** 还有一个timeout的强制参数，提供以秒为单位的超时值（timeout value）。 **ATMT.condition** 和 **ATMT.receive_condition** 有一个可选参数prio（优先值），这样可以强制在哪些条件（conditions）下给顺序（order）赋值（evaluated）。默认的优先值（priority）是0。我们称有同样优先级（priority level）的转化（transitions）在不确定的顺序中（undetermined order）。

当触发了一个新的状态（如 raise self.MY_NEW_STATE()），相应的转化方法（transitions methods）就会被调用（called）。在状态方法（state's method）返回(return)后， **ATMT.condition**修饰的方法就会由提升优先值（growing prio）而运行。每当接收（receive）到包并且包被主过滤器（master filter）接受（accept），所有被**ATMT.receive_condition**修饰的方法（methods）就会因优先值增加被调用（called by growing prio）。当到了（reach）我们输入到当前间隔的超时值（timeout），相应的 **ATMT.timeout** 修饰的方法（method）就会被调用。下面是一些转化（transitions）的例子。

```
class Example(Automaton):
    @ATMT.state()
    def WAITING(self):
        pass

    @ATMT.condition(WAITING)
    def it_is_raining(self):
        if not self.have_umbrella:
            raise self.ERROR_WET()

    @ATMT.receive_condition(WAITING, prio=1)
    def it_is_ICMP(self, pkt):
        if ICMP in pkt:
            raise self.RECEIVED_ICMP(pkt)

    @ATMT.receive_condition(WAITING, prio=2)
    def it_is_IP(self, pkt):
        if IP in pkt:
            raise self.RECEIVED_IP(pkt)

    @ATMT.timeout(WAITING, 10.0)
    def waiting_timeout(self):
        raise self.ERROR_TIMEOUT()
```

以上的转化都是与状态WAITING关联的。prio= 的数字越小，优先值越高。

3.3 行为（actions）修饰符

行为(actions)是由**ATMT.action** 函数的结果修饰的方法（methods）。**ATMT.action** 函数取行为（actions）绑定的转化方法（transition method）作为第一个参数，取一个可选的优先值（priority）prio作为第二个参数。默认的优先值是0。一个行为（action）的方法（method）可以被修饰很多次来绑定到很多转化（transitions）上。

```
class Example(Automaton):
    @ATMT.state(initial=1)
    def BEGIN(self):
        pass

    @ATMT.state(final=1)
    def END(self):
        pass

    @ATMT.condition(BEGIN, prio=1)
    def maybe_go_to_end(self):
        if random() > 0.5:
            raise self.END()
    @ATMT.condition(BEGIN, prio=2)
    def certainly_go_to_end(self):
        raise self.END()

    @ATMT.action(maybe_go_to_end)
    def maybe_action(self):
        print "We are lucky..."
    @ATMT.action(certainly_go_to_end)
    def certainly_action(self):
        print "We are not lucky..."
    @ATMT.action(maybe_go_to_end, prio=1)
    @ATMT.action(certainly_go_to_end, prio=1)
    def always_action(self):
        print "This wasn't luck!..."
```

两种可能的输出是：

```
>>> a=Example()
>>> a.run()
We are not lucky...
This wasn't luck!...
>>> a.run()
We are lucky...
This wasn't luck!...
```

4. Automata举例

4.1 一个小例子

```
class HelloWorld(Automaton):
    @ATMT.state(initial=1)
    def BEGIN(self):
        print "State=BEGIN"

    @ATMT.condition(BEGIN)
    def wait_for_nothing(self):
        print "Wait for nothing..."
        raise self.END()

    @ATMT.action(wait_for_nothing)
    def on_nothing(self):
        print "Action on 'nothing' condition"

    @ATMT.state(final=1)
    def END(self):
        print "State=END"
```

运行及结果：

```
>>> a=HelloWorld()
>>> a.run()
State=BEGIN
Wait for nothing...
Action on 'nothing' condition
State=END
```

4.2 一个有网络包参与的例子

```
class Example(Automaton):

    def master_filter(self, pkt):
        return (IP in pkt)

     # BEGIN
    @ATMT.state(initial=1)
    def BEGIN(self):
        print'This is the beginning of the automata.'
        raise self.WAITING()

    #WAITING
    @ATMT.state()
    def WAITING(self):
        print 'This is WAITING state.'

    @ATMT.receive_condition(WAITING)
    def receive_data(self, pkt):
        if ICMP in pkt and pkt[IP].src == '172.31.88.5':
            print"I've got ICMP and will return to wait."
            raise self.WAITING()

    @ATMT.timeout(WAITING, 10)
    def timeout_waiting(self):
        print"I don't get what I want, I'll raise ERROR state."
        raise self.ERROR()

    # ERROR
    @ATMT.state(error=1)
    def ERROR(self):
        print 'Time out.'

    @ATMT.action(timeout_waiting)
    def on_ERROR(self):
        print "Error happened, the automata will be ended."

a = Example()
a.run()
```

有网络包的automata（一般指有@ATMT.receive_condition的automata，即涉及到对收到的包的判断等），在Windows平台下运行
')。此问题官方暂时也无法解决，所以只能在linux平台或其它可以的平台上运行。此例即在linux平台Ubuntu 9.10中运行。

运行结果：

```
xzf@ubuntu:/media/local_/Python/tests$ sudo python automata_example.py
WARNING: No route found for IPv6 destination :: (no default route?)
/usr/local/lib/python2.6/dist-packages/scapy/crypto/cert.py:6: DeprecationWarning: the sha module
is deprecated; use the hashlib module instead
  import os, sys, math, socket, struct, sha, hmac, string, time
/usr/local/lib/python2.6/dist-packages/scapy/crypto/cert.py:7: DeprecationWarning: The popen2
module is deprecated.  Use the subprocess module.
  import random, popen2, tempfile
This is the beginning of the automata.
This is WAITING state.
I've got ICMP and will return to wait.
This is WAITING state.
I don't get what I want, I'll raise ERROR state.
Error happened, the automata will be ended.
Time out.
Traceback (most recent call last):
  File "automata_example.py", line 144, in <module>
    a.run()
  File "/usr/local/lib/python2.6/dist-packages/scapy/automaton.py", line 540, in _do_control
    state = iterator.next()
  File "/usr/local/lib/python2.6/dist-packages/scapy/automaton.py", line 575, in _do_iter
    result=state_output, state=self.state.state)
scapy.automaton.ErrorState: Reached ERROR: [None]
xzf@ubuntu:/media/local_/Python/tests$
```

该自动机的运行机制是，开始状态（BEGIN）后，转入等待状态（WAITING）。有两个转化（transition）同时关联到等待状态（WAITING）。若等待过程中收到来自172.31.88.5的包且其中有ICMP层，则会打印"I've got ICMP and will return to wait"，并raise WAITING状态，重新等待。若等待过程中（由此例中timeout转化的参数设定可知这个过程的时间为10秒）没有收到上述要求的包，则会打印"I don't get what I want, I'll raise ERROR state." 然后raise ERROR状态，自动机运行结束。

此例中在timeout之前手动ping 了一次172.31.88.5，之后一直等待直到timeout时间到。

4.3 Scapy源码中TFTP client的实现

```
class TFTP_read(Automaton):
    def parse_args(self, filename, server, sport = None, port=69, **kargs):
        Automaton.parse_args(self, **kargs)
        self.filename = filename
        self.server = server
        self.port = port
        self.sport = sport

    def master_filter(self, pkt):
        return ( IP in pkt and pkt[IP].src == self.server and UDP in pkt
                 and pkt[UDP].dport == self.my_tid
                 and (self.server_tid is None or pkt[UDP].sport == self.server_tid) )

    # BEGIN
    @ATMT.state(initial=1)
    def BEGIN(self):
        self.blocksize=512
        self.my_tid = self.sport or RandShort()._fix()
        bind_bottom_up(UDP, TFTP, dport=self.my_tid)
        self.server_tid = None
        self.res = ""

        self.l3 = IP(dst=self.server)/UDP(sport=self.my_tid, dport=self.port)/TFTP()
        self.last_packet = self.l3/TFTP_RRQ(filename=self.filename, mode="octet")
        self.send(self.last_packet)
        self.awaiting=1

        raise self.WAITING()

    # WAITING
    @ATMT.state()
    def WAITING(self):
        pass

    @ATMT.receive_condition(WAITING)
```

```
def receive_data(self, pkt):
    if TFTP_DATA in pkt and pkt[TFTP_DATA].block == self.awaiting:
        if self.server_tid is None:
            self.server_tid = pkt[UDP].sport
            self.l3[UDP].dport = self.server_tid
        raise self.RECEIVING(pkt)
@ATMT.action(receive_data)
def send_ack(self):
    self.last_packet = self.l3 / TFTP_ACK(block = self.awaiting)
    self.send(self.last_packet)

@ATMT.receive_condition(WAITING, prio=1)
def receive_error(self, pkt):
    if TFTP_ERROR in pkt:
        raise self.ERROR(pkt)

@ATMT.timeout(WAITING, 3)
def timeout_waiting(self):
    raise self.WAITING()
@ATMT.action(timeout_waiting)
def retransmit_last_packet(self):
    self.send(self.last_packet)

# RECEIVED
@ATMT.state()
def RECEIVING(self, pkt):
    recvd = pkt[Raw].load
    self.res += recvd
    self.awaiting += 1
    if len(recvd) == self.blocksize:
        raise self.WAITING()
    raise self.END()

# ERROR
@ATMT.state(error=1)
def ERROR(self,pkt):
    split_bottom_up(UDP, TFTP, dport=self.my_tid)
    return pkt[TFTP_ERROR].summary()

#END
@ATMT.state(final=1)
def END(self):
```

```
        split_bottom_up(UDP, TFTP, dport=self.my_tid)
        return self.res
```

可由类似如下方式运行：

```
>>> TFTP_read("my_file", "192.168.1.128").run()
```

【Scapy面向测试的应用举例】

例一

要求：抓两个电脑之间通信的包，两电脑的IP分别为172.31.88.5和172.31.81.40，时间为15秒，过滤规则为只要带IP层的包，抓包之后分析所抓到的包中含有TCP的包的数量。

代码实现：

```
from scapy.all import *

def has_n_TCP():                                        #you can change TCP to any other protocol

    cnt = 0                                             #the variable to count the number
    p   = sniff(filter="ip host 172.31.88.5 \
                and 172.31.81.40", timeout=15)          #wait for 15 seconds, you can change the
filter as the demand changes

    for i in range(len(p)):                             #every packet that sniffed
        if p[i].haslayer(TCP):                          #if a packet has TCP
            cnt += 1                                    #count = count + 1

    return (cnt,p)

def main():

    n,p = has_n_TCP()                                   #n = number counted, p = packet list filtered
    p.show()                                            #here shows the packets you've filtered, you
can delete this sentence if you don't neet it
    print "You've got %s TCP."%n

if __name__ == '__main__':
    main()
```

运行结果：

```
E:\Python\tests>python has_n_TCP.py
WARNING: No route found for IPv6 destination :: (no default route?)
0000 Ether / IP / ICMP 172.31.81.40 > 172.31.88.5 echo-request 0 / Raw
0001 Ether / IP / ICMP 172.31.88.5 > 172.31.81.40 echo-reply 0 / Raw
0002 Ether / IP / ICMP 172.31.81.40 > 172.31.88.5 echo-request 0 / Raw
0003 Ether / IP / ICMP 172.31.88.5 > 172.31.81.40 echo-reply 0 / Raw
0004 Ether / IP / ICMP 172.31.81.40 > 172.31.88.5 echo-request 0 / Raw
0005 Ether / IP / ICMP 172.31.88.5 > 172.31.81.40 echo-reply 0 / Raw
0006 Ether / IP / ICMP 172.31.81.40 > 172.31.88.5 echo-request 0 / Raw
0007 Ether / IP / ICMP 172.31.88.5 > 172.31.81.40 echo-reply 0 / Raw
0008 Ether / IP / TCP 172.31.81.40:1152 > 172.31.88.5:http S
0009 Ether / IP / TCP 172.31.88.5:http > 172.31.81.40:1152 RA / Padding
0010 Ether / IP / TCP 172.31.81.40:1152 > 172.31.88.5:http S
0011 Ether / IP / TCP 172.31.88.5:http > 172.31.81.40:1152 RA / Padding
0012 Ether / IP / TCP 172.31.81.40:1152 > 172.31.88.5:http S
0013 Ether / IP / TCP 172.31.88.5:http > 172.31.81.40:1152 RA / Padding
You've got 6 TCP.
E:\Python\tests>
```

结果中的包，为手动ping 172.31.88.5和在浏览器中访问172.31.88.5 产生，若要自动实现网络操作，可参考下例

例二

要求：先开启抓包，然后进行网络操作产生想要抓到的包（比如此例中是用简单的ping命令产生要抓的包），然后将抓到的包返回，再进行下一步操作（本例中是将包简单地显示出来）。

代码实现：

```
from multiprocessing import Process
from scapy.all import *
from subprocess import *



def ping():                                         #This function is the network manipulation for ge
nerating the  packets you want

    cmd = 'ping 172.31.88.5'
    subprocess.Popen(cmd)                           #This will run 'ping 172.31.88.5' in command line
    print"now ping runs"

def capture():

    pro = Process(target=ping)                      #target=the name of the function defined in the
above
    pro.start()

    print"now sniff"                                 #This will run first
    p   = sniff(count=8,filter='icmp')              #sniff() to capture packets, then run the
function defined in the above(ping)
    print"now sniff ends"

    pro.join()                                      #make sure that ping ends then stop sniff?

    return p                                        #This will return the packets that captured to
main()

def main():

    p = capture()                                   #get packets list from capture()

    print 'now show'
    p.show()                                        #show the packets that captured

if __name__ == '__main__':
    main()
```

运行结果：

```
E:\Python\tests>python test_multiprocess_ping.py
WARNING: No route found for IPv6 destination :: (no default route?)
now sniff
WARNING: No route found for IPv6 destination :: (no default route?)
now ping runs
Pinging 172.31.88.5 with 32 bytes of data:
Reply from 172.31.88.5: bytes=32 time<1ms TTL=127
Reply from 172.31.88.5: bytes=32 time<1ms TTL=127
Reply from 172.31.88.5: bytes=32 time<1ms TTL=127
Reply from 172.31.88.5: bytes=32 time<1ms TTL=127
Ping statistics for 172.31.88.5:
 Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
 Minimum = 0ms, Maximum = 0ms, Average = 0ms
now sniff ends
now show
0000 Ether / IP / ICMP 172.31.81.40 > 172.31.88.5 echo-request 0 / Raw
0001 Ether / IP / ICMP 172.31.88.5 > 172.31.81.40 echo-reply 0 / Raw
0002 Ether / IP / ICMP 172.31.81.40 > 172.31.88.5 echo-request 0 / Raw
0003 Ether / IP / ICMP 172.31.88.5 > 172.31.81.40 echo-reply 0 / Raw
0004 Ether / IP / ICMP 172.31.81.40 > 172.31.88.5 echo-request 0 / Raw
0005 Ether / IP / ICMP 172.31.88.5 > 172.31.81.40 echo-reply 0 / Raw
0006 Ether / IP / ICMP 172.31.81.40 > 172.31.88.5 echo-request 0 / Raw
0007 Ether / IP / ICMP 172.31.88.5 > 172.31.81.40 echo-reply 0 / Raw
E:\Python\tests>
```