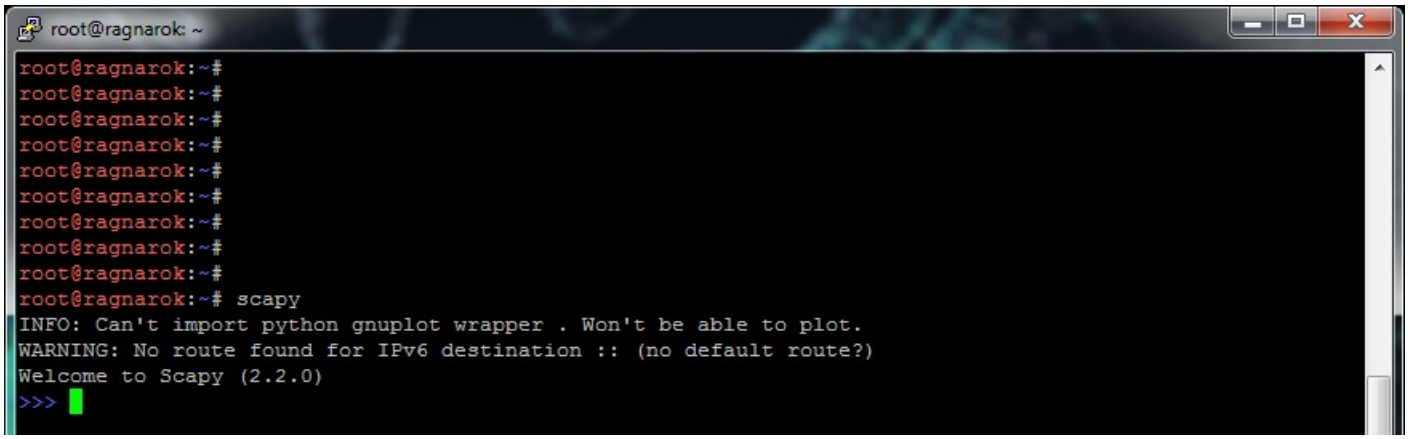
Установку можно провести разными способами, например **apt-get install *python-scapy***, в случае дистрибутивов на основе Debian.
Так же можно просто скачать свежую версию с сайта разработчиков:

```
# cd /tmp
# wget scapy.net
# unzip scapy-latest.zip
# cd scapy-2.*
# sudo python setup.py install
```

После этого запуск происходит непосредственно командой **scapy**.
На экране отобразится примерно так:

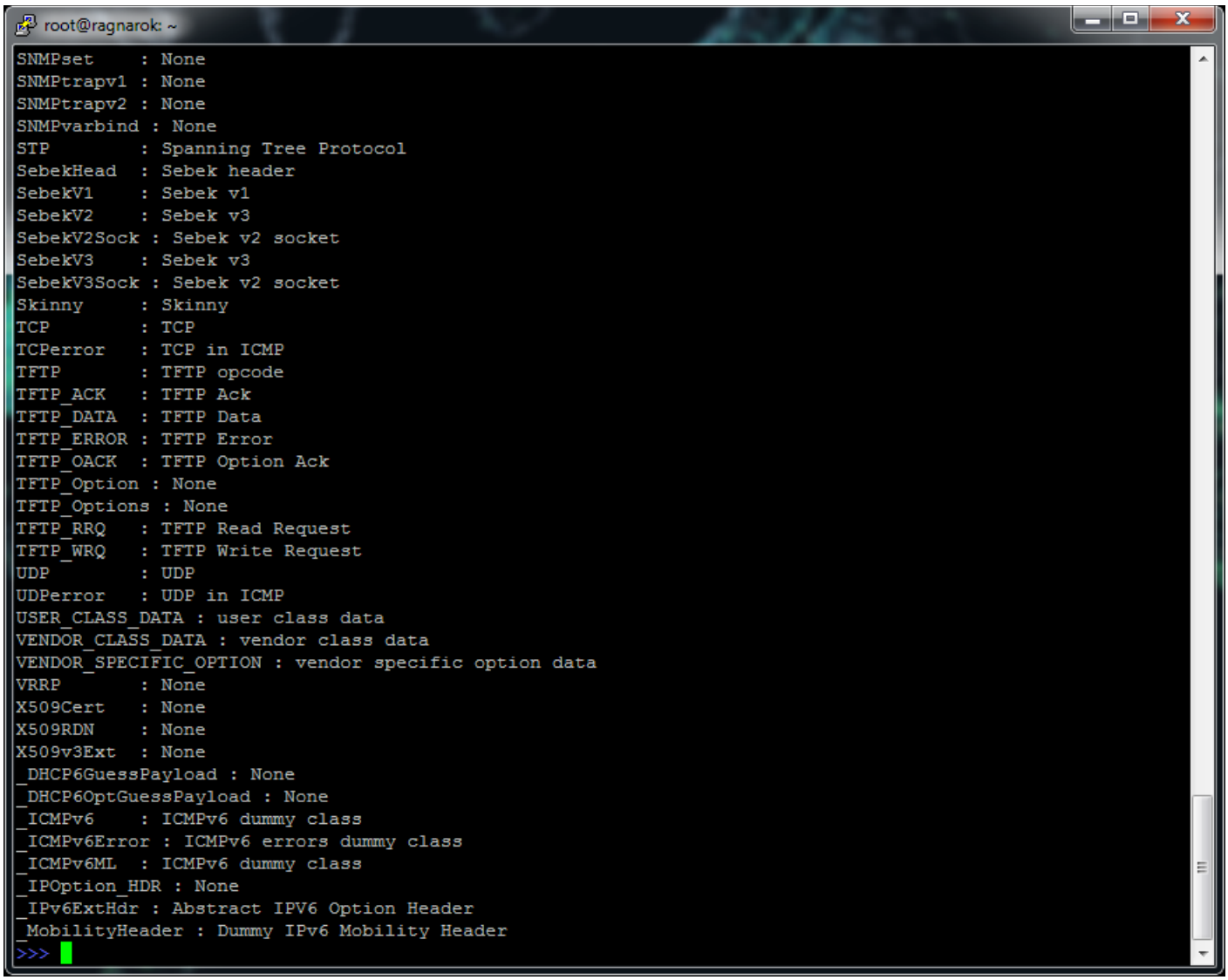


Мы видим стандартное приглашение для ввода, все действия будут выполняться в интерактивном режиме.
Выход происходит комбинацией **Ctrl+D**, либо набрав функцию **exit()**.

Изучаем инструмент

На самом деле Scapy сильно отличается от привычных утилит. Он работает в текстовом режиме, но любое взаимодействие осуществляется не через привычные ключи и параметры командной строки, а через интерпретатор Python'a.
Такой подход вначале может показаться несколько неудобным и непривычным, но со временем и с практикой приходит понимание того, что это было правильным решением, и что это действительно удобно.

Вначале посмотрим на поддерживаемые протоколы, для этого вызовем функцию **ls()**.



Вывалится более 300 разнообразных протоколов, с которыми можно работать, включая прикладные вроде HTTP, транспортные TCP и UDP, сетевого уровня IPv4 и IPv6 и канального уровня Ether (Ethernet).
Важно обращать внимание на регистр: большинство протоколов пишутся в Scapy с заглавными буквами.
Для того чтобы подробно рассмотреть поля определенного протокола, можно вызвать функцую **ls()** с указанием протокола: **ls(TCP)**

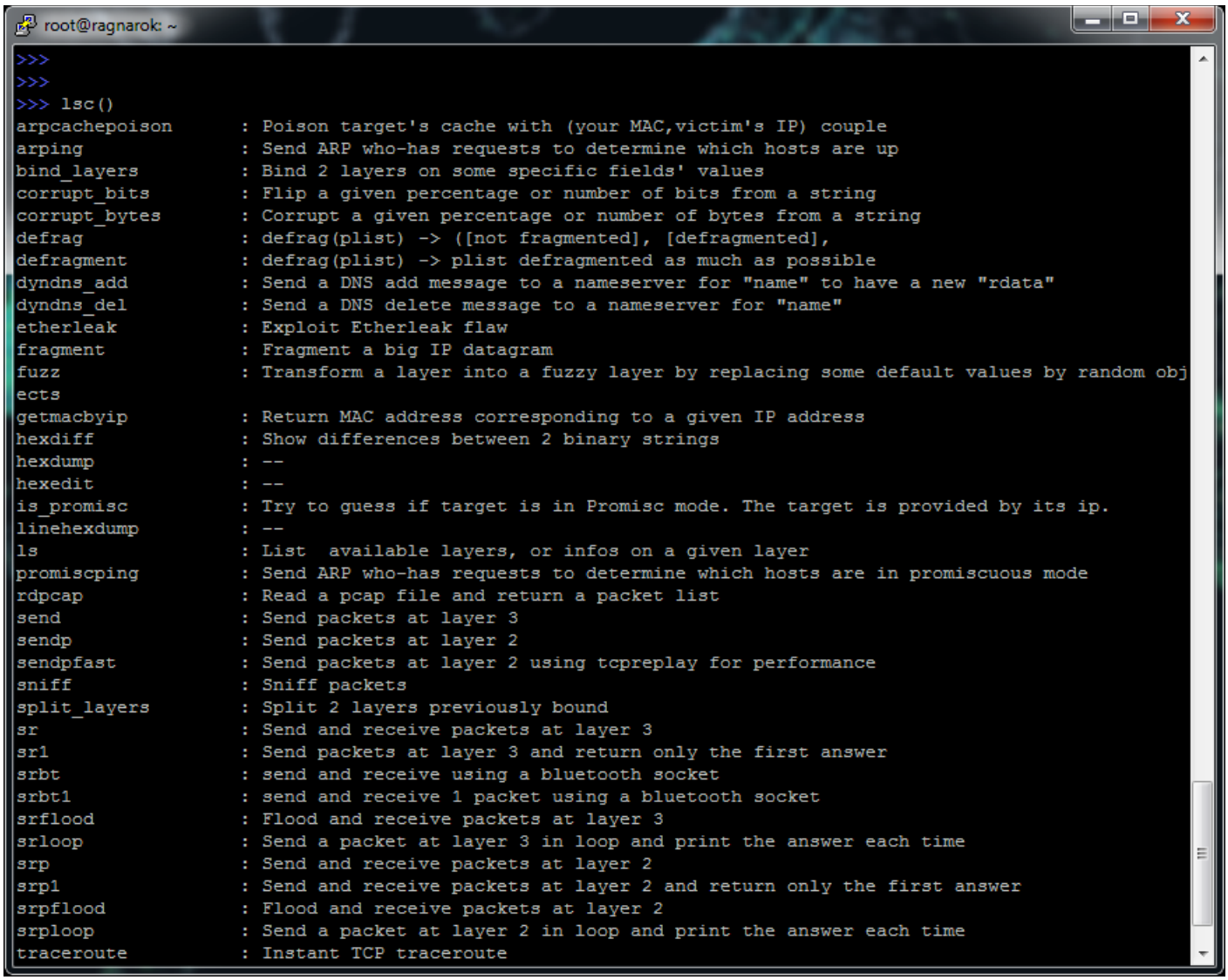
```
>>> ls(TCP)
sport      : ShortEnumField       = (20)
dport      : ShortEnumField       = (80)
seq        : IntField             = (0)
ack        : IntField             = (0)
dataofs    : BitField             = (None)
reserved   : BitField             = (0)
flags      : FlagsField           = (2)
window     : ShortField           = (8192)
chksum     : XShortField          = (None)
urgptr     : ShortField           = (0)
options    : TCPOptionsField      = ({})
>>>
```

В результате будут выведены все поля, которые можно модифицировать в процессе создания пакетов. В скобках показаны значения, которые используются по умолчанию, можно заметить, что порт отправителя 20 (это ftp-data) и порт получателя – 80 (это естественно HTTP), так же установлен флаг SYN (flags = 2).

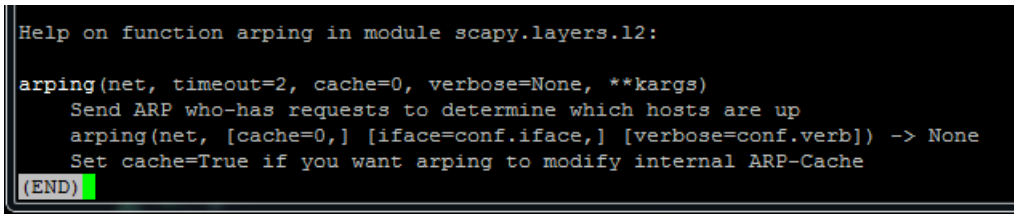
К примеру, если рассмотреть канальный уровень (Ethernet), то тут возможностей будет поменьше:

```
>>>
>>> ls(Ether)
dst        : DestMACField         = (None)
src        : SourceMACField       = (None)
type       : XShortEnumField      = (0)
>>>
```

В дополнение к функции **ls()**, есть полезная функция **lsc()**, которая выведет практически весь основной функционал Scapy:



Для того чтобы получить более подробную информацию о каждой функции, можно использовать **help(*имя_функции*)**, например:



Видим нечто похожее на MAN страницу в Unix системах.
Для выхода можно использовать опять же привычную в Linux клавишу **Q**.

Мы посмотрели на протоколы и функции, теперь можно перейти к делу — к созданию пакетов.

Крафтим

Можно создавать сразу пакеты высоких уровней (сетевого и прикладного), и Scapy автоматически дополнит низлежащие уровни, а можно вручную собирать, начиная с канального уровня.
Разделяются уровни модели OSI символом прямого слэша (/).
Нужно обратить внимание на то, что Scapy читает пакет от нижнего уровня слева, до более высокого справа. Поначалу это может немного сбивать с толку, но после небольшой практики всё станет вполне привычно.
К слову, в терминологии Scapy сетевой пакет разделяется на слои, и каждый слой представляется как экземпляр объекта.
Собранный пакет в упрощенном виде может выглядеть как:
`Ether()/IP()/TCP()/"App Data"`
В большинстве случаев используется только уровень L3, предоставляя Scapy возможность самостоятельно заполнять канальный уровень, на основе информации из ОС.
Меняя значения полей каждого протокола мы меняем стандартные значения (их выводит функция **ls()**).

Теперь создадим какой-нибудь простой пакет.

```
>>>
>>> packet=IP(dst="192.168.10.10")/TCP(dport=22)/"TEST"
>>>
```

Всё очень просто: мы указали адрес назначения, порт и вобщем-то нагрузку в виде слова «TEST».
Сам пакет был незамысловато назван *packet*, мы увидим очень подробно и развернуто наш свежесозданный пакет:
И теперь, выполнив знакомую уже функцию **ls(packet)**:

```
>>>
>>> ls(packet)
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                 = '192.168.10.200' (None)
dst        : Emph                 = '192.168.10.10' ('127.0.0.1')
options    : PacketListField      = []              ([])
--
sport      : ShortEnumField       = 20              (20)
dport      : ShortEnumField       = 22              (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      = {}              ({})
--
load       : StrField             = 'TEST'          ('')
>>>
>>>
>>>
```

Уровни в нем разделяются символами "--".

Вместо того, чтобы создавать пакет за один раз можно создавать его частями:

```
>>>
>>> part3=IP(dst="192.168.10.10")
>>> part4=TCP(dport=22)
>>> part7="TEST"
>>>
```

В этом примере мы создали переменные под каждый уровень модели OSI.
В качестве имен переменных можно использовать и буквы и цифры, при этом, не забывая о регистре.

И теперь собираем всё в один пакет:

```
>>> packet=part3/part4/part7
>>>
>>> ls (packet)
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                 = '192.168.10.200' (None)
dst        : Emph                 = '192.168.10.10' ('127.0.0.1')
options    : PacketListField      = []              ([])
--
sport      : ShortEnumField       = 20              (20)
dport      : ShortEnumField       = 22              (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      = {}              ({})
--
load       : StrField             = 'TEST'          ('')
>>>
```

Видно, что результат получится аналогичный.

Углубляемся в пакеты

Мы уже смотрели на вывод функции **ls()**, но не всегда нужна такая подробная информация о пакете.
Достаточно просто набрать имя переменной и сразу увидеть краткую сводку:

```
>>>
>>> packet
<IP  frag=0 proto=tcp dst=192.168.10.10 |<TCP  dport=ssh |<Raw  load='TEST' |>>>
>>>
```

Так же можно использовать метод **summary()**:

```
>>>
>>> packet.summary()
'IP / TCP 192.168.10.200:ftp_data > 192.168.10.10:ssh S / Raw'
>>>
```

Если же нужно чуть больше информации, то есть метод **show()**:

```
>>>
>>> packet.show()
###[ IP ]###
  version= 4
  ihl= None
  tos= 0x0
  len= None
  id= 1
  flags=
  frag= 0
  ttl= 64
  proto= tcp
  chksum= None
  src= 192.168.10.200
  dst= 192.168.10.10
  \options\
###[ TCP ]###
     sport= ftp_data
     dport= ssh
     seq= 0
     ack= 0
     dataofs= None
     reserved= 0
     flags= S
     window= 8192
     chksum= None
     urgptr= 0
     options= {}
###[ Raw ]###
        load= 'TEST'
>>>
```

Кроме того, можно просмотреть любое поле, просто указав его:

```
>>>
>>> packet.dst
'192.168.10.10'
>>> packet.dport
22
>>>
```

Разумеется, это работает только в том случае, если такие поля уникальны в пределах пакета.
Если, например, взять поле flags, которое присутствует как в TCP, так и в IP, тут уже нужно конкретизировать, что мы хотим увидеть. В противном случае Scapy выведет значение первого найденного поля (IP flags в нашем примере).
Конкретизация происходит путем указания протокола в квадратных скобках:

```
>>>
>>> packet.flags
0
>>> packet[TCP].flags
2
>>>
```

К слову, по умолчанию установленные флаги выводятся в цифровом представлении.

Если все управляющие биты будут включены (установлены в 1), то получим значение равное 255. В нашем случае значение 2 говорит о том, что установлен SYN бит.

Но существует возможность отобразить управляющие биты и в символьном отображении:

```
>>>
>>>
>>> packet.sprintf("%TCP.flags%")
'S'
>>>
```

Как уже говорилось, в любой момент можно достаточно просто поменять значение любого поля:

```
>>>
>>> packet.sport=443
>>> packet
<IP  frag=0 proto=tcp dst=192.168.10.10 |<TCP  sport=https dport=ssh |<Raw  load='TEST' |>>>
>>>
```

А в случае, если поле не является уникальным, то нужно указать протокол:

```
>>>
>>> packet[TCP].flags="SA"
>>> packet
<IP  frag=0 proto=tcp dst=192.168.10.10 |<TCP  sport=https dport=ssh flags=SA |<Raw  load='TEST' |>>>
>>>
```

Вторым способом является использование конструкции **payload**, которая позволяет перепрыгнуть через один уровень (через L3 в нашем случае):

```
>>>
>>>
>>> packet.payload
<TCP  sport=https dport=ssh flags=S |<Raw  load='TEST' |>>
>>> packet.payload.flags
2
>>> packet.payload.flags="SA"
>>> packet
<IP  frag=0 proto=tcp dst=192.168.10.10 |<TCP  sport=https dport=ssh flags=SA |<Raw  load='TEST' |>>>
>>>
```

Здесь мы вначале просматриваем вывод слоев над L3, затем просматриваем значение TCP флагов и устанавливаем для них новое значение.
Кстати, можно даже несколько раз вызвать payload, поднимаясь при этом выше

и выше:

```
>>>
>>> packet
<IP  frag=0 proto=tcp dst=192.168.10.10 |<TCP  sport=https dport=ssh flags=SA |<Raw  load='TEST' |>>>
>>> packet.payload
<TCP  sport=https dport=ssh flags=SA |<Raw  load='TEST' |>>
>>> packet.payload.payload
<Raw  load='TEST' |>
>>>
```

Можно еще посмотреть на содержимое пакета в шестнадцатеричном виде, для этого есть функция **hexdump()**:

```
>>>
>>> hexdump(packet)
0000   45 00 00 2C 00 01 00 00  40 06 E4 A8 C0 A8 0A C8   E..,....@.......
0010   C0 A8 0A 0A 01 BB 00 16  00 00 00 00 00 00 00 00   ................
0020   50 12 20 00 50 41 00 00  54 45 53 54              P. .PA..TEST
>>>
>>>
>>> hexdump(packet[TCP])
0000   01 BB 00 16 00 00 00 00  00 00 00 00 50 12 20 00   ............P. .
0010   50 41 00 00 54 45 53 54                           PA..TEST
>>>
```

Разбираемся с адресацией

Scapy и в деле указания адреса получателя так же проявляет большую гибкость. Масса вариантов — здесь и привычная десятичная форма, и доменное имя и CIDR нотация:

```
>>>
>>>
>>> packet=IP(dst="192.168.10.10")
>>>
>>> packet=IP(dst="192.168.10/24")
>>>
```

В последнем случае пакет будет отправлен на каждый адрес в подсети.

Множество адресов можно задать, просто разделяя их запятой, не забыв про квадратные скобки:

```
>>>
>>>
>>> packet=IP(dst=["192.168.10.1","192.168.10.5","192.168.10.10"])
>>>
```

На этом этапе может возникнуть мысль: «А что если нужно задать множество портов?».
И тут Scapy предоставляет широкие возможности, можно указать как диапазон, так и просто перечислить множество:

```
>>>
>>> packet=IP(dst="192.168.10.10")/TCP(dport=(1,1000))
>>>
>>> packet
<IP  frag=0 proto=tcp dst=192.168.10.10 |<TCP  dport=(1, 1000) |>>
>>>
>>> packet=IP(dst="192.168.10.10")/TCP(dport=[22,25,80])
>>>
>>> packet
<IP  frag=0 proto=tcp dst=192.168.10.10 |<TCP  dport=['ssh', 'smtp', 'http'] |>>
>>>
```

Обращаю внимание на различие в скобках, в случае диапазона они круглые, а в случае множества – квадратные.
И завершая разговор про указание целей, рассмотрим ситуацию, когда нужно отправить множество пакетов на множество портов.
Для того, чтобы увидеть какие пакеты будут отправлены придется задействовать цикл **for**, не забываем, что язык программирования у нас Python.
На самом деле ничего сложного, всё очень логично:

```
>>>
>>> packets=IP(dst="192.168.10.10/30")/TCP(dport=(21,25))
>>>
>>> [a for a in packets]
[<IP  frag=0 proto=tcp dst=192.168.10.8 |<TCP  dport=ftp |>>, <IP  frag=0 proto=tcp dst=192.168.10.8 |<TC
P  dport=ssh |>>, <IP  frag=0 proto=tcp dst=192.168.10.8 |<TCP  dport=telnet |>>, <IP  frag=0 proto=tcp d
st=192.168.10.8 |<TCP  dport=24 |>>, <IP  frag=0 proto=tcp dst=192.168.10.8 |<TCP  dport=smtp |>>, <IP  f
rag=0 proto=tcp dst=192.168.10.9 |<TCP  dport=ftp |>>, <IP  frag=0 proto=tcp dst=192.168.10.9 |<TCP  dpor
t=ssh |>>, <IP  frag=0 proto=tcp dst=192.168.10.9 |<TCP  dport=telnet |>>, <IP  frag=0 proto=tcp dst=192.
168.10.9 |<TCP  dport=24 |>>, <IP  frag=0 proto=tcp dst=192.168.10.9 |<TCP  dport=smtp |>>, <IP  frag=0 p
roto=tcp dst=192.168.10.10 |<TCP  dport=ftp |>>, <IP  frag=0 proto=tcp dst=192.168.10.10 |<TCP  dport=ssh
 |>>, <IP  frag=0 proto=tcp dst=192.168.10.10 |<TCP  dport=telnet |>>, <IP  frag=0 proto=tcp dst=192.168.
10.10 |<TCP  dport=24 |>>, <IP  frag=0 proto=tcp dst=192.168.10.10 |<TCP  dport=smtp |>>, <IP  frag=0 pro
to=tcp dst=192.168.10.11 |<TCP  dport=ftp |>>, <IP  frag=0 proto=tcp dst=192.168.10.11 |<TCP  dport=ssh |
>>, <IP  frag=0 proto=tcp dst=192.168.10.11 |<TCP  dport=telnet |>>, <IP  frag=0 proto=tcp dst=192.168.10
.11 |<TCP  dport=24 |>>, <IP  frag=0 proto=tcp dst=192.168.10.11 |<TCP  dport=smtp |>>]
>>>
```

Вначале мы уже привычно создаем пакет, в котором указываем подсеть и диапазон портов.
Затем, используя цикл, создаем список, где переменной «*a*» присваивается каждый элемент структуры пакета. В Python'e отсутствуют массивы в

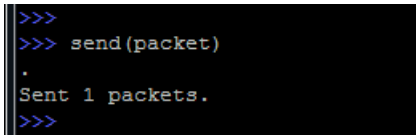
привычном понимании. Вместо них для хранения объектов используются списки.
Мы используем цикл **for**, для того чтобы «распаковать» всю структуру и отобразить ее в таком наглядном виде.

Отправляем пакеты в путь

С таким же размахом и широтой происходит и отправка пакетов:

- функция **send()** – отправляет пакеты, используя сетевой (L3) уровень, никак не обрабатывая ответы. Используется принцип — отправили и забыли;
- функция **sendp()** – отправляет пакеты, используя канальный (L2) уровень, учитываются указанные параметры и заголовки Ethernet кадров. Ответы всё так же не ожидаются и не обрабатываются;
- функция **sr()** – является аналогичной send(), исключение составляет то, что она уже ожидает ответные пакеты;
- функция **srp()** – отправляет и принимает пакеты, уровень L2
- функция **sr1()** – отправляет пакет третьего уровня и забирает только первый ответ, множество ответов не предусматривается;
- функция **srp1()** – аналогично sr1(), только уже канальный уровень.

Каждую из этих функций можно вызвать и без дополнительных параметров, просто указывая имя переменной, содержащей пакет.



Но вместе с тем существует много дополнительных опций, которые могут быть иногда полезны.
Например, *timeout* – укажет, сколько времени (в секундах) нужно ждать до получения ответного пакета, *retry* – сколько раз нужно повторно слать пакет, если ответ не был получен и одна из самых полезных опций – это *filter*, синтаксис которого очень похож на tcpdump.

В качестве наглядного примера отправим пакет в сеть:

```
>>>
>>> sr(packet,timeout=0.1,filter="host 192.168.10.10 and port 23")
Begin emission:
Finished to send 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
(<Results: TCP:1 UDP:0 ICMP:0 Other:0>, <Unanswered: TCP:0 UDP:0 ICMP:0 Other:0>)
>>>
```

Здесь мы используем функцию, которая после отправки ожидает ответ, устанавливаем таймаут 0.1 секунды и фильтруем ответы, которые подпадают под указанное правило.

Как поступать с ответными пакетами?
Можно взять и назначить переменную, которая и будет содержать ответ:

```
>>>
>>> response=sr(packet)
Begin emission:
Finished to send 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
>>>
>>>
>>>
>>> response
(<Results: TCP:1 UDP:0 ICMP:0 Other:0>, <Unanswered: TCP:0 UDP:0 ICMP:0 Other:0>)
>>>
```

А смотреть уже привычным способом, просто вызывая переменную *response*. Видно, что ответ сохранился в двух вариантах – Results и Unanswered, результаты и без ответа, соответственно.
Указывая смещение, можно вывести только необходимую часть ответа:

```
>>>
>>> response[0]
<Results: TCP:1 UDP:0 ICMP:0 Other:0>
>>>
>>> response[1]
<Unanswered: TCP:0 UDP:0 ICMP:0 Other:0>
>>>
```

Или подробную информацию:

```
>>>
>>> response[0][0]
(<IP  frag=0 proto=tcp dst=192.168.10.10 |<TCP  dport=telnet |>>, <IP  version=4L ihl=5L tos=0x0 len=44 i
d=19989 flags= frag=0L ttl=255 proto=tcp chksum=0xd793 src=192.168.10.10 dst=192.168.10.200 options=[] |<
TCP  sport=telnet dport=ftp_data seq=1104310806 ack=1 dataofs=6L reserved=0L flags=SA window=4128 chksum=
0x415b urgptr=0 options=[('MSS', 536)] |<Padding  load='\x00\x00' |>>>)
>>>
```

Если же пакет был отправлен в сеть без указания переменной (например, просто функцией **sr()**), то по умолчанию пакет будет числиться за переменной **"_"** (символ подчеркивания).

Чтобы достать оттуда эти пакеты, можно использовать конструкцию:

```
>>>
>>> res,unans=_
>>>
>>> res
<Results: TCP:1 UDP:0 ICMP:0 Other:0>
>>>
>>> unans
<Unanswered: TCP:0 UDP:0 ICMP:0 Other:0>
>>>
```

При этом разные результаты сохранятся в двух разных переменных (*res* и *unans*).

Более подробный вывод достигается опять же путем указания смещения:

```
>>> res[0]
(<IP  frag=0 proto=tcp dst=192.168.10.10 |<TCP  dport=telnet |>>, <IP  version=4L ihl=5L tos=0x0 len=44 i
d=59054 flags= frag=0L ttl=255 proto=tcp chksum=0x3efa src=192.168.10.10 dst=192.168.10.200 options=[] |<
TCP  sport=telnet dport=ftp_data seq=2455132138L ack=1 dataofs=6L reserved=0L flags=SA window=4128 chksum
=0xb03 urgptr=0 options=[('MSS', 536)] |<Padding  load='\x00\x00' |>>>)
>>>
```

Ловим ответные пакеты

Теперь рассмотрим ситуацию, если пакетов в ответ приходит много.

```
>>>
>>> packet=IP(dst="192.168.10.10")/TCP(dport=(1,100),flags="S")
>>>
>>> res,unans=sr(packet,timeout=10)
Begin emission:
....*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.Finished to send 100 packets.
.*.*.*.*.*.*.*.*.*.*.*.*...................................................
Received 136 packets, got 40 answers, remaining 60 packets
>>>
```

То, что мы увидели, было по сути, самое что ни есть сканирование портов.

Открытые порты будут с флагами SA (SYN/ACK), например:

```
>>>
>>> res[21]
(<IP  frag=0 proto=tcp dst=192.168.10.10 |<TCP  dport=telnet flags=S |>>, <IP  version=4L ihl=5L tos=0x0
len=44 id=65181 flags= frag=0L ttl=255 proto=tcp chksum=0x270b src=192.168.10.10 dst=192.168.10.200 optio
ns=[] |<TCP  sport=telnet dport=ftp_data seq=2061852671 ack=1 dataofs=6L reserved=0L flags=SA window=4128
 chksum=0x1a5f urgptr=0 options=[('MSS', 536)] |<Padding  load='\x00\x00' |>>>)
>>>
```

Мы смотрим именно на пакет по номеру, счет традиционно начинается нуля.

Можно пойти дальше и распаковать этот результат:

```
>>>
>>> sent21,rec21=res[21]
>>>
>>> sent21
<IP  frag=0 proto=tcp dst=192.168.10.10 |<TCP  dport=telnet flags=S |>>
>>>
>>> rec21
<IP  version=4L ihl=5L tos=0x0 len=44 id=65181 flags= frag=0L ttl=255 proto=tcp chksum=0x270b src=192.168
.10.10 dst=192.168.10.200 options=[] |<TCP  sport=telnet dport=ftp_data seq=2061852671 ack=1 dataofs=6L r
eserved=0L flags=SA window=4128 chksum=0x1a5f urgptr=0 options=[('MSS', 536)] |<Padding  load='\x00\x00'
|>>>
>>>
```

Здесь мы извлекли из результата отправленный пакет (под номером 21) и ответ на него.
Но это только один пакет, а как быть, если нужно обработать все пакеты?
В таком случае придется вновь обращаться к циклу **for**:

```
>>>
>>> allsent=[a for a,b in res]
>>>
>>> allrec=[b for a,b in res]
>>>
>>> allsent[2]
<IP  frag=0 proto=tcp dst=192.168.10.10 |<TCP  dport=3 flags=S |>>
>>>
>>> allrec[2]
<IP  version=4L ihl=5L tos=0x0 len=40 id=9201 flags= frag=0L ttl=255 proto=tcp chksum=0x1bc src=192.168.1
0.10 dst=192.168.10.200 options=[] |<TCP  sport=3 dport=ftp_data seq=0 ack=1 dataofs=5L reserved=0L flags
=RA window=0 chksum=0x1996 urgptr=0 |<Padding  load='\x00\x00\x00\x00\x00\x00' |>>>
>>>
```

Берем и разбиваем каждый элемент списка *res* на части *a* и *b*. Затем обрезаем часть “*a*”, заливая это всё в список “*allsent*”.
Аналогично создаем список *allrec*, только уже оставляем другую часть.
Всё это, конечно, хорошо, но хотелось бы в более удобном виде получить список открытых и закрытых портов.

Еще раз посмотрим на список *res*, а точнее на элемент res[0], который состоит из двух частей: пакет, который мы отправили res[0][0], и ответ, который получили res[0][1].

```
>>>
>>> res[0]
(<IP  frag=0 proto=tcp dst=192.168.10.10 |<TCP  dport=tcpmux flags=S |>>, <IP  version=4L ihl=5L tos=0x0
len=40 id=44469 flags= frag=0L ttl=255 proto=tcp chksum=0x77f7 src=192.168.10.10 dst=192.168.10.200 optio
ns=[] |<TCP  sport=tcpmux dport=ftp_data seq=0 ack=1 dataofs=5L reserved=0L flags=RA window=0 chksum=0x19
98 urgptr=0 |<Padding  load='\x00\x00\x00\x00\x00\x00' |>>>)
>>>
>>>
>>> res[0][0]
<IP  frag=0 proto=tcp dst=192.168.10.10 |<TCP  dport=tcpmux flags=S |>>
>>>
>>> res[0][1]
<IP  version=4L ihl=5L tos=0x0 len=40 id=44469 flags= frag=0L ttl=255 proto=tcp chksum=0x77f7 src=192.168
.10.10 dst=192.168.10.200 options=[] |<TCP  sport=tcpmux dport=ftp_data seq=0 ack=1 dataofs=5L reserved=0
L flags=RA window=0 chksum=0x1998 urgptr=0 |<Padding  load='\x00\x00\x00\x00\x00\x00' |>>>
>>>
```

В ответе можно обнаружить три части — заголовок IP (res[0][1][0]), заголовок TCP (res[0][1][1]) и собственно сами данные (res[0][1][2]).

```
>>>
>>> res[0][1][0]
<IP  version=4L ihl=5L tos=0x0 len=40 id=44469 flags= frag=0L ttl=255 proto=tcp chksum=0x77f7 src=192.168
.10.10 dst=192.168.10.200 options=[] |<TCP  sport=tcpmux dport=ftp_data seq=0 ack=1 dataofs=5L reserved=0
L flags=RA window=0 chksum=0x1998 urgptr=0 |<Padding  load='\x00\x00\x00\x00\x00\x00' |>>>
>>>
>>> res[0][1][1]
<TCP  sport=tcpmux dport=ftp_data seq=0 ack=1 dataofs=5L reserved=0L flags=RA window=0 chksum=0x1998 urgp
tr=0 |<Padding  load='\x00\x00\x00\x00\x00\x00' |>>
>>>
>>> res[0][1][2]
<Padding  load='\x00\x00\x00\x00\x00\x00' |>
>>>
```

Используем цикл **for** для извлечения каждого элемента res[N] в переменную *«a»*.

```
>>>
>>> for a in res:
...
```

Теперь в переменной «a»содержится результат для каждого пакета. Другими словами «a» представляет собой ans[N].
Нам остается только проверить значения a[1][1], которые будут означать res[N][1][1] в заголовке TCP.
Если быть еще более точным, требуется значение 18, которое означает

установленные флаги SYN-ACK.

```
>>>
>>> for a in res:
...     if a[1][1].flags==18:
...
```

В тех случаях, когда это условие сработает, мы еще выведем порт отправителя из заголовка TCP:

```
>>>
>>> for a in res:
...     if a[1][1].flags==18:
...         print a[1].sport
...
21
22
80
>>>
```

В итоге, получим результат в виде списка открытых портов.
Все вышеозначенные конструкции набираются за один раз, важно так же уделять внимание отступам (обычно это 4 пробела).

Мы только что вручную написали простой сканер портов, не больше и не меньше.

Листинг 1 Сканер портов

```
>>> packet=IP(dst=«192.168.10.10»)/TCP(dport=(1,100),flags=«S»))

>>> res,unans=sr(packet,timeout=10)

>>> for a in res:
… if a[1][1].flags==18:
… print a[1].sport
```

Сниффер и наоборот

В Scapy входит также и небольшой сниффер, за который отвечает функция **sniff()**.

Естественно, с ним можно использовать фильтры (похожие на фильтры tcpdump), за это отвечает параметр *filter*, так же можно ограничивать количество пакетов с помощью параметра *count*.
Как всегда вызов **help(sniff)** выведет вполне подробную информацию по этой функции.
Не следует забывать, что это сильно упрощенный сниффер, и ожидать от него хорошей скорости особо не приходится.
Стандартная комбинация **Ctrl+C** прервет процесс захвата трафика и выведет результат.
Как и любая неопределенная переменная, результат попадет в **"_"**.
Выполнив метод **summary()**, можно увидеть статистику по захваченным пакетам:

```
>>>
>>> sniff()
^C<Sniffed: TCP:2 UDP:0 ICMP:10 Other:3>
>>>
>>> _.summary()
Ether / IP / TCP 192.168.10.200:ssh > 192.168.10.1:59978 PA / Raw
Ether / IP / TCP 192.168.10.1:59978 > 192.168.10.200:ssh A / Padding
802.3 c2:00:11:58:00:01 > 01:00:0c:cc:cc:cc / LLC / SNAP / Raw
Ether / IP / ICMP 192.168.10.10 > 192.168.10.200 echo-request 0 / Raw
Ether / ARP who has 192.168.10.10 says 192.168.10.200
Ether / ARP is at c2:00:11:58:00:01 says 192.168.10.10 / Padding
Ether / IP / ICMP 192.168.10.200 > 192.168.10.10 echo-reply 0 / Raw
Ether / IP / ICMP 192.168.10.10 > 192.168.10.200 echo-request 0 / Raw
Ether / IP / ICMP 192.168.10.200 > 192.168.10.10 echo-reply 0 / Raw
Ether / IP / ICMP 192.168.10.10 > 192.168.10.200 echo-request 0 / Raw
Ether / IP / ICMP 192.168.10.200 > 192.168.10.10 echo-reply 0 / Raw
Ether / IP / ICMP 192.168.10.10 > 192.168.10.200 echo-request 0 / Raw
Ether / IP / ICMP 192.168.10.200 > 192.168.10.10 echo-reply 0 / Raw
Ether / IP / ICMP 192.168.10.10 > 192.168.10.200 echo-request 0 / Raw
Ether / IP / ICMP 192.168.10.200 > 192.168.10.10 echo-reply 0 / Raw
>>>
```

Вместо захвата трафика из сети, можно прочитать его из заранее сохраненного дампа (pcap файла).

```
>>>
>>> rdpcap("testlab2.pcap")
<testlab2.pcap: TCP:0 UDP:0 ICMP:8 Other:2>
>>>
>>>
>>> _.summary()
Ether / ARP who has 192.168.10.50 says 192.168.10.200
Ether / ARP is at 00:0c:29:bd:d8:86 says 192.168.10.50 / Padding
Ether / IP / ICMP 192.168.10.200 > 192.168.10.50 echo-request 0 / Raw
Ether / IP / ICMP 192.168.10.50 > 192.168.10.200 echo-reply 0 / Raw
Ether / IP / ICMP 192.168.10.200 > 192.168.10.50 echo-request 0 / Raw
Ether / IP / ICMP 192.168.10.50 > 192.168.10.200 echo-reply 0 / Raw
Ether / IP / ICMP 192.168.10.200 > 192.168.10.50 echo-request 0 / Raw
Ether / IP / ICMP 192.168.10.50 > 192.168.10.200 echo-reply 0 / Raw
Ether / IP / ICMP 192.168.10.200 > 192.168.10.50 echo-request 0 / Raw
Ether / IP / ICMP 192.168.10.50 > 192.168.10.200 echo-reply 0 / Raw
>>>
```

Кроме того, можно и наоборот, записать пойманные пакеты в файл, используя функцию **wrpcap()**:

```
>>>
>>> packets=sniff()
^C>>>
>>> packets.summary()
Ether / IP / TCP 192.168.10.200:ssh > 192.168.10.1:59978 PA / Raw
Ether / IP / TCP 192.168.10.1:59978 > 192.168.10.200:ssh A / Padding
Ether / IP / ICMP 192.168.10.10 > 192.168.10.200 echo-request 0 / Raw
Ether / ARP who has 192.168.10.10 says 192.168.10.200
Ether / ARP is at c2:00:11:58:00:01 says 192.168.10.10 / Padding
Ether / IP / ICMP 192.168.10.200 > 192.168.10.10 echo-reply 0 / Raw
Ether / IP / ICMP 192.168.10.10 > 192.168.10.200 echo-request 0 / Raw
Ether / IP / ICMP 192.168.10.200 > 192.168.10.10 echo-reply 0 / Raw
Ether / IP / ICMP 192.168.10.10 > 192.168.10.200 echo-request 0 / Raw
Ether / IP / ICMP 192.168.10.200 > 192.168.10.10 echo-reply 0 / Raw
Ether / IP / ICMP 192.168.10.10 > 192.168.10.200 echo-request 0 / Raw
Ether / IP / ICMP 192.168.10.200 > 192.168.10.10 echo-reply 0 / Raw
Ether / IP / ICMP 192.168.10.10 > 192.168.10.200 echo-request 0 / Raw
Ether / IP / ICMP 192.168.10.200 > 192.168.10.10 echo-reply 0 / Raw
>>>
>>> wrpcap("testlab3.pcap",packets)
>>>
```

И завершая тему сниффинга, можно вызвать Wireshark прямо из интерфейса Scapy, для этого можно использовать одноименную функцию **wireshark()**.

Подробно про Wireshark можно в моей предыдущей статье по адресу http://linkmeup.ru/blog/115.html.

Автоматизация

Всё, что мы рассматривали, происходило непосредственно в интерактивном режиме.

Но, естественно, многие вещи можно автоматизировать, написав скрипты.
Для этого в начале скрипта нужно будет указать:

```
#!/usr/bin/python
```

Знакомый для пользователей ОС Linux, shebang.
(http://en.wikipedia.org/wiki/Shebang_(Unix))

```
from scapy.all import *
```

Импортировать весь функционал Scapy.
После этого уже можно писать необходимые функции.
Важно делать отступы при написании циклов в скриптах, иначе будут появляться сообщения об ошибках, и скрипт не будет работать.
Тут же рассмотрим и подключение дополнительных модулей к Scapy на примере OSPF.
Изначально Scapy не умеет работать с протоколом OSPF.
Если выполнить **load_contrib('ospf')**, то будет сообщение об ошибке: *«ERROR: No module named contrib.ospf»*.
Вначале скачаем модуль, его можно взять тут.
Затем нужно создать каталог *contrib*:

```
# mkdir /usr/lib/python2.7/dist-packages/scapy/contrib
```

И перенести модуль в свежесозданный каталог:

```
# cp ospf.py /usr/lib/python2.7/dist-
packages/scapy/contrib/
```

Теперь если зайти в Scapy и просмотреть список подключенных сторонних модулей (за это, как вы догадались, отвечает функция **list_contrib()**):

```
>>> list_contrib()
ospf: OSPF status=loads
>>>
```

Казалось бы, что уже всё готово, но не тут то было.
При очередной попытке подгрузить ospf модуль:
>>> **load_contrib('ospf')**, получаем всё ту же ошибу *«ERROR: No module named contrib.ospf»*
Для того, чтобы модуль окончательно заработал, осталось создать скрипт инициализации (пустой файл):

```
touch /usr/lib/python2.7/dist-
packages/scapy/contrib/__init__.py
```

После этого, уже можно будет создавать пакеты для OSPF.

Создаем трехэтапное TCP-соединение

Для этого нужно будет поймать SYN/ACK ответ, извлечь из него TCP sequence number, увеличить значение на единицу и, собственно, и поместить полученное значение в поле acknowledgement number.
Непростая задача на первый взгляд, но Scapy может справиться и с ней.
Вначале рассмотрим, что нам нужно, для того чтобы всё прошло успешно.

1) Отправить SYN принимающей стороне:

- собрать заголовок IP, не забыть про адрес отправителя и получателя;
- собрать TCP заголовок, в котором нужно будет указать TCP порты отправителя и назначения, установить TCP флаги (SYN бит) и сгенерировать ISN (Initial Sequence Number).

2) Поймать ответный пакет:

- сохранить ответ;
- извлечь из него TCP sequence number и увеличить это значение на единицу.

3) Создать подтверждение (ACK) на полученный ответный пакет:

- собрать заголовок IP, содержащий такие же адреса отправителя и получателя, как в случае SYN пакета;
- собрать TCP заголовок, с такими же номерами портов, как и в SYN сегменте, но уже установить ACK флаг, увеличить значение ISN на единицу и установить acknowledgement в извлеченный и увеличенный, на втором шаге, sequence number.

Для того чтобы стало еще понятней, рассмотрим уже более подробно, с использованием произвольно взятых значений.
К примеру, соединение прошло таким образом:

> 192.168.10.200 1024 > 192.168.10.50 80 flags=SYN seq=12345
> 192.168.10.50 80 > 192.168.10.200 1024 flags=SYN, ACK seq=9998 ack=12346
> 192.168.10.200 1024 > 192.168.10.50 80 flags=ACK seq=12346 ack=9999

Что в итоге нужно было сделать.

1) Отправить SYN принимающей стороне:

- собрать заголовок IP, в котором указать в качестве отправителя 192.168.10.200 и 192.168.10.50 в качестве получателя;
- собрать TCP заголовок с портом источника (source) 1024 и портом назначения (destination) 80. Так же установить SYN флаг и сгенерировать ISN равный 12345.

2) Поймать ответный пакет:

- сохранить ответ;
- извлечь из него TCP sequence number (9998) и увеличить это значение на единицу, получим 9999.

3) Создать подтверждение (ACK) на полученный ответный пакет:

- собрать заголовок IP, в котором указать в качестве отправителя 192.168.10.200 и 192.168.10.50 в качестве получателя;
- собрать TCP заголовок с такими же портами источника и назначения (1024 и 80 соответственно), установить ACK флаг, увеличить ISN на единицу (12346) и установить acknowledgement в увеличенное значение пойманного seq number (9999).

Начнем собирать пакет:

```
>>>
>>> ip=IP(src="192.168.10.200",dst="192.168.10.50")
>>>
>>> SYN=TCP(sport=1024,dport=80,flags="S",seq=12345)
>>>
>>> packet=ip/SYN
>>>
```

Здесь уже всё должно быть знакомым: пакет собираем из двух частей, инкапсулируя TCP в IP.

Теперь помня о том, что нам нужно будет перехватить ответ, извлечь оттуда sequence number и увеличить на единицу, делаем:

```
>>>
>>> SYNACK=sr1(packet)
Begin emission:
..Finished to send 1 packets.
*
Received 3 packets, got 1 answers, remaining 0 packets
>>>
>>> my_ack=SYNACK.seq+1
>>>
```

Происходит следующее – функция **sr1** отправляет ранее созданный пакет в сеть, а первый пришедший ответ помещается в переменную *SYNACK*.
А затем, используя конструкцию *SYNACK.seq*, извлекаем TCP sequence number, увеличиваем его на единицу и сохраняем в переменной *my_ack*.

Продвигаемся дальше:

```
>>>
>>> ACK=TCP(sport=1024,dport=80,flags="A",seq=12346,ack=my_ack)
>>>
>>> send(ip/ACK)
.
Sent 1 packets.
>>>
```

Создаем новый заголовок TCP и называем его ACK. В нем устанавливается другой флаг (A — ACK) и увеличивается значение sequence number.
Кроме того, в качестве acknowledgement указывается переменная *my_ack*.
Затем собранный пакет выбрасывается в сеть командой **send** (помним, что это L3 команда, которая даже не слушает, что придет в ответ).
Если всё было сделано правильно, то классическое TCP-соединение состоялось. Осталось только создать TCP сегмент без каких-либо флагов и тоже отправить в сеть.

```
>>>
>>> PUSH=TCP(sport=1024,dport=80,flags="",seq=12346,ack=my_ack)
>>>
>>> data="TEXT"
>>>
>>> send(ip/PUSH/data)
.
Sent 1 packets.
>>>
```

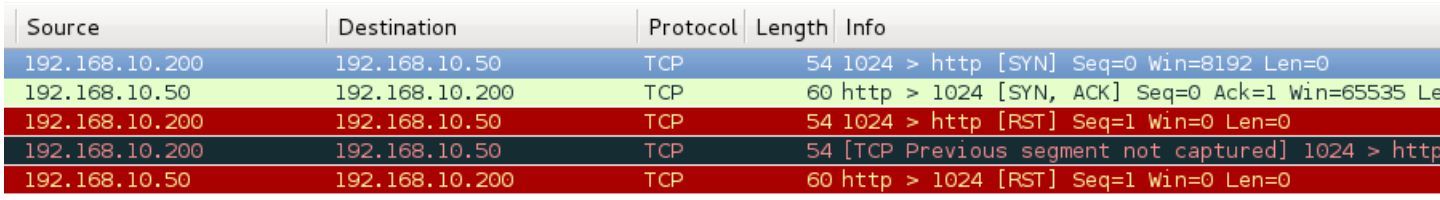
Как можно увидеть, мы в очередной раз создали экземпляр TCP заголовка (в этот раз, назвав его PUSH), без флагов и со всеми остальными знакомыми уже

значениями.
После этого добавили немного данных, используя переменную *data*, и отправили в сеть, используя ту же функцию **send**.
И соответственно от получателя должен будет прийти acknowledgement на этот сегмент.

Листинг 2 TCP-соединение

```
>>> ip=IP(src=«192.168.10.200»,dst=«192.168.10.50»)
>>> SYN=TCP(sport=1024,dport=80,flags=«S»,seq=12345)
>>> packet=ip/SYN
>>> SYNACK=sr1(packet)
>>> my_ack=SYNACK.seq+1
>>> ACK=TCP(sport=1024,dport=80,flags=«A»,seq=12346,ack=my+ack)
>>>send(ip/ACK)
```

Но здесь есть и несколько подводных камней.
Если посмотреть на этот обмен в Wireshark, можно увидеть, что до того как ушел наш ACK пакет, внезапно был отправлен RST:



Source	Destination	Protocol	Length	Info
192.168.10.200	192.168.10.50	TCP	54	1024 > http [SYN] Seq=0 Win=8192 Len=0
192.168.10.50	192.168.10.200	TCP	60	http > 1024 [SYN, ACK] Seq=0 Ack=1 Win=65535 Le
192.168.10.200	192.168.10.50	TCP	54	1024 > http [RST] Seq=1 Win=0 Len=0
192.168.10.200	192.168.10.50	TCP	54	[TCP Previous segment not captured] 1024 > http
192.168.10.50	192.168.10.200	TCP	60	http > 1024 [RST] Seq=1 Win=0 Len=0

Дело в том, что Scapy работает мимо TCP/IP стека ОС. Это означает то, что ОС не подозревает о том, что Scapy отправляет какие-то пакеты.
Соответственно ОС не будет ожидать появления SYN/ACK пакетов. И, следовательно, соединение будет сразу же сброшено.
Очевидно, что это совсем не тот результат, который нам нужен.
Одним из решений такой проблемы будет использование функционала пакетного фильтра, в частности iptables, который сможет блокировать исходящие RST пакеты.
Например, таким образом:

```
# iptables -A OUTPUT -p tcp -d 192.168.10.50 -s
192.168.10.200 --dport 80 --tcp-flags RST RST -j DROP
```

Выполнение такой конструкции приведет к тому, что все исходящие пакеты с

адресом назначения 192.168.10.50 и с адресом отправителя 192.168.10.200 на 80-й порт, с установленным RST флагом, будут отбрасываться.
Пакеты будут все так же генерироваться силами ОС, но они просто не будут вылетать за ее пределы.
В итоге уже ничего не будет мешать Scapy делать полноценную TCP-сессию:

Source	Destination	Protocol	Length	Info
192.168.10.200	192.168.10.50	TCP	54	1024 > http [SYN] Seq=0 Win=8192 Len=0
192.168.10.50	192.168.10.200	TCP	60	http > 1024 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len
192.168.10.200	192.168.10.50	TCP	54	1024 > http [ACK] Seq=1 Ack=1 Win=8192 Len=0

Продолжаем исследования

Используя Scapy, можно находить хосты в сети, среди указанного множества адресов:

```
>>>
>>> res, unans=sr(IP(dst=["192.168.10.1","192.168.10.10","192.168.10.50"])/ICMP())
Begin emission:
Finished to send 3 packets.
.*.*.........^C
Received 13 packets, got 2 answers, remaining 1 packets
>>>
>>> res.show()
0000 IP / ICMP 192.168.10.200 > 192.168.10.50 echo-request 0 ==> IP / ICMP 192.168.10.50 > 192.168.10.200
 echo-reply 0 / Padding
0001 IP / ICMP 192.168.10.200 > 192.168.10.10 echo-request 0 ==> IP / ICMP 192.168.10.10 > 192.168.10.200
 echo-reply 0 / Padding
>>>
```

В этом случае мы используем протокол ICMP и применяем знакомый прием по разделению полученных ответов.

```
>>>
>>> ls(ICMP)
type       : ByteEnumField        = (8)
code       : MultiEnumField       = (0)
chksum     : XShortField          = (None)
id         : ConditionalField     = (0)
seq        : ConditionalField     = (0)
ts_ori     : ConditionalField     = (36563761)
ts_rx      : ConditionalField     = (36563761)
ts_tx      : ConditionalField     = (36563761)
gw         : ConditionalField     = ('0.0.0.0')
ptr        : ConditionalField     = (0)
reserved   : ConditionalField     = (0)
addr_mask  : ConditionalField     = ('0.0.0.0')
unused     : ConditionalField     = (0)
>>>
```

По умолчанию, установлен 8-й тип для ICMP, это и есть классический эхо-запрос.

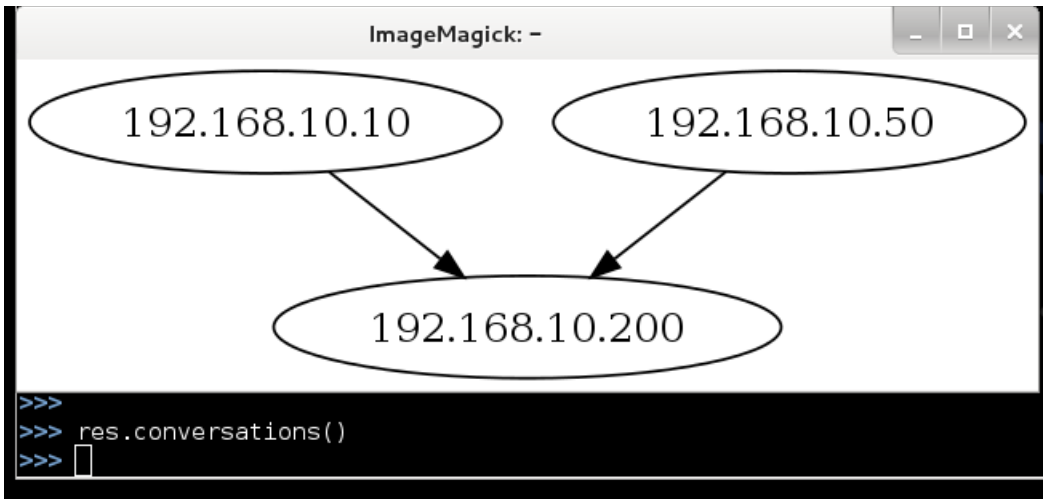
Углубляясь в тему ИБ, попробуем определить версию ОС используя Scapy и nmap.

```
>>>
>>> load_module("nmap")
>>>
>>> conf.nmap_base
'/usr/share/nmap/nmap-os-fingerprints'
>>>
>>> nmap_fp("192.168.10.10",oport=23,cport=10)
Begin emission:
.Finished to send 8 packets.
.*.*.*.*.*.*.*.*.*.*.*.*
Received 25 packets, got 6 answers, remaining 2 packets
(0.9486111111111112, ['Cisco IOS 12.0(7)T (on a 1700 router)'])
>>>
```

Итак, рассмотрим что было сделано.
Вначале был подключен внешний модуль, в данном случае nmap.
Затем проверяем, что у нас есть файл (nmap-os-fingerprints) с отпечатками различных ОС.
И запускаем непосредственно определение удаленной операционной системы, за это отвечает функция **nmap_fp**, где в качестве параметров помимо самой цели, можно еще указать открытый (oport) и закрытый (cport) порты.
Правильно указанные порты помогут сильно улучшить точность определения ОС.

Визуализируем пакеты

Все время мы смотрели на текстовый вывод, местами была псевдографика, но Scapy умеет и выводить некоторые результаты в графическом виде.
Посмотрим, что нам предлагается.
Самое простое — это метод **conversations()**:

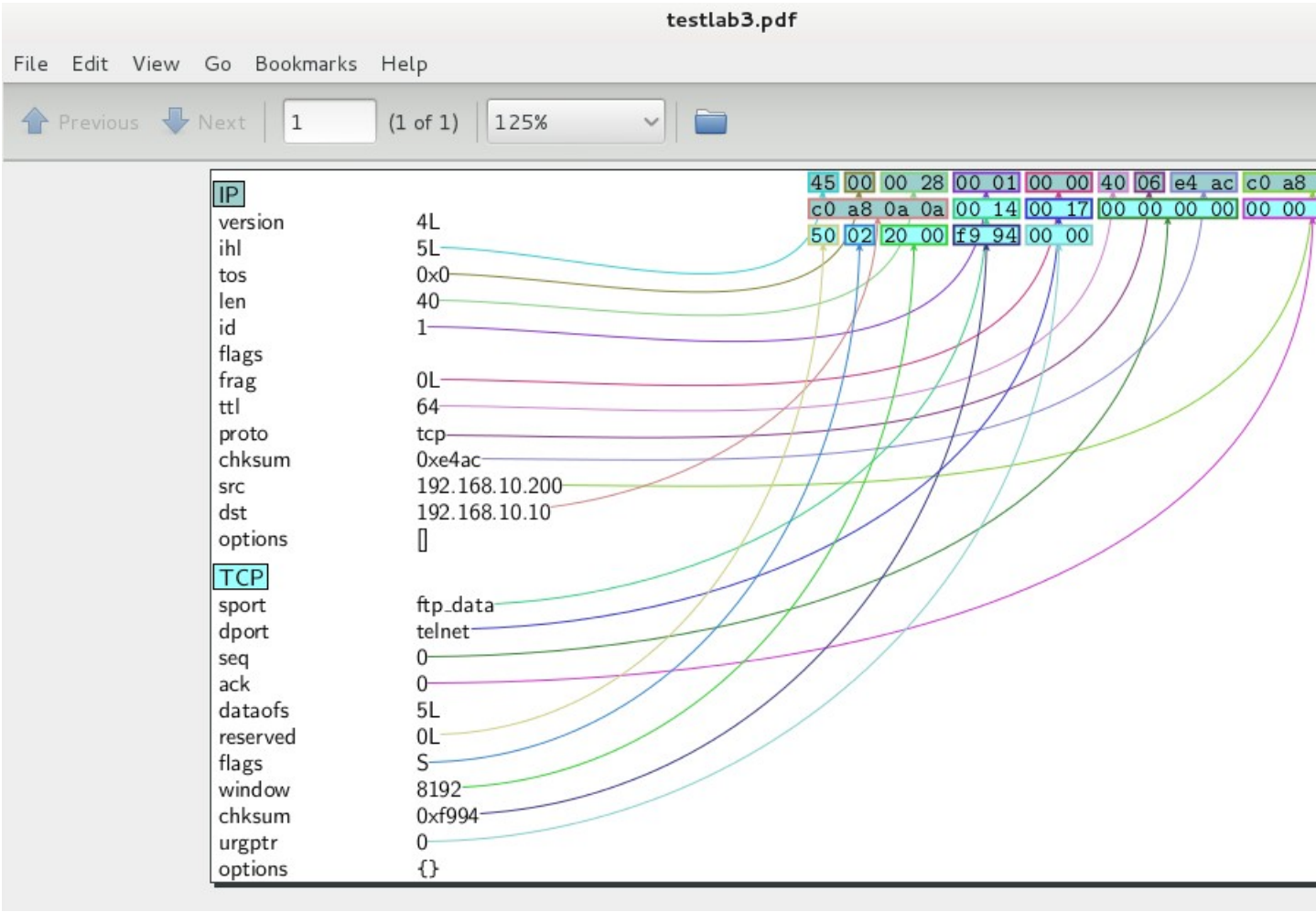


При его выполнении, запустится окно ImageMagick, в котором отрисуется схема нашего обмена пакетами, не ахти красиво, но достаточно наглядно.
Это способ, вероятно, лучше всего подойдет, для визуализации дампов с трафиком.

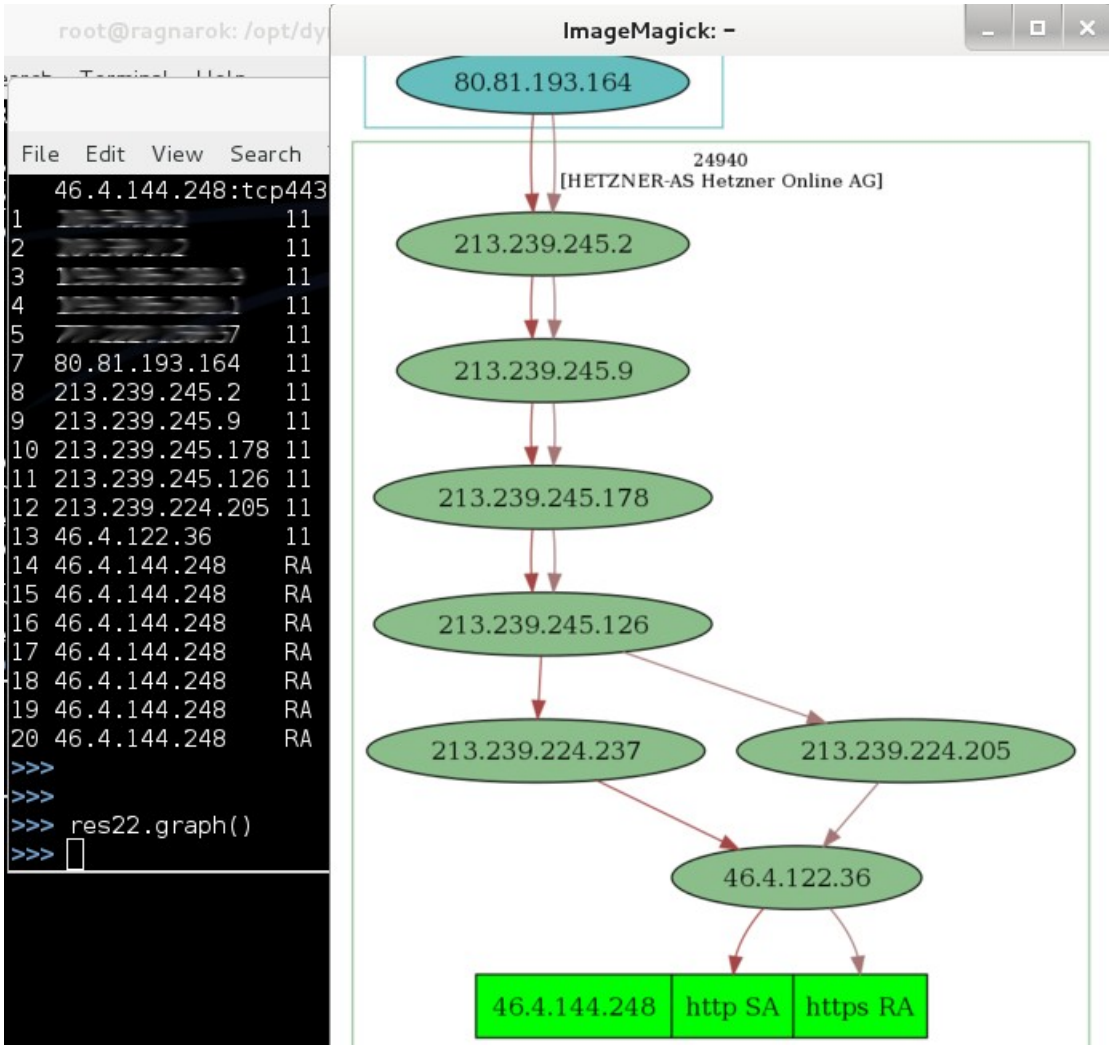
Второй способ заключается в построении 2D графиков, с последующим экспортом их в pdf-файл.

```
>>>
>>> packet=IP(dst="192.168.10.10")/TCP(dport=23,flags="S")
>>>
>>> packet.pdfdump("testlab3.pdf")
>>>
```

За это уже отвечает функция **pdfdump()**.
Результаты выглядят примерно так:

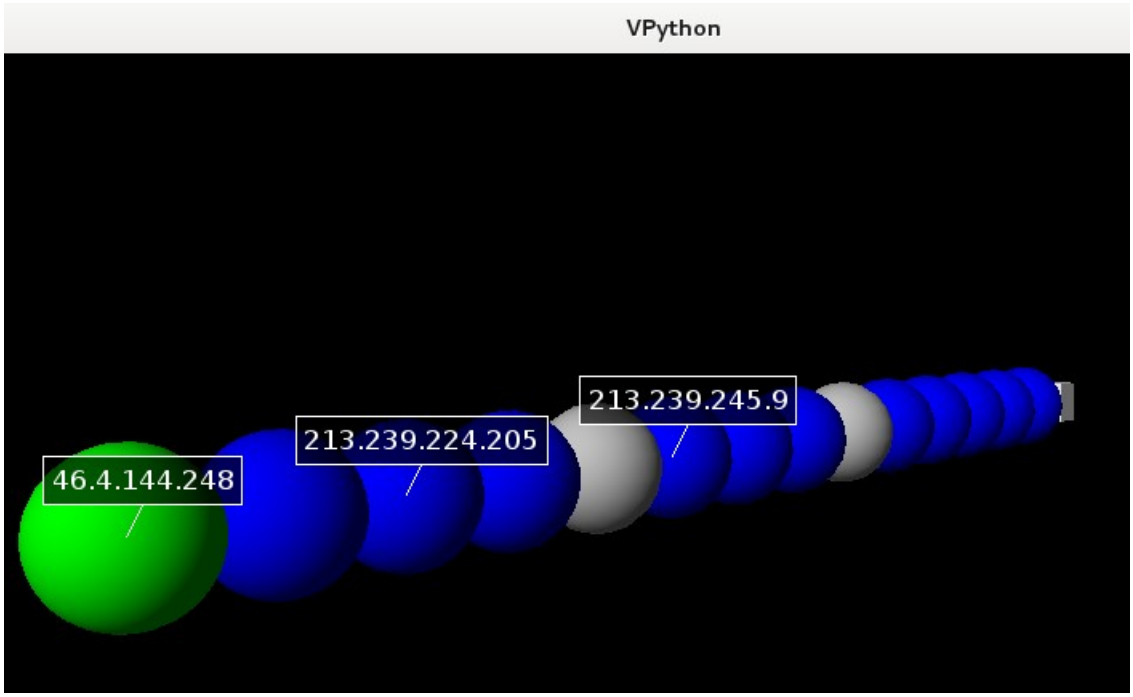


В данном случае уже вполне неплохо.
Кроме того, функция **graph()** опять откроет окно ImageMagick, но уже с детальной прорисовкой:



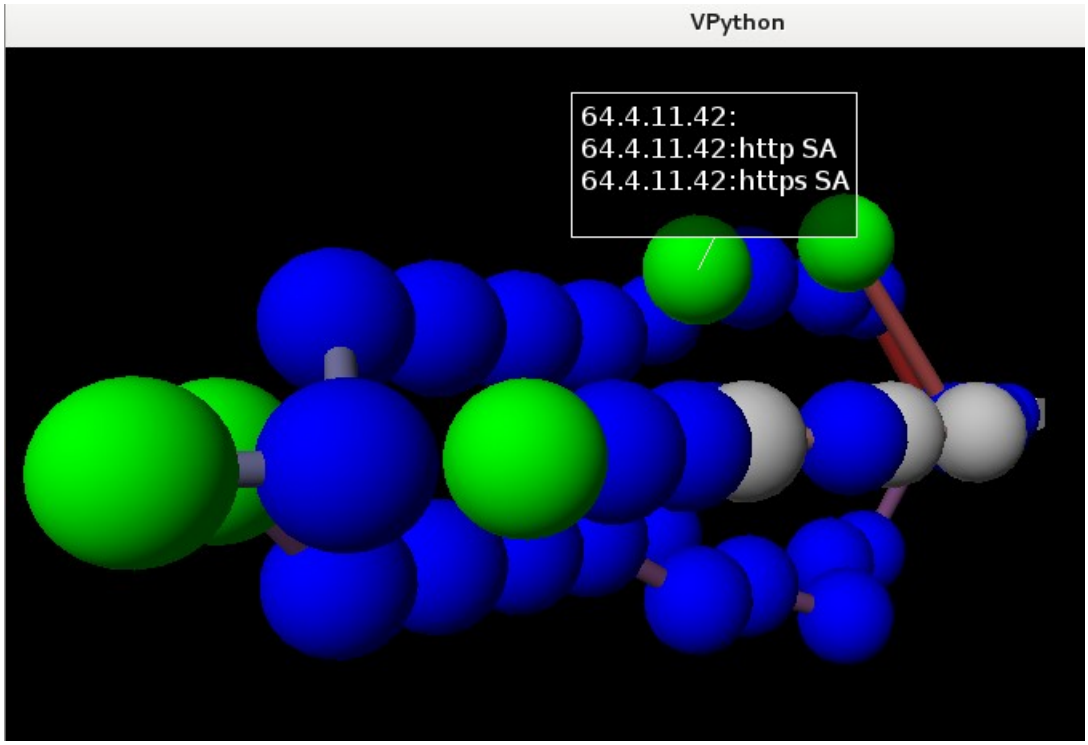
Здесь мы видим результат трассировки, с подробным отображением автономных систем и прочей визуализацией.

И, завершая тему визуализации, а вместе с ней и статью, посмотрим на 3D отображения трассы.
Для этого потребуется VPython и команда **trace3D()**.



Здесь отображена трасса из предыдущего графика.

Но иногда бывают и такие варианты:



В этом примере была проведена трассировка сразу нескольких целей, с

использованием нескольких (80, 443) tcp портов.
Левый клик на любом объекте приведет к появлению IP-адреса над ним, а левый клик с зажатой клавишей CTRL – к отображению более подробной информации — портам, как в этом случае.