Implementing Protocols

> A step-by-step guide

👨‍💻 Guillaume Valadon
✉️ guillaume@valadon.net

Scapy Concepts

> abstracted fields encoding and decoding

- simplified protocols implementations
    - they are called **layers**
- a layer is an object
    - it inherits from the `Packet` object
- each layer is a list of **fields**

Navigating Layers

```
>>> p = Ether() / IP () / UDP()

>>> p[IP]
<IP  frag=0 proto=udp |<UDP  |>>

>>> p[IP].underlayer
<Ether  type=IPv4 |<IP  frag=0 proto=udp |<UDP  |>>>

>>> p[IP].payload
<UDP  |>

>>> p.payload.payload
<UDP  |>
```

- `p[IP]` : access a layer by name
- `p[IP].underlayer` : access its ancestor
- `p[IP].payload` : access its predecessor

Fields

- many types (see `scapy/fields.py`)
  - `ByteField`, `IPField`, `StrField`...
- variants
  - `X*`: display the field in hexadecimal
  - `LE*`: Little Endian integer
  - `Signed*`: signed integer
  - `*Enum*`: use a dictionary to name values

> implementing a layer consists in combining fields

Frequent Patterns

A Simple Layer

```
class UDP(Packet):
    name = "UDP"
    fields_desc = [ShortEnumField("sport", 53,
                                  UDP_SERVICES),
                   ShortEnumField("dport", 53,
                                  UDP_SERVICES),
                   ShortField("len", None),
                   XShortField("chksum", None), ]
```

> see `scapy/layers/inet.py`

- mandatory elements
  - `Packet` inheritance
  - a `name`
  - a `fields_desc` containing a list of fields
- three `ShortField` variants
- default values

Computing Default Values

```
def post_build(self, packet, payload):
    packet += payload
    tmp_len = self.len
    if tmp_len is None:
        tmp_len = len(packet)
        packet = packet[:4]
        packet += struct.pack("!H", tmp_len)
        packet += packet[6:]
    return p
```

> computing the `chksum` value is similar

- `None` is replaced by 0
- `post_build()` method alters the `bytes` values

Matching Answers

```
def hashret(self):
    return self.payload.hashret()

def answers(self, other):
    if not isinstance(other, UDP):
        return 0
    return self.payload.answers(other.payload)
```

> see `UDP` in `scapy/layers/inet.py`

- `hashret()` method
  - constructs a hash common to the query & the answer
- `answers()` method
  - returns `True` if `other` is an answer to `self`

Finding The Next Layer

```
bind_layers(Ether, MACControl, type=MAC_CONTROL_ETHER_TYPE)


def guess_payload_class(self, payload):

    try:
        op_code = (orb(payload[0]) << 8) + orb(payload[1])
        return MAC_CTRL_CLASSES[op_code]
    except KeyError:
        pass

    return Packet.guess_payload_class(self, payload)
```

> see `MACControl` in `scapy/contrib/mpls.py`

- `bind_layers()` function
  - instructs Scapy to tie layers together
- `guess_payload_class()` method
  - returns the class of the payload

Packets As A Single Field

```
class Dot11EltRSN(Dot11Elt):
    name = "802.11 RSN information"
    match_subclass = True
    fields_desc = [
        ByteEnumField("ID", 48, _dot11_id_enum),
        ByteField("len", None),
        LEShortField("version", 1),
        PacketField("group_cipher_suite", RSNCipherSuite(), RSNCipherSuite),
        # --- 8< --- 8< --- 8< ---
```

> see `scapy/layers/dot11.py`

- `PacketField` decodes a field as a packet

Packet As A List Of Fields

```
class GTPEchoResponse(Packet):
    name = "GTP Echo Response"
    fields_desc = [PacketListField("IE_list", [], IE_Dispatcher)]
        # --- 8< --- 8< --- 8< ---
```

> see `scapy/layers/dot11.py`

- `PacketListField` decodes a field as a list of packets
- here `IE_Dispatcher` could be a class or a function that returns a class
  - it makes it possible to handle different types of packets

Notable Fields

Encoding Bits

```
class Dot1AH(Packet):
    name = "802_1AH"
    fields_desc = [BitField("prio", 0, 3),
                   BitField("dei", 0, 1),
                   BitField("nca", 0, 1),
                   BitField("res1", 0, 1),
                   BitField("res2", 0, 2),
                   ThreeBytesField("isid", 0)]
```

> see `scapy/layers/l2.py`

- `BitField` can read less than a byte
  - successive `BitField` must total 8 bits

One Field And Multiple Types

```
class PIMv2JoinPruneAddrsBase(_PIMGenericTlvBase):
    fields_desc = [
        ByteField("addr_family", 1),
        ByteField("encoding_type", 0),
        BitField("rsrvd", 0, 5),
        BitField("sparse", 0, 1),
        BitField("wildcard", 0, 1),
        BitField("rpt", 1, 1),
        ByteField("mask_len", 32),
        MultipleTypeField(
            [(IP6Field("src_ip", "::"),
              lambda pkt: pkt.addr_family == 2)],
            IPField("src_ip", "0.0.0.0")
        ),
    ]
```

> see `scapy/contrib/pim.py`

- `src_ip` could either be `IPField` or `IP6Field`
  - it depends on the `addr_family` field value

Check A Condition

```
class DNS(DNSCompressedPacket):
    name = "DNS"
    fields_desc = [
        ConditionalField(ShortField("length", None),
                         lambda p: isinstance(p.underlayer, TCP)),
        ShortField("id", 0),
        BitField("qr", 0, 1),
        # --- 8< --- 8< --- 8< ---
```

> see `scapy/layers/dns.py`

- the `length` field only exists if the condition is `True`
  - for DNS, it is only valid when TCP is used

Length & Value

```
class HBHOptUnknown(Packet):  # IPv6 Hop-By-Hop Option
    name = "Scapy6 Unknown Option"
    fields_desc = [_OTypeField("otype", 0x01, _hbhopts),
                   FieldLenField("optlen", None, length_of="optdata", fmt="B"),
                   StrLenField("optdata", "",
                               length_from=lambda pkt: pkt.optlen)]
```

> see `scapy/layers/inet6.py`

- `FieldLenField` encodes a count
  - `length_of` is used to compute it
- `StrLenField` stores a value
  - `length_from` is used to get it

Count & Value

```
class ReserveRelease(Packet):
    name = "Reserve / Release"
    fields_desc = [ByteEnumField("rcmd", 0, {0: "Read Reserve List",
                                             1: "Set Reserve List",
                                             2: "Force Set Reserve List"}),
                   FieldLenField("nb_mac", None, count_of="mac_addrs"),
                   FieldListField("mac_addrs", None, MACField("", ETHER_ANY),
                                  count_from=lambda pkt: pkt.nb_mac)]
```

> see `scapy/layers/inet6.py`

- `FieldLenField` encodes a count
  - `count_of` is used to retrieve it
- `FieldListField` stores a list of fields
  - `count_from` is used to get the number of fields

Allow Failures

```
class TCPerror(TCP):
    name = "TCP in ICMP"
    fields_desc = (
        TCP.fields_desc[:2] +
        # MayEnd after the 8 first octets.
        [MayEnd(TCP.fields_desc[2])] +
        TCP.fields_desc[3:]
    )

>> TCPerror(raw(TCP())[:8])
```

> see `scapy/contrib/aoe.py`

- it might be OK to receive fewer data than the full `Packet`
  - only the first fields are decoded

Advanced Patterns

Find A Better Layer

```
class IPv46(IP, IPv6):
    name = "IPv4/6"

    @classmethod
    def dispatch_hook(cls, _pkt=None, *_, **kargs):
        if _pkt:
            if orb(_pkt[0]) >> 4 == 6:
                return IPv6
        elif kargs.get("version") == 6:
            return IPv6
        return IP

conf.l2types.register(DLT_RAW, IPv46)
```

> see `scapy/layers/inet6.py`

- `dispatch_hook()` is used to dynamically change the layer
  - here, it allows choosing `IP` or `IPv6` while parsing Raw IP PCAP files

TCP Reassembly

```python
@classmethod
def tcp_reassemble(cls, data, metadata, session):
    length = struct.unpack("!I", data[4:8])[0] + 8
    if len(data) >= length:
        return DoIP(data)
    return None
```

> see `DoIP` in `scapy/contrib/automotive/doip.py`

- `tcp_reassemble()` allows to decode a layer when enough data is received
  - Scapy reconstructs the TCP session

Modify The Dissection Result

```
def post_dissect(self, s):
    self.decrypt()
```

> see `Dot11WEP` in `scapy/layers/dot11.py`

- `post_dissect()` is called when the layer is fully decoded
  - if the WEP key is known, the frame is decrypted

Discard Extra Bytes

```
def extract_padding(self, pkt):
    return "", pkt
```

> see `TFTP_Option` in `scapy/layers/tftp.py`

- extra bytes could be considered as padding
- here `extract_padding()` informs Scapy, that there is not padding
  - this is useful for `PacketListField`

Adding A New Field

Field States

- **i (internal)**
  - how Scapy manipulates the field
- **m (machine)**
  - how the field is sent over the network
- **h (human)**
  - how the field is displayed for humans

Field State Conversion Methods

- **i2h()**
  - nternal → human
- **i2m()**
  - internal → machine
- **m2i()**
  - machine → internal

> these methods convert field states for specific use cases

Alter A Field Behavior

```
class ThreeBytesField(Field[int, int]):
    def __init__(self, name, default):
        Field.__init__(self, name, default, "!I")

    def addfield(self, pkt, s, val):
        return s + struct.pack(self.fmt, self.i2m(pkt, val))[1:4]

    def getfield(self, pkt, s):
        return s[3:], self.m2i(pkt,
                               struct.unpack(self.fmt,
                                             b"\x00" + s[:3])[0])
```

> see `scapy/fields.py`

- `getfield()` decodes three bytes from `s`
  - it returns the remaining part of `s` and the decode value
- `addfield()` encodes three bytes

Questions?

Going Further

- https://github.com/secdev/scapy
- https://scapy.net
- https://scapy.readthedocs.io