

Scapy_Dojo_V_1

HUGO TROVAO

&

RUSHIKESH D. NANDEDKAR

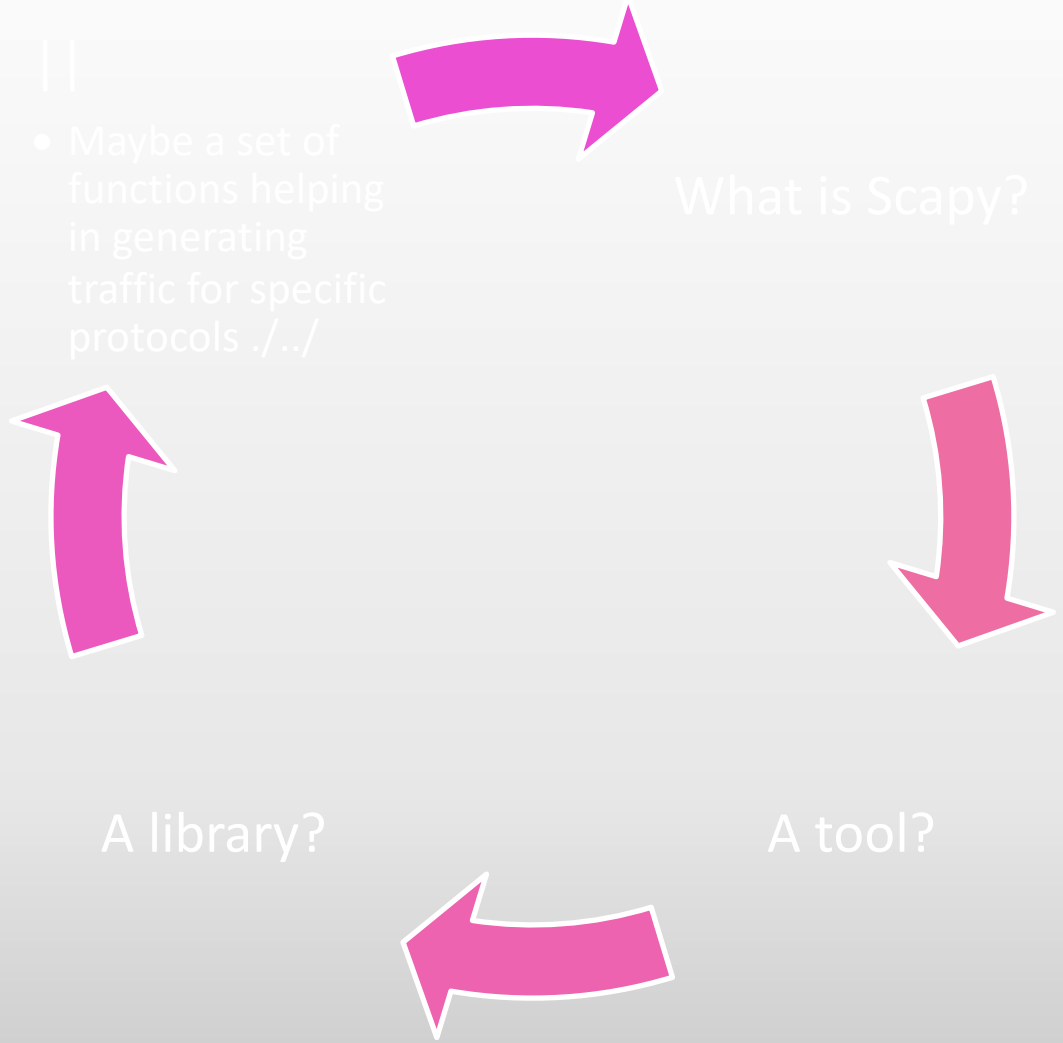
Introduction

What is Scapy?

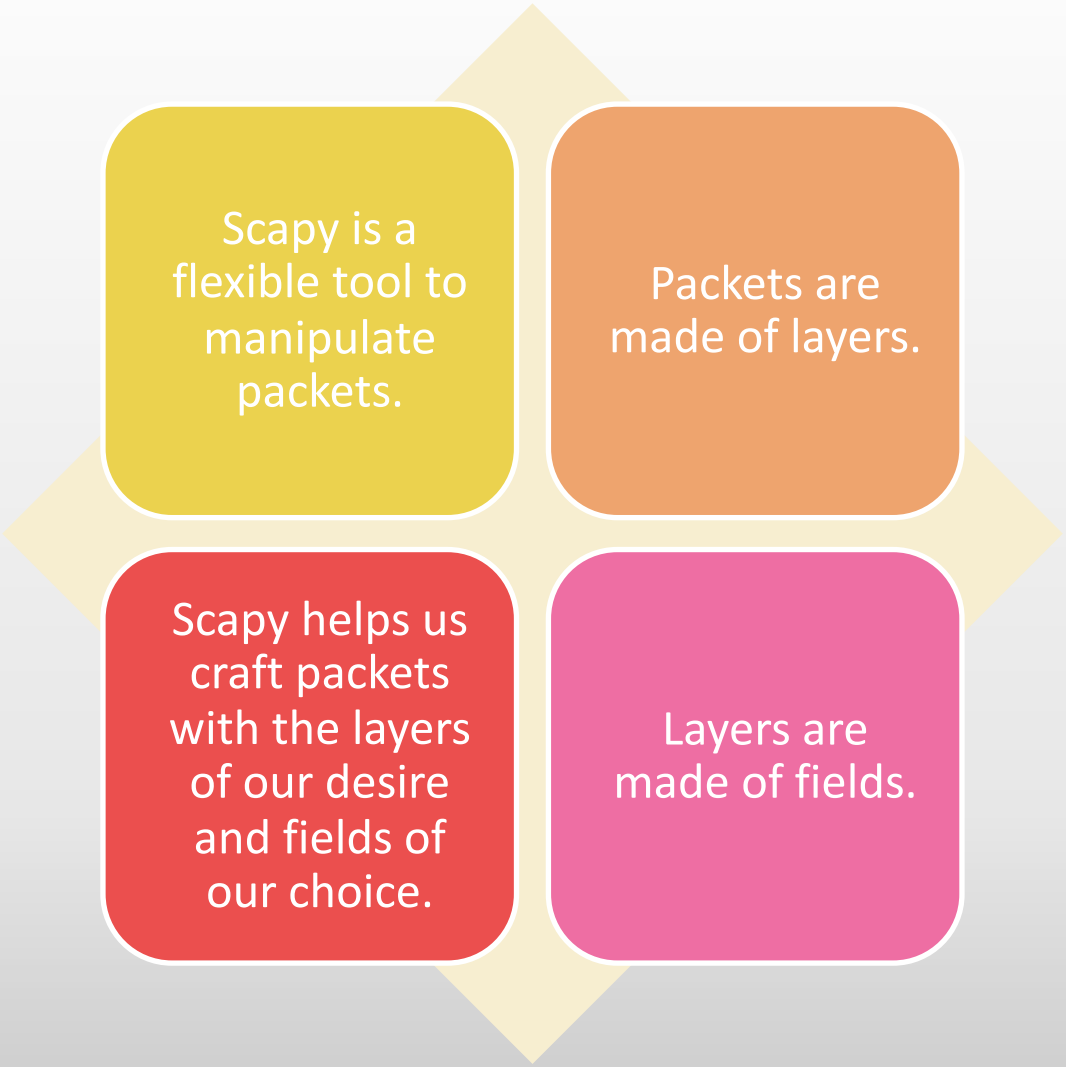
- Maybe a set of functions helping in generating traffic for specific protocols ./../

A tool?

A library?



Anatomy of scapy for our purpose



Layers and Fields



- What is a layer?
  - A unit of packet
  - Ex: Ether()/IP()/TCP()/Raw(payload)
- What is a field
  - Layers are composed by logical parts (fields)
  - Ex: IP(src="10.0.0.1")
- "Automatic" fields
  - Some layers can have fields that are computed in context
  - Ex: TCP(chksum=0x0bad) # this generates an invalid packet

Scapy basics

Packet building in scapy

- Packets are constructed with parameterized layers.
  - Ex: IP(dst="10.0.0.1")/TCP(dport=8080)
- Field layers can be accessed and changed to update an existing layer.
  - Ex: pkt[IP].dst = "10.0.0.2”

Displaying packets

- There are 2 modes of displaying packets.
  - - before "automatic" fields computation (pkt.show())
  - - after "automatic" fields computation (pkt.show2()) [this is what goes on wire!]
- hexdump(pkt) will print packet in hex.
- pkt.summary() will show a short summary of the packet.
- ls(IP, verbose=True) # list protocol fields.
- bytes(pkt) return pkt in wire bytes representation.

Sending packets

- Scapy allows sending packets and frames.
- For data link layer, the prime keyword to send frame is ”sendp”.
  - sendp(layer2_pkt)
- For network layer, the keyword to send packet is ”send”
  - send(layer3_pkt)

Receiving packets

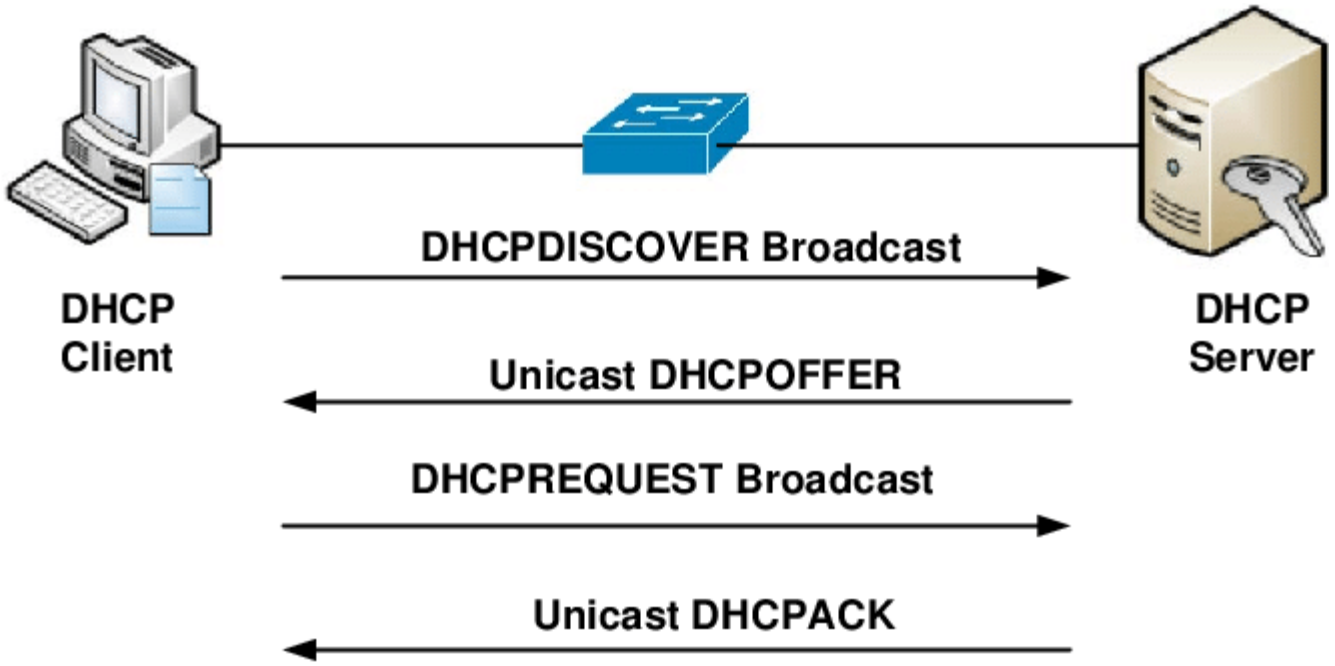
- Packets/frames can be sniffed from wire/air.
  - sniff(count=0, store=True, offline=None, prn=None, lfilter=None, L2socket=None, timeout=None, opened_socket=None, stop_filter=None, iface=None, started_callback=None, *arg, **karg)

...

- Packets can be received according to a response of a sent packet
- srp(pkt) will send pkt and return the responses packet at layer 2
- srp1(pkt) will send pkt and return one response packet at layer 2
- sr(pkt) will send pkt and return the responses packet at layer 3
- sr1(pkt) will send pkt and return one response packet at layer 3
- Returns (ans, unans) where ans is (sent, answ)
- rdpcap(filename, count=-1) reads a pcap file and returns a list of packets

DHCP Servers

DHCP protocol



A different approach - Server

- Sniff dhcp traffic with scapy
- “reverse-engineer” reply packets and use this as your “template” packets
- Modify them at your needs (dns server/gateway address/...)
- Reply to DHCP request packets in the network with your modified packets

Video

A different approach – Client

- Sniff dhcp traffic with scapy
- “reverse-engineer” request packets and use this as your “template” packets
- Modify them at your needs (hw address)
- Reply to DHCP request packets in the network with your modified packets

Video

DNS Servers

DNS/mDNS query viewer

- Sniff some DNS/mDNS traffic
- Dissect and understand the protocol
- Implement a viewer for DNS and/or mDNS queries
  - DNS: 53/udp
  - MDNS: 5353/udp
    - Uses multicast traffic destination IP
- Protocols are similar

Video

DNS/mDNS query responder

- Check **{dns,mdns}-responder.py** for DNS and/or mDNS query responders
  - DNS: 53/udp
  - MDNS: 5353/udp
    - Uses multicast traffic destination IP
- Reply to DNS/mDNS traffic on your network

Video

AJP13

Apache JServ Protocol version 1.3

- Communication protocol for web server/servlet-containers
- Packet oriented protocol
- Binary format to increase performance
- TCP communication with persistent connections
  - Default port: tcp/8009

Making an AJP3 layer

https://tomcat.apache.org/connectors-doc-archive/jk2/common/AJPv13.html

Packet Format (Server->Container)					
Byte	0	1	2	3	4...(n+3)
Contents	0x12	0x34	Data Length (n)		Data

Packet Format (Container->Server)					
Byte	0	1	2	3	4...(n+3)
Contents	A	B	Data Length (n)		Data

Forward Request Packet

For messages from the server to the container of type "Forward Request":

```
AJP13_FORWARD_REQUEST :=
    prefix_code      (byte) 0x02 = JK_AJP13_FORWARD_REQUEST
    method           (byte)
    protocol         (string)
    req_uri          (string)
    remote_addr      (string)
    remote_host      (string)
    server_name      (string)
    server_port      (integer)
    is_ssl           (boolean)
    num_headers      (integer)
    request_headers *(req_header_name req_header_value)
    attributes      *(attribut_name attribute_value)
    request_terminator (byte) OxFF
```

The request_headers have the following structure:

```
req_header_name :=
    sc_req_header_name | (string)  [see below for how this is parsed]

sc_req_header_name := 0xA0xx (integer)

req_header_value := (string)
```

The attributes are optional and have the following structure:

```
attribute_name := sc_a_name | (sc_a_req_attribute string)

attribute_value := (string)
```

Not that the all-important header is "content-length', because it determines whether or not the container looks for another packet immediately.

Message Types

The servlet container can send the following types of messages to the web server:

Code	Type of Packet	Meaning
3	Send Body Chunk	Send a chunk of the body from the servlet container to the web server (and presumably, onto the browser).
4	Send Headers	Send the response headers from the servlet container to the web server (and presumably, onto the browser).
5	End Response	Marks the end of the response (and thus the request-handling cycle).
6	Get Body Chunk	Get further data from the request if it hasn't all been transferred yet.
9	CPong Reply	The reply to a CPing request

The web server can send the following messages to the servlet container:

Code	Type of Packet	Meaning
2	Forward Request	Begin the request-processing cycle with the following data
7	Shutdown	The web server asks the container to shut itself down.
8	Ping	The web server asks the container to take control (secure login phase).
10	CPing	The web server asks the container to respond quickly with a CPong.
none	Data	Size (2 bytes) and corresponding body data.

HTTP method and headers

The structure of request_headers is the following: First, the number of headers num_headers is encoded. Then, a series of header name req_header_name / value req_header_value pairs follows. Common header names are encoded as integers, to save space. If the header name is not in the list of basic headers, it is encoded normally (as a string, with prefixed length). The list of common headers sc_req_header_nameand their codes is as follows (all are case-sensitive):

Name	Code value	Code name
accept	0xA001	SC_REQ_ACCEPT
accept-charset	0xA002	SC_REQ_ACCEPT_CHARSET
accept-encoding	0xA003	SC_REQ_ACCEPT_ENCODING
accept-language	0xA004	SC_REQ_ACCEPT_LANGUAGE
authorization	0xA005	SC_REQ_AUTHORIZATION
connection	0xA006	SC_REQ_CONNECTION
content-type	0xA007	SC_REQ_CONTENT_TYPE
content-length	0xA008	SC_REQ_CONTENT_LENGTH
cookie	0xA009	SC_REQ_COOKIE
cookie2	0xA00A	SC_REQ_COOKIE2
host	0xA00B	SC_REQ_HOST
pragma	0xA00C	SC_REQ_PRAGMA
referer	0xA00D	SC_REQ_REFERER
user-agent	0xA00E	SC_REQ_USER_AGENT

The HTTP method, encoded as a single byte:

Command Name	Code
OPTIONS	1
GET	2
HEAD	3
POST	4
PUT	5
DELETE	6
TRACE	7
PROPFIND	8
PROPPATCH	9
MKCOL	10
COPY	11
MOVE	12
LOCK	13
UNLOCK	14
ACL	15
REPORT	16
VERSION-CONTROL	17
CHECKIN	18
CHECKOUT	19
UNCHECKOUT	20
SEARCH	21
MKWORKSPACE	22
UPDATE	23
LABEL	24
MERGE	25
BASELINE_CONTROL	26
MKACTIVITY	27

Attributes

The attributes prefixed with a ? (e.g. ?context) are all optional. For each, there is a single byte code to indicate the type of attribute, and then a string to give its value. They can be sent in any order (thogh the C code always sends them in the order listed below). A special terminating code is sent to signal the end of the list of optional attributes. The list of byte codes is:

Information	Code Value	Note
?context	0x01	Not currently implemented
?servlet_path	0x02	Not currently implemented
?remote_user	0x03	
?auth_type	0x04	
?query_string	0x05	
?jvm_route	0x06	
?ssl_cert	0x07	
?ssl_cipher	0x08	
?ssl_session	0x09	
?req_attribute	0x0A	Name (the name of the attribut follows)
?ssl_key_size	0x0B	
are_done	0xFF	request_terminator

Response Packets

For messages which the container can send back to the server.

```
AJP13_SEND_BODY_CHUNK :=
  prefix_code   3
  chunk_length  (integer)
  chunk        *(byte)


AJP13_SEND_HEADERS :=
  prefix_code       4
  http_status_code  (integer)
  http_status_msg   (string)
  num_headers       (integer)
  response_headers *(res_header_name header_value)

res_header_name :=
    sc_res_header_name | (string)   [see below for how this is parsed]

sc_res_header_name := 0xA0 (byte)

header_value := (string)

AJP13_END_RESPONSE :=
  prefix_code       5
  reuse             (boolean)


AJP13_GET_BODY_CHUNK :=
  prefix_code       6
  requested_length  (integer)
```

Response headers

The status code and message are the usual HTTP things (e.g. "200" and "OK"). The response header names are encoded the same way the request header names are. See for details about how the the codes are distinguished from the strings. The codes for common headers are:

Name	Code value
Content-Type	0xA001
Content-Language	0xA002
Content-Length	0xA003
Date	0xA004
Last-Modified	0xA005
Location	0xA006
Set-Cookie	0xA007
Set-Cookie2	0xA008
Servlet-Engine	0xA009
Status	0xA00A
WWW-Authenticate	0xA00B

After the code or the string header name, the header value is immediately encoded.

Get body chunk

- The container asks for more data from the request (If the body was too large to fit in the first packet sent over or when the request is chuncked). The server will send a body packet back with an amount of data which is the minimum of the request_length, the maximum send body size (8186 (8 Kbytes - 6)), and the number of bytes actually left to send from the request body.
- If there is no more data in the body (i.e. the servlet container is trying to read past the end of the body), the server will send back an "empty" packet, which is a body packet with a payload length of 0. (0x12,0x34,0x00,0x00)

AJP13 - Layers

Layers

- Layers are made describing fields. Each layer has 3 different representations:
  - Human (h)
  - Internal (I)
  - Machine (m)
- The functions to implement in a layer to customize the construction of the packet representations are: i2m, m2i, h2i, i2h, ...

Fields

- Fields are made describing its representations on wire and for internal use.
- **addfield(pkt, s, val)** and **getfield(pkt, s)** are called for adding or getting a field from a layer

Processing fields

```
def post_build(self, pkt, pay):
    """
    DEV: called right after the current layer is build.

    :param str pkt: the current packet (build by self_buil function)
    :param str pay: the packet payload (build by do_build_payload function)
    :return: a string of the packet with the payload
    """
    return pkt + pay


def dissection_done(self, pkt):
    """DEV: will be called after a dissection is completed"""
    self.post_dissection(pkt)
    self.payload.dissection_done(pkt)


def post_dissection(self, pkt):
    """DEV: is called after the dissection of the whole packet"""
    pass

def post_dissect(self, s):
    """DEV: is called right after the current layer has been dissected"""
    return s


def pre_dissect(self, s):
    """DEV: is called right before the current layer is dissected"""
    return s
```

What to implement

- AJP13Header
- AJP13ForwardRequest
- Pascal String Field
- Data length calculation

Video

AJP13 - Fuzzing

Rand generators

- Fuzzing fields with Rand...() generators is easy

- RandInt(), RandShort(), RandIP() RandString(), ... on fields we want to fuzz

1) loop sending a packet
    1) Every time the packet is built it call the generators and create new random values
    2) don’t forget to save packets for posterior analysis [after anomaly]
2) observe logs in fuzzed application

What to implement – AJP13 as Web Server

- Generate a fuzzing template with Rand generators with AJP13ForwardRequests
- Fuzz ajp13 service on Tomcat on the VM
  - */opt*/apache-tomcat-.../
    - bin/catalina.sh start
- Observe logs
  - logs/

Video

What to implement – AJP13 as Servlet Container

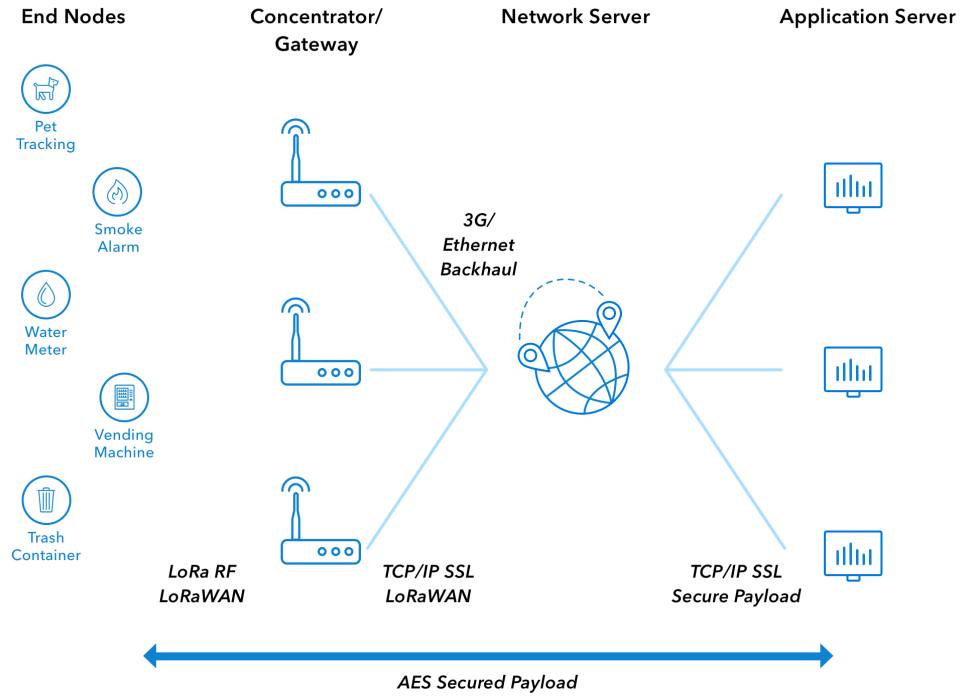
- Check **test_as_server.py** example, and customize it
  - Launch it, it will listen in 8009/tcp and reply to web server requests in AJP protocol
- Launch Apache httpd
  - Configure httpd with the lines from the config in the sources directory
    - Configure modules and insert a ProxyPass directive for ajp13
  - bin/httpd -X
- Start making requests to Apache AJP endpoint
  - while [[1]] ; do curl localhost/ajp; done
- Observe logs

Video

LoRaWAN

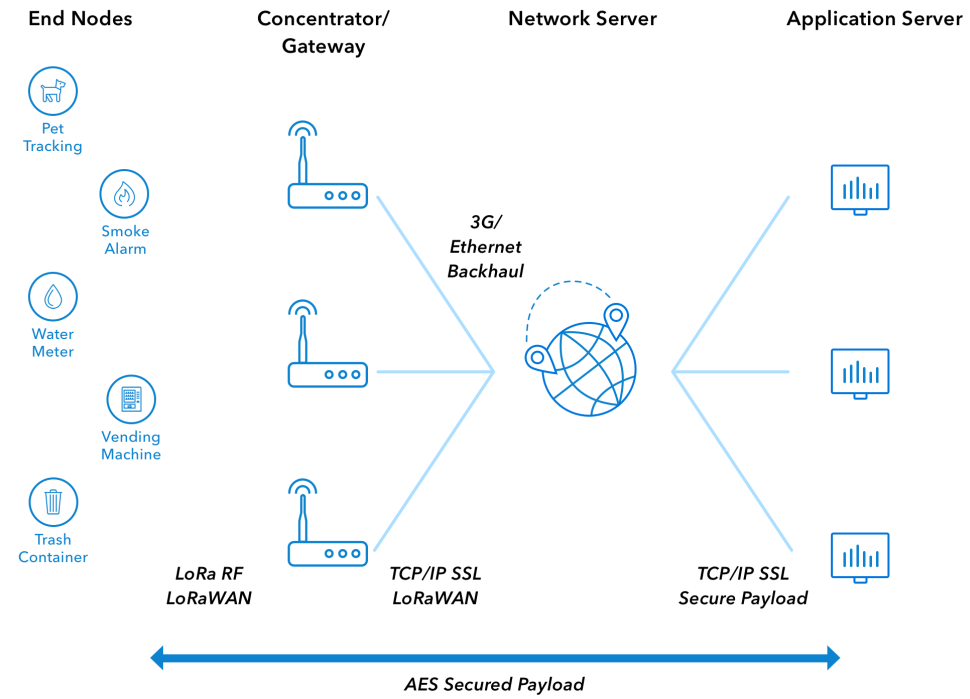
LoRaWAN

- LoRaWAN is protocol on top of LoRa modulation
- Used in IoT
- Used in star topology
- Essential components
  - Application server
  - Gateway
  - Node
  -



LoRaWAN Our setup

- End Nodes use LoRa modulation to talk to Gateway
- Gateway use 1700/udp port to talk to application server
  - Semtech protocol
- LoRaWAN protocol is used for End Nodes to talk to Application Server



LoRaWAN Our setup

- LoRaWAN payloads are encrypted with AES
- End Nodes can join the network with:
  - ABP (pre shared keys)
  - OTAA (keys are exchanged in JoinRequests)
- The communication between the Node and Application server is encrypted. Node talks to Gateway through LoRa and the Gateway uses the Internet to reach the application server.
- Gateway talks to the Application server using messages with gateway and reception information and the received message from the node. In our case its SemtechUDP protocol and LoRaWAN 1.0.1.

LoRaWAN Our setup – Protocol

- LoRaWAN Class-A, Version 1.0.1
- https://lora-alliance.org/resource-hub/lorawanr-specification-v101

Protocol Layers

Radio PHY layer:



Figure 5: Radio PHY structure (CRC* is only available on uplink messages)

PHYPayload:



Figure 6: PHY payload structure

MACPayload:



Figure 7: MAC payload structure

FHDR:



Figure 8: Frame header structure

Figure 9: LoRa message format elements

Protocol Layers

4.1 MAC Layer (PHYPayload)

Size (bytes)	1	1..*M*	4
PHYPayload	MHDR	MACPayload	MIC

4.2 MAC Header (MHDR field)

Bit#	7..5	4..2	1..0
MHDR bits	MType	RFU	Major

4.2.2 Major version of data message (Major bit field)

Major bits	Description
00	LoRaWAN R1
01..11	RFU

Table 2: Major list

Protocol Layers

4.2.1 Message type (MType bit field)

The LoRaWAN distinguishes between six different MAC message types: **join request**, **join accept**, **unconfirmed data up/down**, and **confirmed data up/down**.

MType	Description
000	Join Request
001	Join Accept
010	Unconfirmed Data Up
011	Unconfirmed Data Down
100	Confirmed Data Up
101	Confirmed Data Down
110	RFU
111	Proprietary

Table 1: MAC message types

4.3.1 Frame header (FHDR)

The **FHDR** contains the short device address of the end-device (**DevAddr**), a frame control octet (**FCtrl**), a 2-octets frame counter (**FCnt**), and up to 15 octets of frame options (**FOpts**) used to transport MAC commands.

Size (bytes)	4	1	2	0..15
FHDR	DevAddr	FCtrl	FCnt	FOpts

For downlink frames the FCtrl content of the frame header is:

Bit#	7	6	5	4	[3..0]
FCtrl bits	ADR	RFU	ACK	FPending	FOptsLen

For uplink frames the FCtrl content of the frame header is:

Bit#	7	6	5	4	[3..0]
FCtrl bits	ADR	ADRACKReq	ACK	RFU	FOptsLen

Protocol Layers

4.3.2 Port field (FPort)

If the frame payload field is not empty, the port field must be present. If present, an **FPort** value of 0 indicates that the **FRMPayload** contains MAC commands only; see Chapter 5.4 for a list of valid MAC commands. **FPort** values 1..223 (0x01..0xDF) are application-specific. **FPort** values 224..255 (0xE0..0xFF) are reserved for future standardized application extensions.

Size (bytes)	7..22	0..1	0..*N*
MACPayload	FHDR	FPort	FRMPayload

N is the number of octets of the application payload. The valid range for *N* is region specific and is defined in Section 7

N should be equal or smaller than:

$$N \leq M - 1 - (\text{length of } \textbf{FHDR} \text{ in octets})$$

where *M* is the maximum MAC payload length.

SemtechUDP

- UDP based communication on port 1700
- Based on binary header and json payloads
  - we'll be interested in the data of json payload where the LoRaWAN protocol is being exchanged
- https://github.com/Lora-net/packet_forwarder/blob/master/PROTOCOL.TXT

In practice

- Use the vm and:
  - Unpack IQ samples zip to /tmp
  - Build and install gr-lora
  - Launch scapy-radio and use LoRa radio module
  - gnuradio.sniffradio(radio=”LoRa”)
  - Dissect sniffed packets

Video