

利用libpcap库解析802.11帧

一、相关原理

1.1 libpcap

libpcap (Packet Capture Library) 即数据包捕获函数库，是Unix/Linux平台下的网络数据包捕获函数库。它是一个独立于系统的用户层包捕获的API接口，为底层网络监测提供了一个可移植的框架。著名的软件TCPDUMP就是在libpcap的基础上开发而成的

libpcap可以实现以下功能：

- 数据包捕获：捕获流经网卡的原始数据包
- 自定义数据包发送：任何构造格式的原始数据包
- 流量采集与统计：网络采集的中流量信息
- 规则过滤：提供自带规则过滤功能，按需要选择过滤规则

关于libpcap库的基本使用，可以参照[Writing a Basic Packet Capture Engine](http://www.tcpdump.org/writing-a-basic-packet-capture-engine/)

1.2 IEEE 802.11

802.11是国际电工电子工程学会（IEEE）为无线局域网制定的标准。目前在802.11的基础上开发出了802.11a、802.11b、802.11g、802.11n、802.11ac。并且为了保证802.11更加安全的工作，开发出了802.1x、802.11i等协议

使用802.11协议的设备连接到无线网络需要三步

1. 扫描（获得网络信息）
2. 认证（确认身份）
3. 连接（确定连接、交换数据）

802.11的帧格式类似于以太网，但是更加复杂。并且为了解决无线网络的缺陷，需要添加额外的功能。所有802.11的帧分为三类：

- 管理帧
- 控制帧
- 数据帧

帧格式



图 3-9：一般的 802.11 MAC 帧



图 3-10：Frame control 位

由于本实验需要解析出进行通信的硬件MAC地址，因此需要了解To DS、From DS标记位和4个地址的关系如下：

function	To DS	From DS	Address1	Address2	Address3	Address4
IBSS	0	0	DA	SA	BSSID	Not use
TO AP	1	0	BSSID	SA	DA	Not use
FROM AP	0	1	DA	BSSID	SA	Not use
WDS	1	1	RA	TA	DA	SA

IBSS（Independent BasicService Set）独立基本服务集

BSSID（Basic Service SetIdentifier）基本服务集标识符（为AP的MAC地址）

要在同一个区域划分不同的局域网络，可以为工作站指定所要使用的基本服务集。在基础网络里，BSSID即是基站无线界面所使用的MAC地址。而对等网络(也就是所谓的p2p)则会产生一个随机的BSSID，并Universal/Localbit设定为1，以防止与其他官方指定的MAC地址产生冲突。

DA（Destination Address）目的地址

SA（Sender Address）源地址

RA（Receiver Address）接收端地址

TA（Transmission Address）发送端地址

WDS（WirelessDistribution System）无线分布式系统

WDS（wireless distribution system），无线分布系统：构建在FHSS或DSSS底下，可让基地台与基地台之间得以沟通，比较不同的是有WDS的功能是可当做无线网络的中继器，且可多台基地台对一台。

欲了解更多关于802.11帧格式的相关信息，请参照[IEEE 802.11的帧格式介绍](#)

1.3 Radiotap

本文介绍的实现方式是工作Ubuntu20.04(Linux)下的，通过wireshark或tcpdump抓到的无线网卡数据包，每一数据帧前面都有一个叫 **radiotap** 的协议头，它包含了信号强度、噪声强度、信道、时间戳等信息。**radiotap** 比传统的 **Prism** 或 **AVS** 头更有灵活性，成为 **ieee802.11** 事实上的标准。支持 **radiotap** 的系统较多，如 **Linux**、**FreeBSD**、**NetBSD**、**OpenBSD**，还有 **windows** (需使用 **AirPcap**)

协议头部结构

```
struct ieee80211_radiotap_header {
    u_int8_t      it_version;    /* set to 0 */
    u_int8_t      it_pad;
    u_int16_t     it_len;        /* entire length */
    u_int32_t     it_present;    /* fields present */
} __attribute__((packed));
```

一般情况整个头部共8byte 32bit8(versiont) + 8(pad) + 16(len) = 32(present)

1. 当前版本version字段始终为0
2. pad字段是为了补齐四个字节而置0的字段因此radiotap开头均为\x00\x00
3. len为整个radiotap数据层的长度不需要解析时方便直接跳过
4. present为radiotap协议数据的位掩码某位为1时表示这个位代表的存在存放在头部后面。

比如bit5(下标)表示后面存在信号强度数据bit31表示还有另一个present字段存在。

因为位掩码的存在raditap协议的数据是不定长的也让radiotap变得很灵活当出现新的字段时不会破坏现有的解析器。

当出现了不能理解的radiotap数据可以通过len直接跳过继续解析上层数据。

5. 因为present字段可能存在多个所以说上面的头部长可能会变长但要注意8byte对齐以0补位比如WN722N抓到的

二、实现步骤

1. 查看机器上无线网卡的状态

打开终端输入以下命令：

```
iwconfig
```

可以得到

```
lo          no wireless extensions.

enp0s25     no wireless extensions.

wlp4s0      IEEE 802.11  ESSID:"HBTSG"
            Mode:Managed  Frequency:5.3 GHz  Access Point: 90:E7:10:32:0F:91
            Bit Rate=144.4 Mb/s   Tx-Power=22 dBm
            Retry short limit:7   RTS thr:off   Fragment thr:off
            Power Management:on
            Link Quality=62/70   Signal level=-48 dBm
            Rx invalid nwid:0   Rx invalid crypt:0   Rx invalid frag:0
            Tx excessive retries:0   Invalid misc:102   Missed beacon:0
```

可以看到我的设备上有三个网卡，但只有wlp4s0可以接受到无线信号，其所工作的协议为802.11但是可以看到其所处的Mode为Managed，此时尝试用tcpdump或wireshark抓包会发现抓到的包都是以太帧的格式

这是因为wifi driver会自动把wireless frame转成ethernet frame后再给kernel，这样kernel里面的protocol stack会比较处理好

因此我们需要将该网卡的Mode更改为Monitor（监听模式）

但是若我们直接尝试去修改本机上唯一无线网卡（上述的wlp4s0）可能需要关闭网卡再打开，可能会导致本机无线网卡无法正常工作，因此一般我们不直接使用它作为mnitor模式，而是建立一个别名：

```
iw wlp4s0(此处需要更换成实验机器上无线网卡的标识符) interface add mon0 type monitor
ifconfig mon0 up
```

2. 利用tcpdump或wireshark对无线数据包进行抓取

若利用tcpdump命令：

```
tcpdump -i mon0 -w foo.cap
# 将捕捉到的包信息保存到foo.cap文件中
```

命令运行一段时间后即可Ctrl-C打断运行

如果利用wireshark抓包需要将抓到的包保存为.cap文件格式

3. 编写.c源文件

```
#include <stdio.h>
#include <signal.h>
#include <pcap.h>
#include <stdlib.h>
#include <string.h>
#include <getopt.h>

void pcap_callback(u_char* userarg, const struct pcap_pkthdr* handle, const
u_char* packet) {
    printf("-----\n");
    printf("-----802.11-----\n");
    // radiotap 头部数据结构
    struct radiotap_header {
        uint8_t it_rev;
        uint8_t it_pad;
        uint16_t it_len;
    };
    uint8_t offset = 0;
    struct radiotap_header* rtaphdr;
    rtaphdr = (struct radiotap_header *) packet;
    offset = rtaphdr->it_len;
    // 读取radiotap后跳过头部
    printf("Radiotap Header\n");
    printf("\tHeader revision: %d\n", rtaphdr->it_rev);
    printf("\tHeader pad: %d\n", rtaphdr->it_pad);
    printf("\tHeader Length: %d\n", rtaphdr->it_len);

    char* type;

    uint8_t toDS_fromDS_flag;

    u_char* DA, *SA, *RA, *TA, *BSSID, *add1, *add2, *add3, *add4;

    // 获取To DS和From DS标志位
    toDS_fromDS_flag = packet[offset+1] & 0x03;

    // 根据标记字段读取802.11包所属的type, 即帧类型
    switch (packet[offset]) {
        case 0x00:
            type = "Association request";
            break;
        case 0x10:
            type = "Association response";
            break;
        case 0x20:
            type = "Reassociation request";
            break;
        case 0x30:
            type = "Reassociation response";
            break;
        case 0x40:
            type = "Probe request";
            break;
        case 0x50:
            type = "Probe response";
```

```
        break;
    case 0x60:
        type = "Timing advertisement";
        break;
    case 0x80:
        type = "Beacon";
        break;
    case 0x90:
        type = "ATIM";
        break;
    case 0xA0:
        type = "Disassociation";
        break;
    case 0xB0:
        type = "Authentication";
        break;
    case 0xC0:
        type = "Deauthentication";
        break;
    case 0xD0:
        type = "Action";
        break;
    case 0xE0:
        type = "Action No Ack";
        break;
    case 0x74:
        type = "Control wrapper";
        break;
    case 0x84:
        type = "Block Ack Request";
        break;
    case 0x94:
        type = "Block Ack";
        break;
    case 0xA4:
        type = "PS Poll";
        break;
    case 0xB4:
        type = "RTS";
        break;
    case 0xC4:
        type = "Clear to send";
        break;
    case 0xD4:
        type = "ACK";
        break;
    case 0xE4:
        type = "CF-End";
        break;
    case 0xF4:
        type = "CF-End+CF-Ack";
        break;
    case 0x08:
        type = "Data";
        break;
    case 0x18:
        type = "Data+CF-Ack";
        break;
```

```

        case 0x28:
            type = "Data+CF-Poll";
            break;
        case 0x38:
            type = "Data+CF-AC";
            break;
        case 0x48:
            type = "Null function";
            break;
        case 0x58:
            type = "CF-ACK";
            break;
        case 0x68:
            type = "CF-Poll";
            break;
        case 0x78:
            type = "CF-Ack+CF-Poll";
            break;
        case 0x88:
            type = "QoS";
            break;
        case 0x98:
            type = "QoS Data+CF-Ack";
            break;
        case 0xA8:
            type = "QoS Data+CF-Poll";
            break;
        case 0xB8:
            type = "QoS Data+CF-Ack+CF-Poll";
            break;
        case 0xC8:
            type = "QoS Null";
            break;
        case 0xE8:
            type = "QoS CF-Poll";
            break;
        case 0xF8:
            type = "QoS CF-Ack+CF-Poll";
            break;
        default:
            printf("Unknown packet: %02hhx \n", packet[offset]);
            break;
    }
    printf("IEEE 802.11 %s, Flags:\n", type);

    // 获取包携带的四个地址的头指针
    add1 = packet+(offset+4);
    add2 = packet+(offset+10);
    add3 = packet+(offset+16);
    add4 = packet+(offset+24);

    // 根据标志位信息获取各Mac地址信息
    switch(toDS_fromDS_flag) {
        case 0:
            DA = add1;
            SA = add2;
            BSSID = add3;

```

```

        printf("\tDestination MAC address:
%02hhx:%02hhx:%02hhx:%02hhx:%02hhx\n",
            DA[0], DA[1], DA[2], DA[3], DA[4], DA[5]);
        printf("\tSender MAC address:
%02hhx:%02hhx:%02hhx:%02hhx:%02hhx:%02hhx\n",
            SA[0], SA[1], SA[2], SA[3], SA[4], SA[5]);
        printf("\tBSSID: %02hhx:%02hhx:%02hhx:%02hhx:%02hhx:%02hhx\n",
            BSSID[0], BSSID[1], BSSID[2], BSSID[3], BSSID[4],
BSSID[5]);

        break;
    case 1:
        DA = add3;
        SA = add2;
        BSSID = add1;
        printf("\tDestination MAC address:
%02hhx:%02hhx:%02hhx:%02hhx:%02hhx:%02hhx\n",
            DA[0], DA[1], DA[2], DA[3], DA[4], DA[5]);
        printf("\tSender MAC address:
%02hhx:%02hhx:%02hhx:%02hhx:%02hhx:%02hhx\n",
            SA[0], SA[1], SA[2], SA[3], SA[4], SA[5]);
        printf("\tBSSID: %02hhx:%02hhx:%02hhx:%02hhx:%02hhx:%02hhx\n",
            BSSID[0], BSSID[1], BSSID[2], BSSID[3], BSSID[4],
BSSID[5]);

        break;
    case 2:
        DA = add1;
        BSSID = add2;
        SA = add3;
        printf("\tDestination MAC address:
%02hhx:%02hhx:%02hhx:%02hhx:%02hhx:%02hhx\n",
            DA[0], DA[1], DA[2], DA[3], DA[4], DA[5]);
        printf("\tSender MAC address:
%02hhx:%02hhx:%02hhx:%02hhx:%02hhx:%02hhx\n",
            SA[0], SA[1], SA[2], SA[3], SA[4], SA[5]);
        printf("\tBSSID: %02hhx:%02hhx:%02hhx:%02hhx:%02hhx:%02hhx\n",
            BSSID[0], BSSID[1], BSSID[2], BSSID[3], BSSID[4],
BSSID[5]);

        break;
    case 3:
        RA = add1;
        TA = add2;
        DA = add3;
        SA = add4;
        printf("\tReceiver MAC address:
%02hhx:%02hhx:%02hhx:%02hhx:%02hhx:%02hhx\n",
            RA[0], RA[1], RA[2], RA[3], RA[4], RA[5]);
        printf("\tTransmission MAC address:
%02hhx:%02hhx:%02hhx:%02hhx:%02hhx:%02hhx\n",
            TA[0], TA[1], TA[2], TA[3], TA[4], TA[5]);
        printf("\tDestination MAC address:
%02hhx:%02hhx:%02hhx:%02hhx:%02hhx:%02hhx\n",
            DA[0], DA[1], DA[2], DA[3], DA[4], DA[5]);
        printf("\tSender MAC address:
%02hhx:%02hhx:%02hhx:%02hhx:%02hhx:%02hhx\n",
            SA[0], SA[1], SA[2], SA[3], SA[4], SA[5]);

```

```

        printf("\tDestination MAC address:
%02hhx:%02hhx:%02hhx:%02hhx:%02hhx\n",
               BSSID[0], BSSID[1], BSSID[2], BSSID[3], BSSID[4],
               BSSID[5]);

        break;
    default:
        printf("Error To DS & From DS flags: %d\n", toDS_fromDS_flag);
        break;
    }

}

int main(int argc, char** argv) {
    if (argc < 2) {
        // 用法为./$< [.cap文件的路径], 如./80211sniffer foo.cap
        printf("Usage: ./80211sniffer [filepath]\n");
        return 0;
    }
    char errbuf[PCAP_ERRBUF_SIZE];
    int status = 0;
    int option_index = 0;
    char* file = argv[1];
    pcap_t* handle;
    // 打开文件对抓包进行读取
    handle = pcap_open_offline(file, errbuf);
    if(handle) {
        // 一直解析包直到文件结束, 对解析到的包调用回调函数pcap_callback
        pcap_loop(handle, -1, pcap_callback, NULL);
    } else {
        printf("error is %s \n", errbuf);
        exit(1);
    }
    pcap_close(handle);
    return 0;
}

```

编译命令

```
gcc 80211.c -l pcap -o 80211sniffer
```

运行

```
./80211sniffer foo.cap
```

附: 实验环境

OS: Ubuntu 20.04.1 LTS

Linux version:5.8.0-43-generic (bulld@lcy01-amd64-018)

gcc --version: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0

