ESP32-WROOM Promiscuous Mode Packet Sniffing

Abir Mojumder

Abstract—A paper describing a program that allows users to sniff WiFi packets using the Espressif ESP32 module.

Index Terms—Keywords should be taken from the taxonomy (http://www.computer.org/mc/keywords/keywords.htm). Keywords should closely reflect the topic and should optimally characterize the paper. Use about four key words or phrases in alphabetical order, separated by commas (there should not be a period at the end of the index terms)

1 Introduction And Motivation

This document reviews the experiment performed with the Espressif ESP32-WROOM module for packet sniffing in the WiFi Access Point's Promiscuous mode. In the Design and Implementation section we will review the libraries, IDE and a complete overview of the packet sniffer program. I wanted to try this experiment after taking an undergraduate course at Iowa State University (CPRE 430) where we were given a virtual machine that was setup with Wireshark and was assigned to look at some of the different packets that were being captured and be able to identify some of the information seen in those packets. The ESP32 module comes with a WiFi module that can capture all visible packets in the air when in promiscuous mode. The program I have written uses the promiscuous mode and parses the data into a PCAP format used by Wireshark. This data is output to the serial port of a computer which is then read by a python program "SerialShark.py" made by GitHub user "xdavidhu" that writes to a file (in PCAP format) and is finally re-read by the Wireshark program.

2 Problem Statement

Although this project doesn't aim to solve any real-life issue, it does look at the issue of internet anonymity. At my current apartment, the WiFi service is fairly unsecure, allowing anyone to connect to the network. With sufficient knowledge of network systems, an attacker may be able to get important/private information out of someone's computer if sent to another unsecure destination especially since the attacker is also part of the mesh network. This is one of the reasons that I chose to try an experiment such as this, to find out just how much I can learn by spending $15 (the price of a pair of ESP32 modules).

3 Design and Implementation

3.1 Hardware

The ESP32 Module is the only additional hardware required for this experiment. I have placed it securely on a breadboard. It is connected to the USB port of a computer via a Micro-USB cable

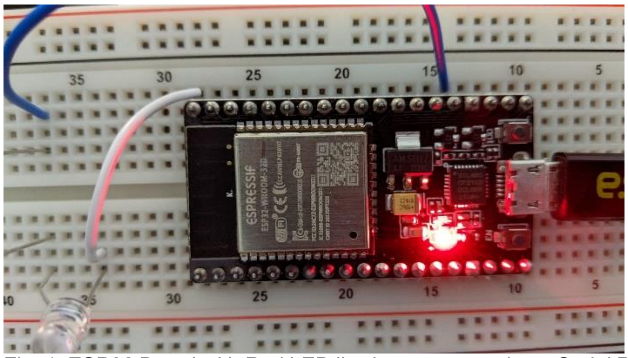


Fig. 1. ESP32 Board with Red LED lit when connected to a Serial Port

3.2 Software

The software required to develop the program and flash it to the ESP32 board is the Arduino IDE. Espressif provides the GitHub repository for the ESP32 Board Library to use with the Arduino IDE. Wireshark tool is a free program for analysing network packets and will be the main GUI for looking at the captured packets. SerialShark is a python program written by "xdavidhu" which is required for this experiment to work and can be found at his Github repository. PySerial is a python library that allows Linux systems to give access to serial ports to user made software (SerialShark needs this).

3.3 Program Implementation

The packet sniffer is written in C++ using the Arduino IDE. The required libraries and initialization configurations were taken from the Espressif ESP32 WiFi API documentation [1]. Since the program must output data that is readable by Wireshark, a PCAP header information needs to be send out through the serial port first. The header information tells wireshark some basic but important information such as the Endianness of the data, what PCAP version is being used, the maximum length of captured packets, timestamp offset, data link type and a few other header data that remain zero as they

• *Abir Mojumder is a Graduate student at Iowa State University. Ames, IA 50014. E-mail : abir99@iastate.edu*

are optional. This information can be found at Wireshark's Wiki Page [2].

The next step is to setup the ESP32 to operate in promiscuous mode. The Previously mentioned WiFi API documentation tells us the minimum required configuration to capture packets. The steps are:

1. Initialize the stack configuration to default
2. Initialize the wifi module using the configuration from step 1.
3. Enable the WiFi storage RAM
4. Set Wifi to Access Point mode
5. Enable the WiFi module
6. Set WiFi to Promiscuous Mode
7. Set the WiFi scanning channel
8. Define a callback function with a packet is received

Once the initialization is done, we need to deal with the received packet by writing a callback function as mentioned in step 8 above. The function receives two arguments: a packet buffer and the packet type (data link type), we can ignore the packet type for this experiment. The structure of the packet is shown below, taken from Łukasz Podkalicki's blog on ESP32 [3]:

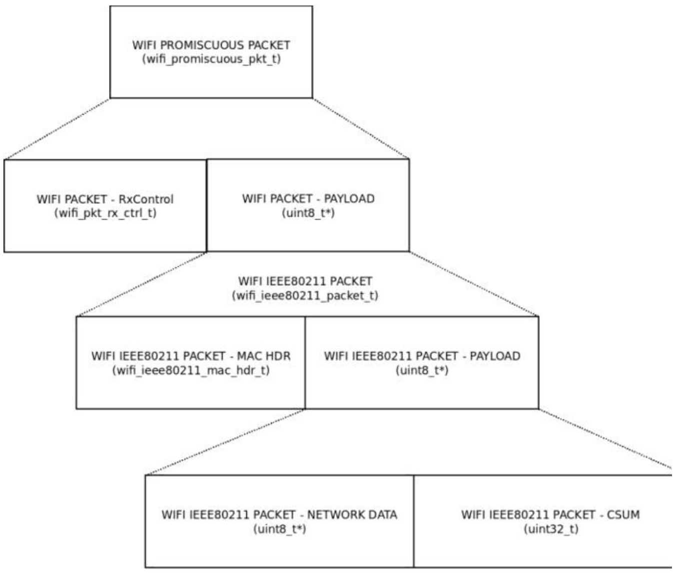

WIFI PROMISCUOUS PACKET
(wifi_promiscuous_pkt_t)
WIFI PACKET - RxControl
(wifi_pkt_rx_ctrl_t)
WIFI PACKET - PAYLOAD
(uint8_t*)
WIFI IEEE80211 PACKET
(wifi_ieee80211_packet_t)
WIFI IEEE80211 PACKET - MAC HDR
(wifi_ieee80211_mac_hdr_t)
WIFI IEEE80211 PACKET - PAYLOAD
(uint8_t*)
WIFI IEEE80211 PACKET - NETWORK DATA
(uint8_t*)
WIFI IEEE80211 PACKET - CSUM
(uint32_t)

Fig. 2. A visual breakdown of the captured packet

As seen from the image, the received packet has a RxControl and a Payload. The RxControl is important as it tells us the size of the payload which the program later uses to make sure the parsed packet isn't too big. The 'payload' itself has the MAC header and Payload. During initial testing of the program, the mac header could be used to parse out sender and receiver MAC addresses and output to the serial monitor in the Arduino IDE. However, with the use of Wireshark, we can directly send the entire payload in single bytes of data which the tool can read and format into the GUI. We also two other critical information: Timestamp of packet received in seconds and microseconds. This is necessary by the Wireshark program as defined by the PCAP format rules. Earlier I mentioned that the program outputs some PCAP header information; the timestamp requirement in seconds and microseconds if defined by one of the header information called the "magic_number". Depending on the magic number, you could send timestamps in nanoseconds too.

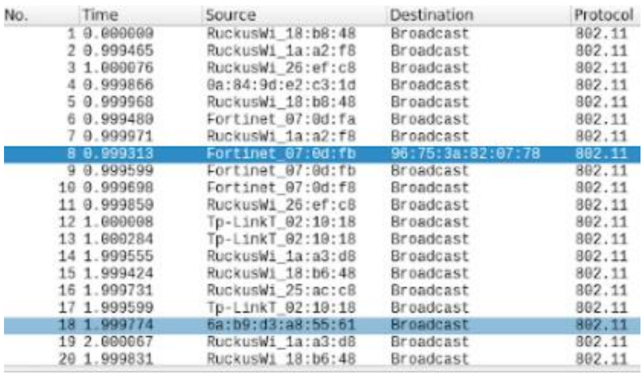
Now that we have the timestamp in seconds and microseconds, packet length and the packet itself, we can send it out via the serial port. But first it requires parsing, since the 4 fields of data are in 32-bit unsigned integers. A helper method is written that converts 32-bit input to 4-byte output. It does so by creating a byte long unsigned integer variable, taking the input and shifting the bits to the right by 8 bits incrementaly and storing it byte by byte. Ie: buf[1] = input >> 8 and buf[2] = input >> 16 and so on. Once that is done, use the command 'Serial.write' to write the formatted data to the serial port. Note that this formatting is not necessary for the payload itself since it already comes in an array of bytes, so you can simply write to serial by doing 'Serial.write(payload, payload_len)'.

The bits to bytes conversion were necessary because the SerialShark program only reads a byte at a time from the serial port. I was able to figure that out by working backwards from the SerialShark code. This part of the program also checks to make sure the data packet is not greater than 65535 bytes (defined as the 'snaplen' in the PCAP header information) since the ESP32 has a WiFi RAM smaller than the frame size of 65536 bytes.

So far, the main functionality of the program has been covered. Now we will look at an additional feature of channel hopping as the module allows us to scan channels 1 through 11. The program initially starts at channel 1, and has an interval of 250 milliseconds to hop to the next channel. An infinitely looping method checks to see how long it has been since the last time channel was changed. If greater than the interval, increase the channel to 1. Once we reach channel 12, this is not a valid channel so we reset back to channel 1. This concludes the overview of the packet sniffer program using the ESP32.

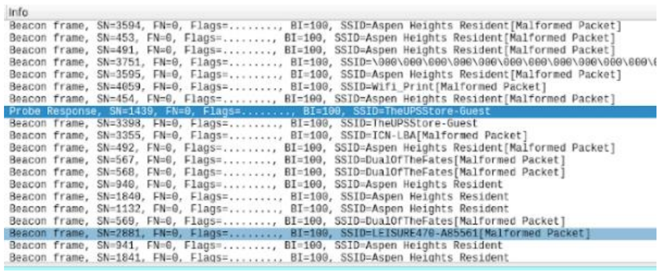
4 Result of Experiment

After flashing the program into the ESP32 module and running the SerialShark python program in an Ubuntu machine, we can see several packets being captured in the Wireshark GUI. The image is split into multiple pieces for better visibility.



```
No.    Time       Source               Destination           Protocol
     1 0.000000   RuckusWi_18:b8:48    Broadcast             802.11
     2 0.999465   RuckusWi_1a:a2:f8    Broadcast             802.11
     3 1.000076   RuckusWi_26:ef:c8    Broadcast             802.11
     4 0.999866   0a:84:9d:e2:c3:1d    Broadcast             802.11
     5 0.999968   RuckusWi_18:b8:48    Broadcast             802.11
     6 0.999480   Fortinet_07:0d:fa    Broadcast             802.11
     7 0.999971   RuckusWi_1a:a2:f8    Broadcast             802.11
     8 0.999313   Fortinet_07:0d:fb    96:75:3a:82:07:78     802.11
     9 0.999599   Fortinet_07:0d:fb    Broadcast             802.11
    10 0.999698   Fortinet_07:0d:f8    Broadcast             802.11
    11 0.999850   RuckusWi_26:ef:c8    Broadcast             802.11
    12 1.000008   Tp-LinkT_02:10:18    Broadcast             802.11
    13 1.000284   Tp-LinkT_02:10:18    Broadcast             802.11
    14 1.999555   RuckusWi_1a:a3:d8    Broadcast             802.11
    15 1.999424   RuckusWi_18:b6:48    Broadcast             802.11
    16 1.999731   RuckusWi_25:ac:c8    Broadcast             802.11
    17 1.999599   Tp-LinkT_02:10:18    Broadcast             802.11
    18 1.999774   6a:b9:d3:a8:55:61    Broadcast             802.11
    19 2.000067   RuckusWi_1a:a3:d8    Broadcast             802.11
    20 1.999831   RuckusWi_18:b6:48    Broadcast             802.11
```

Fig. 3. Packets captured with timestamp, source, and destination



```
Info
Beacon frame, SN=3594, FN=0, Flags=........, BI=100, SSID=Aspen Heights Resident[Malformed Packet]
Beacon frame, SN=453, FN=0, Flags=........, BI=100, SSID=Aspen Heights Resident[Malformed Packet]
Beacon frame, SN=491, FN=0, Flags=........, BI=100, SSID=Aspen Heights Resident[Malformed Packet]
Beacon frame, SN=3751, FN=0, Flags=........, BI=100, SSID=\000\000\000\000\000\000\000\000\000\000\000\
Beacon frame, SN=3595, FN=0, Flags=........, BI=100, SSID=Aspen Heights Resident[Malformed Packet]
Beacon frame, SN=4059, FN=0, Flags=........, BI=100, SSID=Wifi_Print[Malformed Packet]
Beacon frame, SN=454, FN=0, Flags=........, BI=100, SSID=Aspen Heights Resident[Malformed Packet]
Probe Response, SN=1439, FN=0, Flags=........, BI=100, SSID=TheUPSStore-Guest
Beacon frame, SN=3398, FN=0, Flags=........, BI=100, SSID=TheUPSStore-Guest
Beacon frame, SN=3355, FN=0, Flags=........, BI=100, SSID=ICN-LBA[Malformed Packet]
Beacon frame, SN=492, FN=0, Flags=........, BI=100, SSID=Aspen Heights Resident[Malformed Packet]
Beacon frame, SN=567, FN=0, Flags=........, BI=100, SSID=DualOfTheFates[Malformed Packet]
Beacon frame, SN=568, FN=0, Flags=........, BI=100, SSID=DualOfTheFates[Malformed Packet]
Beacon frame, SN=940, FN=0, Flags=........, BI=100, SSID=Aspen Heights Resident
Beacon frame, SN=1840, FN=0, Flags=........, BI=100, SSID=Aspen Heights Resident
Beacon frame, SN=1132, FN=0, Flags=........, BI=100, SSID=Aspen Heights Resident
Beacon frame, SN=569, FN=0, Flags=........, BI=100, SSID=DualOfTheFates[Malformed Packet]
Beacon frame, SN=2881, FN=0, Flags=........, BI=100, SSID=LEISURE470-A85561[Malformed Packet]
Beacon frame, SN=941, FN=0, Flags=........, BI=100, SSID=Aspen Heights Resident
Beacon frame, SN=1841, FN=0, Flags=........, BI=100, SSID=Aspen Heights Resident
```

Fig. 4. Additional packet information

From the images above, we can see the source BSSID along with destination address which is sometimes displayed as the MAC address or BSSID or is a Broadcast message.

You may have noticed on Figure 4 alongside the SSID "[Malformed Packet]". This is due to the packet capturing capability of the ESP32 as the packet size is too big, being cut off or simply not picking up the next packet in the sequence of packets.

5 Conclusion