Ryan Mia

4/9/19

Wi-fi Deauthentication application on Raspberry Pi

**Abstract**

The project deliverable is a Wi-Fi deauthentication application ported onto a Raspberry Pi 3. Using a packet receiving/sending library as well as a kernel modification for the Raspberry Pi to enable monitor mode and packet injection, the application is able to sniff packets in order to obtain a list of MAC addresses of both clients and access points. Then the second half of the application sends deauthentication packets to each MAC address repeatedly to disable Wi-Fi for both the client and host. The final product can send deauthentication packets at a quick enough rate in an ideal test environment to never allow connection to Wi-Fi at all.

**Introduction**

The main purpose for pursuing this project goal is to understand more about wireless security and the flaws in IEEE 802.11 wireless standard (Wi-Fi). This application is a Wi-Fi deauthenticator, not to be confused with a Wi-Fi jammer. A Wi-Fi jammer sends a series of signals around the common Wi-Fi frequency bands (2.4GHz and 5GHz) in order to interfere with packets being sent at those frequencies, thus blocking Wi-Fi signals in the range of the attack. A Wi-Fi deauthentication device, by contrast sends a standard network message to the devices in order to kick them off of the network. Creating this application was a good way of testing the security or the lack thereof of the currently set up Wi-Fi system. This project was for research purposes only Wi-Fi deauthentication may be illegal in various jurisdiction; the output of this project should not be used other than for research demonstration purposes.

In order to create a working application two tools were used. The first is Nexmon, a firmware patch for Kali Linux OS to enable monitor mode and packet injection on the Raspberry PI 3.[1] Without this, the Raspberry Pi would be unable to obtain a list of clients and host MAC addresses. Scapy, a package for Python which has built in functions for the crafting of deauthentication packets, sending packets, sniffing packets, and checking packets for different attributes was also used in this project.[2]

**Discussion**

The development of this application involved setting up the environment, creating the application, and then executing the application in the terminal. There were also some assumptions made in the test environment this application can run under.

*Setup Environment*

The device used to host the application is a Raspberry Pi 3. This device is a cheap portable computer that usually runs a Linux operating system. Note the use of a Raspberry Pi 3 instead of a Raspberry Pi. This is important as the feature of having a Wi-Fi chip on this device is essential to the application.

There are multiple steps in set up of the environment. First a specific operating system must be installed with a kernel modification. The standard OS kernels for a Raspberry Pi 3 do not allow monitor operation with the on-board Wi-Fi chip. In order to get around this, a Kali Linux OS with Nexmon preinstalled on it was flashed onto the SD card that goes in the Raspberry Pi. This OS can be downloaded from:

https://whitedome.com.au/re4son/download/sticky-fingers-kali-pi/

Once the Raspberry Pi 3 is running on this OS, monitor mode can be enabled on the device.

Next, a packet sniffing and sending package was installed on the device. Scapy was chosen as it has more useful functions than similar packages. To install Scapy this command was executed in the terminal:

$pip install scapy

There are some limitations of using a Raspberry Pi 3. The built-in Wi-Fi chip is not very strong so the range of the application is somewhat limited. In addition, sometimes when executing the program, the application may not start properly, or may run extremely slowly. In such cases a reboot is usually sufficient to get the program working properly again.

*Application*

As an initial step, the application sets conf.iface=’wlan0mon’. This line ensures that Scapy utilizes the interface in monitor mode for all sniffing and packet sending.

Next, the program runs two threads concurrently. One thread sniffs packets and interprets the incoming data in order to add to the client and access point list for the other thread. The other thread switches the channel and sends deauthentication packets to all of the clients and access points on the current channel. The client and access point lists are global variables so the list is updated in the sniffing thread, the deauthentication thread immediately has access to the new information.

The sniffing thread uses Scapy’s sniff command. The program is set up so the get\_cli\_n\_hos function is called repeatedly with a parameter ‘sniffed’ which is the packet currently being sniffed. This function will add clients and hosts in range of the Raspberry Pi based on different parameters in the sniffed packet. Initially it is important to ensure that the packet is a Wi-Fi packet using the .haslayer(Dot11) function which is a Scapy function that returns true if the packet is of type Dot11.[3]

Then the packet is checked to see if it is a request or response packet. These type of packets hold information which allows access points to be added to the hosts list.[4] In order to check this, the .haslayer(Dot11Beacon) and .haslayer(Dot11ProbeResp) function calls are used. Then the bssid, ssid, and channel are extracted using parsing from the packet. The proper syntax for the function calls to extract these values are modeled from the program by Dan McInerney.[5] The bssid is the MAC address of the access point and the ssid is the access point Wi-Fi name.[6] A check is done to ensure that the bssid is not equivalent to any bssid already on the hosts list. This ensures the list is not being flooded with duplicate access points, slowing down the application. If the bssid is unique then a structure which includes the [bssid, channel, ssid] is appended to the host list.

After the logic to potentially append a host is complete, a similar process is performed to add any new clients. The type of packet is checked to be either type 1 or 2 (management or data), both of which are in the correct format to obtain client information. If so, then the client is appended if the client and access point MAC address are not already in the client list, and that one of the addresses matches an address in the host list. This ensures there are no duplicate clients and that a client is not found whose access point has not been found yet, as information from the access point is needed as part of the data in the client list. If these conditions are met, then the client list appends a structure of [source address, receiver address, channel, ssid]. The ssid and channel are taken from the matching host entry, as this information is not in the packet checked.

The second thread runs function channel\_hop\_n\_deth. This function takes in the interface as its only parameter, but relies on the updating of global variables clients and hosts. The function is one large infinite loop which loops from channels 1-11 (the maximum number of channels in the U.S).[7] The channel is changed by calling command:

$iw dev wlan0mon set channel *channelnumber*

where *channelnumber* is a variable of the current channel number. Inside the infinite loop, the list of clients and hosts are looped through in order to find any that have the same channel as the current channel. For each client on the client list that has the matching channel two deauthentication packets are sent. One packet is sent to the client, and the other is sent to the access point that the client is connected too. For each access point on the same channel one deauthentication packet is sent to the access point with a default client of FF:FF:FF:FF:FF:FF. The deauthentication packets are created using the Dott11() function which creates a Wi-Fi packet, and the Dot11Deauth() function which specifies the packet to be of type deauthentication. Then the packets are sent using send(). All three of these functions can be found in the Scapy package.

*Execution and Assumptions*

In order to run the application these few commands should be run in the terminal first:

$ifconfig wlan0 down

$iwconfig wlan0 mode monitor

$ifconfig wlan0 up

$airmon-ng start wlan0

The network wlan0 is disabled to allow modifications. Then monitor mode is enabled on the interface. This is allowed due to the Nexmon kernel modification. The interface can be put back up now. The last command starts the monitor interface and creates a new network called ‘wlan0mon’. After using these commands, the new interface called ‘wlan0mon’ now appears when typing the ‘iwconfig’ command. The monitor mode allows for sniffing of packets and to obtain a list of clients and hosts.

Now to execute the application, go to /home/pi/jam/ and run command:

$sudo python no\_more\_wifi.py

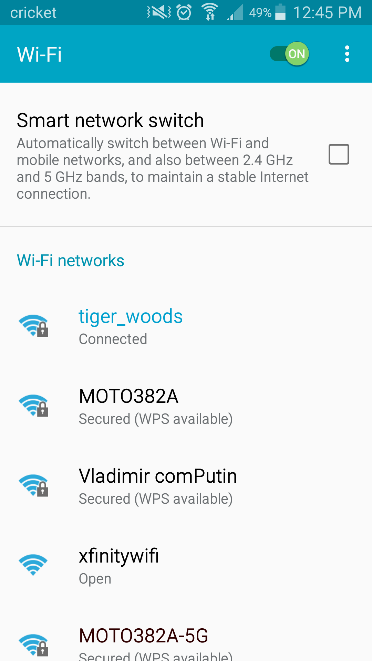
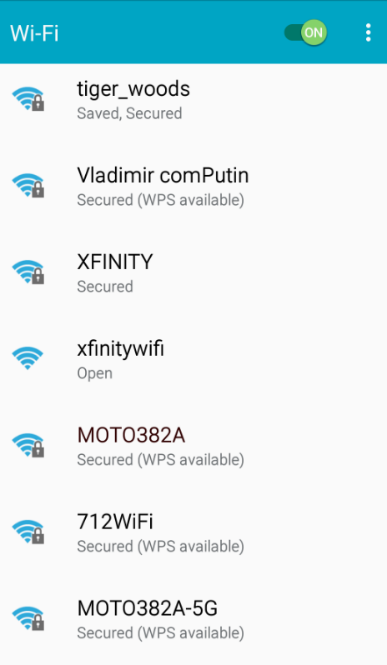
The program output will display the list of clients and access points each time of channel switching as well as the packet address 1,2, and 3 for each one sent.

Some assumptions were made in order to improve efficiency and to work better in a stationary location with a relatively small number of clients and access points. This was chosen as the default to be optimized for as it is the most likely conditions to be running such an application under, especially considering the application should only be run in a test environment. Because there is never a wipe on the hosts and client lists, if there are a very large list due to moving around or being in a very congested area the application will slow down immensely. In extreme cases this can cause the program to stop being effective as deauthentication packets will not be sent rapidly enough to immediately kick clients off the network. However, the advantage is that the program never needs to recreate the clients and hosts list and will be more efficient.

Lastly, an assumption is made that the deauthentication is taking place in the United State, as channels range from 1-11, but international channels range from 1-13. This is an easy fix that could be added as a command line argument in the future.

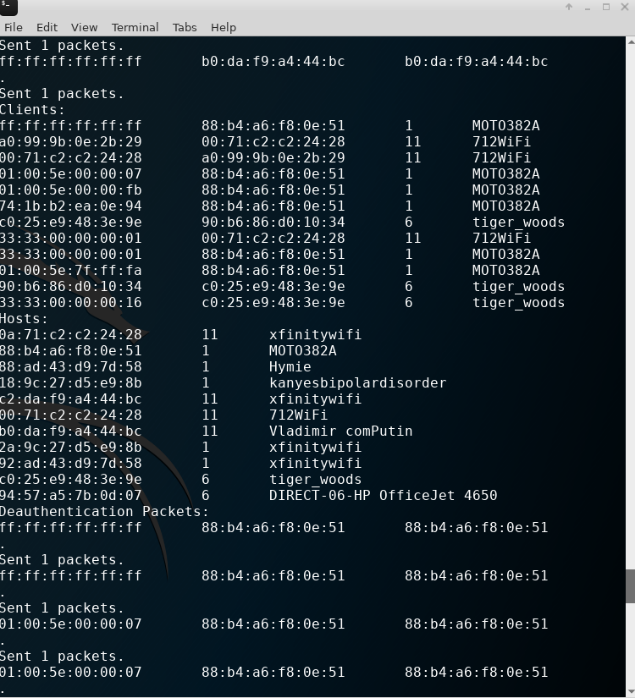
**Results**

The result of this application can be seen in the video presentation. When a cell phone is connected to the Wi-Fi, and the program is executed in a couple of seconds the cell phone will be disconnected. Then, depending on the current running speed and strength of the connection at the time, the Wi-Fi will either stay disabled indefinitely or pop in and out of connection, only being connected for approximately half a second at a time. The screenshots of a cellular device before and after the program is connected can be seen in **Figure 1** and **Figure 2**, respectively.



**Figure 2**: Wi-Fi disconnected during application execution

**Figure 1:** Wi-Fi connected before application execution

In addition to the visible results on the Wi-Fi channels, there is also a terminal output that outputs very quickly the list of hosts and clients after each iteration through the packet sniffing as well as the deauthentication packets MAC addresses when they send. A small snippet of this output can be seen in **Figure 3**.

**Figure 3:** A portion of the terminal output when the application is running.

**Conclusion**

This project was chosen specifically to learn more about 802.11, how it works, and a current security flaw. Using the monitor mode on a Raspberry Pi Wi-Fi chip and packet sniffing, it is possible to obtain a list of MAC addresses from both clients and access points in range of the Raspberry Pi. Then deauthentication packets can be created and sent to the clients and access points rapidly in order to disable the Wi-Fi for all users in range. Packet sniffing is a common security flaw in many communication systems and, in addition to the ease of creating and sending deauthentication packets, makes 802.11 especially vulnerable to attacks such as these. The application developed can disable Wi-Fi for clients in range of the device as well as returning a list of hosts and clients in the surrounding areas.

**References and Software Sources**

[1] Nexmon GitHub files: <https://github.com/seemoo-lab/nexmon>

[2] Scapy main webpage, installation instructions, and .zip download: <https://scapy.net/>

[3] Nathanson, Jonathan. “Dot11 Basics: What Is Dot11, Why Do I Need It and How

Does It Work?” The Excell Help Centre, Nov. 2018, excellgroupplc.zendesk.com/hc/en-gb/articles/206434729-Dot11-Basics-What- is-Dot11-why-do-I-need-it-and-how-does-it-work-.

[4] “Haktip 23 – WiFi 101: Probe Requests and Responses.” Hak5, 2011,

[www.hak5.org/episodes/haktip-23](http://www.hak5.org/episodes/haktip-23).

[5] McInerney, Dan, wifijammer, (2014), GitHub repository,

<https://github.com/DanMcInerney/wifijammer>

[6] “BSSID, SSID, and ESSID.” SourceDaddy, 2018,

sourcedaddy.com/networking/bssid-ssid-and-essid.html.

[7] Albano, Samantha. “WiFi Channels Explained.” Minim, Minim Inc, 9 July 2018,

www.minim.co/blog/wifi-channels-explained.