Exploitations of Wireless Interfaces via Network Scanning

Nathalie Domingo*, Bryan Pearson†, and Yier Jin‡
* Department of Electrical and Computer Engineering, Carnegie Mellon University
†Department of Computer Science, Stetson University
‡Department of Electrical and Computer Engineering, University of Central Florida
ndomingo@andrew.cmu.edu, peabryan95@gmail.com, yier.jin@eecs.ucf.edu

***Abstract*—In brainstorming for ways to exploit Internet of Things (IoT) security, we envisioned the following premise: an attack which compromises the security of multiple wireless interfaces. In this project, we attempt to reveal information about a region of networks; this data includes each network's location, security, signal strength, and user activity. In exposing such data, we hope to shed light on the security problems associated with wireless networks.**

I. Introduction

Network security is one of the most imperative areas of computer security in the modern era. The number of devices which require network access in order to function grows on a daily basis; because of this, network protection is more important than it has ever been before. Connecting wirelessly to these networks has become standard among devices, thus leading to the necessity of WiFi. Since it is no longer necessary to physically connect to the network, more vulnerabilities arise when dealing with wireless networks. In particular, more information, regarding both the network and the users, becomes available, making it imperative to secure these networks [1]. This notion provides our largest incentive in creating our attack; our long-term goal was to build something that would affect a large number of users and emphasize the importance of securing wireless networks.

The paper is organized as follows. Section II discusses the current state of wireless networks and related applications. In Section III we explain our initial attempts in creating this device. The implementation of our final device is explained in Section IV. The results from the test of the device is discussed in Section V. Finally, conclusions are presented in Section VI.

II. State-of-the-Art

Wireless networks have become more popular as they have many advantages over wired networks, but with the advantages also come new security problems. The main difference between wireless and wired networks is the way information is transmitted. In order to get information from a wired network, there must be a physical connection, while with a wireless network, all data transmission is done through the radio frequency, which allows for a lot more interceptions of data [2]. While this poses a disadvantage over wired networks, wireless networks are faster in terms of network configuration, more cost effective and easier to integrate with already existing networks [1]. These tremendous advantages over wired networks has led to their popularity and large interest in wireless network security. The main focus of securing wireless networks is on preventing unauthorized access, eavesdropping, and tampering of transmitted messages [3].

Some efforts have already been made to standardize WiFi security, which has led to WEP, WPA, and RSN standard protocols. All of these protocols are aimed at encrypting data in an attempt to prevent the security risks associated with transmitting sensitive information over the radio frequency. Wireless Equivalent Privacy (WEP) was created so that wireless networks would have the same security as wired networks. This protocol, given enough time and effort, can be broken, thus leading to the WPA standard. WiFi Protected Access (WPA) serves as an improvement over WEP in terms of data encryption, user authentication, and error detection. While this protocol is an improvement, when passphrases consisting of less than 20 characters are used, which is considered common, the network is more open to attacks. Lastly, in order to accommodate the improvement in wireless device capabilities, Robust Security Networks (RSN), or WPA2, was created as the final wireless security solution. The main improvement over WPA is that it facilitates the communication between wireless access points and wireless devices through a new encryption and authentication scheme [2].

Despite all the efforts to enhance wireless security, wireless networks still broadcast sensitive information, something that we exploit in the device explained in this paper.

There are various applications available that collect network information, most of which rely on using a mobile device. Two applications that fall into this category are WiFi Collector [4] and OpenWebGIS [5]. WiFi Collector is an Android app that collects various information about the available wireless networks. The information collected includes the network name, the signal level, and the location and time the information was taken, which is then displayed in a list format in the app, all of which is exportable to external mapping applications. OpenWebGIS has more features than WiFi Collector, as it can be used to collect and map any type of data. As of 2016, OpenWebGIS has a detected WiFi access points option which collects various information and then maps it. The information collected includes the SSID, the coordinate point of where the information was taken, the address of the access point, the authentication and encryption schemes, and the data and time the data was collected. Chosen information is then available in list format and coordinate points are viewable automatically on a map.

Although these applications are available, they have mostly been created with the purpose of helping people keep track of where they can connect to networks. While the device we propose in this paper has a similar data collection aspect, it is aimed at exposing security vulnerabilities in wireless networks by showing the amount of sensitive information that networks reveal, all of which is displayed in a user friendly way.

III. Initial Attempts

In order to access the Raspberry Pi while outside, it was necessary to turn the Pi into an AP (access point) since there was not an available network on which we could SSH into the Pi. When turning the Raspberry Pi into an AP, the main issue encountered was trying to maintain the Pi's built in functions, such as WiFi and Ethernet. The first attempt at doing this disregarded the fact that the data collection script required the use of WiFi, and thus the configure AP on the built in WiFi had to be removed. Instead the AP was created via a WiFi dongle. Once the dongle was used, more problems arose as the built in WiFi, necessary for the data collection script, and Ethernet, necessary for testing purposes, stopped working. This problem was fixed by reverting configuration files to their original state and paying close attention to the configurations that were changed when repeating the process.

Additionally, we encountered problems with communicating with the GPS module. The network data needed to be mapped out, so it was essential to know the location of the Raspberry Pi when the data was being collected. A GPS module provided this information, but proved difficult to communicate with. The GPS module was connected to the Pi via UART, but because a newer Pi was utilized for this project, there was a lack of documentation on which port corresponded to the serial port. Documentation for an older model was used and it identified the `/dev/AMA0` port as the designated serial port. When testing the GPS, it was determined that this was not the correct serial port and that it was actually being used for the built in Bluetooth, a feature in only the newer model. Thus, it was important to figure out the correct serial port to in order to communicate with the GPS, an essential component in our project.

IV. Implementation

A Raspberry Pi 3 model B running the Raspbian Jessie kernel version 4.4 and an Adafruit Ultimate GPS Breakout module was used in this project to carry out the objective. The Raspberry Pi was used as a means to communicate with the GPS and to collect data about the available APs, while the GPS was used to determine where the data point was collected. In order to amplify the GPS's ability to correctly capture its coordinates, an external antenna was used. The Raspberry Pi must first be remotely accessed via the AP created through the WiFi dongle attached to the Pi in order to run a Python script that collects all the data on the available networks and writes it to a CSV file. Afterwards, a second Python script must be ran that converts the outputted CSV file to a KML file which can be viewed on a map. Figure 1 shows the Pi and GPS setup used to collect data.

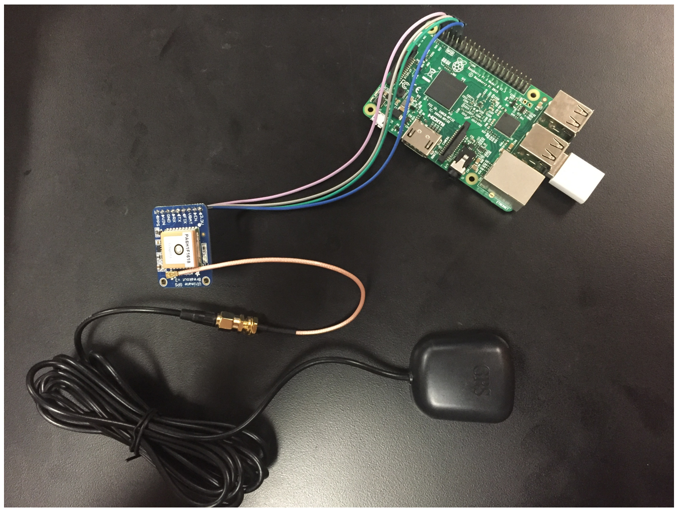


Fig. 1: Device set-up used to collect data about the available access points

By using the WiFi dongle, the Raspberry Pi can act as an AP, which can then be connected to by any computer, thus allowing for access to the Raspberry Pi through SSH or a tightVNC viewer. Since this version of Raspberry Pi has a built-in WiFi module, configurations regarding the WiFi dongle had were made to affect wlan1, instead of wlan0. The first step to doing this is setting up a DHCP server which allows WiFi connections to automatically get IP addresses. Next, `wlan1` must be given a static IP, and the AP credentials must be created through the `hostapd` program. Lastly, in order to automate this process, i.e. to ensure that the AP will be available automatically when the Pi boots up, a daemon must be set up that enables both `hostapd` and the DHCP server. Additionally, in order to allow access to the Pi's Internet connection via the AP, the network address translation must be configured. After the AP is configured and after connecting to the network created, the Pi's command line was available via SSH and the desktop through a tightVNC viewer. In order to be able to see the GUI through a tightVNC viewer, a tightVNC server must be set up on the Raspberry Pi. This too can be automated by placing the necessary commands in the Pi's `/etc/rc.local` file.

In order to map out the network information, it was essential to be able to grab the coordinates of where the network data was collected. In order to do this, an Adafruit Ultimate GPS Breakout was used. Every time data was collected, the coordinates of the Pi were taken from the GPS module. This was done by connecting the GPS to the correct pins in the Pi's UART header and then setting up communications between the two. Communication was set up by enabling UART communication and starting the GPS daemon, `gpsd`, on the correct serial port, which in this case was `/dev/serial0` or `/dev/ttyS0`. Additionally, although the GPS is powered through the Pi, a coin cell was used to allow the GPS to get a better signal, as per the manufacturer's suggestion.

The first stage of our attack involves executing the data retrieval script, which will ultimately output a CSV file that we can later use. The script, written in Python, must run in root in order to execute many of the required commands. The program starts by creating an empty CSV file with writing privileges so that we can append data to the file later. The script immediately writes a single line to the CSV file which contains the headers of each category. This is because each datum in the first line of a CSV file is always interpreted as the header of that respective column (refer to Figure 5 for an illustration). Following this, the script asks the user to input the total length, in minutes, that the script should run; after this time has passed, the program will automatically terminate. In addition, the script will ask for the wait time before it reconnects to APs that have already been accessed; in other words, if the script connects to an AP named `Access`, the script will have to wait n minutes (n being the input from the user) before connecting to `Access` again. The delay is implemented in order to force the script to connect to a variety of APs, because the network scanner will, by default, connect to the AP with the strongest signal. However, the delay also ensures that we can still collect multiple samples from the same network to obtain more accurate data.

Once the user inputs these settings, the script enters a loop which will continuously acquire and append data to the CSV file until the program eventually terminates. The loop first determines whether the script should terminate by checking if enough time has passed; if not, it continues. Next, it checks whether enough time has passed for the delay to reset. If enough time has passed, the timer resets, and the string containing all inaccessible APs is cleared, allowing reconnection to those APs; otherwise, the script continues.

Before scanning for nearby networks, the script will first attempt to capture the latitude, longitude, and date information from the GPS module that was attached beforehand. The script communicates with the GPS by first opening a serial port in the location `/dev/serial0` with a baud rate (the speed at which the GPS transfers its data) of 9600 bits per second. The port must then flush its input buffer contents, since the buffer updates automatically. Flushing the buffer is critical, as this will ensure that we only receive the newest reading from the GPS. After flushing the contents, we then attempt to read the next line of the buffer stream, which is presumed to be the most recent line. If this line exists, we parse it into a readable format using the Python library `pynmea2`. The script then uses the parsed format to obtain the latitude and longitude of its position, as well as the time that it retrieved this data. One caveat of this process is that the GPS takes a short moment to add data to its buffer, meaning it will usually take several attempts before we can recognize the data. Therefore, we use another loop to repeatedly poll whether we can recognize the buffer data; once we can, we retrieve the GPS data and exit the loop. Otherwise, after fifty attempts, the script will set the contents for GPS data as "not found;" this would most likely occur due to a weak GPS signal.

Next, the script attempts to ensure that the wireless interface is permitted to scan for APs. To achieve this, the script communicates with the Linux terminal via the os library, and running the `popen` command, allowing it to specify a terminal command and to save its return value. We run the terminal command `iwlist wlan0 scan 2>/dev/null | grep ESSID`, which will return the IDs of all nearby APs. If scanning is not permitted, then the command will return a blank string, in which case the script resets the wireless interface and runs the command again, until eventually it returns at least one SSID. To reset the interface, we run the terminal command `ifconfig wlan0 down` to shut down the interface, and `ifconfig wlan0 up` to start it back up. Scanning permissions are generally enabled by default when the interface reboots, although its occasional inconsistencies mean that we must always recheck for scanning permissions before connecting to another AP.

Once the script has finished checking the scan permissions, it obtains the SSID of the next available AP. To accomplish this, we run the scan command mentioned previously, and obtain the first AP that does not appear in the string containing the list of inaccessible APs. We must also ensure that the script ignores the WiFi dongle attached to the Raspberry Pi (we simply ignore the AP if it has the SSID "PiAP", which is the name of the WiFi dongle). Once we have acquired the SSID, we can retrieve the encryption type and signal strength of the network, as well as the number of devices connected to the network - assuming it is an open AP. To obtain the encryption, we scan the network similarly as before, albeit with one key difference: `iwlist wlan0 scan 2>/dev/null | egrep 'ESSID|Encryption'`. In this command, the terminal returns the SSID of all nearby networks along with their encryption settings - an encryption of "off" signifies an open network, while an encryption of "on" signifies a closed network. We compare our acquired SSID to each SSID in the return value until a match is found, and then we grab the corresponding encryption. The process of obtaining the signal strength is nearly identical, except the command is now `iwlist wlan0 scan 2>/dev/null | egrep 'ESSID|Quality'`. Signal strength of WiFi is measured in dBm (decibel-milliwatts), where a dBm of -30 indicates the strongest achievable strength and a dBm of -90 and below indicates a very weak connection.

As mentioned before, we will only attempt to retrieve the number of devices connected to the network if it is not encrypted. This is because we use Address Resolution Protocol (ARP) scanning to retrieve this information, which requires a network connection in order to function. ARP is a protocol that can determine a MAC address from a given IP address. By "scanning" the network, ARP sends message packets to all hosts on the network, and essentially counts how many of those hosts receive the packet. To connect to the network, we run the terminal command `iwconfig wlan0 essid [ID]`, where]`[ID]` corresponds to our network SSID. This command will configure the wireless interface such that we can specify which network to associate with. Afterwards, we run the command `arp-scan --interface=wlan0 --localnet` to send ARP packets to all hosts on the network. This will return a list of hidden IP address that we

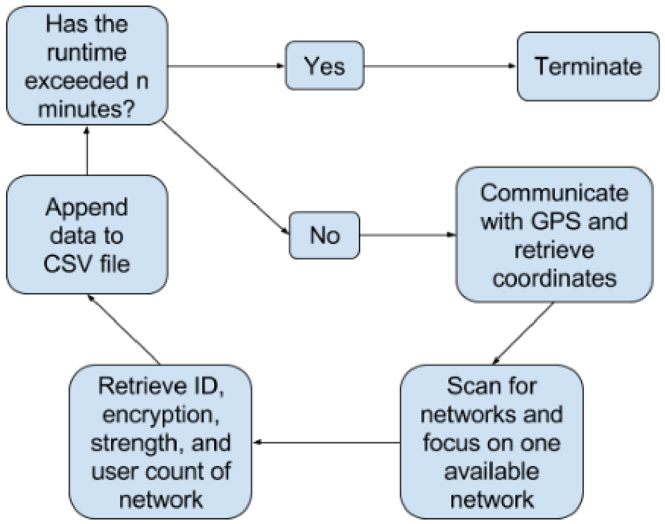

Has the runtime exceeded n minutes?
Yes
Terminate
No
Communicate with GPS and retrieve coordinates
Scan for networks and focus on one available network
Retrieve ID, encryption, strength, and user count of network
Append data to CSV file

Fig. 2: The process the data collection script goes through

Signal Category	dBm Range	Color
Very Strong	>= -50	Dark Green
Good	<-50 and >= -65	Light Green
Weak	<-65 and >= -80	Yellow
Poor	<-80	Red

TABLE I: A chart showing the signal categories and their corresponding signal strength range and color

can count up in order to determine the number of connected devices. This method of retrieving device count data, while occasionally serviceable, is quite inefficient.

The final step in the script is a simple matter of appending gathered information to the CSV file created earlier. Afterwards, the script begins at the start of the loop, and the process repeats. An illustration of the script is provided in Figure 2.

After creating the CSV file with all of the data, it is converted into a KML file through a Python script. The script first parses the CSV file produced and sorts the data based on the network. Two dictionaries are kept to keep track of various information that must be sorted by network. The first dictionary keeps track of all the coordinates that belong to the same signal category. The signal categories were created in order to be able to create a map that resembles a contour map. All of the coordinates belonging to the same category, either poor, weak, good, or strong, are connected and colored in to create a polygon that is colored based on the signal strength. Table I is a chart with the signal categories and their corresponding signal strength ranges and polygon colors. This is done to be able to differentiate where the signal is the best and to get the closest representation of where the access point originates, which is where the signal is the strongest. Additionally, the same dictionary keeps track of the network's average information, which includes the most number of users, least number of users, average number of users, encryption, and the timestamp of when the last data point was taken.

On top of having the different polygons representing each

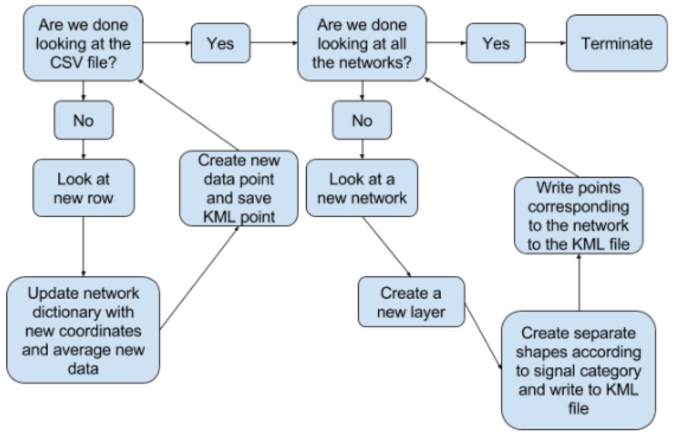

Are we done looking at the CSV file?
Yes
Are we done looking at all the networks?
Yes
Terminate
No
Look at new row
Create new data point and save KML point
Update network dictionary with new coordinates and average new data
No
Look at a new network
Create a new layer
Write points corresponding to the network to the KML file
Create separate shapes according to signal category and write to KML file

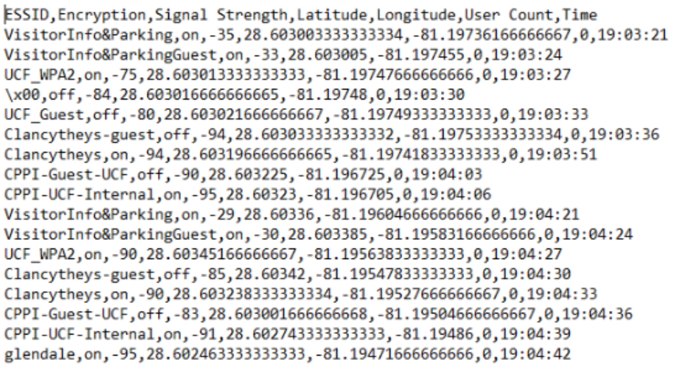
Fig. 3: The process the CSV to KML script goes through

signal level, each data point is also individually mapped in order to easily display the individual data points collected. The second dictionary is used to keep track of all the points that must be written to the KML file. Instead of creating a KML point after parsing through the CSV file, the point is created while going through the rows in the CSV file. KML points are created by creating a placemark, and then specifying the point coordinates. Additionally, in order to attach a textbox, the style of the textbox and the actual text, which is referred to as a description, has to be specified. The placemark created for each individual point is what is saved in the dictionary, which then makes it easy to write it to the KML file according to the network it belongs to.

After parsing through the CSV file, each network is written to the KML file as a new layer. Within each network layer, polygons are first created referring to the signal category and then every corresponding data point is written. The polygons are created by creating a placemarker, and then within the placemarker specifying the polygon style, which includes the signal category color and that it should be filled in. After specifying the style, the polygon is actually created by specifying the coordinates that should be connected and to clamp the polygon to the ground. Lastly, in order to have a text box pop up when the polygon is clicked on, the text style and the actual text description must be specified. This KML file can then be read by Google Maps and Google Earth. The best viewing platform for the number of layers that are most likely will be created is Google Earth, where each layer can be individually viewed and overlapping points can be viewed independently on the map. Figure 3 shows the complete process the python script goes through to convert the CSV file to a KML file.

V. Experimental Results

In order to test out our implementation, we collected data around the University of Central Florida (UCF). We took a shuttle that went around the perimeter of the campus and collected 322 data entries, each consisting of the network ESSID, the encryption, the signal strength, latitude, longitude,



```
ESSID,Encryption,Signal Strength,Latitude,Longitude,User Count,Time
VisitorInfo&Parking,on,-35,28.603003333333334,-81.19736166666667,0,19:03:21
VisitorInfo&ParkingGuest,on,-33,28.603005,-81.197455,0,19:03:24
UCF_WPA2,on,-75,28.603013333333333,-81.19747666666666,0,19:03:27
\x00,off,-84,28.603016666666665,-81.19748,0,19:03:30
UCF_Guest,off,-80,28.603021666666667,-81.19749333333333,0,19:03:33
Clancytheys-guest,off,-94,28.603033333333332,-81.19753333333334,0,19:03:36
Clancytheys,on,-94,28.603196666666665,-81.19741833333333,0,19:03:51
CPPI-Guest-UCF,off,-90,28.603225,-81.196725,0,19:04:03
CPPI-UCF-Internal,on,-95,28.60323,-81.196705,0,19:04:06
VisitorInfo&Parking,on,-29,28.60336,-81.19604666666666,0,19:04:21
VisitorInfo&ParkingGuest,on,-30,28.603385,-81.19583166666666,0,19:04:24
UCF_WPA2,on,-90,28.60345166666667,-81.19563833333333,0,19:04:27
Clancytheys-guest,off,-85,28.60342,-81.19547833333333,0,19:04:30
Clancytheys,on,-90,28.603238333333334,-81.19527666666667,0,19:04:33
CPPI-Guest-UCF,off,-83,28.603001666666668,-81.19504666666667,0,19:04:36
CPPI-UCF-Internal,on,-91,28.602743333333333,-81.19486,0,19:04:39
glendale,on,-95,28.602463333333333,-81.19471666666666,0,19:04:42
```

Fig. 4: First 18 lines of the CSV file produced from the data collection script

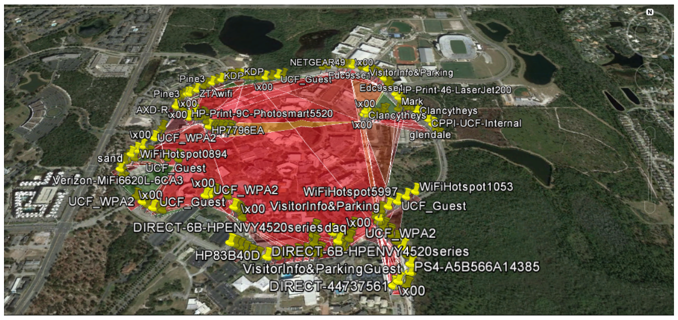


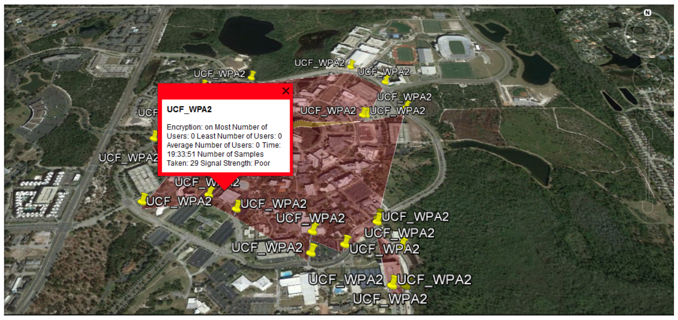
Fig. 5: All the layers of the data collected shown in Google Earth

user count, and the time. When we were collecting data, we ran into problems concerning the method used for determining the user count of the network. Specifically, the script would pause for a very long time, causing us to not be able to collect data and have a huge gap between the consecutive coordinates. We were concerned that this would continue to happen and that it would result in us not having a very representative map, so we removed the user count method and simply have the user count for every network be 0. Figure 4 shows an excerpt of the CSV file produced from the data collection script.

Since the campus shuttle moved slowly, we were able to collect multiple coordinates for each network with varying signal strengths, allowing for quite a few data polygons. Additionally, most of the coordinate points were not overlapping, which is ideal when looking at multiple networks on the same map. Although the data polygons cover most of the UCF campus, it is not an accurate depiction of available access points within the campus. Data was only taken along the perimeter, and thus these maps are not a good representation of the entire campus. Overall, the results were as we expected and serve as an initial test of the capability of the device created. After mapping the collected data to Google Earth, the experimental results are shown in Figures 5, 6, and 7.

VI. Conclusion

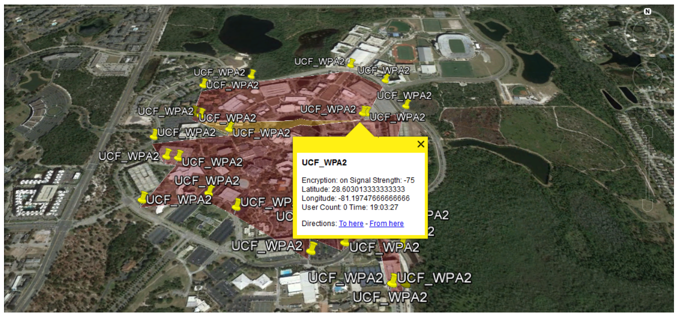
In this paper we have described a project that should be treated as a first step in a much larger attack. The project involves collecting informative data about the available networks and displaying that information in an easily viewed platform. The data collected includes the network names, the encryption status, and ideally, the number of users. Then, in order to give a realistic picture as to where the networks are available, a contour-like map is created to display the signal strength, along with the network data collected, by utilizing a GPS breakout board. This project can be used to figure out the best network to attack, something that is important when creating and deploying an attack that will affect the most users.



UCF_WPA2

Encryption: on Most Number of Users: 0 Least Number of Users: 0 Average Number of Users: 0 Time: 19:33:51 Number of Samples Taken: 29 Signal Strength: Poor

Fig. 6: One layer isolated in Google Earth and typical text box



UCF_WPA2

Encryption: on Signal Strength: -75
Latitude: 28.603013333333333
Longitude: -81.19747666666666
User Count: 0 Time: 19:03:27

Directions: To here - From here

Fig. 7: One layer isolated in Google Earth and individual data point text box

VII. Acknowledgment

The support for this work was provided by the National Science Foundation REU program under Award No. 1560302. Any opinions, findings, and conclusions and recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] M.-k. Choi, R. J. Robles, C.-h. Hong, and T.-h. Kim, “Wireless network security: Vulnerabilities, threats and countermeasures,” *International Journal of Multimedia and Ubiquitous Engineering*, vol. 3, no. 3, pp. 77–86, 2008.

[2] H. I. Bulbul, I. Batmaz, and M. Ozel, “Wireless network security: comparison of wep (wired equivalent privacy) mechanism, wpa (wi-fi protected access) and rsn (robust security network) security protocols,” in *Proceedings of the 1st international conference on Forensic applications and techniques in telecommunications, information, and multimedia and workshop*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008, p. 9.

[3] H. Boland and H. Mousavi, “Security issues of the ieee 802.11 b wireless lan,” in *Electrical and Computer Engineering, 2004. Canadian Conference on*, vol. 1. IEEE, 2004, pp. 333–336.

[4] Http://www.nirsoft.net/android/wifi_collector.html.

[5] Http://openwebgisystem.blogspot.com/2016/01/creating-wifi-map-and-monitoring-access.html.