Concordia
UNIVERSITY

Reverse engineering of AirCrack software

Summary

Report on the reverse engineering performed on aircrack software. Provide architecture, design and implementation details of this program thanks to static analysis. Written by Concordia Students for “Management of Evolving Systems” course.

Keywords

aircrack, wireless networks, encryption, WEP, FMS attack, replay attack, korek, initialization vector, specification, architecture, design.

Mihail Roman – mike_galaxy@hotmail.com
Laurent Fallet – laurent.fallet@insa-rouen.fr
Sumit Chandel – sumit_chandel2003@yahoo.ca
Najib Nassif – najib_na@cse.concordia.ca

May 2005

Foreword on objectives of this report

The software aircrack has been created and maintained by Christophe Devine. As there is no change or maintenance to perform on aircrack by our team, the objective of our report is to provide documentation.

Our analysis is starting from scratch because no document was provided at the beginning.

We have therefore used static analysis principles and tools. The report contains 3 main parts: general information surrounding WEP and wireless networks, specification recovery and design recovery. Even if the static analysis has been done in a bottom-up way (design, then specification), specification is presented first, to preserve understanding. Our analysis is mainly based on flow analysis (control and data flow).

We do not know if Christophe created documentation for himself while developing aircrack. If he did, this paper might provide him an alternative view of his software, and show him another way to present it.

That’s the reason why this document will be proposed to Christophe.

Table of Contents

Table of Figures

Introduction / Context

This report aims to present a comprehensive document to aid in the understanding of AirCrack, a WEP encryption key cracker.

First, some general concepts about wireless networks and more especially encryption will be explained. This includes the WEP algorithm that uses RC4 as the encryption algorithm. We will also talk about FMS attacks, the most famous cracking algorithm. Those concepts are needed to understand the way the program operates. If you're already familiar with wireless networks and WEP encryption, feel free to skip it.

Then we will explain the objectives of AirCrack: what does it do, and how does it do it?

In a second part, we will explain the architecture and design of AirCrack, its general components and how they are used. Diagrams obtained by reverse engineering tools will help in this task.

The third part details the implementation of the cracking in the software. Although it might be a little technical, it's the most interesting part for crypto analysts.

Part 1 - General Information

1.1 - A word on AirCrack

AirCrack is an 802.11 WEP key cracker. Its main purpose is to find the secret key used during a communication in a wireless network. The program collects certain number of packets from the network, dumps them in a file, and once an enough variety of packets are collected, it starts the cracking and obtain the WEP secret key. The need for diverse packets is crucial, because the more different pieces of information it has, the better are the chances of cracking.

The author is Christophe Devine, a French software engineer fond of cryptography and computer science. Information and download area of AirCrack can be found at the address http://www.cr0.net:8040/code/network/. AirCrack is distributed under the GNU General Public License.

1.2 - Wired Equivalent Privacy (WEP)

Wired Equivalent Privacy (WEP) is part of the IEEE 802.11 standard and is a scheme used to secure wireless networks. Because a wireless network broadcasts messages using radio waves, it is principally susceptible to eavesdropping; WEP was designed to provide comparable confidentiality to a traditional wired network, hence the name Wired Equivalent Privacy.

However, several serious weaknesses were identified by cryptographers, and WEP was superseded by Wi-Fi Protected Access (WPA) in 2003, and then by the full IEEE 802.11i standard also known as WPA2 in 2004. Despite the inherent weaknesses, WEP provides a bare minimal level of security that can deter casual snooping, and this is where AirCrack blows the WEP security protocol out the window.

1.2.1 - Encryption process

The following figure explains the encryption process. Briefly, an IV (which stands for Initialization Vector) is appended to the secret key. The aim of theses IVs is to ensure that the value used as a seed for the RC4 algorithm is always different. A gold rule is to never reuse an old key.

The key goes through the RC4 to create the keystream, which is XORed with the plaintext. The IV used to encrypt the packet is appended to the encrypted packet in order to be able to decrypt it.

Getting the IV is apparently not enough to achieve to decrypt a packet.

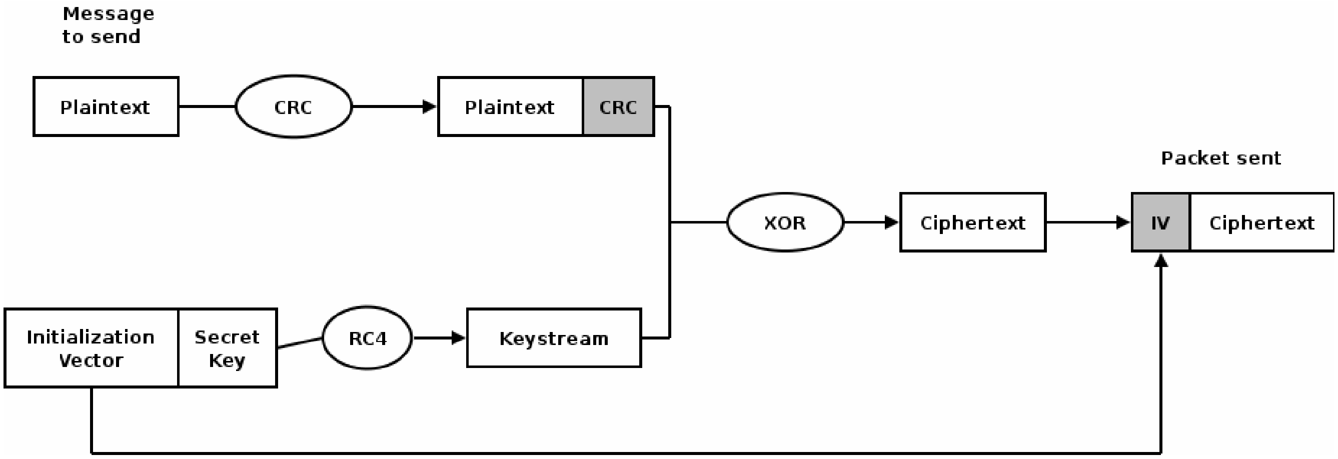
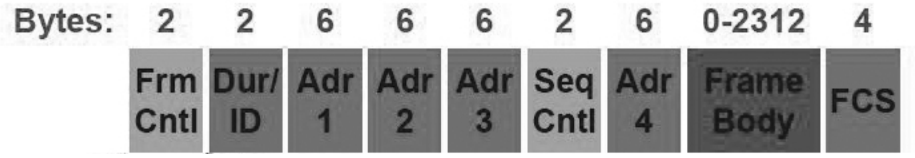


Figure 1 - WEP encryption process

The length of the IV is 3 bytes (24 bits), and the length of the secret key can be 5, 13, 29 or 61 bytes. When a key is claimed 64 bits long, it consists of 24 bits of IV and 40 of secret key (the same applies for 128 bits long, 104 bits of secret key). In cryptography, only the secret part of a key counts. WEP is therefore weaker than it claims it is.

1.2.2 - Wireless Packet Format

In order to get a better understanding on how WEP secures it’s packets against snooping and sniffing, let’s have a look to a regular wireless non encrypted packet.



Bytes:	2	2	6	6	6	2	6	0-2312	4
	Frm Cntl	Dur/ ID	Adr 1	Adr 2	Adr 3	Seq Cntl	Adr 4	Frame Body	FCS

Figure 2 - Non-Encrypted Wireless Packet

The header of the packet starts at the beginning up to the frame body. The first 2 bytes concerns frame control, holding information about the protocol, the type, if there are other fragments, etc. Sometimes this part might have a length of 8 bytes in case LLC is used. After the duration/ID bytes are stored 4 addresses of 6 bytes length each and the sequence control.

The frame body is the payload, and at the end the Frame Sequence Check, usually computed with the CRC algorithm.

The following figure displays the format of an encrypted wireless packet, and highlights the encrypted and non-encrypted parts.

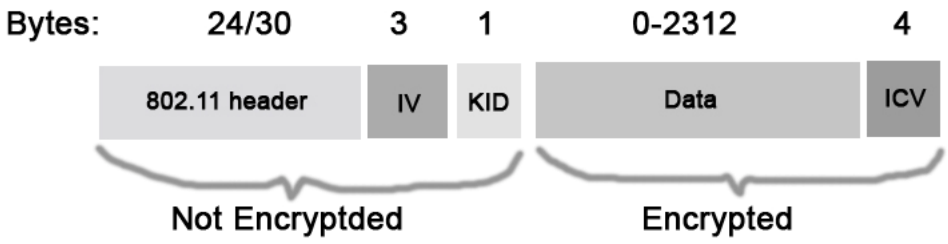


Figure 3 - WEP Encrypted Wireless Packet

As a matter of fact, in order to route the packet through the network, the packet header cannot be encrypted, because it holds destination and transmission addresses.

This diagram shows that the IV is not encrypted (because it is needed during the decryption process), and AirCrack takes advantage of this fact. It uses these IVs (specifically unique IVs) in order to crack the WEP key and allow a malicious user access to the wireless network.

Let’s now have a look at the encryption part of WEP.

1.3 - RC4 Encryption Algorithm

The encryption algorithm that WEP uses is called RC4.

In cryptography, **RC4** (or **ARCFOUR**) is the most widely-used software stream cipher and is used in popular protocols such as Secure Sockets Layer (SSL) (to protect Internet traffic) and WEP (to secure wireless networks). RC4 falls short of the high standards of security set by cryptographers, and some ways of using RC4 lead to very insecure cryptosystems, especially in our case and the WEP protocol. There are two main algorithms’ that are part of the RC4 encryption algorithm; the KSA, Key Scheduling Algorithm, is the first part of the encryption process and the second part is the PRGA. The Pseudo Random Generation Algorithm is the part of RC4 process that outputs a streaming key based on the KSA’s pseudo random state array.

1.3.1 - Key Scheduling Algorithm (KSA)

Here is the algorithm of KSA:

```
KSA(K)
K[] = Secrete Key array
Initialization:
For i = 0 to  N - 1
  S[i] = i
j = 0
Scrambling:
For i = 0 to N - 1
  j = j + S[i] + K[i mod l]
  Swap(S[i], S[j])
```

The Key scheduling algorithm is quite simple. It initializes an array of values with values of 0 to N in the position of the array a[0] to a[N]. It then performs a series of swapping between the values of the array in order to scramble the values.

The next step is performed by the Pseudo Random Generation Algorithm.

1.3.2 - Pseudo Random Generation Algorithm (PRGA)

Here is the algorithm of PRGA:

```
PRGA(K)
Initialization:
i = 0
j = 0
Generation Loop:
  i = i + 1
  j = j + S[i]
  Swap(S[i], S[j])
  Output z = S[S[i] + S[j]]
```

As the name indicates this algorithm is used to generate a random key. It goes about doing this by first initializing the two array indicators to zero. It then increments the two indicators separately and swaps values from the array using the two indicators. From this method/algorithm someone with programming knowledge can see that it is not the best pseudo random algorithm possible, but it is effective enough for the purpose of RC4 encryption. Together the KSA and PRGA are the heart of the RC4 algorithm and are also the reason why it is vulnerable. Because WEP keys rely intimately upon the RC4 algorithm the weakness of the RC4 algorithm is passed on the WEP key and as such makes it susceptible to being cracked.

1.4 - Weaknesses of WEP

When WEP is active in a wireless network, each wireless packet is encrypted separately with the RC4 cipher stream generated by a 64-bit key. This key is composed of a 24-bit initialization vector (IV) and a 40-bit secret key. The encrypted packet is generated with a bitwise exclusive OR (XOR) of the original packet and the RC4 stream. IVs are chosen by the sender’s computer and are changed periodically so that every packet won't be encrypted with the same cipher stream. The IV is sent in the clear with each packet and is not encrypted with the RC4 key. An additional 4-byte Integrity Check Value (ICV) is computed on the original packet and appended to the end. The ICV is also encrypted with the RC4 cipher stream unlike the IV.

WEP has been widely criticized for more than one apparent weakness; among those weaknesses we include a few in the following section.

1.4.1.1 - Key management and key size

One of the major problems in the WEP standard is that key management is not specified and without interoperable key management, keys will tend to be long-lived and of poor quality. Most wireless networks that use WEP have one single WEP key shared between every node on the network. Access points and client stations must be programmed with the same WEP key. Since synchronizing the change of keys is tedious and difficult, keys are hardly ever changed. And on top of that fact, the 802.11 standard does not specify any WEP key sizes other than 40 bits & 104 bits.

1.4.1.2 - The IV is too small

The WEP's IV size of 24 bits provides a maximum of 16,777,216 different RC4 cipher streams for a given WEP key, for any key size. It is important to remember that the RC4 cipher stream is XOR-ed (a simple & easily reversible operation) with the original packet to give the encrypted packet that is transmitted, and on to further weaken it’s position, the IVs that are sent with each packet are not encrypted and are transmitted in the clear.

The general problem brought up with this point is IV reuse. After a few hours of operation in a heavily loaded wireless network, IVs possibilities start to be exhausted, and IVs begin to repeat themselves. Moreover NIC manufacturers implement the IV generation algorithm as they want, so sometimes you find a simple increment of the IV, starting at 0. So if you restart your computer, you’ll start sending the same IVs again and again.

The direct consequence is that if the RC4 cipher stream for a given IV is found, an attacker can decrypt subsequent packets that were encrypted with the same IV. He can also forge his own packets and submit “evil” packets to users on that wireless network.

1.4.1.3 - The ICV algorithm is not appropriate

The WEP ICV is based on CRC-32, an algorithm for detecting noise and common errors in transmission. CRC-32 is an excellent checksum for detecting errors, but an awful choice for a cryptographic hash. The CRC-32 ICV is a linear function of the message. Hence `CRC(x^y) = CRC(x)^CRC(y).`

Without going into details, it means that an attacker can modify an encrypted message and easily fix the ICV so the message appears authentic. Being able to modify encrypted packets provides for a nearly limitless number of very simple attacks. For example, an attacker can easily make the victim's wireless access point decrypt packets for him. Simply capture an encrypted packet stream, modify the destination address of each packet to be the attacker's wired IP address, fix up the CRC-32, and retransmit the packets over the air to the access point. The access point will happily decrypt the packets and forward them to the attacker through the wired network.

A better-designed method of packet control for ICV would use algorithms such as MD5 or SHA-1.

1.4.1.4 - Considerations on the Key ID

The key ID, or KID, also called key index, is an identifier of the shared secret key to use. It enables to use many different keys on the same link. The shared secret key chosen by the sender will be concatenated with the IV; the receiver will know which shared secret to use thanks to the KID.

The KID takes 1 byte in the packet; in fact it consists of 2 bits of information and 6 bits of padding. Therefore only 4 possible keys can be used per network.

1.5 - FMS Attack

1.5.1 - Cracking process

AirCrack uses the FMS attack in exploiting the network and getting the encryption key. The FMS attack named after its inventors Fluhrer, Mantin, and Shamir implements a passive cipher text-only attack against RC4 as used in WEP.

The principle is the same as in every cracking involving XOR: catch the encrypted data, and if you know the plain text, you'll recover the key!

We'll go through a few step of cracking to explain how the cracking can be done. This explanation originally comes from ref [3].

Before getting into the details, we need to define a few values:

- The captured weak IV is 3, 255, 7. This value was chosen because it was tested and is known to be a real weak IV.
- The pre-shared password is 22222. Although a hacker would not know this before cracking WEP, we need to define it so you can see the cracking process in action.
- N is 256
- The initialization process of the state array has already occurred and seeded the state array with the 256 values.

Here is the key array as a hacker would see it after capturing the IV.

K[0] = 3	K[1] = 255	K[2] = 7	K[3] = ?	K[4] = ?	K[5] = ?	K[6] = ?

Next, we need to define and track the state array values, i value, and j value. This will be done before each loop is processed, so you can see how the values change. We will not show all 256 state array values because they are useless to the cracking WEP process. Only the first four state array values and any value that has changed will be shown.

```
KSA loop 1
i=0 j=0 S[0]=0 S[1]=1 S[2]=2 S[3]=3
j=j + S[i] + K[i mod l] = 0 + S[0] + K[0] = 0 + 0 + 3 = 3 so j = 3
```

In this equation, you can see that the j and i value were 0, which is used by the S[] array (S[0] = 0) and the K[] array (K[0] = 3). This resulted in the values of 0, 0, and 3 being added together to assign the value of 3 to j. This value is then passed on to the swap function below.

```
i=0  j=3
Swap (S[i], S[j]) so Swap (S[0] , S[3])
As S[0] = 0 and S[3] = 3 we have S[0] = 3 , S[3] = 0
```

In this process, you can see that values held in S[0] and S[3] are swapped. This is an important process to watch, but remember there is a 5% chance that the values held in S[0] _ S[3] will not change after the first 4 KSA/PRGA loops.

```
KSA loop 2
i=1  j=3  S[0]=3  S[1]=1  S[2]=2  S[3]=0
j=j + S[i] + K[i mod l] = 3 + S[1] + K[1 mod 8] = 3 + 1 + 255 = 259 mod
256 = 3 so j = 3
i=1, j=3
Swap(S[i], S[j]) is Swap (S[1] , S[3])
As S[1]=1 , S[3]=0 we have S[1]=0 , S[3]=1
```

Note that in this loop the value of i increases by one and that a modulus operation was performed to determine the value of j. It is only coincidental that j = 3 again.

```
KSA loop 3
i=2  j=3  S[0]=3  S[1]=0  S[2]=2  S[3]=1
j=j + S[i] + K[i mod l] = 3 + S[2] + K[2] = 3 + 2 + 7 = 12 so j = 12
i=2, j=12
Swap(S[i], S[j]) is Swap (S[2] , S[12])
As S[2]=2 , S[12]=12 we obtain S[2]=12 , S[12]=2
```

Note that up to this point, only KNOWN values are used. Any hacker can reproduce this process up to this point. However, in the next step, the secret key is unknown, so a hacker has to stop.

```
KSA loop 4
i=3  j=12  S[0]=3  S[1]=0  S[2]=12  S[3]=1  S[12]=2
j=j + S[i] + K[i mod l] = 12 + S[3] + K[3] = 12 + 1 + ? = ?
i=3, j=?
Swap(S[i], S[j]) is Swap (S[3] , S[?])
As S[3]=1 , S[?]=? we just obtain S[3]=?? , S[??]=1
```

So, now a hacker is up against a wall. However, what if there was a way to determine the `j` value at this point? Fortunately, for a hacker, there is a way. A simple XOR calculation enables to determine this value from the first iteration of the PRGA process.

Knowing this, let’s reflect on the XOR process that creates the encrypted data. The final step of the RC4 process is to XOR a PRGA byte per byte with the plaintext data. Since XOR works in both directions, we also know that we can get deduce the first byte of the PRGA if we XOR the first byte of the encrypted data with the first byte of plaintext. But what is the plaintext?

Fortunately, for a hacker this is easy thanks to the SNAP header (the value is 170_{10} or AA_{16}) and the use of a sniffer to capture the encrypted byte. In our example, we will provide the captured encrypted byte value (165 in decimal), which changes from packet to packet.

The following equation illustrates the XOR process:

```
z = 0xAA(SNAP) XOR Ciphertext byte1 = 170 (Dec) XOR 165 (Dec) = 15
```

As a result of this XOR calculation, a hacker can deduce that the PRGA value is 15 (decimal). Now, he can reverse-engineer the PRGA process, and use this to determine the missing j value. First, let's remind ourselves of the known loop values as they would occur entering loop 4 of the KSA. Remember, these values can be easily reproduced by the use of the IV values.

```
KSA loop 4
i=3  j=12  S[0]=3  S[1]=0  S[2]=12  S[3]=1  S[12]=2
1. Initialization:
2. i=0
3. j=0
4. Generation:
5. i = i + 1 = 0 + 1 = 1
6. j = j + S[i] = 0 + S[1] = 0 + 0 = 0
7. Swap (S[i], S[j]) is Swap (S[1] , S[0])
   so S[1]=0 , S[0]=3 becomes S[1]=3 , S[0]=0
8. z = S[S[i] + S[j]] = S[S[1] + S[0]] = S[3 + 0] = S[3] = ?
9. ?=15 and S[3] =15 at KSA4
```

From the previous discussion, you know that i will always equal 1 for the first iteration of the PRGA (line 5). This then means that j will always equal S[0] (line 6). As we can see from the KSA loop 4 input values, S[1] = 0. This then results in j being assigned the value of 0 (line 6). The values held in S[i] and S[j] are then swapped, which means that S[1] is swapped with S[0] resulting in S[1] = 3 and S[0] = 0 (line 7). These values are then added together, and used to pull a value from the state array. In this case, the combined S[i] and S[j] values = 3 (line 8). However, the S[3] value referenced to here is from the completion of KSA loop 4, which is unknown to us. Fortunately, due to the XOR process, we know that the resulting value is 15, which means that S[3] will equal 15 at the output of KSA loop 4. Knowing this, a hacker only needs to reverse the KSA loop 4 process to deduce the secret key value.

Let's now walk though this as a hacker would.

```
KSA loop 4
i=3  j=12  S[0]=3  S[1]=0  S[2]=12  S[3]=1  S[12]=2
S[3]=15 , S[15]=S[3]t-1 gives S[3]=15 , S[15]=1
```

First, we know that the final step in the KSA loop is to swap values. Knowing the values of the state array after loop 4 completes and before it starts is important. Thanks to the XOR 'weakness', we know S[3] will equal 15, and we can make an educated guess that S[15] will hold the value held by S[3] before loop 4, which is 1 in this case. As a result, a hacker can deduce that S[3]=15 and S[15]=1 after the swap.

```
Swap (S[3] , S[15]) so S[3]=1 , S[15]=15
```

Next, a hacker swaps the values held in these positions, which leaves S[3] equaling 1.

```
j=j + S[i] + K[i mod 256] = 12 + S[3] + K[3] = 12 + 1 + K[3] = 15
```

A hacker then plugs the values into the equation that would produce the j value. This fills in all the fields except the value of the secret key array.

```
K[3] = 15 – 12 – 1 = 2
```

After a simple reverse calculation, the value 2 is produced, which is the first byte of out secret key!

1.5.2 - Replay attack

The attack explained here above assumes that there is enough wireless traffic in order to collect sufficient amount of packets with unique IVs. However, in the case when there is no network traffic, it is possible to force the access point to generate new packets, and therefore new IVs. Basically the attacker will send old ARP requests to the base station, therefore invoking ARP responses. Here is a diagram illustrating how an artificial traffic could be generated:

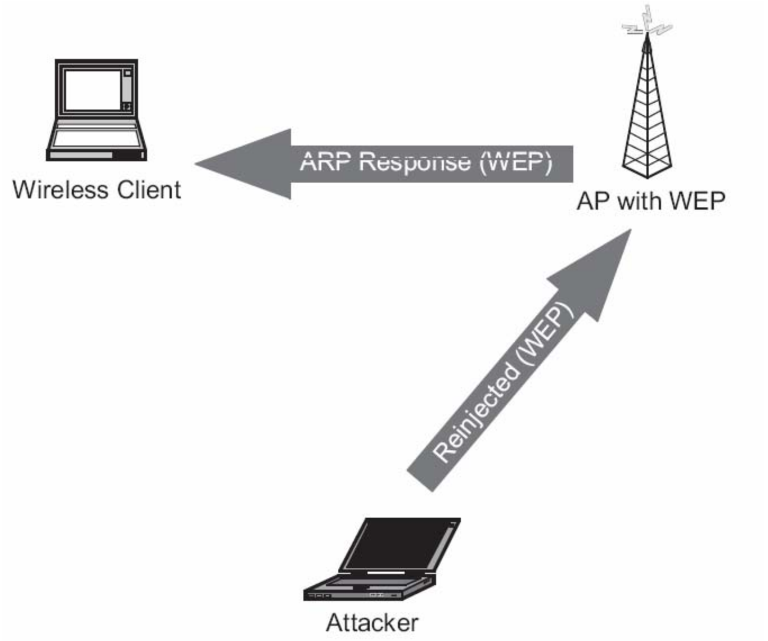


Figure 4 - ARP replay attack

The attacker floods the network with the re-injected ARP's request packets. This results in a flood of ARP responses, which then the attacker captures to initiate the FMS attack.

AirCrack provides a component to perform this attack. That's why Aircrack can be considered as a cracking solution rather than only a cracker program.

Part 2 - Architecture

Although Aircrack is a relatively small security program in terms of LOC, there is something to say about the tools it employs and the interactions between these tools and the central cracking component (aircrack.c).

Figure 5 below represents the overall implementation view of the software. In this section each component or tool is analyzed in terms of its required input and produced output.

Following these details, information explaining the component interaction is explored, demonstrating how Aircrack as a program can be run completely independently from other software or tools needed for WEP cracking.

This section of the report then goes into some details about architectural and design considerations that may or may not have been present in the implementation of the tool itself.

2.1 - Overview

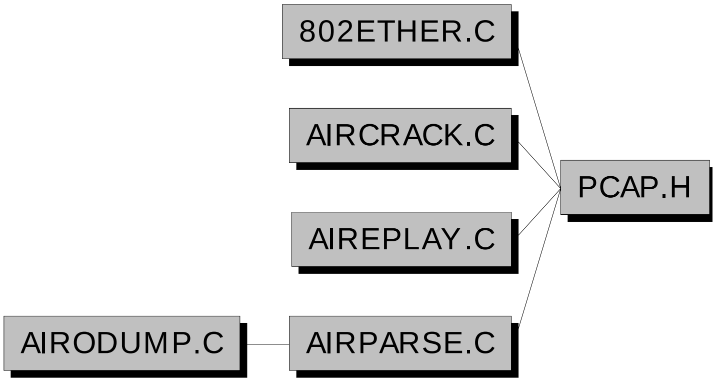


Figure 5 – Aircrack Component Layout

The overall explanation of this figure is that the airodump.c component uses airparse.c to verify the validity of the packets it is capturing (i.e. if they contain unique IVs that have not yet been captured, if they have the potential to allow key cracking and if they are well-formed packets). These packets, stored in a pcap.h and csv file, are fed to aircrack.c where the cracking is performed. A more detailed explanation of component interaction is given in section 2.3 - .

2.2 - Aircrack and Supporting Tools

Here detailed descriptions in terms of the objective and the input/outputs needed and produced, respectively, required for each component to play its part in Aircrack's cracking of the WEP key, are explained. Although aircrack.c itself is the part of the program that performs the cracking, the supporting tools allow Aircrack to be a standalone program that can gather everything it needs to perform an attack on and in the presence of the targeted wireless network.

2.2.1 - Aircrack.c

2.2.1.1 - Objectives

Aircrack.c is the central component of the Aircrack program. Essentially, this is where the cracking mechanism is employed and the WEP key of the targeted wireless network is recovered.

2.2.1.2 - Input/Output Analysis

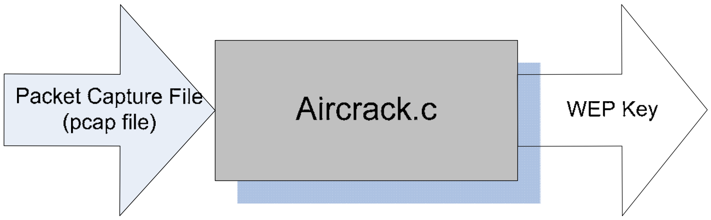


Figure 6 – Aircrack.c Input/Output Sequence

In Figure 5Figure 6 above, we illustrate the inputs to and the outputs generated from the aircrack.c component. The input consists of a packet capture file (pcap file), produced by airodump.c. Basically, the pcap file is a large set of packets that airodump.c was able to capture from the network. To crack a 40-bit WEP key, ~200,000 packets are needed whereas for a 104-bit WEP key, ~500,000 packets are required. The output of this component is the WEP key itself, which the component itself uses to verify if the key it found is indeed the WEP key that the target wireless network is using.

2.2.2 - Airodump.c

2.2.2.1 - Objectives

This component is responsible for “gathering the building blocks” of the security cracking program. Airodump.c listens to all the traffic transacting from the targeted network and dumps the packets to a packet capture file (pcap). Airdump.c uses airparse.c to select specific packets that have the potential to grant Aircrack the information necessary to crack the file, and drop those packets lacking this potential.

This component could be replaced by any other network sniffer programs that could capture packets and store them in a pcap file

2.2.2.2 - Input/Output Analysis

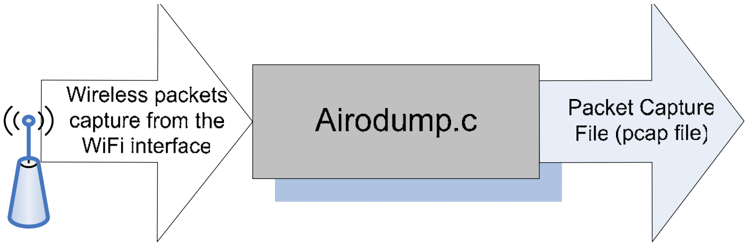

Wireless packets capture from the WiFi interface
Airodump.c
Packet Capture File (pcap file)

Figure 7 – Airodump.c Input/Output Sequence

The input to airodump.c is the wireless packets that are being transacted through the network. This requires a wireless NIC that must be set in promiscuous mode to capture all traffic flowing in the vicinity of the network. A WiFi interface allows airodump.c to take these packets in and write them to the pcap file. The output itself is the pcap file as well as a CSV (Comma Separated Values) file with captured packet data, separated by commas.

2.2.3 - Airparse.c

2.2.3.1 - Objectives

The objective of airparse.c is to extract those packets that have the potential to allow the cracking process to be successful. For example, it does not admit packets that are smaller than the 802.11 header size into the pcap file and skips uninteresting control and management frames. It also keeps a running tally of unique IVs and access points to the target wireless network.

2.2.3.2 - Input/Output Analysis

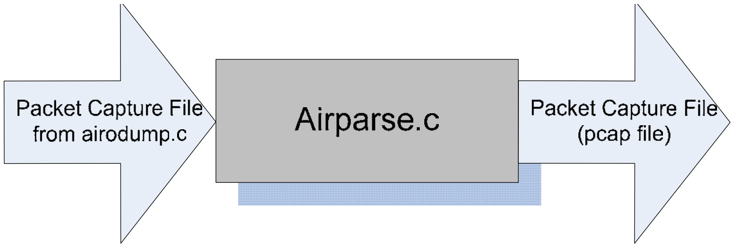

Packet Capture File from airodump.c
Airparse.c
Packet Capture File (pcap file)

Figure 8 – Airparse.c Input/Output Sequence

The input to airparse.c is the partial dumps in the pcap file generated by airodump.c. It parses the packets in the pcap file it receives, and recopies those packets which have the potential to allow aircrack.c to be successful in cracking the WEP key. The output is therefore another pcap file with the unyielding packets removed.

2.2.4 - Aireplay.c

2.2.4.1 - Objectives

The objective of aireplay.c is to generate traffic from the network when there is not much network activity present. aireplay.c replays ARP requests to induce responses from the access point and get more traffic to analyze (or dump).

2.2.4.2 - Input/Output Analysis

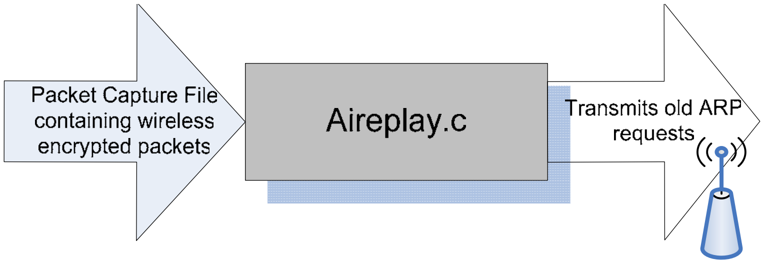

Packet Capture File containing wireless encrypted packets
Aireplay.c
Transmits old ARP requests

Figure 9 – Aireplay.c Input/Output Sequence

As illustrated in the figure above, the input to aireplay.c is a packet capture file. aireplay.c can be run completely separately from the rest of the Aircrack program and tools. The output of the aireplay.c component is old ARP requests that are continuously retransmitted. Of course, in order to capture these packets, airodump.c would be running at the same time as the packets are being replayed on a separate machine.

2.2.5 - 802ether.c

This program is used at the end of the chain, after finding the key. Its purpose is to decrypt wireless WEP encrypted packets and save them in the Ethernet format.

The output is an Ethernet pcap file that can be read in ethereal or any pcap analyzer. The aim is usually to extract communications, TCP threads, and get the whole communication between two hosts. It enables to catch emails, attachments, all the activity on the wireless network.

2.3 - Components Interaction

The diagram below shows a more detailed representation of the way that these components interact and rely on each other for data and the success of the WEP cracking process.

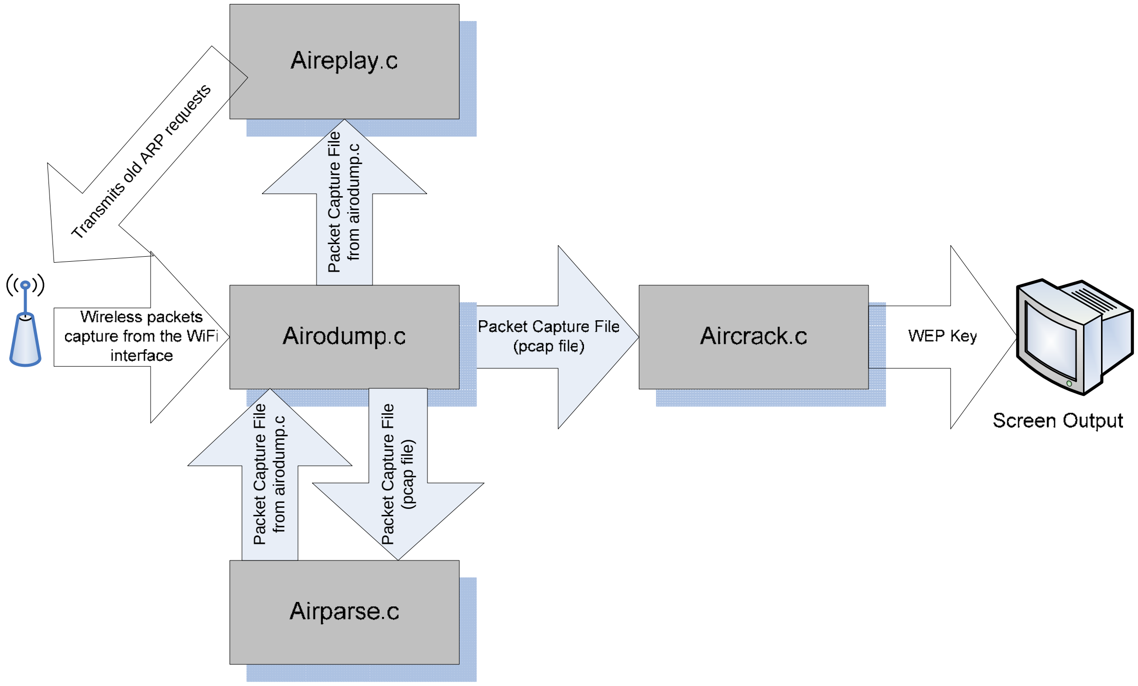

Aireplay.c
Transmits old ARP requests
Packet Capture File from airodump.c
Wireless packets capture from the WiFi interface
Airodump.c
Packet Capture File (pcap file)
Aircrack.c
WEP Key
Screen Output
Packet Capture File from airodump.c
Packet Capture File (pcap file)
Airparse.c

Figure 10 – Aircrack Component Interaction Diagram

Wireless packets are captured through the WiFi interface and dumped by airodump.c into a pcap file. This pcap file is then passed to airparse.c to filter out only those packets that have the potential to bring the cracking process to success. airparse.c then passes this new pcap file to the airodump.c component which then passes the file to aircrack.c. In cases where the traffic is too low to pick up the needed volume of packets, aireplay.c will read the pcap file and replay ARP requests from it to generate more traffic. airodump.c will then pick up the responses to these requests and dump more traffic to pass on to aircrack.c. Finally, once aircrack.c has finished performing the cracking operation, it displays to the screen the WEP key (or displays a message telling that the key could not be recovered).

2.4 - Architecture and Design Considerations

The architecture of Aircrack resembles most to a specialized pipe-and-filter architecture, or more specifically a batch-sequential system. This architecture is established by the fact that each component depends on the other's output, and some components in the architecture can be replaced as long as they provide the required input to the components it is connected to. For example, airodump.c can be replaced by any network sniffer, as long as a pcap file is produced. The only components that could not be replaced are aircrack.c itself and airparse.c as airodump.c and airparse.c are too tightly coupled. This pipe-and-filter system is working on an extremely small project relative to the kind of software architectures that make use of this schema.

The design of the software itself was very low quality according to the standards of design practice in the software engineering field. The entire program was coded in C, therefore practices of object-oriented design do not apply. Variables used in the program were all declared globally, so coupling was very high and locality of functions and variable use was impossible to discern without the use of sophisticated reverse engineering tools.

This program would definitely not be able to handle changes well since the design exhibits poor cohesion and high coupling. For example, adapting to a new chipset for capturing wireless network traffic would entail changes throughout the entire implementation of the software (ripple effect). Redesigning this software, therefore, would be a difficult task.

Part 3 - Design and implementation

The previous part of the report was describing the general architecture of Aircrack. This part is going into the cracking system with much more details.

3.1 - General idea of cracking

The method is based on the FMS attack. 17 attacks are implemented up to now. It consists mainly in finding, thanks to assumptions of the FMS attack, the likelihood of a byte to appear at a given position.

Each attack will perform its calculations and according to the result, it will vote or not for the byte being evaluated. The vote is symbolized by incrementing a counter (an integer) by one.

For a given byte of the key, some of the 256 possibilities will be rated. Please note that the byte possibility being evaluated is found by the program; it is not a random value and aircrack is not performing a brute force attack.

After the votes, hypothesis are sorted for each byte of key. Combinations are tested to try a supposed secret key by decrypting random packets with the secret key found.

In fact, this process is done dynamically, that is some votes are calculated and even if there are only a few results some keys might be tried. In case of success, the process is much faster than waiting for complete results. In case of failure, vote’s calculation goes on.

3.2 - aircrack.c details

In this section we’ll explain what are the methods and algorithms contained in aircrack.c.

However, understanding such algorithms might be difficult without having a look at variables, their structure and what data they store. This will be explained in the section of the function using the given variable. No mention will be done about variables declaration as it is usually done in the main and used globally.

3.2.1 - Function call graph

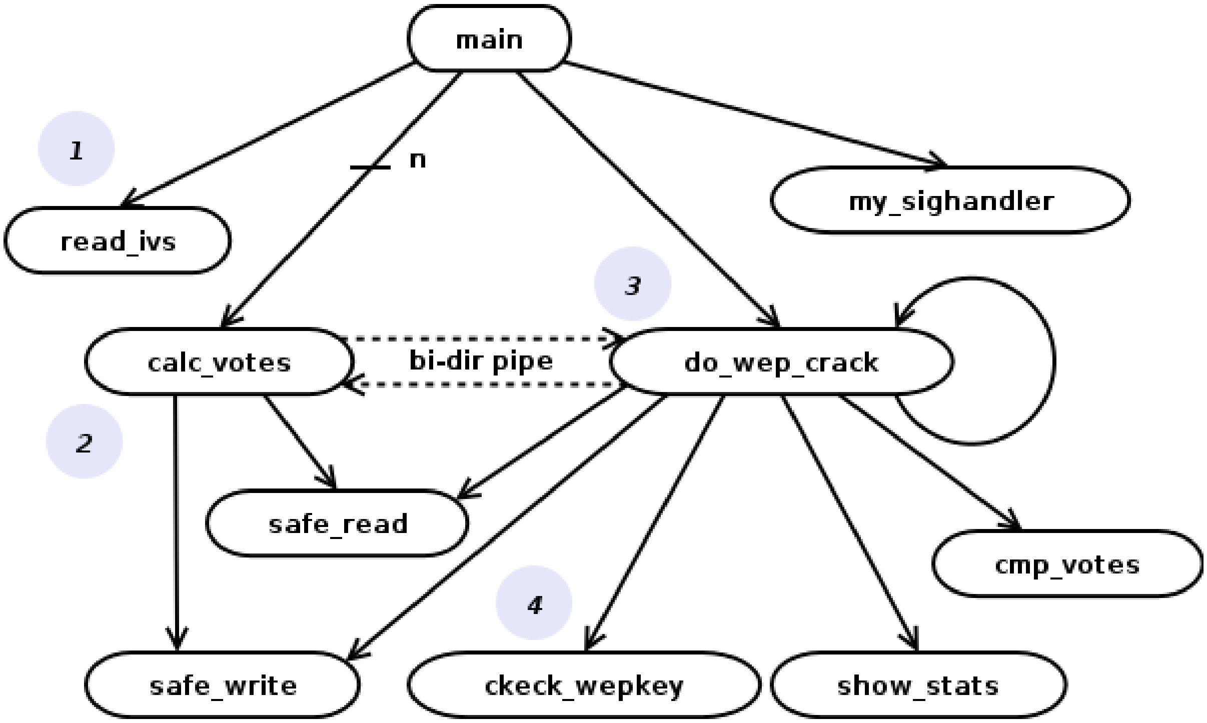


Figure 11 - Function call graph of aircrack.c

3.2.1.1 - Order of call of main functions

This graph explains what are the methods of aircrack.c and in which order they are called. The methods are detailed in the following sections. `read_ivs` is the first method called; next `calc_votes` can be called *n* times, *n* representing the number of child processes instantiated by the main. This number can be input by command line, and is 1 by default.

The function `do_wep_crack` is called for each byte of the key, and controls the different `calc_votes` processes which execute a calculation. Communication between the 2 methods is achieved through pipes, 2 pipes for the 2 directions. `do_wep_crack` is called recursively on the next byte key as long as they key length is not reached.

Finally `check_wepkey` is called each time `do_wep_crack` wants to test if a key combination is correct or not.

3.2.1.2 - safe_read and safe_write

Those functions are used to ensure that the communication throughout pipes is correct. Basically it reads in the pipe and checks the number of bytes read. That’s why both `calc_votes` and `do_wep_crack` use it.

3.2.1.3 - cmp_votes and my_sighandler

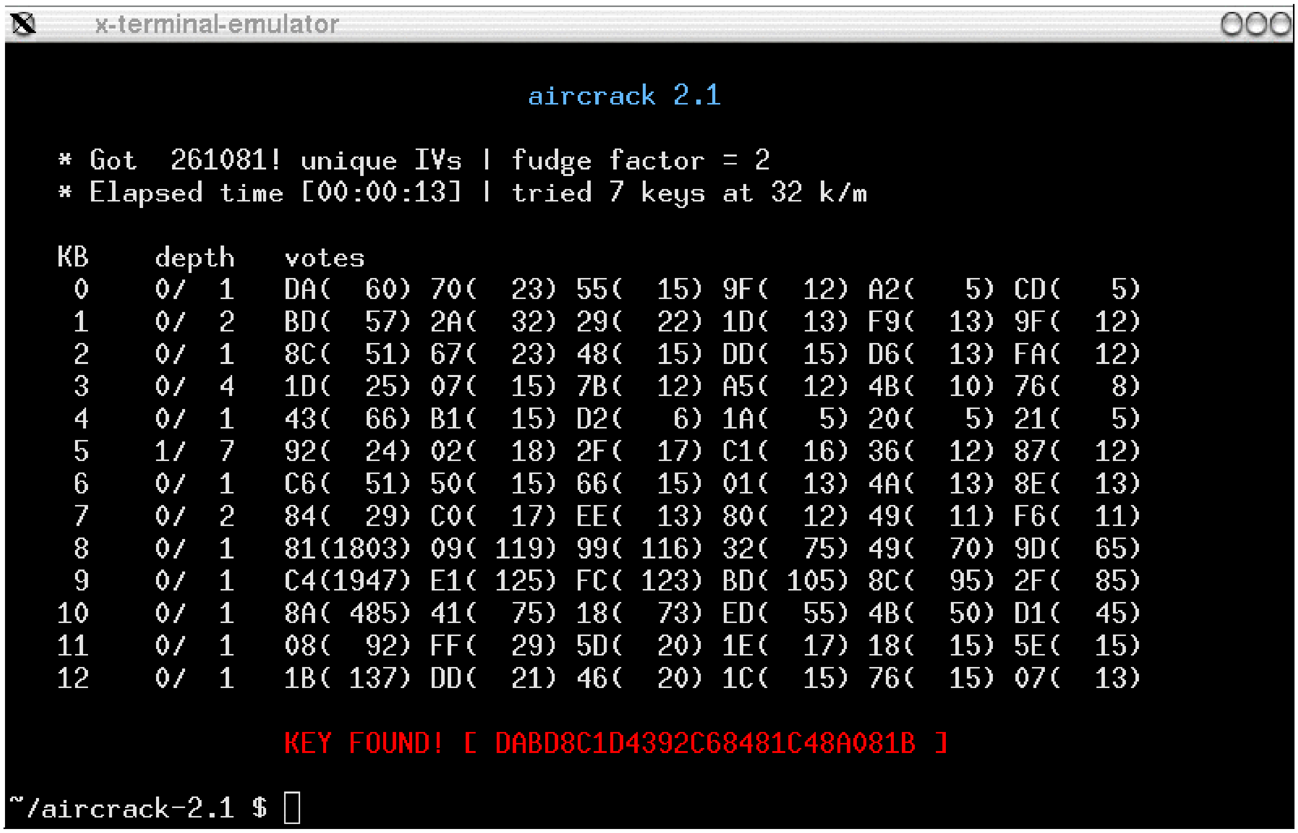
`cmp_votes` is a very basic function to compare 2 numbers and return =, < or >. The reason of the creation of a routine for this task is that the field to compare is part of a structure, so the function is accessing it directly.

`my_sighandler` is responsible for catching terminating signals (like control-C for example). In this case it exits the program more neatly than interrupting the process by force.

3.2.1.4 - show_stats

This function basically displays all relevant information gathered by the program in a terminal. Here is the content of the screen, going left to right, top-down:

- number of unique IVs
- fudge factor
- running time since the loading of the file
- statistics about the speed of the program, number of key tried
- KB: key byte
- depth of the search achieved over the number of branches estimated
- votes: byte hypothesis and number of votes for this byte
- and very often the secret key



```
                                aircrack 2.1

   * Got  261081! unique IVs | fudge factor = 2
   * Elapsed time [00:00:13] | tried 7 keys at 32 k/m

   KB    depth   votes
    0    0/  1   DA(  60) 70(  23) 55(  15) 9F(  12) A2(   5) CD(   5)
    1    0/  2   BD(  57) 2A(  32) 29(  22) 1D(  13) F9(  13) 9F(  12)
    2    0/  1   8C(  51) 67(  23) 48(  15) DD(  15) D6(  13) FA(  12)
    3    0/  4   1D(  25) 07(  15) 7B(  12) A5(  12) 4B(  10) 76(   8)
    4    0/  1   43(  66) B1(  15) D2(   6) 1A(   5) 20(   5) 21(   5)
    5    1/  7   92(  24) 02(  18) 2F(  17) C1(  16) 36(  12) 87(  12)
    6    0/  1   C6(  51) 50(  15) 66(  15) 01(  13) 4A(  13) 8E(  13)
    7    0/  2   84(  29) C0(  17) EE(  13) 80(  12) 49(  11) F6(  11)
    8    0/  1   81(1803) 09( 119) 99( 116) 32(  75) 49(  70) 9D(  65)
    9    0/  1   C4(1947) E1( 125) FC( 123) BD( 105) 8C(  95) 2F(  85)
   10    0/  1   8A( 485) 41(  75) 18(  73) ED(  55) 4B(  50) D1(  45)
   11    0/  1   08(  92) FF(  29) 5D(  20) 1E(  17) 18(  15) 5E(  15)
   12    0/  1   1B( 137) DD(  21) 46(  20) 1C(  15) 76(  15) 07(  13)

                 KEY FOUND! [ DABD8C1D4392C68481C48A081B ]

~/aircrack-2.1 $
```

Figure 12 - Screenshot of aircrack.c

3.2.2 - read_ivs

3.2.2.1 - Operation perfomed

This function is in charge of extracting wireless packets from the capture file, and building an array of data used for the cracking.

Basically, it opens the pcap file (file containing all captured packets), then performs some checks about the header of this file to ensure the proper format of the capture. The second stage is done as long as there are packets to read in the file:

- it checks the packet is encrypted
- it filters packets according to the MAC address given in input if there is one
- it checks if the packet's IV already exists or not
- it adds the IV and the first 2 encrypted bytes to the `ivbuf` variable

3.2.2.2 - ivbuf

The one dimensional array `ivbuf` is filled by the function `read_ivs` as follows:

iv_0	iv_1	iv_2	eb_0	eb_1	iv_0	iv_1	iv_2	eb_0	eb_1	…	iv_0	iv_1	iv_2	eb_0	eb_1
from 1st packet					from 2nd packet					…	from last packet				

Table 1 - ivbuf variable

This array contains the initialization vector from each packet (3 bytes) and the first 2 encrypted bytes of the payload. The IV can be recovered easily as it is unencrypted in the packet. The first 2 encrypted bytes are positioned just after the header of the packet, more especially after the KeyID.

This is done for each valuable packet found by `read_ivs`.

Below is the code explaining the affectation of the variable `ivbuf`:

```
        ivbuf[nb_ivs * 5    ] = h80211[z    ];
        ivbuf[nb_ivs * 5 + 1] = h80211[z + 1];
        ivbuf[nb_ivs * 5 + 2] = h80211[z + 2];
        ivbuf[nb_ivs * 5 + 3] = h80211[z + 4];
        ivbuf[nb_ivs * 5 + 4] = h80211[z + 5];
```

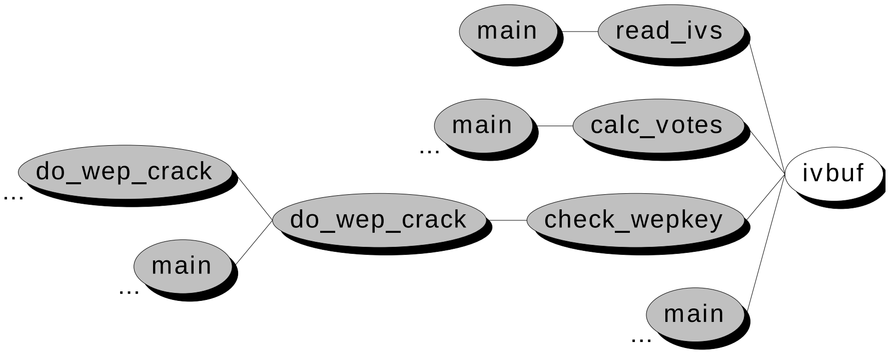


Figure 13 - ivbuf caller tree

`ivbuf` is used in all functions of aircrack.c. It is storing data which is heavily used in the cracking process.

3.2.2.3 - nb_ivs , xv

As you can see, the variable `nb_ivs` is used to shift in `ivbuf` and recover each structure {IV, first 2 encrypted bytes}. It is the number of IVs in the array. As the structure is 5 bytes long, it is multiplied by 5, and the offset enable to recover a specific byte.

`xv` is also used to navigate through `ivbuf` array.

3.2.3 - calc_votes

3.2.3.1 - Operation performed

This function calculates votes for each byte hypothesis. It performs the attacks on a specific byte of the key. 17 attacks are implemented, based on the KoreK attacks and on FMS principles.

As said before, the communication with the parent process is done through pipes. That’s why the first operation done by calc_votes is to read the current state of the secret key array from the pipe. As permutations are done in the algorithm, to reverse it you need at least the beginning of the key, and on which key byte the function is working.

Then it goes through a decryption process, therefore using KSA and PRGA. It performs a number of iteration equal to the number of key bytes discovered. It goes to the ‘unknown boundary’ and can start the cracking there.

The cracking, in fact the votes of each attack, consists of tests according to the current state of the key array and state array. If the condition is fulfilled, the vote counter is incremented.

This function uses the data structure `votes`, explained in the next subsection.

3.2.3.2 - N_ATTACKS, KoreK_attacks

These variables are useful to understand the way the program runs.

`N_ATTACKS` is a constant representing the number of attacks implemented in aircrack. This number increased release after release, as more attacks were found. There are currently 17 attacks.

`KoreK_attacks` is an enumeration holding all possible attacks. The letter ‘*A*’ means attack, *u* means unstable, *s* means stable, and the number next to the letter represent the probability of success of the attack (15%, 13% or 5%, please refer to page 3 and 4 of ref [10] for explanations of these values). In fact it is closely linked to the probability that the value won’t swap during RC4.

```
#define N_ATTACKS 17

enum KoreK_attacks
{
    A_u15,                      /* semi-stable  15%             */
    A_s13,                      /* stable       13%             */
    A_u13_1,                    /* unstable     13%             */
    A_u13_2,                    /* unstable ?   13%             */
    A_u13_3,                    /* unstable ?   13%             */
    A_s5_1,                     /* standard      5% (~FMS)      */
    A_s5_2,                     /* other stable  5%             */
    A_s5_3,                     /* other stable  5%             */
    A_u5_1,                     /* unstable      5% no good ?   */
    A_u5_2,                     /* unstable      5%             */
    A_u5_3,                     /* unstable      5% no good     */
    A_u5_4,                     /* unstable      5%             */
    A_s3,                       /* stable        3%             */
    A_4_s13,                    /* stable       13% on q = 4    */
    A_4_u5_1,                   /* unstable      5% on q = 4    */
    A_4_u5_2,                   /* unstable      5% on q = 4    */
    A_neg                       /* helps reject false positives */
};
```

3.2.3.3 - votes

The variable `votes` is used to store the number of votes performed by each attack in favor of a given byte hypothesis, for a specified byte of the key. This means that this array is used for cracking one byte only. For a 13 bytes long key, this array will be used 13 times.

As you can see in the table, attacks come from the `KoreK_attacks`. The bytes are the possibilities for a byte to be the key at a precise position. Only a few examples of votes are given in the table, but usually the whole table is filled, by 0 if there have been no votes.

	Byte hypothesis **256 possibilities**							
Index *(dec)*	0	1	2	…	78	…	254	255
Byte *(hex)*	00	01	02	…	4E	…	FE	FF
A_u15								
A_s13								
A_u13_1								
A_u13_2								
A_u13_3								
A_s5_1								
A_s5_2								
A_s5_3				…	…	…	…	…
A_u5_1				…	24	92	19	…
A_u5_2				…	86	11	145	…

A_u5_3				…	13	73	20	…
A_u5_4				…	5	24	37	…
A_s3				…	78	148	13	…
A_4_s13				…	…	…	…	…
A_4_u5_1								
A_4_u5_2								
A_neg								

Table 2 - votes variable

`votes` is used in `do_wep_crack` and `calc_votes`.

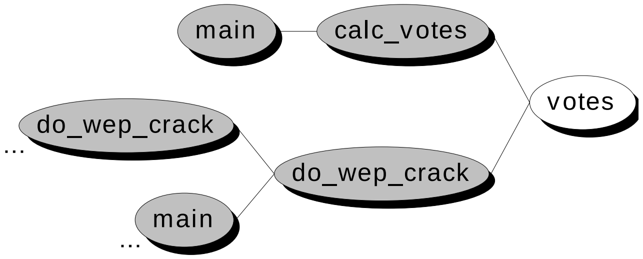


Figure 14 - votes caller tree

3.2.4 - do_wep_crack

3.2.4.1 - Operation performed

This function organizes the byte per byte cracking through calc_votes childs.

1. it sends the current byte key and state array to each child through pipes
2. it recovers the arrays `votes` for each key byte through pipes
3. it compensates the votes with a stability factor; this factor is the level of confidence mathematically derived from the attacks. Some attacks are known to be 5% sure, some others 13% or 15%.
4. it sorts votes, enabling to choose the most likely candidate
5. it deepens the attacks as much as necessary or specified
6. if the total key is recovered (5 or 13), `check_wepkey` is called
7. otherwise, calls itself recursively to crack the next byte key

Only the main (and most complex) steps will be explained, as well as the major variables.

3.2.4.2 - wpoll

This 2 dimensions array contains the structure `byte_stat`. `byte_stat` is basically 2 members, `index` and `votes`.

`wpoll` contains the votes for all byte hypotheses for all bytes in the key. Those votes are sorted, so the column structure used in variable `votes` is no more possible, as column index need to be different for each byte key.

The number of lines depends on the length of the key.

As a drawing is always clearer than a long text, here is the structure of the variable.

		Byte hypotheses 256 structures sorted by votes							
		0	1	2	…	78	…	254	255
Key bytes (5, 13 or more...)	0	index: F0 votes: 241	index: 7B votes: 124	index: 23 votes: 37	index: ... votes: …	…	…	…	…
	1	index: AB votes: 397	index: C1 votes: 258	index: 1D votes: 54	index: ... votes: …				
	2								
	3								
	4								
	5								
	6								
	7								
	8								
	9								
	10								
	11								
	12								

Table 3 - wpoll variable

It is exactly the data displayed on the screenshot of Figure 12. The first column contains the key bytes having the biggest number of votes. It is therefore likely that it is the secret key, especially if the number of votes is very high.

3.2.4.3 - coeff_attacks

The enumeration `KoreK_attacks` stress that the confidence in the attacks are different. This should be reflected in the voting process. This is done thanks to the `coeff_attacks` variable.

```
int coeff_attacks[4][N_ATTACKS] =
{
    { 15, 13, 12, 12, 12, 5, 5, 5, 3, 4, 3, 4, 3, 13, 4, 4, 0 },
    { 15, 13, 12, 12, 12, 5, 5, 5, 0, 0, 0, 0, 3, 13, 4, 4, 0 },
    { 15, 13,  0,  0,  0, 5, 5, 5, 0, 0, 0, 0, 0, 13, 0, 0, 0 },
    {  0, 13,  0,  0,  0, 5, 0, 0, 0, 0, 0, 0, 0,  0, 0, 0, 0 }
};
```

It stores the probability of success of each attack (in the row). There are 4 rows because it is possible to adjust the ‘stability’ of the attacks. This parameter value is 0 by default and can be input by command line (with ‘-s’). It cancels some attacks by affecting a factor of 0. However there is no mention whatsoever of this parameter in the help or in the documentation. It seems to be a hidden parameter used for testing purposes.

3.2.4.4 - Eventually... explaining step 3

The variable `wpoll` is therefore the combination of the `votes` recovered by `calc_votes` childs and the balancing of these votes by `coeff_attacks`.

```
  for( i = 0; i < 256; i++ ) {
     for( n = 0; n < N_ATTACKS; n++ ) {
        wpoll[B][i].votes += coeff_attacks[stability][n] * votes[n][i];
     }
     wpoll[B][i].votes -= 20 * votes[A_neg][i];
  }
```

The last line is a safekeeping of false positive results of attacks. It lessens a lot the likelihood of the key byte depending how many votes the `A_neg` attacks gave.

This piece of code is the main calculation of the votes.

3.2.4.5 - Step 5: brute force control

Attacks are perfomed from some input packets, but not all. A limit should be determined for the sufficient number of votes which enables to determine the secret key. That’s what the fudge factor is in charge of controlling.

A fudge factor (variable `fudge[]`) is determined for each key byte. The fudge threshold, `ffact`, is given in command line. Its value is 2 by default.

```
    for( fudge[B] = 1; fudge[B] < 256; fudge[B]++ )
        if( wpoll[B][fudge[B]].votes < wpoll[B][0].votes / ffact )
            break;
```

It basically iterates until it finds a vote which is `ffact` times smaller than the biggest number of votes. The number of iterations gives you the fudge factor of the key byte. As `do_wep_crack` is called on each key byte, a different fudge factor is found for each key byte.

This fudge factor gives the limit of the deepness to go to during the search. As you can see with this iteration:

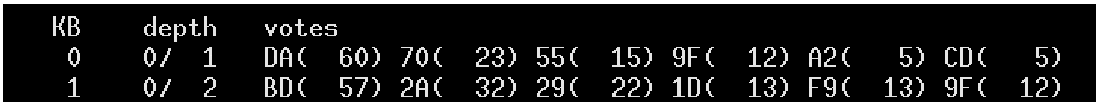
```
    for( depth[B] = 0; depth[B] < fudge[B]; depth[B]++ )
    {
        constructs the key from results
        cracks next key byte or checks the key
    }
```

The depth of the search will not exceed the fudge factor. The body of the loop consists of taking the most likely key byte solution, append it to the key, check if it gives a good secret key size and check the validity of this key.

It is possible to exit the loop before going to the maximum deepness if the key if found.

If the loop ends naturally it means that the key has not been found, and all permitted paths have been unsuccessful.

As you can see in the screenshot of Figure 12 or below, the column indicating ‘depth’ displays the depth level achieved out of the depth permitted (the fudge factor).



```
 KB    depth   votes
  0    0/  1   DA(  60) 70(  23) 55(  15) 9F(  12) A2(   5) CD(   5)
  1    0/  2   BD(  57) 2A(  32) 29(  22) 1D(  13) F9(  13) 9F(  12)
```

A closer look enables to check the calculation of the fudge factor. The first key byte has a vote of 60 for DA solution. The next solution has 23 votes, so less than the half of the first one. Therefore the fudge factor is 1. Naturally it means: “given the fact that there are many votes for this solution, it doesn’t worth going further than the first vote”.

The second key byte has votes of 52, 32, 22. 32 is not less than half of 52, so the iteration would go further. 22 is less than half of 52, so the fudge factor value is 2. We invite the reader to check the process with the other lines of the screenshot of Figure 12, even if we won’t go on in this report.

3.2.5 - check_wepkey

When a hypothesis of the key is found, how to check it is the valid secret key?

The principle is very easy. Take encrypted data, decrypt it with the key and check the result with the plaintext (that you already know). If it matches the key is correct, otherwise it’s not.

As you might have figured out, the SNAP header will be used again as the plaintext.

There are the steps of the checking:

- take a random encrypted packet (in fact structure IV + 1st 2 encrypted bytes, which are the encrypted form of the SNAP header)
- decrypt it thanks to the IV and the hypothetical secret key
- check if it equal 0xAA (value of SNAP header)

This process is done on multiple packets in order to ensure it’s not just luck.

Above 4 matches aircrack considers the key is correct.

Conclusion

Afterward

This report focused on the main component of aircrack, the cracking process. Other programs are easier to understand, as it is always the same kind of packet manipulation.

The report was written in a top down way, and following the opportunistic model, as during our research and reverse engineering process.

One of the tools which helped us a lot is the couple aiCall / aiSee. The first one generates flow graphs and the second one displays the graph. The flow control is there well displayed and enables to follow the different edges (true or false condition, breaks, exiting statements, calls…). Another useful tool when looking at the graph is CCRider. This program gives much information on all symbols (functions, variables, constants, etc). You easily know how they are declared, and where they are read / killed.

However all these tools cannot make the developer or maintainer escape the best and oldest technique in reverse engineering, which is code reading. It is sometimes long and hard, and some pieces of code need to be read ten times, but it certainly helps to turn gibberish code into design, architecture, and specifications.

What about WEP?

Efforts are underway within the wireless community to address these outstanding security issues. The 802.11 committee has formed its "i" subcommittee to explore the problems with the WEP protocol and to correct them.

The temporal key integrity protocol (TKIP) is an interim solution already being pushed by the 802.11i committee. Under TKIP, the client starts with a 128-bit "temporal key" (TK) that is then combined with the client's MAC address and with an initialization vector to create a key that is used to encrypt data via the RC4. Temporal keys are changed every 10,000 packets, addressing the glaring key reuse issue in WEP and making WLAN systems considerably more secure.

TKIP systems are expected to be available during the second quarter of 2002, and will be backward compatible with existing 802.11b hardware. While TKIP represents a step forward, it is widely seen as a stopgap measure that has not been widely tested. Many industry experts are pointing the recently adopted Advanced Encryption Standard (AES) protocol as the best long term security option. AES equipped wireless LAN hardware should be available in 2003.

Another alternative is to use IPsec for wireless security. IPsec is the Internet Engineering Task Force (IETF) protocol for providing encryption and authentication for IP traffic. IPsec is characterized by a strong key exchange, or "handshake" function, that is used to establish a connection and strong data encryption and authentication that ensure both confidentiality and integrity.

The IPSec protocol includes two main parts: a key-exchange function (a "handshake") that determines how the two sites will communicate, and a bulk encryption/authentication function that manages the encryption and authentication of the actual data being communicated.

WLAN LAN systems provide tremendous flexibility to corporate users. However, wireless users should not make the mistake of believing that any WEP-based security option provides anything more than a way of keeping the honest people out. The bottom line is that there will never be a fully secure network protocol, because each time that a new protocol is developed there will be those that will try and find a way to crack the system, just like AirCrack manages to crack an 802.11b/g wireless network. Eventually there will be a program that will crack the new standard 802.11i.

References

[1] Interesting code: http://www.netstumbler.org/showpost.php?p=89036&postcount=11

[2] Information and algorithms, sources: http://www.cs.miami.edu/~burt/learning/Csc524.052/

[3] The BIBLE for RC4: http://www.airscanner.com/pubs/wep.pdf

[4] WEP generalities: http://www.securityfocus.com/printable/infocus/1814 (part 1) - http://www.securityfocus.com/printable/infocus/1824 (part 2)

[5] FMS attack: "Weaknesses in the Key Scheduling Algorithm of RC4" by Scott Fluhrer, Itsik Mantin and Adi Shamir: http://citeseer.ist.psu.edu/rd/20942602%2C474384%2C1%2C0.25%2CDownload/http://citeseer.ist.psu.edu/cache/papers/cs/24255/http:zSzzSzwww.cs.rice.eduzSz%7EastubblezSzwepzSzwep_attack.pdf/stubblefield01using.pdf

[6] WEP vulnerabilities: http://www.phptr.com/articles/printerfriendly.asp?p=102230

[7] More on how WEP works: http://www.isaac.cs.berkeley.edu/isaac/wep-faq.html

[8] The RC4 Encryption Algorithm: http://www.cebrasoft.co.uk/encryption/rc4.htm

[9] FMS, cracking: http://sorbo.darkircop.org/wlan/sorwep.txt

[10] Forum with KoreK and devine: http://www.netstumbler.org/showthread.php?t=12489&page=1&pp=15

[11] 802.2 LLC: http://ckp.made-it.com/ieee8022.html

[12] SNAP: http://www.ecsl.cs.sunysb.edu/cse591/RFMon.pdf

[13] Theory about 802.11 (wlan) : http://www.cs.pdx.edu/~jrb/netsec/lectures/80211/theory.txt

[14] MAC wireless packet structure : http://www.irean.vt.edu/courses/ececs4570/content2004/lecture07_MAC.pdf

[15] Theory about 802.2/802.3 (ethernet) : http://www.inetdaemon.com/tutorials/lan/ethernet/frame_format.html

[16] 802.2 Theory : http://www.usc.edu/dept/engineering/eleceng/Adv_Network_Tech/Html/datacom/sld024.htm