The Fragmentation Attack in Practice

Andrea Bittau*
a.bittau@cs.ucl.ac.uk

September 17, 2005

Abstract

The 802.11 encryption standard *Wired Equivalent Privacy* (WEP) is still widely used today despite the numerous discussions on its insecurity. Although WEP definitely faces serious security problems, there is no single tool which will recover any WEP key with minimal effort from the user and in a very short amount of time.

In this paper, we present a mechanism which allows an attacker to send *arbitrary* data on a WEP network after having eavesdropped a *single* data packet. Many common WEP attacks require gathering large amounts of data before they may be performed whereas ours requires only one, making it much quicker and more practical.

We implemented a fully automatic version of this attack which enables even non technical people to perform it and recover a key without effort and in a relatively short period. Hopefully this will induce people to abandon WEP as their wireless security solution—it is no longer the case that only skilled and patient attackers may recover the key in practice.

1 Introduction

Everyone knows that WEP is broken but only a minority manage to recover keys successfully in practice. The reason for this is partly because most of these attacks require long waiting times or a more skilled use of various tools. As many believe that it is unlikely for an attacker to spend hours around their network, people prefer to adopt WEP rather than seeking more sophisticated and possibly more difficult to manage security solutions. This will no longer be the case as with the fragmentation attack, hours become minutes.

By far the most famous WEP attack is the one related to weak IVs formally described in [5]. Interestingly enough, these weak RC4 keys were first noted in 1995 [15] before the 802.11 standard [8] became available in 1999! It is clear that WEP was designed without too much thought or research effort. The weak IV attack relies on statistical data and calculates which key is the most probable. Each guess made for a particular key byte is correct $\approx 5\%$ of the time. For this reason, it is necessary to collect a large volume of statistics (data packets in this case) before a candidate key byte reaches a reasonable probability of being correct. In practical scenarios 1,000,000 data packets could be necessary for a full key to be recovered. If following a link of a WWW page causes 100 packets of traffic, a user on the network may need to view 1,000 links before the WEP key is potentially broken—not an immediate task!

Following the publication of the weak IV attack numerous attempts to improve it were made. An example is that of [6] which generalizes the attack to include multiple output bytes. The weak IV attack relies on the first clear-text byte of each data packet to be known—an easy requirement as most packet headers commence with constant values. The generalized attack utilizes the second and third byte too. The improvement is not drastic as there are only a few weak IVs for additional out-

*University College London, Computer Science Dept.

put bytes and the probability they yield for a key byte being correct is low.

Methods for guessing key bytes with 13% chance of success are mentioned in [14] although not thoroughly explained. Possibly these have been used in private implementations (the author of this paper had one [3]). A description of these 13% probability attacks, including many more, was finally published in [10] and implemented in the `aircrack` [4] tool. This was a drastic improvement—only $\approx 500,000$ data packets are now required.

However, many real wireless networks are rather "silent" requiring the attacker to have much incentive in order to passively gather large amounts of data for a long period of time. The most common optimization is the replay attack. An *Address Resolution Protocol* (ARP) request, identifiable by its length and broadcast destination, is eavesdropped and replayed on the network. This will cause an ARP reply to be generated. The attacker may therefore flood the network and capture the resulting traffic. The only problem with this attack is the rate with which data can be injected and the rate with which traffic is created. Many implementations require two wireless cards—one for transmission and one for reception. This attack is quite effective although the user must have some knowledge on how to perform it—it is not *totally* trivial (unless a sophisticated tool is developed).

An entirely different approach in breaking WEP is illustrated and used by the `WEPWedgie` [12] tool. Here, the attacker is able to transmit arbitrary data without knowing the key. The requirement for accomplishing this is that the network must use shared-key authentication, which is rarely used today. The attack presented in this paper uses a similar strategy to `WEPWedgie` although the requirement is much more practical—a single data packet of any type needs to be eavesdropped.

The rest of this paper is organized as follows. In Section 2 the minimum required background knowledge on WEP is presented. Following that, the fragmentation attack is described in Section 3. The description will include how to initiate the attack and different ways in which the attack may be completed, either by recovering the key or decrypting data via other means. Section 4 analyzes the difficulties, if any, in implementing the attack in practice. It includes information on the hardware used and the description of the tool developed by the authors. The results (such as cracking time) obtained from running this tool are presented in Section 5. Finally we conclude in Section 6.

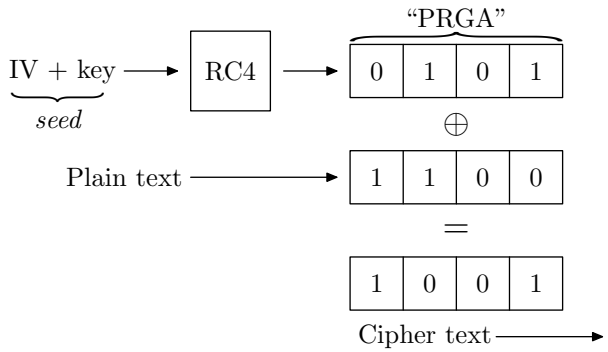

Figure 1: WEP encryption.

2 WEP Operation

WEP is a stream cipher used to encrypt the data portion of data packets in 802.11 networks. The 802.11 header is always in clear-text and both management and control frames are entirely transmitted unscrambled. WEP is built on top of RC4 [13] and the 802.11 standard specifies a 40-bit pre-shared key, although 104-bit keys are commonly used today. A *seed* is constructed by prepending a 24-bit *initialization vector* (IV) to the secret key. This seed is used to setup RC4's state via the *Key Scheduling Algorithm* (KSA). The output of RC4's *Pseudo Random Generation Algorithm* (PRGA) is XOR-ed with the clear-text in order to produce the cipher-text. The pseudo-random sequence produced by RC4 will be referred to as PRGA throughout the rest of this paper. The whole process of WEP encryption is summarized in Figure 1. Decryption is performed in an analogous way by performing a XOR of the cipher-text with the PRGA.

When encrypting, a checksum (CRC32) of the

data is appended to the message body and is encrypted with the rest of the data. This checksum allows the the decrypting process to verify whether decryption was successful.

The IV chosen for producing RC4's seed is prepended in clear-text in the payload being sent. Normally, each data packet will have a different IV and in practice implementations tend to use an incremental counter for their IV generation. The weak IV attack operates by eavesdropping particular IVs which reveal information about key bytes from the KSA. Some modern implementations of WEP tend to filter out these IVs and not use them in order to make attacks more difficult.

2.1 PRGA

Each different IV will produce a different PRGA as the RC4 seed will have changed. A single secret key will have 2^{24} different PRGAs. In essence, WEP encryption and decryption is a XOR with one of these PRGAs. In order to decrypt a packet, the PRGA produced from the IV specified in the packet must be XOR-ed with the clear-text. To encrypt a packet, the requirements are more relaxed—XOR the clear-text with *any* PRGA. It is enough to have a single PRGA and keep transmitting each time using that same IV which generated the PRGA.

XOR (denoted as $\oplus$) is a symmetric operation. That is, clear $\oplus$ PRGA = cipher, and cipher $\oplus$ PRGA = clear. Furthermore, cipher $\oplus$ clear = PRGA. Therefore, one possible way of recovering PRGA is by knowing the cipher-text and clear-text and by performing a XOR operation on the two. The cipher-text could easily be obtained by eavesdropping a packet from the network. If the PRGA is recovered from that cipher-text then transmission may occur by always using the same IV used in the captured packet. That PRGA may also be used to decrypt future packets which use that IV. However the problem of calculating the PRGA from the cipher-text is knowing the clear-text.

In Shared Key authentication, the *Access Point* (AP) sends a 128 byte challenge to the client. The client responds with the encrypted version of the challenge in order to authenticate. By eavesdropping this transaction, 128 bytes of PRGA may trivially be obtained—both the clear-text and cipher-text traverse the network! This is exactly the vulnerability `WEPWedgie` exploits. The 802.11 standard actually describes this scenario and discourages implementations to re-use the IV used in the authentication transaction in order to prevent attackers from decrypting future packets which happen to use that particular IV. What it fails to mention is the fact that transmission with that same IV may occur indefinitely from an intruder.

An interesting aspect of this attack is that the key is never known nor recovered. Many ciphers rely on the complexity of the key for their security. This attack may be launched with the same simplicity no matter how long and complex the key is. In fact, it is totally independent of the KSA and from RC4 in general—it would work even if a stream cipher other than RC4 was used to generate the PRGA.

An extension the `WEPWedgie` tool provides is transmitting data using the source IP address of an Internet host the attacker controls. This way the attacker may receive data on that host and obtains a 2 way communication channel. It may be useful in order to circumvent firewalls—perhaps the network is firewalled from the Internet, but not from the wireless.

The `WEPWedgie` attack will only work on networks with Shared Key authentication which are almost extinct today. The fragmentation attack is an attempt use the approach of `WEPWedgie` in all wireless networks and not be limited only to the ones which use Shared Key authentication.

3 The Attack

In order to recover PRGA, clear-text must be known. How much clear-text is know in all packets? Each 802.11 data packet is *Logical Link Control* (LLC) encapsulated. The LLC header is virtually always followed by *Sub-Network Access Protocol* (SNAP). The LLC/SNAP header, illustrated

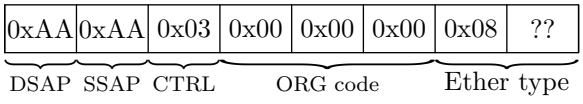

Figure 2: LLC/SNAP header contained in practically all 802.11 data frames.

in Figure 2, is 8 bytes in length and contains (almost always) constant fields except for the Ethernet Type which defines the protocol which follows in the stack. The two common candidates for this field are IP or ARP. The value for both of these protocols commence with `0x08` making the first 7 bytes of clear-text always known with a very high degree of probability.

If IP packets were distinguishable from ARP packets, then 8 bytes of PRGA would be known. Deciding whether a given packet is IP or ARP generally is not too difficult in practice. All ARP packets have a fixed length of $8 + 8 + 20 = 36$ (the size of LLC/SNAP + ARP header + ARP data) and are broadcast in the case of ARP requests. Any packet which has a larger size than this may safely be considered an IP packet. There is an exception. Some hardware seems to pad ARP packets (or short packets in general) to 54 bytes. In such cases, packets larger than 54 bytes can be considered to be IP and broadcast packets of 54 bytes to be ARP. Anyway, it is possible to build a list of hardware which performs this padding and infer whether or not it is present in this network by deducing the manufacturer of the AP from its MAC address prefix [7].

In general, at least 8 bytes of clear-text are known.[1] This means that 8 bytes of PRGA may be recovered after eavesdropping a single data packet (by performing a XOR). At this point, arbitrary data of 8 byte length may be sent. In reality, the scenario is slightly worse. Only 4 bytes of useful data could be transmitted as 32 bits are required for the checksum, which is also ciphered. With only 4 bytes available the attacker could not get very far—the LLC/SNAP header itself is 8 bytes long... not to mention IP!

[1] In reality, more bytes of clear-text are known. For example the IP version and header length is generally constant. In the case of ARP packets, much more is known as the ARP header is virtually all constant.

3.1 Fragmentation

The 802.11 standard defines fragmentation at a MAC layer. During fragmentation, WEP encryption is performed independently on each individual fragment. Nothing stops each fragment from having the same IV. It is therefore possible to send data in chunks of 4 bytes with the 8 bytes of PRGA that has been recovered. The fragment number field is 4 bits allowing a maximum of $2^4 = 16$ fragments. This would allow an attacker to inject $4 \times 16 = 64$ bytes of arbitrary data. A minimal IP packet requires $8 + 20 = 28$ (LLC/SNAP + IP header) bytes of data. It is therefore possible to send $64 - 28 = 36$ bytes of IP data on the network. Note that at this stage IP fragmentation may also be used if more data needs to be sent.

To bootstrap the attack, the attack may discover more PRGA. Since arbitrary data can be injected, a long broadcast frame may be sent, which the AP will then relay. The AP will transmit it as a single frame as it has no incentive to fragment it into tiny chunks. The clear-text is definitely known (the attacker generated the frame) and the relayed frame can be trivially eavesdropped, ultimately revealing a larger portion of PRGA. This may be repeated until a large enough PRGA is obtained, perhaps of a size equal to the *Maximum Transmission Unit* (MTU) after which fragmentation is no longer necessary in order to transmit frames.

3.2 Decrypting Data

Not having the ability to receive data is a great limitation in most cases. There is a generic technique which may be applied in order to decipher an arbitrary packet in a linear amount of time with respect to the packet length.

Suppose that a packet has been eavesdropped and the attacker needs to decrypt it. If the PRGA for the IV used in that packet was known, the clear-text would be recovered by performing a

XOR with the cipher-text. The first 8 bytes of PRGA for that packet are known (as discussed previously) and data may be transmitted using them. For example, a broadcast frame may be sent, and the AP will relay it. Note that if the 8 bytes of the calculated PRGA were incorrect, the AP would *not* relay the packet. An attacker may guess the 9^{th} byte of PRGA and try transmitting a broadcast frame in 9 byte chunks. If the AP relays the frame, the guess was correct. Thus, after at most 256 guesses, the 9^{th} byte of PRGA will be revealed. This means that the 9^{th} clear-text byte may also be calculated (XOR cipher-text and PRGA). The next byte of PRGA may now be guessed and the process continues until the whole frame has been decrypted.

The 802.11 specifications does not allow fragments of odd length (except for the last fragment) but all implementations the authors came across do permit them, therefore making this particular technique feasible. If implementations complied to the specification, the attacker could simply make only the last fragment longer.

A naive implementation of this technique could use a timeout in order to decide whether the AP did not relay the frame because of an incorrect PRGA guess. A quicker implementation could exploit the fact that multicast frames are also relayed and that MAC addresses in the 802.11 header are in clear-text. If the current PRGA guess is `0xAB` a multicast frame destined to `01:00:5E:00:00:AB` (corresponds to the IP address 224.0.0.x) may be transmitted. Effectively all guesses (0–255) may be sent in parallel and only one will be relayed. The correct guess may be read off the multicast destination address in the relayed frame. This technique will reveal the next PRGA byte in the order of seconds making the decryption of an entire packet take only a couple of minutes.

3.2.1 Decrypting ARP Packets

Useful and simple packets to decrypt are ARP packets. The benefit of decrypting them is discovering the IP addresses present in the network.

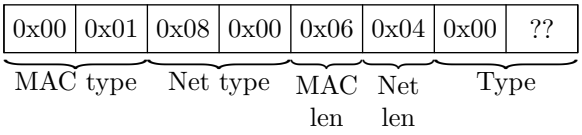

Figure 3: Format of the ARP header. This header is followed by the source MAC and IP addresses.

The format of the ARP header is illustrated in Figure 3. It is followed by the sender's MAC and IP addresses. The first *really* unknown value in an ARP packet is actually the source IP, therefore no time is spent decrypting useless information. The request type is known: if it is a broadcast, it is an ARP request, else it is an ARP reply. The source MAC address is also known—it is transmitted in clear-text in the 802.11 header! Decrypting the 4 bytes of the source IP will take less than a minute and strictly speaking, only 3 bytes are necessary to reveal the network address. Knowing an IP address allows the attacker to "assign" himself an address and communicate with other hosts—potentially, even with the Internet!

3.2.2 IV Dictionary

Another technique may be used to decrypt data in the long run although it is both time and space consuming. Each time the AP relays a frame, it will most likely use a different IV. The attacker may flood the network with broadcast frames and calculate the PRGA for the relayed version of each frame. As these frames will use different IVs the attacker may slowly build an "IV dictionary"—a database of all PRGAs for this particular network. If a packet (encrypted using a specific IV) is sent on the network, the attacker may check whether he obtained the PRGA for that IV and decrypt the packet. If the broadcast frames used in building the dictionary are the size of an MTU, then any frame size may be decrypted for the PRGAs obtained so far. Once the dictionary is complete, the network may be fully used without ever knowing the key!

Considering the Ethernet MTU of 1500 bytes,

the 24 bit IV space will cause the dictionary to be of at least $2^{24} \times 1500 = 25,165,824,000$ bytes ($\approx$ 23.4GB). Also, $2^{24} = 16,777,216$ packets would have to be relayed by the AP. However, hardware which filters out weak IVs may lower these figures significantly—"helping" against one attack, aids in the achievement of another one!

It is possible to build a dictionary with entries containing a PRGA length of less than the size of the MTU. Often the attacker is only interested in the first couple of bytes in a packet which normally contain login data (such as POP3, FTP and telnet credentials). If the attacker wishes to use the Internet connectivity of the wireless network, he may setup a VPN with an Internet server which only uses small packet sizes and route all traffic through that VPN. To compare this approach against a full dictionary, using 128 byte PRGA entries, a dictionary of only 2GB would be required and its construction would be much quicker as smaller packets could be sent at a higher rate.

3.3 Recovering the Key

To recover the WEP key one must rely on the weak IV attack which requires gathering many data packets. Being able to inject arbitrary data may speed this process up a lot. Instead of blindly replaying ARP packets (which is today's most effective technique) an attacker may choose to ping the broadcast IP address. Hopefully this will cause multiple replies to return effectively causing a "smurf" *Denial Of Service* (DoS) attack.

A more adventurous approach would be attempting to contact a host on the Internet. A requirement is obtaining the router's MAC address. Many times this is the MAC address of the AP itself. Another technique would be to rely on heuristics and send an ARP request to IP addresses which terminate with a .1 or .254. The ARP reply is easily detected by its fixed length and the destination MAC address (the attacker's). The source MAC (clear in the 802.11 header) of the ARP reply hopefully will be the router's MAC. Alternatively, the attacker may analyze the traffic of the network and see which destination MAC address is the most popular and infer it being an Internet gateway.

A benefit of causing an Internet host to flood is speed. No longer does the attacker need to generate a packet, wait for an 802.11 ACK and finally wait for the AP to relay it. The attacker simply passively waits for packets transmitted by the AP and ACKs them. It is often difficult to transmit and receive "concurrently" with wireless devices making it hard to quickly inject and receive traffic. This is why many replay attack implementations suggest using two physical wireless cards. If the Internet is flooding, the attacker only needs to send a packet once in a while, potentially to keep the *Network Address Translation* (NAT) connection alive (in the case of private networks). The identity of this host may be hidden by spoofing the source IP address whilst flooding. It will also work with NAT by sending the "flood request" packets to the spoofed host, which for example, may be down or firewalled.

Note that if the weak IV attack is being performed, packets can be minimal in size as only the first 1–3 data bytes are ever used. The number of unique IVs is what matters (essentially the frequency of packets). It is therefore possible even for a low bandwidth Internet connection to cause a high number of packets per second as they may be of minimum size.

3.4 Hybrid Attack

While waiting for enough packets to be captured in order to perform the weak IV attack, it is possible to build the IV dictionary, thus carrying out both attacks concurrently. Although the AP will most likely be using a different IV while it is flooding, other wireless stations may transmit with IVs for which the attacker has already recovered the PRGA, enabling him to decrypt their traffic.

If the Internet is being used as a flood source while building the dictionary, some care must be taken. The *Time To Live* (TTL) field in the IP packet will be different and so will the resulting

IP checksum. Assuming the forward and reverse path are symmetric (same number of hops) it is possible to calculate the change in the TTL. The attacker transmits a packet to the Internet host with a pre-defined TTL making it possible for the host to calculate the change which occurs. The host may encode this delta in the length of a packet it transmits. For example it may transmit a packet of length $x + \Delta\text{TTL}$. The attacker will be able to spot packets destined to itself in this specific length range of $x + \max(\Delta\text{TTL})$. By knowing the change in TTL, the attacker can conclude what the TTL of the received packet should be, since the Internet host also will transmit with a pre-defined value. The attacker can also calculate the new IP checksum and therefore recover the PRGA correctly.

Since MTU length of PRGA normally needs to be recovered, a low bandwidth host is no longer optimal—packets need to be large *and* frequent. The attacker may therefore instruct the Internet host to either send MTU sized packets for a dictionary attack or minimum sized packets for the weak IV attack depending on the network speed. Ultimately the attacker may also send feedback to the Internet host about congestion and flow control in order to avoid packet loss and obtain an optimal flood rate.

If a mechanism for resetting the IV generator existed, only a minimal dictionary would need to be built. Once the IV generator exceeds the last IV the attacker possesses, the generator could be forced to be reset and cause the station to transmit with known IVs. This area has not been researched yet but the most likely way of doing this would be to cause the AP to crash, or maybe even simpler, force clients to disassociate.

4 Implementation

Like all other WEP attacks, this particular one seems great in theory. However, we were keen in seeing it work out in practice. The main hurdle to pass was determining whether the hardware available in the market would allow the sending of raw 802.11 frames. It is well known that many wireless cards support the commonly called "monitor mode" which allows the driver to read all raw 802.11 frames. It is less obvious how to inject data. The first attempt was made using Intersil Prism2 cards [9].

4.1 Prism2 Based Cards

Prism2 cards support "host AP" mode which allows creating an AP in software. The kernel is responsible for sending particular management frames and encapsulating data frames—exactly what is needed for this attack. A Linux driver named `airjack` [11] uses this technique to allow a user to inject (almost) any raw 802.11 frame and receive traffic concurrently. After initial experimentation, it is easy to discover that the Prism2 firmware changes some fields in the 802.11 header before transmission, such as the sequence and fragment number.

However, there is a somewhat known work around to this problem and the solution is briefly mentioned in [2]. Prism2 cards have an auxiliary (AUX) debug port which provides raw access to the card's memory. The basic idea is to queue the packet in the normal way for transmission and locate its header via the AUX port. Just after instructing the card to transmit, one could busy wait reading a header byte (such as the duration) through the AUX port until it is modified. At that point, the firmware has done its processing and is about to send it off to the radio. Just before it is able to transmit the packet physically, the modified packet header bytes can be re-written via the AUX port. It is a race condition which is virtually always won in practice.

Our implementation was for the FreeBSD `wi` driver. First, all the airjack functionality was implemented. This allows the sending of raw frames (with no firmware re-transmits) and the receiving of all frames. After that, the AUX overwriting was implemented in order to send fragments properly (need control over the fragment and sequence numbers). There were also other subtle aspects

which were encountered and needed to be resolved. For example, the More Fragment bit needs to be set only when the overwrite is performed and not when the packet is queued normally. Another issue is that when the WEP bit is set, it is cleared after the firmware processing although the packet does indeed get transmitted as a WEP one. However, it is important not to set the WEP bit when performing the re-write or the card will not transmit. Obviously none of this is documented which caused the authors to have great fun.

The main limitation with our Prism2 implementation has to do with the fact that reception is difficult after transmission. Possibly this is due to the fact that the author of this paper is a totally unskilled kernel developer or the card indeed has some limitations. For example it is not possible to receive the ACKs after each fragment is sent (802.11 requires each fragment to be acknowledged).

4.2 Atheros based cards

Atheros [1] cards are mostly software radios making them ideal for packet injection. The FreeBSD `ath` driver has been modified in order to allow the sending and reception of raw frames. Reception is easily achieved by simply changing the RX filter to accept all frames, including control frames. Control frames turned out to be very useful as the card is able to spot ACKs for data being sent, making it possible to implement re-transmissions in the attack.

Atheros will readily send out WEP fragments. It does however mangle the fragment and sequence number although the fix is much simpler compared to the Prism2 approach. The packet needs to be queued with a type of '2' (which indicates PS-Poll frames). In order to eliminate re-transmissions from the firmware, a flag indicating that the packet requires no ACKs exists.

The only limitation with the Atheros implementation is the inability to send ACKs in time. Either ACKs are not sent properly, or more likely, they are sent too late. This probably occurs as the card might be transmitting after DIFS time rather than SIFS. There surely is a solution, but currently this problem has not been looked into. The work around is to simply have any other wireless card in range with the same MAC address as the one being used in the attack. The firmware of that card will blindly send an ACK when data arrives for it.

4.3 The Tool: `wesside`

Our goal was to provide an easy to use tool. Although the wireless drivers need to be patched before the tool may be used, it does not mean that the user must be a computer literate—providing a Live CD is an option.

Our final product is called `wesside` and when launched with *no* command line arguments does the following:

1. Channel hops looking for a WEP network.
2. Once a network is found, it tries to authenticate.
   (a) If authentication fails, it attempts to find a MAC address to spoof.
3. Once authenticated, it associates.
4. After it eavesdropped a single data packet, it discovers at least 128 bytes of PRGA by sending out larger broadcasts and intercepting the relayed packets.
5. After it eavesdropped an ARP request, it decrypts the IP address by guessing the next four bytes of PRGA using multicast frames.
6. It floods the network with ARP requests for the decrypted IP address.
7. Launches `aircrack` (v2.1) each 100,000 packets captured for 1 minute and attempts to decrypt the key. If 3,000,000 packets have been captured so far, the cracking time is increased to 10 minutes and cracking is started every 1,000,000 packets.

The tool could be much smarter, and it is, but only when launched with at least one command line argument: the IP of an Internet host. The tool acts as described previously but with these differences:

- After the IP network is discovered, an ARP request for the IP address terminating with ".1" is sent and the reply is waited for (the tool assumes this will be the router). The IP address ending in ".123" is assigned to the attacker.
- After the MAC address of the router is obtained, a UDP packet to the Internet host is sent each 5 seconds. ARP requests also continue to be sent in order to maintain the attacker's IP address in the router's ARP cache. This allows IP traffic to be forwarded back to the attacker.
- The Internet host may either:
  1. Sends short packets. In this case, weak IV key decryption occurs periodically as described previously.
  2. Sends MTU length packets. In this case, other than periodically attempting the weak IV attack, the tool also builds a dictionary. The tool will bind to a TAP interface. This TAP interface may be used for transmission and occasionally for reception whenever a packet with a known IV traverses the wireless network.

  The attacker may choose whether to run the short or MTU length packet server on the Internet host.

There are *many* ways in which this tool could be made smarter. For example it may ping a broadcast address when launched with no arguments or it may use better heuristics for determining the router's MAC address. However, our main goal was to provide a working and usable proof of concept. If the results prove to be outstanding, then it might be worth developing a more sophisticated tool.

5 Evaluation

In this section, we show the performance of the attack tool which has been developed. The level of how much the fragmentation attack is a threat in practice will be evident by the results. The tool was tested by performing experiments on common everyday hardware. The setup for all the experiments is as follows:

- AP: Linksys WRT54G.
- Attacker: Laptop, P4 2.4GHz, 512MB RAM, Atheros 802.11g.
- "Internet" host: Laptop, Celeron 400MHz, 200MB RAM, 100Mb Ethernet attached to AP WAN port.

In all cases, once the `wesside` tool is started by the attacker, a *single* ARP request is generated by a host attached to the LAN port of the AP. This is the requirement for the bootstrap of the fragmentation attack. Although any packet type will allow PRGA determination, the IP discovery has only been implemented on ARP packets and for that reason such a packet is generated. This requirement does not distort results too much. Firstly, IP packets (and thus the source IP) may be decrypted in the same way as ARP packets without taking too long. Secondly, a smarter version of the tool may cause wireless clients to de-authenticate from the AP by spoofing a de-authentication management frame (management frames are sent in cleartext). Once the client re-associates, the most likely data it will immediately send is an ARP request for its own IP for the router's.

This setup is also best case in terms of connectivity with the "Internet" as the flood host is connected directly to the WAN port, effectively having a 100Mbit/s connection. When presenting the results, we will calculate how much bandwidth is actually being used in order to demonstrate that this throughput could be achieved on most real Internet links.

The efficiency of the attack is measured in two stages. The first stage is the "preparation" to

the flooding (bootstrapping)—i.e. determining network parameters and its configuration before flooding may commence. The metric used in this stage is time—the quicker the better.

The second part of the attack either involves decrypting the key, building a dictionary or both. In all cases, the more data packets received, the better. For this part of the attack, the metric used is the packets received per second.

5.1 Bootstrap speed

The bootstrap of the attack includes:

- Determining PRGA.
- Determining an IP.
- Determining the router's MAC address (relies on the ".1" IP address heuristic).

All of this is independent of the key size and its complexity as no KSA attacks are being performed. Thus, the metric obtained for the bootstrapping procedure should be very similar across all networks and configurations.

The results are as follows. From when the first data packet is eavesdropped, it takes 1 second to determine 144 bytes of PRGA. For 1500 bytes of PRGA it takes less than 2 seconds.

It takes ≈ 6 seconds to decrypt a single byte of the IP and < 30 seconds to decrypt the whole IP address.

Finally, determining the router's MAC address takes less than a second, simply because we "force" the router to have an IP address ending with ".1". This assumption is realistic and may work in many cases. Another way of instantly forwarding packets would be to use the AP's MAC address as a destination—APs normally forward packets themselves.

In short, it takes less than a minute for an attacker to be able to transmit any data and determine an IP address on a WEP wireless network after he eavesdropped a single ARP packet. This metric is quite network independent as no assumptions are being made about the WEP key.

5.2 Flood rate

The are two main ways in which the network may be flooded:

1. Send ARP requests. This emulates the simple replay attack (good for comparisons).
2. Use the Internet to flood. This is the main intent of this tool.

Additionally, there are two main "modes" of flooding:

1. Short packets. These are useful only for the weak IV attack.
2. MTU sized packets. These are useful in all cases, but the flooding rate is potentially lower.

These two modes of flooding have been implemented only for Internet based flooding.

The results are as follows. The simple replay-like attack will generate ≈ 350 unique (no retransmissions) packets per second (p/s). The Internet flood with small packets will cause ≈ 950 p/s. These packets are UDP with 5 bytes of data and their total packet size on the Internet will roughly be $14 + 20 + 8 + 5 = 47$ bytes (size of Ethernet + IP + UDP headers + data). Thus the Internet traffic generated is approximately $47 \times 8 \times 800 = 357,200$ bits/s. An ADSL with 512Kbit/s downstream could easily handle that. To compare all these metrics with "normal" traffic, a `ping -f` with no replies and 56 data bytes generates ≈ 550 p/s. An FTP download generates ≈ 150 p/s. Beware that the FTP download was occurring from an 11Mbit/s client so a 802.11g link would yield a higher rate. However, what is important to note is that "normal" traffic on a network will almost never exceed the rate of data sent by a `ping -f`.

The MTU size packet flood for populating the dictionary will generate ≈ 50 p/s. This low rate is due to an implementation issue. Currently the way the dictionary is stored is in a directory hierarchy. The PRGA for the IV `0xAABBCC` will

Flood source	≈ p/s
802.11b client FTP download.	150
LAN client `ping -f` (no replies).	550
ARP replay-like attack.	350
Internet flood (short packets).	950
Internet flood (MTU sized packets) & dictionary logging in directories.	50
Internet flood (MTU sized packets) & dictionary logging in a file.	250

Table 1: Approximate packet rates reached when using different flooding mechanisms.

be located in the path "`/AA/BB/CC`". Creating directories is a slow process so the bottleneck is the actual filesystem—not the network. By simply logging the dictionary in a file (which may then be post-processed) it is possible to generate ≈ 250 p/s. This rate will require a 2.8Mbit/s Internet connection which is not always available. With 50 p/s a 512Kbit/s connection should suffice and completing the dictionary would take at least $\frac{2^{24}}{50\times60\times60} = 93$ hours. Generating 250 p/s would yield a full dictionary in at least ≈ 17 hours.

A full dictionary attack is not quite practical unless in extreme cases—investigating how to reset the IV generator would be one option. Also, a dictionary containing PRGAs of less than MTU size would require much less traffic possibly making this approach more practical. For example, using a PRGA length of $\frac{1}{3}$ MTU should yield approximately triple the packet rate ($\approx 250 \times 3$) requiring ≈ 6 hours of flooding.

All the packet rates obtained from all these experiments are summarized in Table 1.

5.3 Cracking time

Although key cracking using the weak IV attack is outside the scope of this work, some results are presented here. Initially, all cracking is run for less than 1 minute and with the default "fudge" factor (the breadth with which the key is searched amongst the possible candidates) of 2. Cracking commences each 100,000 packets captured. After 3,000,000 packets have been obtained, the cracking time is increased to 10 minutes and cracking occurs each 1,000,000 packets captured. The reason being that as the number of packets increase, the time to load the cracking process and the time it needs to perform calculations increases. Therefore it was not sensible to crack for only a single minute when a very large number of packets had to be processed.

The experiments involve a very limited sample of keys which were generated by reading from `/dev/urandom`. Although this sample is insignificant, it does give a vague idea on how many packets are required and how long the tool takes to complete the whole attack. The time taken for the attack is measured from the launch of the tool until the key is displayed to the user. The results are displayed in Table 2.

Different keys have diverse properties requiring a different volume of traffic to be gathered before they are cracked. An interesting thing to note is that the 40-bit candidate keys all took about 5 minutes to be recovered! Conversely, 104-bit keys seem to take much longer and there is no evident trend from the tested sample on how many packets could be required on average. In some cases (such as the key requiring 10M packets) a dictionary attack could have been effective too.

The worst case scenario of this sample is an attack time of about 3 hours while the best case took only about 2 minutes. A wild guess could be that most keys will be recovered in ≈ 1 hour—only experience will tell.

In many cases it was possible to obtain the key with less packets by simply running the cracking process for longer or with a higher fudge factor. An important consideration to make is that the further into the process, the laptop's CPU would slow down significantly (even to 50% of its throughput) because of temperature rise—a common phenomenon whilst cracking! Cracking on a desktop with proper cooling would have been more ideal.

Also, note that the IVs being captured are incremental in a linear way. Only the AP generates traffic in this test network and no one else. In a real scenario, there would be a larger mix of collected

Key	Packets	Time (m)
2C:CE:FC:1D:2B	100,000	1.93
80:19:B8:3F:C8	200,000	3.83
6F:34:11:BC:A3	200,000	4.30
91:B7:C0:A7:F7	300,000	5.45
3B:07:DA:02:B7	300,000	5.60
EB:A6:50:D0:2B:DA:CC:B7:E1:B7:E8:50:59	1,700,000	30.77
D9:06:CA:9E:EA:B3:18:CD:24:9F:2E:5E:10	2,400,000	42.85
5E:02:F4:83:FE:F6:27:10:21:EC:8E:87:27	2,700,000	49.17
64:AC:EE:55:B7:7E:27:93:09:6B:78:00:78	9,000,000	156.58
41:0A:68:52:5B:BE:C7:64:D7:09:FC:CC:BB	10,000,000	181.28

Table 2: Approximate packets required for cracking random keys and the total attack time.

IVs as other stations would be transmitting too with an IV generator at a different phase. Thus, the number of packets needed may be totally different in a real network. Also the IV space covered is always starting from 0 onwards. It may be that more weak IVs lie towards the end of the 24 bit space.

6 Conclusions and Future Work

The fragmentation attack proves to be highly practical as its bootstrap can be completed in less than a minute. Although the key is not recovered and it is not possible to receive data during this interval, the network is under a great threat as the attacker may transmit arbitrary data. It is trivial to use this power in order to flood the network and aid other WEP cracking techniques.

There are two main questions related to this attack which still need to be assessed. The first one is how can the flood rate be increased, i.e., what is the maximum number of packets per second which may be caused to traverse a wireless network. If the rate doubles, the time needed the gather weak IVs will halve—a significant improvement for cracking the key. The other aspect to examine is whether there is a systematical way to reset the IV generator so that only small dictionaries are required in order to all decrypt traffic.

WEP networks with low traffic were considered to be "safe" since it would require an attacker to wait many hours, perhaps days, before the key could be recovered. With the fragmentation attack, this is no longer the case as the key could potentially be recovered in less than a couple of hours. Furthermore, an attacker may cause severe damage by launching DoS attacks or by ARP spoofing hosts. We believe that the fragmentation attack was the missing link in providing an efficient and practical mechanism for breaking WEP.

7 Acknowledgments

The author of this paper would like to thank Joshua Lackey for discovering and disclosing the fragmentation vulnerability. This whole attack was a collaboration of numerous people and the author of this paper simply gathered the knowledge, filled in the missing bits and provided an implementation.

The author would also like to thank David Hulton, Anton Rager and Michael Lynn for much information on this attack. A big thank you goes to Alex Lee who lend his Atheros card to the author. Finally many thanks to Mark Handley who allowed the author to work on this while he should have been developing protocols for Mark rather than breaking other ones!

References

[1] Atheros Communications. Atheros chipset. http://www.atheros.com.

[2] J. Bellardo and S. Savage. 802.11 denial-of-service attacks: Real vulnerabilities and practical solutions. 2003.

[3] A. Bittau. Additional weak IV classes for the FMS attack. 2003. http://www.darkircop.org/sorwep.txt.

[4] C. Devine. aircrack, 2004. http://www.cr0.net:8040/code/network/.

[5] S. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the Key Scheduling Algorithm of RC4. *Lecture Notes in Computer Science*, 2259:1–24, 2001.

[6] D. Hulton. Practical Exploitation of RC4 Weaknesses in WEP Environments. 2002.

[7] IEEE. MAC address prefix list (OUI). http://standards.ieee.org/regauth/oui/oui.txt.

[8] IEEE. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 1999.

[9] Intersil. Prism2 chipset. http://www.intersil.com/globespanvirata.

[10] KoreK. Next generation of WEP attacks?, 2004. http://www.netstumbler.org/showpost.php?p=93942&postcount=35.

[11] M. Lynn. airjack, 2003. http://sourceforge.net/projects/airjack/.

[12] A. Rager. WEPWedgie, 2003. http://sourceforge.net/projects/wepwedgie/.

[13] R. L. Rivest. *The RC4 Encryption Algorithm*. RSA Data Security, Inc., Mar. 12, 1992. (Proprietary).

[14] A. Stubblefield, J. Ioannidis, and A. Rubin. Using the Fluhrer, Mantin, and Shamir Attack to Break WEP. 2001.

[15] D. Wagner. Weak Keys in RC4, 1995. http://www.cs.berkeley.edu/~daw/my-posts/my-rc4-weak-keys.