AT&T Labs Technical Report TD-4ZCPZZ

Using the Fluhrer, Mantin, and Shamir Attack to Break WEP

Revision 2

August 21, 2001

Adam Stubblefield
Rice University
astubble@cs.rice.edu

John Ioannidis
Aviel D. Rubin
AT&T Labs – Research, Florham Park, NJ
{ji,rubin}@research.att.com

Using the Fluhrer, Mantin, and Shamir Attack to Break WEP

Adam Stubblefield*
Computer Science Dept.
Rice University
astubble@cs.rice.edu

John Ioannidis

Aviel D. Rubin

AT&T Labs – Research, Florham Park, NJ
{ji,rubin}@research.att.com

Abstract

We implemented an attack against WEP, the link-layer security protocol for 802.11 networks. The attack was described in a recent paper by Fluhrer, Mantin, and Shamir. With our implementation, and permission of the network administrator, we were able to recover the 128 bit secret key used in a production network, with a passive attack. The WEP standard uses RC4 IVs improperly, and the attack exploits this design failure. This paper describes the attack, how we implemented it, and some optimizations to make the attack more efficient. We conclude that 802.11 WEP is totally insecure, and we provide some recommendations.

1 Introduction

Wireless networking has taken off, due in large part to the availability of the 802.11 standard. While another standard, Bluetooth, is also gaining in popularity, the longer range and higher speeds achieved by 802.11 make it the protocol of choice for wireless LANs. Office buildings, conferences, and even many residences now offer 802.11 connectivity. The PC cards that are most often used in these networks provide a security protocol called Wired Equivalent Privacy (WEP).

WEP is easy to administer. The device using the 802.11 card is configured with a key, that in practice usually consists of a password or a key derived from a password. The same key is deployed on all devices, including the access points. The idea is to protect the wireless communication from devices that do not know the key.

*Research done while a summer intern at AT&T Labs

Borisov, Goldberg and Wagner demonstrated some security flaws in WEP [1]. They explained that WEP fails to specify how IVs for RC4 are specified. Several PC cards reset IVs to zero every time they are initialized, and then increment them by one for every use. This results in high likelihood that keystreams will be reused, leading to simple cryptanalytic attacks against the cipher, and decryption of message traffic. The authors verified this experimentally and describe other weaknesses as well. For example, the space from which IVs are chosen is too small, virtually guaranteeing reuse, and leading to the same cryptanalytic attacks just described. The paper also shows that message authentication in WEP is broken.

Fluhrer, Mantin, and Shamir describe a passive ciphertext-only attack against RC4 as used in WEP [4]. The attack exploits the method in which the standard describes using IVs for the RC4 stream cipher. In their paper, the authors state, *Note that we have not attempted to attack an actual WEP connection, and hence do not claim that WEP is actually vulnerable to this attack.* Based on the description in their paper, we successfully implemented the attack, proving that WEP is in fact completely vulnerable. The purpose of this paper is to describe our implementation, along with some enhancements to improve the performance of the attack.

2 Overview of the WEP attack

In this section we present an overview of the WEP protocol and review briefly how the attack of Fluhrer, Mantin, and Shamir can be applied to WEP. For a detailed description of WEP we refer the reader to the official 802.11 standard [7].

Encryption in WEP uses a secret key, k, shared between an access point and a mobile node. To compute a WEP frame, the plaintext frame data, M, is first concatenated with its (non-cryptographic) checksum $c(M)$, to produce $M \cdot c(M)$ (where $\cdot$ denotes concatenation). Then, a per packet initialization vector (IV) is prepended to the secret key, k, to create the packet key, $IV \cdot k$. The RC4 stream cipher is then initialized using this packet key, and the output bytes of the cipher are exclusive-ored (denoted $\oplus$) with the checksummed plaintext to generate the ciphertext:

$$C = (M \cdot c(M)) \oplus RC4(IV \cdot k)$$

The actual WEP data is the per-packet IV prepended to this ciphertext, C.

2.1 The Known IV Attack of Fluhrer, Mantin, and Shamir

For completeness, we include a short description of the attack of Fluhrer, Mantin, and Shamir [4] here. We refer the reader to the original paper for the motivation

and details.

To begin, we describe the structure of the RC4 stream cipher (a full description can be found in [9]). RC4 consists of two parts, a key scheduling algorithm and an output generator. In WEP, the key scheduling algorithm uses either a 64-bit packet key (40-bit secret key plus 24-bit IV) or a 128-bit key (104-bit secret key plus 24-bit IV) to set up the RC4 state array, S, which is a permutation of $\{0,\ldots,255\}$. The output generator uses the state array S to create a pseudorandom sequence.

The attack utilizes only the first word of output from the pseudorandom sequence, so we focus our attention there. The equation for this first byte of output is given by $S[S[1]+S[S[1]]$. Thus, after the key setup stage, this first byte depends on only three values of the state array ($S[1]$, $S[S[1]]$, $S[S[1]+S[S[1]]$). The attack is based on our ability to derive information about the key by observing this value. We defer the discussion of how to recover the value of this first byte from a WEP ciphertext stream until Section 3.

To mount the attack, we search for IVs that place the key setup algorithm into a state which leaks information about the key. Using the terminology of Fluhrer *et al.*, we refer to these key-leaking cases as *resolved*. It is simple to test whether a particular packet provides an IV and output byte that result in a resolved condition, though we refer the reader to the Fluhrer *et. al.* paper for the conditions under which they occur[1]. Each resolved packet leaks information about only one key byte, and we must correctly guess each key byte before any packet gives us information about a later key byte.

We say we must "guess" each key byte as the attack is statistical in nature; each resolved packet gives us a 5% chance of guessing a correct key byte and a 95% chance of guessing incorrectly. However, by looking at a large number of these resolved cases, we can expect to see a bias toward the true key bytes.

3 Implementation

In implementing this attack, we had three goals. First and foremost, we wanted to verify that the attack could work in the real world. Second, we were interested in how cheaply and easily the attack could be launched. Lastly, we wanted to see what improvements could be made to both the general RC4 attack and the WEP attack in particular. In this section we report on our success at the first two goals, while reserving discussion about attack optimizations to Section 4.

[1] It is important to use the criteria given in section 7 rather than the criteria given in appendix A. The IVs listed in appendix A are only a subset of the IVs which can resolve. We return to this in section 4 of this paper.

3.1 Simulating the Attack

Before trying to break WEP, we created a simulation of the RC4 attack to both verify our understanding of the weakness and to gather information about how many resolved packets we could expect would be required when mounting the actual attack. The coding of the simulated attack took under two hours, including a few optimizations. The simulation showed that the attack was always able to recover the full key when given 256 probable resolved cases.[2] We also observed that although 60 resolved cases (the number recommended in the Fluhrer *et. al.* paper) were usually enough to determine a key byte, there were instances in which more were required. Because at this point we had not thoroughly investigated how accurately we would be able to determine the first output byte of the RC4 pseudorandom sequence, we also simulated the effect that sometimes guessing wrong would have on the attack. We were pleased to see that as long as the number of incorrect guess was kept small, the correct key byte would still be returned, though sometimes more resolved cases were needed.

3.2 Capturing the Packets

Surprisingly, capturing WEP encrypted packets off of our wireless network proved to be the most time consuming part of the attack. There are a number of commercial software programs that are able to both capture and decode 802.11 packets, such as NAI's "Sniffer" and Wildpacket's "AiroPeek," though both products cost thousands of dollars. Because we wanted to show that the attack could be done by an adversary with limited resources, we purchased a $100 Linksys wireless card, based on the Intersil Prism II chipset. We made this choice because the Prism II allows much of its computation to be completed in software and because there was a Linux driver available that could grab raw WEP encrypted packets. Though we did not know it at the time, this chipset has been used by others to mount dictionary and brute force attacks against WEP.[3]

We used both the `linux-wlan-ng prism2` driver[4] and a modified version of Tim Newsham's patch to re-enable raw packet monitoring,[5] to get the card working in Linux. We were then able to use a modified version of the packet sniffer `ethereal`[6] to capture raw WEP encrypted packets and to decode the data necessary for our attack tool.

[2] Cases corresponding to IVs of the form (B+3, 255, N) as in the Fluhrer *et. al.* paper.

[3] See Blackhat '01 presentation at `http://www.lava.net/~newsham/wlan/WEP_password_cracker.ppt`

[4] Available from `http://www.linux-wlan.com/`

[5] Available from `http://www.lava.net/~newsham/wlan/`

[6] Available from `http://www.ethereal.com/`

There is one problem with using this card as opposed to a more sophisticated solution. The prism2 chipset does request a transmission time-slot even when in monitor mode. Many inexpensive basestations do not report this, though a software hack can allow Linux computers running as access points to register an SNMP trap each time that a node joins or leaves the network [5]. This information does not directly indicate likely attackers, but could be combined with other information in an IDS to locate users who register with a basestaion but not with whatever network level access controls exist. Also, we know of no practical reason why this "registration" with the network is necessary; there may even exist consumer 802.11 chipsets which support listening without registering (perhaps even the prism2 chipset in some other undocumented mode).

Even with the hardware and software problems, from the time that we first decided to look at this problem, it took less than a week for the the card to be ordered and shipped, the test-bed to be set up, the problems to be debugged, and a full key to be recovered.

3.3 Mounting the Attack

The last piece in actually mounting the attack was determining the true value of the first plaintext byte of each packet, so that we could could infer the first byte of the pseudorandom sequence from the first ciphertext byte. We originally looked at `tcpdump` output of decrypted traffic (using a correctly keyed card[7]), and were planning on using packet length to differentiate between ARP and IP traffic (both of which have well known first bytes in their headers) as these were by far the two most common types of traffic on our network. After implementing this, however, we discovered that the attack didn't seem to work. We then tried hand decrypting packets to determine whether `tcpdump` was working correctly and discovered that an additional 802.2 encapsulation header is added for both ARP and IP traffic.[8] This discovery actually made the attack even easier, as all IP and ARP packets would now have the same first plaintext byte (0xAA, the SNAP designation).[9] If the network in question also carries legacy IPX traffic, the first plaintext byte will not be 0xAA for these packets. However, as we showed in our simulation, as long as the IP and ARP packets greatly outnumber the IPX packets, the attack is still possible. If the network carries mostly IPX traffic, the attack should be modified to use either 0xFF or 0xE0 instead of 0xAA.

Although our actual attack used the improvements discussed in the next section, we present an outline of how a naive attack could work here. It is interesting

[7] Note that a correctly keyed card is not needed; we simply used one to design the attack.

[8] We eventually traced this back to RFC 1042 [8].

[9] Some vendors, such as Cisco use a proprietary OID [2]. Fortunately, it also beings with 0xAA.

```
RecoverWEPKey()
    Key[0...KeySize] = 0
    for KeyByte = 0...KeySize
        Counts[0...255] = 0
        foreach packet → P
            if P.IV ∈ {(KeyByte + 3, 0xFF, N) | N ∈ 0x00...0xFF}
                Counts[SimulateResolved(P, Key)] += 1
        Key[KeyByte] = IndexOfMaximumElement(Counts)
    return Key
```

Figure 1: The basic attack on WEP. Depending on the actual key used, this attack can take between 4,000,000 and 6,000,000 packets to recover a 128-bit key. The `SimulateResolved` function computes the value described in section 7.1 of Fluhrer *et al.*

to note that even this baseline version of the attack would still be successful in a short period of time (a day or two at most) and with an even smaller amount of computation when compared to the improved implementation, assuming that the wireless network in question had a reasonable amount of traffic.

To begin, we collected a large number of packets from our wireless network. To speed the process up for some of our experiments late at night when network volume was low, we artificially increased the load on the wireless network by ping flooding a wireless node. (We could have waited until more traffic was created; this is not an active attack.) Because we are able to predict the value of the first byte of any plaintext, the fact that we changed the makeup of the network traffic did not affect these experiments. In looking at the IVs of these collected packets, we discovered that the wireless cards use a simple counter to compute the IV, wherein the first byte is incremented first.[10]

Figure 1 shows the basic attack used to recover a WEP key. In section A.1 of Fluhrer *et. al.*, the authors postulate that 4,000,000 packets would be sufficient with this baseline attack; we found the number to be between 5,000,000 and 6,000,000 for our key. This number is still not unreasonable, as we were able to collect that many packets in a few hours on a partially loaded network.

4 Improving the attack

In this section we discuss several modifications that can be made to improve the performance of the key recovery attack on WEP. While not necessary for the com-

[10] Other cards have been reported to choose IVs at random, to count in big endian order, or to switch between two IVs. This last class are cards are not vulnerable to the attack in this paper, although they break badly under the attacks of Borisov *et al.* [1].

promise to be effective, they can decrease both time and space requirements for an attacker.

4.1 Choosing IVs

In the baseline attack (the one described in Appendix A of Fluhrer *et. al.*), only IVs of a particular form are considered (those corresponding to $(KeyByte+3, 0xFF, N)$ where *KeyByte* is the current KeyByte we are guessing and N is unrestricted). However, we found that there are other IVs that can result in a resolved state, and that testing all IVs instead of only the subset suggested by the Fluhrer *et. al.* paper can be done in parallel with receiving packets. This conclusion was verified by Adi Shamir [10], who also noted that these packets appear more often for higher key bytes.

4.2 Guessing Early Key Bytes

As the Fluhrer, Mantin, and Shamir attack works by building on previously discovered key bytes, recovering early key bytes is critical. There are two approaches that we tried both separately and together. The first utilized the way that the IVs were generated, namely that we would receive packets that resolved for lots of different key bytes before necessarily receiving enough resolving packets to predict the early key bytes.[11] We would therefore use the resolving cases that we had received to narrow down the possibilities for the early key bytes. We were then able to test candidate keys by determining if the WEP checksum on a decrypted packet turned out correctly.

The second approach exploited the poor key management available in WEP implementations. Since WEP keys have to be entered manually, we assumed that instead of giving clients a long string of hex digits, a user memorable passphrase would be used. After examining the test wireless cards at our disposal, we determined that the user-memorable passphrase is simply used raw as the key (i.e. the ASCII is used; no hashing is done). Although hashing does not protect against a dictionary attack, it would have helped in this circumstance, as we were able to determine directly whether each key byte was likely to be part of a user memorable passphrase by checking whether the byte value corresponded to an ASCII letter, number, or punctuation symbol.

This pair of optimizations turned out to provide an astounding decrease in the number of packets required. In parallel with receiving packets (on another machine, though this is not really necessary), we were continually attempting to guess the key by choosing the most likely candidates based on the resolved cases we had

[11] See Figure 6 of Fluhrer *et. al.*; resolved cases are much more likely to occur for later key bytes.

```
RecoverWEPKeyImproved(CurrentKeyGuess, KeyByte)
    Counts[0...255] = 0
    foreach packet → P
        if Resolved?(P.IV)
            Counts[SimulateResolved(P,CurrentKeyGuess)] += Weight(P,CurrentKeyGuess)
    foreach SelectMaximalIndexesWithBias(Counts) → ByteGuess
        CurrentKeyGuess[KeyByte] = ByteGuess
        if Equal?(KeyByte,KeyLength)
            if CheckChecksums(CurrentKeyGuess)
                return CurrentKeyGuess
        else
            Key = RecoverWEPKeyImproved(CurrentKeyGuess,KeyByte+1)
            if notEqual?(Key,Failure)
                return Key
        return Failure
```

Figure 2: The improved attack on WEP. Depending on the actual key used, this attack can take between 1,000,000 and 2,000,000 packets to recover a 128-bit key. The `SimulateResolved` function computes the value described in section 7.1 of Fluhrer *et al.*, the `CheckChecksums` checks to see if a key causes the checksums in the WEP packets to come out correctly, and the `Resolved?` predicate checks to see if a given packet results in a resolved condition. The `SelectMaximalIndicesWithBias` function corresponds to the optimization in section 4.2. The `Weight` function returns 3 if the resolved case corresponds to a special resolved case as described in section 4.3, and 1 otherwise.

already gathered. In the event of "ties" for the next most likely byte, we gave priority first to (in order): lowercase letters, uppercase letters, numbers, symbols, other byte values.

4.3 Special Resolved Cases

As Shamir pointed out to us, there are cases when a resolved case can provide an even better indication as to a particular key byte. If there is a duplication among the three values at positions $S[1]$, $S[S[1]]$, and $S[S[1]+S[S[1]]$ (i.e. these are only two distinct values), then the probability that these positions in the S permutation remain unchanged jumps from $e^{-3} \approx 5\%$ to $e^{-2} \approx 13\%$. We can thus treat the evidence from these cases as about three times more convincing as a standard resolved case.

4.4 Combining the Optimizations

Figure 2 shows the key recovery algorithm after all of the improvements described above. The improvements drop the number of packets required from around 5,000,000 to around 1,000,000.

5 Discussion

There are many variables that can affect the performance of the key recovery attack on WEP. In this section we summarize the effect of some of these variables and look at how the WEP design could be slightly altered to prevent this particular attack.

5.1 IV Selection

Since the WEP standard does not specify how IVs should be chosen, there are a variety of IV generation in use in current 802.11 cards. The majority of cards seem to use one of three methods: counters, random selection, or value-flipping (i.e. switching between two IV values). This attack is possible with either of the first two types of IV selection. Value-flipping prevents this attack at the expense of reusing the pseudorandom stream every other packet. This is not a reasonable trade-off.

Counter modes are the most accommodating of this attack. In these cards, the IV is incremented with each packet sent (starting either at 0 or at some random value when the card is powered on). With counter mode cards, an attacker is practically guaranteed a nice distribution of resolving packets among the key bytes. Random selection of each IV is not much better, as there are enough expected resolved cases that although the distribution might not be quite as good as the counter modes, it won't be much worse.

In short, there does not seem to be a way of choosing IVs to mitigate the effects of this attack without explicitly testing each IV and key pari to see if it resolves before sending it. This would require extra processing power and would decrease the already small space of IVs.

5.2 Key Selection

The lack of key management in WEP certainly contributes to the ease of the key recovery attack. Most networks use a single shared key between the basestation and all mobile nodes. Besides the suite of "disgruntled ex-employee who knows the key" style attacks, there is also the problem of distributing this key to the users. Many sites use a human memorable password to easy this key distribution. There is however no standard way of mapping these passwords to a WEP key. The current solution is mapping the ASCII value directly to a key byte. We would recommend switching to either using a secure (non-memorable) WEP key or having the key setup software hash the password to the key using a cryptographic hash function. Note that neither of these solutions prevent the attack, only make it slightly more difficult.

There do exist proprietary solutions that allow each mobile node to use a distinct WEP key, most notably Cisco's LEAP protocol. LEAP sets up a per-user, per-session WEP key when a user first authenticates to the network. This complicates the attack, but does not prevent it so long as a user's "session" lasts sufficiently long. We would recommend securely rekeying each user after every approximately 10,000 packets.

5.3 RC4

RC4 is an efficient stream cipher that can be used securely. The implementation of RC4 in SSL is not affected by the Fluhrer *et. al.* attack. The reasons are that SSL pre-processes the encryption key and IV by hashing with both MD5 and SHA-1 [3]. Thus different sessions have unrelated keys. In addition, in SSL, RC4 state from previous packets is used in future packets, so that the algorithm does not rekey after each packet.

A further recommendation (RSA Security Inc.'s standard recommendation) is for applications to discard the first 256 bytes of RC4 output. This may be a bit expensive for very small packets, but if session state is maintained across packets, that cost is amortized.

In summary, RC4 can be used as part of a security solution. However, care must be taken when implementing it so that key material is not leaked. One of the risks of algorithms that have such caveats is that protocol designers without a strong grounding in cryptography and security may not be aware of the correct way to implement them, and this is exactly what happened in the case of WEP.

6 Conclusions and recommendations

We implemented the attack described by Fluhrer *et. al.* in several hours. It then took a few days to figure out which tools to use and what equipment to buy to successfully read keys off of 802.11 wireless networks. Our attack used off of the shelf hardware and software, and the only piece we provided was the implementation of the RC4 attack, along with some optimizations. We believe that we have demonstrated the ultimate break of WEP, which is the recovery of the secret key by observation of traffic.

Since our technical report appeared, others have duplicated our results. Although we did not release our code, there are now two publicly available tools for breaking WEP keys. As always, once security attacks become known, exploits are available to *script kiddies*, who do not need to understand the technical details to break systems. The two tools that we know of are `Airsnort` and `WEPCrack`.

Given this attack, we believe that 802.11 networks should be viewed as insecure. We recommend the following for people using such wireless networks.

- Assume that the link layer offers no security.
- Use higher-level security mechanisms such as IPsec [6] and SSH [11] for security, instead of relying on WEP.
- Treat all systems that are connected via 802.11 as external. Place all access points outside the firewall.
- Assume that anyone within physical range can communicate on the network as a valid user. Keep in mind that an adversary may utilize a sophisticated antenna with much longer range than found on a typical 802.11 PC card.

The experience with WEP shows that it is difficult to get security right. Flaws at every level, including protocol design, implementation, and deployment, can render a system completely vulnerable. Once a flawed system is popular enough to become a target, it is usually a short time before the system is defeated in the field.

Acknowledgments

We thank Bill Aiello, Steve Bellovin, Scott Fluhrer, Bob Miller, Ron Rivest, Adi Shamir, Dave Wagner, and Dan Wallach for helpful comments and discussions.

We informed Stuart Kerry, the 802.11 Working Group Chair, that we successfully implemented the Fluhrer, et al. attack. Stuart replied that the 802.11 Working Group is in the process of revising the security, among other aspects, of the standard and appreciates this line of work as valuable input for developing robust technical specifications.

References

[1] BORISOV, N., GOLDBERG, I., AND WAGNER, D. Intercepting mobile communications: The insecurity of 802.11. *MOBICOM 2001* (2001).

[2] CAFARELLI, D. Personal communications, 2001.

[3] DIERKS, T., AND ALLEN, C. *The TLS Protocol, Version 1.0.* Internet Engineering Task Force, Jan. 1999. RFC-2246, `ftp://ftp.isi.edu/in-notes/rfc2246.txt`.

[4] FLUHRER, S., MANTIN, I., AND SHAMIR, A. Weaknesses in the key scheduling algorithm of RC4. *Eighth Annual Workshop on Selected Areas in Cryptography* (August 2001).

[5] HAMRICK, M. Personal communications, 2001.

[6] KENT, S., AND ATKINSON, R. Security architecture for the Internet protocol. Request for Comments 2401, Internet Engineering Task Force, November 1998.

[7] L. M. S. C. OF THE IEEE COMPUTER SOCIETY. Wireless LAN medium access control (MAC) and physical layer (PHY) specifications. *IEEE Standard 802.11, 1999 Edition* (1999).

[8] POSTEL, J., AND REYNOLDS, J. K. Standard for the transmission of IP datagrams over IEEE 802 networks. Request for Comments 1042, Internet Engineering Task Force, Feb. 1988.

[9] SCHNEIER, B. *Applied Cryptography - Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 1994.

[10] SHAMIR, A. Personal communications, 2001.

[11] YLONEN, T. SSH - secure login connections over the Internet. *USENIX Security Conference VI* (1996), 37–42.