Two Attacks on WEP

CS 165 (Computer Security), Fall 2020
Due date: November 4, 2020

We have seen in class that the Wired Equivalent Protocol (WEP) had serious weaknesses due to the flawed manner in which systems designers understood or applied cryptographic principles.

In this assignment, you will mount two attacks on the Wired Equivalency Protocol (WEP). The first attack exploits the attacker's ability to modify encrypted messages without detection due the linearity of CRC-32 and the weak manner in which WEP uses encryption. The second attack deduces the value of plaintext bytes without actually breaking the encryption, by exploiting a weakness in the wireless networking part of the WEP protocol.

1 Overview of WEP operation

Each WEP client shares a password P with the Access Point (AP). To send a message M, the client proceeds as follows:

1. Use the CRC-32 algorithm to compute the 32-bit "Integrity Check Value" CRC_M of M.
2. Choose a random 24-bit "Initialization Vector" I_M, and form the key $K_M = P \| I_M$.
3. Use the RC4 algorithm to generate the "key stream" $X_M = \text{RC4}(K_M)$.
4. Form the ciphertext $C_M = X_M \oplus [M \| CRC_M]$.
5. Transmit the packet $[I_M \| C_M]$. Here, I_M is in plaintext. C_M is the encryption of M.

The receiver proceeds as follows:

1. When $[I_M \| C_M]$ comes in, select the first 24 bits as I_M, and form the key $K_M = P \| I_M$.
2. Use the RC4 algorithm to generate the "key stream" $X_M = \text{RC4}(K_M)$.
3. Recover the plaintext $[M \| CRC_M] = C_M \oplus X_M$.
4. Take the last 32 bits of this plaintext as CRC_M. Treat the rest as the message M, and verify that its CRC-32 value matches the CRC_M in the last 32 bits of the incoming packet.

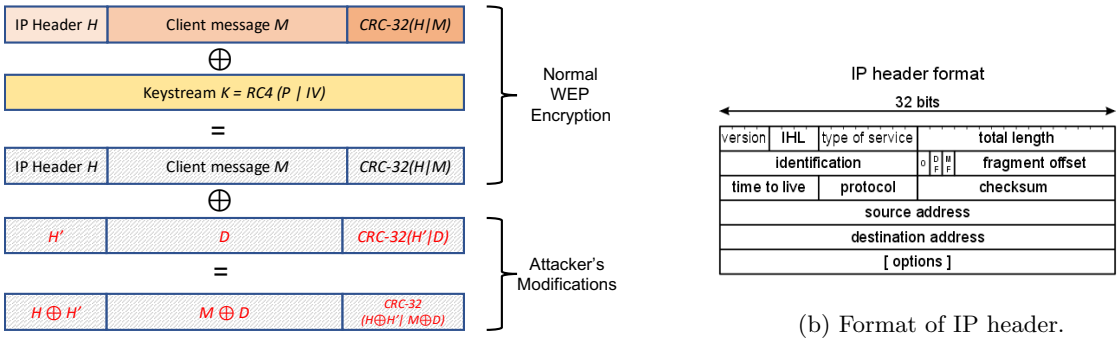
2 Overview of CRC 32 computation

The CRC-32 algorithm computes the 4-byte checksum of an input data stream by operating on it one byte at a time.

1. The CRC value is initialized to `0xFFFFFFFF` before the input is processed.
2. The following is repeated for each incoming input byte:
   (a) Each input byte is **xor**-ed with the last byte of the current CRC value.
   (b) This 8-bit value is used to index into a table of precomputed 32-bit patterns.
   (c) The CRC is now updated follows: right-shift the current CRC by 8 bits and do an **xor** with the pattern obtained from the table in the previous step.
3. When all input bytes have been processed, the CRC is **xor**-ed with `0xFFFFFFFF`.

3 The packet-redirect attack on WEP

In this clever attack, the attacker tricks the Access Point (AP) so that it decrypts the packet and sends it to him. This is done using the message modification attack that exploits the linearity property of CRC-32, allowing the adversary to modify messages under encryption.



(a) Message modification attack on WEP.

(b) Format of IP header.

Figure 1: WEP encryption and message modification attacks, and the structure of an IP header.

Figure 1a shows how the adversary can modify the packet under encryption, and Figure 1b shows the structure of the IP header in the encrypted part of the WEP packet. Say that a WEP client whose IP address is `A.B.C.D` wants to send a message M to a destination with IP address `E.F.G.H`.

The client is not connected directly to the destination `E.F.G.H`, but it is connected to a WEP Access Point (AP), with which it shares a key. The IP software at the client creates an IP packet, filling in the proper values for the source and destination addresses in the IP header. This packet is sent wirelessly via WEP to the AP, with both the message and the IP header encrypted. The AP decrypts this packet, and sends it via a wired connection to the host with IP address `E.F.G.H`.

This situation is shown in Figure 2, where Alice is sending packets to Bob through the wireless access point. The blue packets she generates have her IP address in the source field of the packet, and Bob's IP address in the destination field of the packet. Under normal circumstances, the packets are received by the AP, decrypted using the IV value in the packet and the key shared between Alice and the AP. These packets are then forwarded to Bob over a wired connection. Bob and the AP have already established this connection, which may or may not also be secured by encryption.

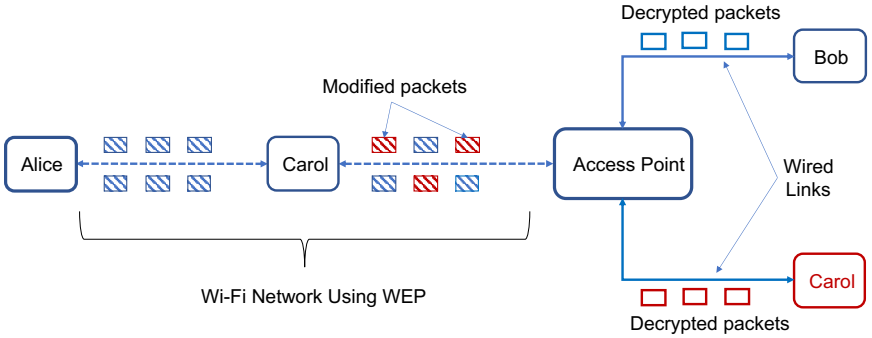


Figure 2: The packet redirect attack on WEP.

3.1 The attack

However, as Figure 2 shows, the attacker Carol can interpose herself between Alice and the AP, and exploit WEP weaknesses to render the encryption useless. Instead of trying to recover the WEP key or compromising the encryption in any way, she simply changes the destination address in the IP header part of the encrypted WEP packet. Modified packets are shown in red in the figure.

When the AP receives a modified packet, it regenerates the RC4 keystream, decrypts the packet, and sends it to the destination specified in the IP header. In Figure 2, unmodified packets carry Bob's IP address in the IP header destination field, and are sent to Bob. However, all modified packets carry Carol's IP address in the IP header's destination field, are sent to Carol. Carol has tricked the AP into decrypting the packets and sending them to her.

3.2 Your assignment

You are to set up the scenario shown in Figure 2. There will be five entities: `Alice`, `Carol-WEP` (on the wireless link), the `AP`, `Bob`, `Carol` (where Carol receives the plaintext packets from `AP` on a wired network). All communication is via sockets. These entities are connected as in Figure 2. We assign the IP address `169.235.16.75` to `Alice`, `141.212.113.199` to `Bob`, and `128.2.42.95` to `Carol`.

1. Form packets at `Alice`, such that packet i contains the plaintext message "Packet i", where i is a 4-byte hexadecimal value appearing as ASCII characters.
2. Prefix each packet with an IP header, which will have the IP address for `Alice` in the source field and the IP address for `Bob` in the destination field. Ignore the other IP header fields.
3. Compute a CRC-32 checksum for the packet, and attach it to the end of the packet.
4. Generate a random 24-bit IV. Use the `RC4` library call in Linux/Unix to encrypt the packet with the IP header and the CRC.
5. Prefix the encrypted packet with the plaintext IV you just generated, and send it to `Carol-WEP`.
6. `Carol-WEP` forwards all packets to the AP. *However, `Carol-WEP` changes the destination address in every second packet. The IP address for `Bob` is changed to the IP address for `Carol`.*
7. When a packet arrives at `AP`, it removes the IV prefix, uses RC4 to generate the keystream, and decrypts the rest of the packet. It first validates the packet using the CRC. It then looks at the destination address in the IP header, and sends it to `Bob` or `Carol` depending on the destination address in the IP header.

You are to generate 20 packets at `Alice`, and print the contents of the packets received by `Bob` and `Carol`. The output should have the format "Bob received ⟨packet contents⟩" and "Carol received ⟨packet contents⟩".

4 The "chop-chop" attack

Table 1 shows a 6-byte plaintext $D_0D_1D_2D_3D_4D_5$, and its 4-byte CRC-32 checksum $J_3J_2J_1J_0$. We will show how to guess the last byte (D_5) of the plaintext of the message by chopping it off, and manipulating the CRC-32 integrity checksum to make it correct, as in Table 2.

When an Access Point (AP) receives a packet in WEP, it checks its CRC-32 checksum to verify that the packet is not corrupted. If the checksum is invalid, the AP assumes that the packet is corrupt, and silently drops it, without any acknowledgement.

This is a weakness in the protocol. An attacker can determine whether the checksum is valid by seeing whether the AP acknowledges it. This weakness can be combined with the linearity property of CRC-32 to formulate an attack called the *chop-chop* attack. It is possible to recover the plaintext, *without breaking the cryptosystem*, by guessing the plaintext byte-by-byte.

	Data bytes						CRC-32			
Plaintext	D_0	D_1	D_2	D_3	D_4	$\boldsymbol{D_5}$	J_3	J_2	J_1	J_0
	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$
Keystream	K_0	K_1	K_2	K_3	K_4	$\boldsymbol{K_5}$	K_6	K_7	K_8	K_9
Ciphertext	S_0	S_1	S_2	S_3	S_4	$\boldsymbol{S_5}$	S_6	S_7	S_8	S_9

Table 1: Six-byte message $D_0D_1D_2D_3D_4D_5$ and its checksum $J_3J_2J_1J_0$.

	Data bytes					CRC-32			
Plaintext	D_0	D_1	D_2	D_3	D_4	I_3	I_2	I_1	I_0
	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$
Keystream	K_0	K_1	K_2	K_3	K_4	K_5	K_6	K_7	K_8
Ciphertext	R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8

Table 2: Five-byte message $D_0D_1D_2D_3D_4$ and its checksum $I_3I_2I_1I_0$.

To guess D_5, the attacker chops off the last plaintext byte D_5 by dropping the last ciphertext byte S_5 (see Table 1) to get a 5-byte message. If nothing else is done, the AP sees an incorrect checksum on decryption, and discards the packet. However, if the attacker can somehow place the correct checksum in the next 4 bytes, the AP will correctly decrypt and validate the remaining ciphertext bytes $R_0R_1R_2R_3R_4$.

This situation is shown in Table 2, where the checksum $I_3I_2I_1I_0$ corresponds to the plaintext $D_0D_1D_2D_2D_4$. In this case, the AP will acknowldege the packet, and the attacker will know that he succeeded in correcting the checksum. Here is how the attacker can pull this off.

4.1 Guessing the last byte

The attacker leaves D_0 to D_4 unchanged, but drops D_5. Now, I_3 appears (see Table 2) in the same position as D_5 previously did (see Table 1). *The attacker will try to guess D_5 by repeatedly guessing the value of $X = I_3 \oplus D_5$ and adjusting the CRC-32 checksum according to the guess.* The AP will acknowledge the packet when his guess is correct. Since there are only 256 possibilities for X, he is guaranteed to ultimately succeed.

- The ciphertext bytes R_0, R_1, R_2, R_3, R_4 will be the same as S_0, S_1, S_2, S_3, S_4.
- In Table 2, notice that $R_5 = I_3 \oplus K_5 = I_3 \oplus (D_5 \oplus D_5) \oplus K_5 = (I_3 \oplus D_5) \oplus (D_5 \oplus K_5)$. Since $X = I_3 \oplus D_5$ and $S_5 = D_5 \oplus K_5$ (Table 1), we have $R_5 = X \oplus S_5$. The attacker knows S_5, so he can compute the value of R_5 corresponding to his guess for X.
- R_6, R_7, R_8 are computed by reversing one step of the CRC-32 algorithm, based on our guess for X. There's a correspondence among $I_2I_1I_0$ and $J_3J_2J_1$ because CRC-32 shifts them back but D_5 "pushes" them forward again. They are not necessarily keeping the same values, but their difference depends only on X, which we have guessed.
- J_0 depends only on X. $K_9 = S_9 + J_0$. We have guessed the last message byte and the last byte of keystream.

We will guess X by trial and error. The access point will discard invalid frames and help us in guessing the value of X. Once we know X, we have guessed the last byte of the message. We can repeat to get the entire message.

4.2 What you are to do

You will do this part of the assignment in plaintext, not under encryption. You will approach the "chop-chop" method as a process of reversing the CRC-32 checksum.

1. Treat the last two bytes of your SID as a message, and compute its CRC-32 checksum. Illustrate the "chop-chop" technique on the plaintext.
2. Guess the last byte of this message by dropping it and recomputing the checksum for the remaining byte, under your guess for the last byte. Assume that the AP will tell you whether your corrected checksum is valid.

4.2.1 An example

We illustrate how to conduct this attack for the two-byte message "PQ". We begin by computing the CRC-32 checksum for "PQ".

1. Initialize `CRC` to `0xffffffff`.
2. Since 'P' is `0x50`, index into table is `0xaf`. Table entry is `0x4669be79`.
3. New `CRC` is (`0x4669be79` $\oplus$ `0x00ffffff`) = `0x46964186`.
4. Since 'Q' is `0x51`, index into table is `0xd7`. Table entry is `0x18b74777`.
5. New `CRC` is (`0x18b74777` $\oplus$ `0x00469641`) = `0x18f1d136`
6. Final $\oplus$ with `0xffffffff` gives `CRC = 0xe70e2ec9`.

We will now see how to reverse this computation, starting with the CRC `0xe70e2ec9`.

1. Reversing the last step, (`0xffffffff` $\oplus$ `0xe70e2ec9`) = `0x18f1d136`.
2. The top byte of this value (`0x18`) matches index `0xd7` in the table. We pick out this value, which is `0x18b74777`.
3. We reverse the earlier step by computing (`0x18b74777` $\oplus$ `0x18f1d136`) = `0x00469641`.
4. Since this value is `CRC >> 8`, we know that `CRC` must be of the form `0x469641??`, where the last byte is unknown.
5. We now try different possibilities for this last byte, using the AP's help. We send just the first byte of the message and the checksum `0x469641??` with different values in the last checksum byte. The AP signals the checksum validity by acknowledging our packet.
6. In this case, the AP acknowledges the packet only when we use the `CRC` value `0x46964186`.
7. We know from step 4 of the (forward) CRC calculation that the table index `0xd7` comes from the **xor** of the last byte of the `CRC` with the input byte. This means that the input byte must have been (`0xd7` $\oplus$ `0x86`) = `0x51`. This value corresponds to ASCII 'Q'. We have recovered the last byte of the message!

5 The CRC-32 algorithm

```
const uint32_t crc_tbl[] = {
0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419, 0x706af48f,
0xe963a535, 0x9e6495a3,0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0x97d2d988,
0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91, 0x1db71064, 0x6ab020f2,
0xf3b97148, 0x84be41de,0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7,
0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec,0x14015c4f, 0x63066cd9,
0xfa0f3d63, 0x8d080df5,0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0xa2677172,
0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b,0x35b5a8fa, 0x42b2986c,
0xdbbbc9d6, 0xacbcf940,0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0xabd13d59,
0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0x56b3c423,
0xcfba9599, 0xb8bda50f, 0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924,
0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d,0x76dc4190, 0x01db7106,
```

```
0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0xe8b8d433,
0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818, 0x7f6a0dbb, 0x086d3d2d,
0x91646c97, 0xe6635c01, 0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e,
0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457, 0x65b0d9c6, 0x12b7e950,
0x8bbeb8ea, 0xfcb9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65,
0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2, 0x4adfa541, 0x3dd895d7,
0xa4d1c46d, 0xd3d6f4fb, 0x4369e96a, 0x346ed9fc, 0xad678846, 0xda60b8d0,
0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9, 0x5005713c, 0x270241aa,
0xbe0b1010, 0xc90c2086, 0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17, 0x2eb40d81,
0xb7bd5c3b, 0xc0ba6cad, 0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a,
0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683, 0xe3630b12, 0x94643b84,
0x0d6d6a3e, 0x7a6a5aa8, 0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1,
0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe, 0xf762575d, 0x806567cb,
0x196c3671, 0x6e6b06e7, 0xfed41b76, 0x89d32be0, 0x10da7a5a, 0x67dd4acc,
0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5, 0xd6d6a3e8, 0xa1d1937e,
0x38d8c2c4, 0x4fdff252, 0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0x48b2364b,
0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60, 0xdf60efc3, 0xa867df55,
0x316e8eef, 0x4669be79, 0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236,
0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f, 0xc5ba3bbe, 0xb2bd0b28,
0x2bb45a92, 0x5cb36a04, 0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b, 0x5bdeae1d,
0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a, 0x9c0906a9, 0xeb0e363f,
0x72076785, 0x05005713, 0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38,
0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21, 0x86d3d2d4, 0xf1d4e242,
0x68ddb3f8, 0x1fda836e, 0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777,
0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c, 0x8f659eff, 0xf862ae69,
0x616bffd3, 0x166ccf45, 0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2,
0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db, 0xaed16a4a, 0xd9d65adc,
0x40df0b66, 0x37d83bf0, 0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9,
0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6, 0xbad03605, 0xcdd70693,
0x54de5729, 0x23d967bf, 0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94,
0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d
};

Function CRC32
   Input:
      data:  Bytes     //Array of bytes
   Output:
      crc32: uInt32    //32-bit unsigned crc-32 value
//Initialize crc-32 to starting value
crc32 <- 0xFFFFFFFF
//CRCTable is an array of 256 32-bit constants
for each byte in data do
   tblIndex <- (crc32 xor byte) & 0xFF;
   crc32 <- (crc32 >> 8) xor crc_tbl[tblIndex]
//Finalize the CRC-32 value by inverting all the bits
crc32 <- crc32 xor 0xFFFFFFFF
return crc32
```