Cracking WEP40 using a FMS attack

Group members:
Christoforos Seas cseas@kth.se
Emil Sjölander emilsjol@kth.se
Anass Inani inani@kth.se
Giovanni Prete prete@kth.se

Overview

For our project we implemented a WEP-attack, aimed at the WEP40 protocol. We implemented a FMS attack. Ideally we would have implemented it on captured data, but it proved difficult to acquire a router that had WEP functionality since it is so old and unsafe and thus not available on newer routers. Instead we generated packets to simulate network traffic, and implemented the attack on those instead. We managed to more often than not crack the keys we provided for the sample data we generated.

We chose to do this for our project since we had an interest in WiFi security, which is so common in our lives. Surely there would be some older parts of wifi security we could delve deeper into in order to improve our understanding of modern wifi security. We found that WEP was previously the standard but was shown to be very insecure and thus thought it would be a good idea to implement an attack and try to learn something from the vulnerabilities we re-discover.

Background

WEP, wired equivalent privacy, was ratified as security standard for wifi 1999. Only a few years later however, in 2001, it was shown to be insecure and insufficient. A part of why it was so weak was that the USA had restrictions on security protocols that could be exported, which limited the key size to 64 bits in total. Nowadays there are multiple ways to crack it, and it has been shown to be crackable in a few minutes using publicly available tools by the FBI. One such tool is aircrack-ng, which can perform the same thing we implemented in our project. Attempts to fix WEP were ineffective, and it was later superseded by WPA/WPA2.

WEP 40 works by concatenating a 24 bit IV with a 40 bit key. If run in this configuration it is commonly called WEP40, due to the size of the key. During the algorithm, the IV is not secret. It is a fast and simple algorithm. Versions with larger keys are available, such as WEP104. The key is generated with RC4.

RC4, Rivest Cipher 4, is a stream cipher. It was initially secret but the source code was later leaked. The idea for RC4 is to generate a pseudorandom keystream and use it to encode the plaintext by bitwise XOR. It works by first performing a key-scheduling algorithm (KSA) followed by a pseudo-random generation algorithm (PRGA). First, in KSA, it fills an array with permutations of the key, called S. It then performs a similar operation in PRGA, mixing in the values of the key. It encrypts and decrypts on a byte-basis.

Vulnerability summary

Since RC4 is a stream cipher, each key must not be repeated. The IV is intended to help prevent this repetition. However, a 24 bit IV is not sufficient given enough traffic on a network. If there is not enough traffic, an attacker could find ways to cause it, for example by sending out traffic of their own on the network in order to get other computers to respond. One such attack is the Fluhrer, Mantin and Shamir attack (FMS attack). They have discovered that certain IVs are vulnerable to disclosing information about the key. Given that an attacker knows the first byte of the key and the IV, their attack could, given enough packets, estimate the next value of the key. This could be repeated for the next byte, given that enough packets of a specific sort vulnerable for the next byte of the key have been collected, and so on. Unluckily for WEP, the first byte of the key is often 0xAA (hexadecimal format, known as the SNAP header) which means the FMS attack often succeeds given enough packets to analyze.

Implementation:

Firstly, we wanted to test fms attack, so we created a python program to generate packets using a given key with maximum size 5 bytes. These packets have weak IVs and the first byte of the payload (which the plaintext is 0xAA, the SNAP Header). It generates all possible packets with both forms.

First form:

IVs of type (a+3, 255, x), a ranging from 0 to key length and x ranging from 0 to 255

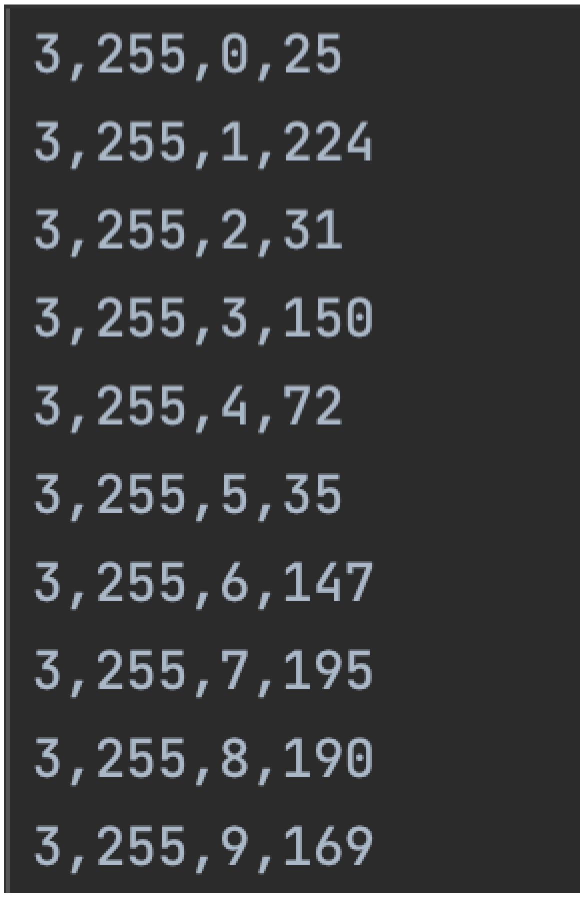
Second form:

(v, 257-2, 255), v ranging from 2 to 255.

In other words, first form has values (3, 255, 0), (3, 255, 1), until (3, 255, 255), and then (4, 255, 0), (4, 255, 1), until (4, 255, 255), and this pattern repeats for until a = 4, so the last 256 values are (7, 255, 0), (7, 255, 1), until (7, 255, 255).

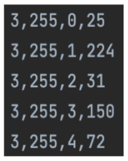
The second form has values (2, 255, 255), (3, 254, 255), until (255, 2, 255), so in the form (v, 257-v, 255), where v is in range [2, 255].

Example for key: 0x4341434343 (“CACC”):

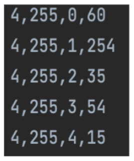


```
3,255,0,25
3,255,1,224
3,255,2,31
3,255,3,150
3,255,4,72
3,255,5,35
3,255,6,147
3,255,7,195
3,255,8,190
3,255,9,169
```

FIRST FORM:

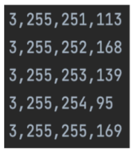


```
3,255,0,25
3,255,1,224
3,255,2,31
3,255,3,150
3,255,4,72
```

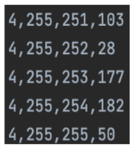


```
4,255,0,60
4,255,1,254
4,255,2,35
4,255,3,54
4,255,4,15
```

••• •••

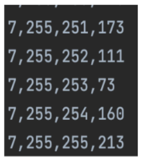


```
3,255,251,113
3,255,252,168
3,255,253,139
3,255,254,95
3,255,255,169
```



```
4,255,251,103
4,255,252,28
4,255,253,177
4,255,254,182
4,255,255,50
```

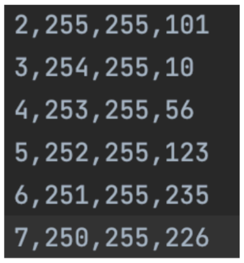
•••



```
7,255,251,173
7,255,252,111
7,255,253,73
7,255,254,160
7,255,255,213
```

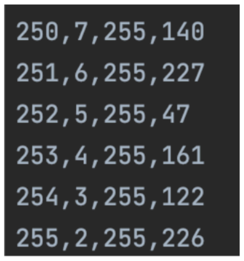
```
3, 255, 0, 0xAA ^
first keystream byte

…

3, 255, 255, 0xAA ^
first keystream byte

…

4, 255, 255, 0xAA ^
first keystream byte

…

7, 255, 255, 0xAA ^
first keystream byte
```

SECOND FORM:



```
2,255,255,101
3,254,255,10
4,253,255,56
5,252,255,123
6,251,255,235
7,250,255,226
```

•••

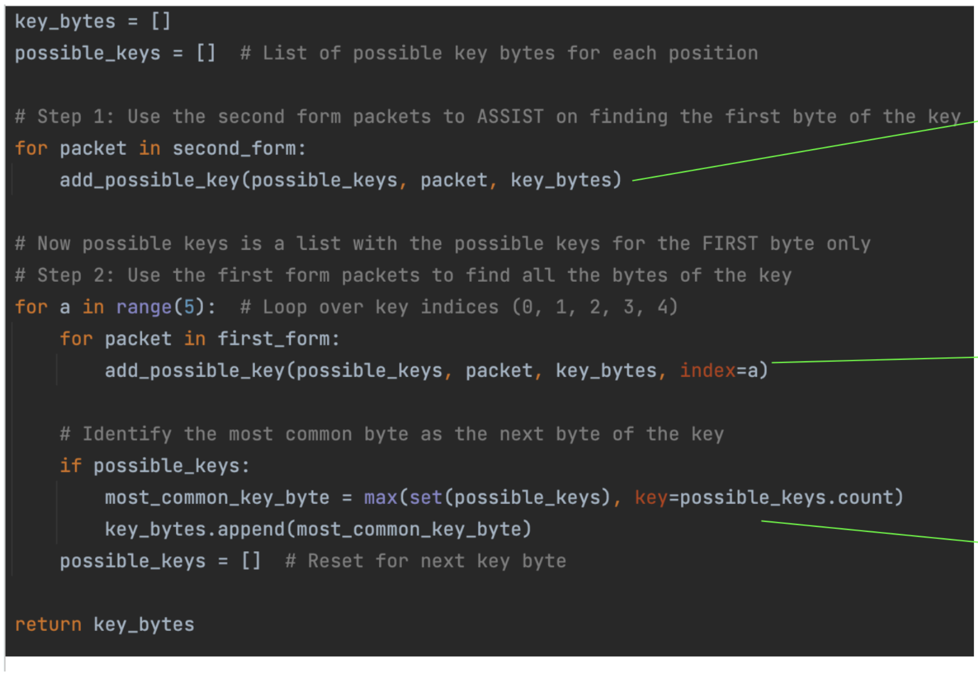


```
250,7,255,140
251,6,255,227
252,5,255,47
253,4,255,161
254,3,255,122
255,2,255,226
```

```
2, 255, 255, 0xAA ^
first keystream byte

…

255, 2, 255, 0xAA ^
first keystream byte
```

FMS Attack:

Then we run our program which finds the WEP key either from the CSV or the .pcap file.



```python
key_bytes = []
possible_keys = []  # List of possible key bytes for each position

# Step 1: Use the second form packets to ASSIST on finding the first byte of the key
for packet in second_form:
    add_possible_key(possible_keys, packet, key_bytes)

# Now possible keys is a list with the possible keys for the FIRST byte only
# Step 2: Use the first form packets to find all the bytes of the key
for a in range(5):  # Loop over key indices (0, 1, 2, 3, 4)
    for packet in first_form:
        add_possible_key(possible_keys, packet, key_bytes, index=a)

    # Identify the most common byte as the next byte of the key
    if possible_keys:
        most_common_key_byte = max(set(possible_keys), key=possible_keys.count)
        key_bytes.append(most_common_key_byte)
    possible_keys = []  # Reset for next key byte

return key_bytes
```

Without index it adds the possible values for the first key byte using the second form

Add the possible values of the key byte a (first, then second, etc.)

Find the most common key byte value and assume that this is the original key byte at position a

Workload

Christoforos Seas:

1. Implemented packet generator algorithm (CSV)
2. Implemented WEP cracking algorithm (from CSV)
3. Created draft presentation (skeleton)
4. Created the implementation slides and report
5. Finished README.md

Anass Inani:

1. Contributed in the presentation
2. Created Readme.md

Giovanni Prete:

1. Configured WEP wireless router
2. Created LAN network to exchange packets
3. Captured encrypted wireless packets via network interface
4. Wrote pcap parser for preliminary packet elaboration
5. Contributed in the presentation

Emil Sjölander

1. Communicated and attempted to coordinate group
2. Contributed to idea and structure of project
3. Performed research and assisted in development of attack
4. Contributed to slides regarding intro, history, overview as well as WEP40 and RC4
5. Contributed to report regarding background, overview as well as vulnerability explanation