Lab Work 1: RC4 Attack

Aleix Galan Figueras - 26062500G
Data Protection
Universitat Politècnica de Catalunya

29 de setembre de 2020

Introduction

The idea behind this attack is to exploit a vulnerability in RC4 produced by the encryption of the same first byte repeatedly. This case can be usually produced in the Internet due to the existence of various internet protocols where the header is the same in all the encrypted messages. This attack also requires of an IV prepended to the long term key of at least 3 bytes, and access to a bunch of messages. These two requirements are also feasible to obtain in a modern scenario.

The first step of the attack will be to guess the first byte of the clear message (`m[0]`) detecting the special IV value of *01FFx* where x is the third byte of the IV value. Then, with the found `m[0]` we will be able to obtain the long term key bytes by doing a similar operation in the IV values of *03FFx*, *04FFx*, etc. The process will be further explained in the correspondent section.

Generating data

To generate the RC4 encryption data (IVs and cipher-text) that will be used to demonstrate the attack we will use OpenSSL calls via Bash language. The script will simply iterate through the possible IV values, call the OpenSSL commands to encrypt and save the result (IV, cipher-text) in a text file. We will save the data used to recover `m[0]` and the data used to recover the key (`k`) in two different files to facilitate the future handling. The data will be saved in a csv format.

```bash
#!/bin/bash

subkey='000102030405060708090a0b0c'
constant_m0='a'

#### Generating m0 data ####

iv_base='01ff'
echo "IV,Ciphertext" > data_m0.txt

for iv in {0..255};
do
    iv_hex=$(printf "%02x" $iv)
    key="$iv_base$iv_hex$subkey"
    echo -n $iv_base$iv_hex"," >> data_m0.txt
    echo -n $constant_m0 | openssl enc -K $key -rc4 | xxd -p >> data_m0.txt
done

##### Generating k data #####

echo "IV,Ciphertext" > data_k.txt

for f_iv in {3..16};
do
    f_iv_hex=$(printf "%02x" $f_iv)
    for l_iv in {0..255};
    do
        l_iv_hex=$(printf "%02x" $l_iv)
        key="$f_iv_hex"ff"$l_iv_hex$subkey"
        echo -n $f_iv_hex"ff"$l_iv_hex"," >> data_k.txt
        echo -n $constant_m0 | openssl enc -K $key -rc4 | xxd -p >> data_k.txt
    done
done

```

Figura 1: Bash script to generate RC4 data.

Obtaining `m[0]`

To obtain m[0] we must exploit the IV value `01FFx`. Each cipher-text byte is produced by XORing a message byte with a key-stream byte that is derived from the key. The vulnerability is found in that the key-stream byte used to compute the first cipher-text byte is often $x + 2$ (where x is the third byte of the IV value).

Then, all that we have to do is generate all possible `m[0]` values and, for each IV in `01FFx`, XOR the m[0] value with the cipher-text and find if the result is equal to $x+2$. If it's equal, we add 1 to the frequency of that m[0]. At the end of all the `01FF` values, the `m[0]` with most frequency will be the real `m[0]`.

The following code in Python reads the csv file with all the IV and cipher-text values. For each line in the file tries all `m[0]` values and add 1 to the list position of that `m[0]` if it complies with $k+2$. Then just obtain the position with the maximum frequency and verifies the value.

```python
############# Guessing m0 #############

m0_freq_list = []

for m0_value in range(0,255):
    m0_freq_list.append(0)

with open('data_m0.txt', mode='r') as csv_file:
    csv_reader = csv.DictReader(csv_file)
    for row in csv_reader:
        for m0_value in range(0,255):
            keystream_value = m0_value ^ int('0x'+row['Ciphertext'],0)
            suposed_value = (int('0x'+row['IV'][-2:],0) + 2) % 255
            if keystream_value == suposed_value:
                m0_freq_list[m0_value] += 1

frequency = max(m0_freq_list)
m0 = m0_freq_list.index(frequency)
verification = chr(m0) == o_m0
print('m[0]: ' + chr(m0) + '\t\t(with freq. ' + str(frequency) +
        ')\t\t' + str(verification))
print('=============================')

```

Figura 2: Python code to recover the `m[0]` value.

Obtaining the long-term key

Once we have determined the `m[0]` value, we can recover all the value of the key-stream for all the first bytes just XORing the `m[0]` with the first cipher-text byte. There is another vulnerability that links the value of the key-stream with bytes of the long-term key. This vulnerability also uses specific IV values and a probability approach.

In order to obtain the first byte of the long-term key we have to find the IV value of `03FFx`. Then, the first byte of the key-stream will often be $x + 6 + k[0]$ where `k[0]` is the first byte of the long-term key. So we have to generate all possible values of `k[0]` and find the one with more probability.

The second byte of the long-term key is found in the IV values `04FFx` and complies with $x + 10 + k[0] + k[1]$, and so on. The counter term in the guessing formula is a summation starting at 3 for the first byte.

The following Python code reads the csv file with all the IVs and cipher-texts. Starting at the values for the first byte of the long-term key reads all the correspondent IVs and checks what `k` value complies with the formula. When the last IV of the current block is detected, gets the `k` with the highest frequency and verifies the result. Then, updates the counter values for the formula and the IV number and proceeds with the next byte.

```python

############# Guessing key #############

recovered_key = []
iv_counter = 3
guess_counter = 6
k_byte = 0

with open('data_k.txt', mode='r') as csv_file:
    csv_reader = csv.DictReader(csv_file)
    for row in csv_reader:

        if row['IV'] == '0'+format(iv_counter, 'x')+'ff00':
            # First value of this k
            # Set the frequence list to 0
            k_freq_list = []
            for k_value in range(0,255):
                k_freq_list.append(0)

        for k_value in range(0,255):
            # Computes the guessing for this IV value
            keystream_value = m0 ^ int('0x'+row['Ciphertext'],0)
            suposed_value = (int('0x'+row['IV'][-2:],0) + guess_counter + k_value) % 255
            if keystream_value == suposed_value:
                k_freq_list[k_value] += 1

        if row['IV'] == '0'+format(iv_counter, 'x')+'ffff':
            # Last value of this k
            # Saves and compares the guessed k
            frequency = max(k_freq_list)
            k = k_freq_list.index(frequency)
            recovered_key.append(k)
            verification = k == int('0x'+o_key[k_byte*2:k_byte*2+2],0)
            print('k['+str(k_byte)+']: ' + "{0:#0{1}x}".format(k,4) +
                    '\t(with freq. ' + str(frequency) + ')\t\t' + str(verification))

            # Update the counters for next k guessing
            iv_counter += 1
            guess_counter += iv_counter + k
            k_byte += 1

```

Figura 3: Python code to recover the long-term k value.

Example of execution

This is the result of executing the Python code. The files with the code can be found on `https://github.com/Algafix/problemes-dprot/tree/master/practica1`.

First execute the file `generate_rc4_data.sh` (this may take a while, you can skip this step and use the already generated files). Then, execute the python file `rc4_crack.py`.

```
message is: a
key is: 000102030405060708090a0b0c

=============================
m[0]: a         (with freq. 36)         True
=============================
k[0]: 0x00      (with freq. 9)          True
k[1]: 0x01      (with freq. 10)         True
k[2]: 0x02      (with freq. 13)         True
k[3]: 0x03      (with freq. 12)         True
k[4]: 0x04      (with freq. 10)         True
k[5]: 0x05      (with freq. 14)         True
k[6]: 0x06      (with freq. 13)         True
k[7]: 0x07      (with freq. 8)          True
k[8]: 0x08      (with freq. 6)          True
k[9]: 0x09      (with freq. 9)          True
k[10]: 0x0a     (with freq. 9)          True
k[11]: 0x0b     (with freq. 8)          True
k[12]: 0x0c     (with freq. 7)          True
=============================

 Key recovered!
        000102030405060708090a0b0c == 000102030405060708090a0b0c
```

Figura 4: Execution of the RC4 crack.