Lab#1 - RC4 practical attack

1. The attack scenario

In this section we propose a easy-to-implement key-recovery attack against a particular mode of operation of RC4.

We need several assumptions to launch a successful attack:

1. The initialization value iv is a 3-byte counter prepended to a 13-byte long-term key, making a 16-byte string that is used as the actual RC4 key.
2. The iv is incremented in every new encryption (from *01FF00* to *FFFFFF*).
3. RC4 is used in a communication system to send some well-structured packets with a constant header of at least one byte m[0].
4. The attacker has access to many encryptions, so that he can wait for the use of some specific iv values.

The attack is based on the following facts:

1. For any byte *x*, using *iv=01FFx* results in a keystream such that its first byte equals *x+2* with high probability.
2. The first keystream byte produced with *iv=03FFx* is, with a noticeable probability, *x+6+k[0]*, where *k[0]* denotes the first byte of the long-term key. Similarly, *iv=04FFx* produces the first keystream byte equal to *x+10+k[0]+k[1]* with a noticeable probability.
3. In general, for *i* ranging from 0 to 12, using *iv=zFFx* where *z* is the hexadecimal representation of *i+3* often produces the first keystream byte equal to *x+d[i]+k[0]+k[1]+...+k[i]*, where *d[i]* is

the constant
1+2+...+(i+3).

2. Simulating the attack

2.1 Channel Eavesdropping

To simulate the attack I wrote a python script which generates a 1-byte message, a 13-bytes key and an IV incrementing from 01FF00 to 0FFFFF.

For each of the IV values mentioned before, it concatenates it with the key and encrypt m[0] with a python RC4 implementation called ARC4 from the cryptodome package v3.19.0.

After each m[0] encryption the IV used and the ciphertext generated are stored in a text file.

```
# generate 1-byte message
message = get_random_bytes(1)
# generate 13-bytes key
key = get_random_bytes(13)

# create file, empty it if exists
cipher_file = "./ciphertexts.txt"
open(cipher_file, "w").close()

# generate all IVs from 01FF00 to 0FFFFF
for i in range(1, 17):
        # from 01 to 0F
        beg = format(i, "02X")
        for j in range(256):
                # from 00 to FF
                end = format(j, "02X")
                iv = beg + "FF" + end
                enc_key = bytes.fromhex(iv) + key_byte
                cipher = ARC4.new(enc_key)
                c = cipher.encrypt(message)
                with open(cipher_file, "a") as cifile:
                        cifile.write(f"{iv} {c.hex()}\n")
```

A sample of how the encryption results looks like:

```
01FF00 64
01FF01 a6
01FF02 a1
01FF03 a0
01FF04 4c
...
0FFFFB fb
0FFFFC 55
0FFFFD a7
0FFFFE 73
0FFFFF 36
```

2.2 Fact 1

📌 For any byte *x*, using *iv=01FFx* results in a keystream such that its first byte equals *x+2* with high probability.

To demonstrate fact 1, I wrote a python script which:

1. Opens the file containing all the ciphertexts
2. Extracts the lines starting by 01FF
3. Compute all the $c[0] \oplus (x+2)$
4. Compare the message guessed with the original message

```
# open file & extract lines
with open(cipher_file, "r") as cifile:
    iv_and_cipher = cifile.readlines()[0:256]

guesses = []

# compute XOR for each line
for line in iv_and_cipher:
    parts = line.split()
    x = parts[0][-2:]
```

```
    c = parts[1]
    # c[0] XOR (x+2)
    result = (int(c, 16) ^ (int(x, 16) + 2)) % 256
    guesses.append(hex(result))

# Find most frequent result
counts = Counter(guesses).most_common(1)
msg_guessed = counts[0][0]
nb_times = counts[0][1]

print(f"[MSG] Expected {message} got {msg_guessed}, found {nb_
```

Example output:

```
[MSG] Expected 0x80 got 0x80, found 46 times
```

2.3 Fact 2

📌 The first keystream byte produced with *iv=03FFx* is, with a noticeable probability, *x+6+k[0]*, where *k[0]* denotes the first byte of the long-term key.

Similarly, *iv=04FFx* produces the first keystream byte equal to *x+10+k[0]+k[1]* with a noticeable probability.

2.3.1 For IV=03FFx

My python script:

1. Opens the file containing all the ciphertexts
2. Extracts the lines starting by 03FF
3. Computes all the $(c[0] \oplus m[0]) - x - 6$
4. Compares the key byte guessed with the expected one

```
# open file & extract lines starting by 03FF
with open(cipher_file, "r") as cifile:
    iv_and_cipher = cifile.readlines()[512:768]
```

```
key_guesses = []

# compute XOR for each line
for line in iv_and_cipher:
    parts = line.split()
    x = parts[0][-2:]
    c = parts[1]
    # (c[0] XOR m[0]) - x - 6
    guess = (int(c, 16) ^ int(msg_guessed, 16)) - int(x, 16)
    key_guesses.append(hex(guess))

# Find most frequent result
counts = Counter(key_guesses).most_common(1)
k0 = counts[0][0]
nb_times = counts[0][1]

print(f"[KEY 0] Expected {key.hex()[0:2]} got {k0}, found {nb_
```

Example output:

```
[KEY 0] Expected 0x7a got 0x7a, found 13 times
```

2.3.2 For IV=04FFx

My python script:

1. Opens the file containing all the ciphertexts
2. Extracts the lines starting by 03FF
3. Computes all the $(c[0] \oplus m[0]) - x - 10 - k[0]$
4. Compares the key byte guessed with the expected one

```
# open file & extract lines starting by 04FF
with open(cipher_file, "r") as cifile:
    iv_and_cipher = cifile.readlines()[768:1024]

key_guesses = []
```

```
# compute XOR for each line
for line in iv_and_cipher:
    parts = line.split()
    x = parts[0][-2:]
    c = parts[1]
        # (c[0] XOR m[0]) - x - 10 - k[0]
    guess = ((int(c, 16) ^ int(msg_guessed, 16))-int(x, 16)-1
        key_guesses.append(hex(guess))

# Find most frequent result
counts = Counter(key_guesses).most_common(1)
k1 = counts[0][0]
nb_times = counts[0][1]

print(f"[KEY 1] Expected {key.hex()[2:4]} got {k1}, found {nb
```

Example output:

```
[KEY 1] Expected 0xf3 got 0xf3, found 11 times
```

2.4 Fact 3

📌 In general, for *i* ranging from 0 to 12, using $iv = zFFx$ where *z* is the hexadecimal representation of $i + 3$ often produces the first keystream byte equal to $x + d[i] + k[0] + k[1] + ... + k[i]$, where $d[i]$ is the constant $1 + 2 + ... + (i + 3)$.

My python script:

1. Loop through i from 0 to 12
2. Extracts the lines of IV iFFx (01FFx, 03FFx, etc...)
3. Computes all the $(c[0] \oplus m[0]) - x - d[i] - k[0] - k[1] - ... - k[i]$
4. Prints all the guessed key bytes

```
all_guesses = []
# from i=0 to i=12
```

```
for i in range(13):
    # compute lines to be extracted
    iv_beg = 256 * (i + 3) - 256
    iv_end = 256 * (i + 3)
    with open(cipher_file, "r") as cifile:
        iv_and_cipher = cifile.readlines()[iv_beg:iv_end]

    key_guesses = []

        # compute XOR for each line
    for line in iv_and_cipher:
        parts = line.split()
        x = parts[0][-2:]
        c = parts[1]
                # compute d[i]=1+2+...+(i+3)
        d = sum(range(1, i + 4))
        # (c[0] XOR m[0]) - x - d[i] - k[0] - k[1] - ... - k[
        result = (
            (int(c, 16) ^ int(msg_guessed, 16))
            - int(x, 16)
            - d
            - sum(int(x, 16) for x in guessing_key[0 : i + 1]
        ) % 256
        key_guesses.append(hex(result))

        # Find most frequent result
        counts = Counter(key_guesses).most_common(1)
        k = counts[0][0]
        nb_times = counts[0][1]
        all_guesses.append(k)

        print(f"[KEY {i}] {k}, found {nb_times} times")

guessed_key = [byte[2:] for byte in guessing_key]
result = "0x" + ''.join(guessed_key)
print(f"Guessed : {result}")
print(f"Original: 0x{key.hex()}")
```

Example output:

```
[KEY 0]: 0xda, found 9 times
[KEY 1]: 0xeb, found 14 times
[KEY 2]: 0xb7, found 14 times
[KEY 3]: 0xde, found 8 times
[KEY 4]: 0xd0, found 13 times
[KEY 5]: 0x3a, found 15 times
[KEY 6]: 0x71, found 10 times
[KEY 7]: 0x31, found 9 times
[KEY 8]: 0xc1, found 16 times
[KEY 9]: 0x86, found 12 times
[KEY 10]: 0x6c, found 7 times
[KEY 11]: 0xca, found 9 times
[KEY 12]: 0x7d, found 8 times
Guessed : 0xdaebb7ded03a7131c1866cca7d
Original: 0xdaebb7ded03a7131c1866cca7d
```

In this example I could guess every key bytes successfully but it happens that I get between 1 and 4 wrong guesses. Usually the wrong guesses are not too far from the original one.

3. A Practical attack

I downloaded all the files `bytes_xxFFxx.dat` and formatted all the lines to a prettier look:

From:

```
0X01FF00 0XDB
0X01FF01 0X60
0X01FF02 0XE8
0X01FF03 0X74
0X01FF04 0X95
...
```

To:

```
01FF00 DB
01FF01 60
01FF02 E8
01FF03 74
01FF04 95
...
```

Then I applied the same piece of code than for Fact 1 and Fact 3:

```
cipher_file = "./prof_ciphertexts.txt"

# GUESS MESSAGE

# open file & extract lines
with open(cipher_file, "r") as cifile:
    iv_and_cipher = cifile.readlines()[0:256]

guesses = []

# compute XOR for each line
for line in iv_and_cipher:
    parts = line.split()
    ivx = parts[0][-2:]
    ciphertext = parts[1]
        # c[0] XOR (x+2)
    result_hex = hex((int(ciphertext, 16) ^ (int(ivx, 16) + 2
    guesses.append(result_hex)

# Find most frequent result
counts = Counter(guesses).most_common(1)
msg_guessed = counts[0][0]
nb_msg = counts[0][1]
guessing_key = []

# GUESS KEYS

for i in range(12):
        # compute lines to be extracted
    iv_beg = 256 * (i + 3) - 256
    iv_end = 256 * (i + 3)

        # open file & extract lines
    with open(cipher_file, "r") as cifile:
        iv_and_cipher = cifile.readlines()[iv_beg:iv_end]

    key_guesses = []
```

```python
        # compute XOR for each line
    for line in iv_and_cipher:
        parts = line.split()
        ivx = parts[0][-2:]
        ciphertext = parts[1]
        d_i = sum(range(1, i + 4))
        # (c[0] XOR m[0]) - x - d[i] - k[0] - k[1] - ... - k[
        result = (
            (int(ciphertext, 16) ^ int(msg_guessed, 16))
            - int(ivx, 16)
            - int(d_i)
            - sum(int(x, 16) for x in guessing_key[0 : i + 1]
        ) % 256
        key_guesses.append(hex(result))

        # Find most frequent result
    counts = Counter(key_guesses).most_common(1)
    k = counts[0][0]
    guessing_key.append(k)

# print results
guessed_key = [byte[2:] for byte in guessing_key]
result = "0x" + "".join(guessed_key)
print(f"I think the key is: {result}")
print(f"I think the msg is: {msg_guessed}, found {nb_msg} tim
```

After running this piece of code I found the following results:

```
I think the key is: 0x44b44a85fa1f26dc60fa6abe56
I think the msg is: 0xc2, found 40 times
```