Data Protection LAB 1

Team: Oriol Lalaguna Royo , Tsagiopoulou Sofia

Data generation

To implement the attack, we simulated the eavesdropping of a communication channel by generating a message of 1 byte and a key of 13 bytes with the created executable bash file *data.sh*.

```
k=$(openssl rand -hex 13)
m=$(openssl rand -hex 1)
me=$(echo "$m"| xxd -r -p)
echo "$m" >> message2
echo "$me" >> message
```

Generating the random message and key (data.sh)

The rc4 encrypts the message byte by byte and the keystream that is used for the encryption is a combination of a nonce (iv value) and a random key. In our case the iv value is a 3 bytes counter with values 01FFxx, 03FFxx… with xx being a counter value. Each value of the ivs was concatenated with the rand key to encrypt the one-byte message. Then the ciphers and the ivs were stored into files, these files in real communication channels are transferred from the sender to the receiver.

Both, the message, and the key, were generated with openssl rand command. For the encryption of the message it was mandatory to revert the hex message into binary with the command 'xxd -r -p'. The -p is giving us only the plaintext without the additional information like line number.

For each value of the ivs (01ff,03ff,04ff,05ff,06ff,07ff,08ff,09ff,0aff,0bff,0cff,0dff,0eff,0fff) its counter (xx) is increased by one for 255 times and every value concatenated with the random key to generate the key that was used for the encryption.

```
z=0
p=ff
w=xx
l=15

counter=0
for v in {01,03,04,05,06,07,08,09,0a,0b,0c,0d,0e,0f};
do
    for ((i=0; i<=$n; i++));
    do
        myvar=$v$p
        echo "$myvar"
        printf -v x %x $(($counter))
        if (($i<=$l))
        then
            key=$myvar$z$x
        else
            key=$myvar$x
        fi
        echo "$key"

        keystream=$key$k
        echo "$keystream" > key
        cipher=$(echo -n "$me"| openssl enc -K "$keystream" -nosalt -rc4| xxd -p)
```

Value concatenation (data.sh)

The encryption is done by openssl command using -n command to remove the '/n' character from the message and without salt (as with salted encryption we will get a cipher greater than 1byte). The output of the encryption is in hex dump plaintext. The ciphers of an iv are stored in a file and the corresponding ivs in another file. Then these files are combined in one, with space as delimiter.

```
            echo "0X$cipher" |tr '[a-z]' '[A-Z]' >> /home/ubuntu/dp/ciphers$myvarca
        echo "0X$key" |tr '[a-z]' '[A-Z]'>> /home/ubuntu/dp/ivs$myvarca
        let counter++
    done
counter=0
paste -d ' ' /home/ubuntu/dp/ivs$myvarca /home/ubuntu/dp/ciphers$myvarca > /home/ubuntu/dp/bytes_$myvarca$w.dat
done
```

File concatenation (data.sh)

To generate exactly the right format of the file for the attack we transformed the lower-case letters in upper case.

The final format of the files is like the below.

```
0X0AFF00 0X2A
0X0AFF01 0X99
0X0AFF02 0X83
0X0AFF03 0XD4
0X0AFF04 0XE4
0X0AFF05 0XA6
0X0AFF06 0XC0
0X0AFF07 0XCB
0X0AFF08 0X5E
0X0AFF09 0X33
0X0AFF0A 0X50
0X0AFF0B 0XAE
0X0AFF0C 0X3F
0X0AFF0D 0XA0
0X0AFF0E 0XB8
0X0AFF0F 0X90
0X0AFF10 0XD2
0X0AFF11 0XD1
0X0AFF12 0XD4
0X0AFF13 0X76
0X0AFF14 0XE9
0X0AFF15 0XBE
0X0AFF16 0X51
0X0AFF17 0X88
0X0AFF18 0X72
0X0AFF19 0XA3
0X0AFF1A 0X09
0X0AFF1B 0X3F
0X0AFF1C 0X4C
0X0AFF1D 0XF5
0X0AFF1E 0X26
0X0AFF1F 0XB1
```

Output example

Guessing the secret values

After successfully generating all the required .dat files and ensuring the format is the correct one comparing it to the official .dat files, we proceeded to code in python all the execution to decrypt the message and key from that files. That executable python file is called *simulateattack.py*.

The procedure to execute the attack is the following. First, we created a function to read all the information of ivs and ciphertext doing a split and save it to variables.

```python
def readtext(filetoread):
    global mylines
    global myivs
    mylines = []
    myivs = []
    with open (filetoread, 'rt') as myfile:
        for myline in myfile:
            myivs.append(myline.strip('\n').split(" ")[0])
            mylines.append(myline.strip('\n').split(" ")[1])
```

Readtext function (simulateattack.py)

Then, using the above function we did all the XOR for the different fact1, fact2 and fact3 procedures to read each different file, because each one contains different information. The function of XOR for the message decryption (fact 1) computing the 256 values of c[0] XOR (x+2) for the file *bytes_01FFxx.dat* is the following.

```python
def XOR_M():
    readtext("bytes_01FFxx.dat")
    global myxor_m
    global m
    myxor_m = []
    i = 0
    while i < 256:
        y = int(mylines[i].split("X")[1], 16)                # convert hex to int with or without 0x
        if i == 254:                                         # 255 + 2 = 0
            z = 0
        if i == 255:                                         # 256 + 2 = 1
            z = 1
        else:
            z = int(myivs[i].split("01FF")[1], 16) + 2
        myxor_m.append(y^z)
        i = i + 1
    m = most_frequent(myxor_m)
```

Fact1 function (simulateattack.py)

The function XOR for the key0 decryption (fact 2) to recover k[0] as the most frequent value of (c[0] XOR m[0])-x-6 for the file *bytes_03FFxx.dat* is the following.

```python
def XOR_K0():
    readtext("bytes_03FFxx.dat")
    global k
    k = []
    myxor_k0 = []
    i = 0
    while i < 256:
        y = int(mylines[i].split("X")[1], 16)                    # convert hex to int with or without 0x
        z = int(myivs[i].split("03FF")[1], 16)
        myxor_k0.append(((y^m)-z-6)%256)
        i = i + 1
    k.append(myxor_k0)
```

Fact2 function (simulateattack.py)

The function XOR for the rest of the key bytes using the rest of the files is the following (fact 3).

```python
def XOR_KX(num, o):
    if num == 10:
        num = 'A'
    if num == 11:
        num = 'B'
    if num == 12:
        num = 'C'
    if num == 13:
        num = 'D'
    if num == 14:
        num = 'E'
    if num == 15:
        num = 'F'
    readtext("bytes_0{}FFxx.dat".format(num))
    myxor_kx = []
    i = 0

    while i < 256:
        y = int(mylines[i].split("X")[1], 16)                    # convert hex to int with or without 0x
        z = int(myivs[i].split("0{}FF".format(num))[1], 16)
        if o == 0:
            myxor_kx.append(((y^m)-z-10-k0)%256)
        if o == 1:
            myxor_kx.append(((y^m)-z-(1+2+3+4+5)-k0-k1)%256)
        if o == 2:
            myxor_kx.append(((y^m)-z-(1+2+3+4+5+6)-k0-k1-k2)%256)
        if o == 3:
            myxor_kx.append(((y^m)-z-(1+2+3+4+5+6+7)-k0-k1-k2-k3)%256)
        if o == 4:
            myxor_kx.append(((y^m)-z-(1+2+3+4+5+6+7+8)-k0-k1-k2-k3-k4)%256)
        if o == 5:
            myxor_kx.append(((y^m)-z-(1+2+3+4+5+6+7+8+9)-k0-k1-k2-k3-k4-k5)%256)
        if o == 6:
            myxor_kx.append(((y^m)-z-(1+2+3+4+5+6+7+8+9+10)-k0-k1-k2-k3-k4-k5-k6)%256)
        if o == 7:
            myxor_kx.append(((y^m)-z-(1+2+3+4+5+6+7+8+9+10+11)-k0-k1-k2-k3-k4-k5-k6-k7)%256)
        if o == 8:
            myxor_kx.append(((y^m)-z-(1+2+3+4+5+6+7+8+9+10+11+12)-k0-k1-k2-k3-k4-k5-k6-k7-k8)%256)
        if o == 9:
            myxor_kx.append(((y^m)-z-(1+2+3+4+5+6+7+8+9+10+11+12+13)-k0-k1-k2-k3-k4-k5-k6-k7-k8-k9)%256)
        if o == 10:
            myxor_kx.append(((y^m)-z-(1+2+3+4+5+6+7+8+9+10+11+12+13+14)-k0-k1-k2-k3-k4-k5-k6-k7-k8-k9-k10)%256)
        if o == 11:
            myxor_kx.append(((y^m)-z-(1+2+3+4+5+6+7+8+9+10+11+12+13+14+15)-k0-k1-k2-k3-k4-k5-k6-k7-k8-k9-k10-k11)%256)

        i = i + 1
    k.append(myxor_kx)
```

Fact3 function (simulateattack.py)

Then, we did a loop to find the most repeated secret value for the k generated string and save it to local variables.

```
while filenum < 16:
    XOR_KX(filenum, ifile)
    filenum = filenum +1
    ifile = ifile +1
    if ifile == 1:
        k1 = most_frequent(k[1])
    if ifile == 2:
        k2 = most_frequent(k[2])
    if ifile == 3:
        k3 = most_frequent(k[3])
    if ifile == 4:
        k4 = most_frequent(k[4])
    if ifile == 5:
        k5 = most_frequent(k[5])
    if ifile == 6:
        k6 = most_frequent(k[6])
    if ifile == 7:
        k7 = most_frequent(k[7])
    if ifile == 8:
        k8 = most_frequent(k[8])
    if ifile == 9:
        k9 = most_frequent(k[9])
    if ifile == 10:
        k10 = most_frequent(k[10])
    if ifile == 11:
        k11 = most_frequent(k[11])
    if ifile == 12:
        k12 = most_frequent(k[12])
```

```
def most_frequent(List):
    return max(set(List), key = List.count)
```

Getting the most freqüent value (simulateattack.py)

Then we get the key values filled with 0 in case the output is an individual number or letter.

```
message = hex(most_frequent(myxor_m))[2:].zfill(2)
key0 = hex(most_frequent(k[0]))[2:].zfill(2)
key1 = hex(most_frequent(k[1]))[2:].zfill(2)
key2 = hex(most_frequent(k[2]))[2:].zfill(2)
key3 = hex(most_frequent(k[3]))[2:].zfill(2)
key4 = hex(most_frequent(k[4]))[2:].zfill(2)
key5 = hex(most_frequent(k[5]))[2:].zfill(2)
key6 = hex(most_frequent(k[6]))[2:].zfill(2)
key7 = hex(most_frequent(k[7]))[2:].zfill(2)
key8 = hex(most_frequent(k[8]))[2:].zfill(2)
key9 = hex(most_frequent(k[9]))[2:].zfill(2)
key10 = hex(most_frequent(k[10]))[2:].zfill(2)
key11 = hex(most_frequent(k[11]))[2:].zfill(2)
key12 = hex(most_frequent(k[12]))[2:].zfill(2)
```

Filling the value with 0 and without 0x (simulateattack.py)

For the most frequent value we used the *.count* integrated python library function for the saved k string. And finally, we print the final output using the following code.

```
print("key is {}{}{}{}{}{}{}{}{}{}{}{}{} and message is {}".format(format(key0),format(key1),format(key2),format(key3),form
print("Gathering keystream first bytes for IV=01FFxx ... done")
print("Guessing m[0] ... done")
print(" Guessed m[0]= {} (with freq. {}) *** OK ***".format(format(message),myxor_m.count(most_frequent(myxor_m)), 'x'))
print("Gathering keystream first bytes for IV=03FFxx ... done\nGuessing k[0] ... done")
print(" Guessed k[0]= {} (with freq. {}) *** OK ***".format(format(key0),k[0].count(most_frequent(k[0])),'x'))
print("Gathering keystream first bytes for IV=04FFxx ... done\nGuessing k[1] ... done")
print(" Guessed k[1]= {} (with freq. {}) *** OK ***".format(format(key1),k[1].count(most_frequent(k[1])),'x'))
print("Gathering keystream first bytes for IV=05FFxx ... done\nGuessing k[2] ... done")
print(" Guessed k[2]= {} (with freq. {}) *** OK ***".format(format(key2),k[2].count(most_frequent(k[2])),'x'))
print("Gathering keystream first bytes for IV=06FFxx ... done\nGuessing k[3] ... done")
print(" Guessed k[3]= {} (with freq. {}) *** OK ***".format(format(key3),k[3].count(most_frequent(k[3])),'x'))
print("Gathering keystream first bytes for IV=07FFxx ... done\nGuessing k[4] ... done")
print(" Guessed k[4]= {} (with freq. {}) *** OK ***".format(format(key4),k[4].count(most_frequent(k[4])),'x'))
print("Gathering keystream first bytes for IV=08FFxx ... done\nGuessing k[5] ... done")
print(" Guessed k[5]= {} (with freq. {}) *** OK ***".format(format(key5),k[5].count(most_frequent(k[5])),'x'))
print("Gathering keystream first bytes for IV=09FFxx ... done\nGuessing k[6] ... done")
print(" Guessed k[6]= {} (with freq. {}) *** OK ***".format(format(key6),k[6].count(most_frequent(k[6])),'x'))
print("Gathering keystream first bytes for IV=0AFFxx ... done\nGuessing k[7] ... done")
print(" Guessed k[7]= {} (with freq. {}) *** OK ***".format(format(key7),k[7].count(most_frequent(k[7])),'x'))
print("Gathering keystream first bytes for IV=0BFFxx ... done\nGuessing k[8] ... done")
print(" Guessed k[8]= {} (with freq. {}) *** OK ***".format(format(key8),k[8].count(most_frequent(k[8])),'x'))
print("Gathering keystream first bytes for IV=0CFFxx ... done\nGuessing k[9] ... done")
print(" Guessed k[9]= {} (with freq. {}) *** OK ***".format(format(key9),k[9].count(most_frequent(k[9])),'x'))
print("Gathering keystream first bytes for IV=0DFFxx ... done\nGuessing k[10] ... done")
print(" Guessed k[10]= {} (with freq. {}) *** OK ***".format(format(key10),k[10].count(most_frequent(k[10])),'x'))
print("Gathering keystream first bytes for IV=0EFFxx ... done\nGuessing k[11] ... done")
print(" Guessed k[11]= {} (with freq. {}) *** OK ***".format(format(key11),k[11].count(most_frequent(k[11])),'x'))
print("Gathering keystream first bytes for IV=0FFFxx ... done\nGuessing k[12] ... done")
print(" Guessed k[12]= {} (with freq. {}) *** OK ***".format(format(key12),k[12].count(most_frequent(k[12])),'x'))
```

Code of the final output (simulateattack.py)

This is the output of the execution of the *simulateattack.py* script with the random generated values of the *data.sh* script:

```
ubuntu@ubuntu-VirtualBox:~/shared/DProtection$ python simulateattack.py
key is 73fc5e322c4b98dd95a9d873f5 and message is 7c
Gathering keystream first bytes for IV=01FFxx ... done
Guessing m[0] ... done
 Guessed m[0]= 7c (with freq. 31) *** OK ***
Gathering keystream first bytes for IV=03FFxx ... done
Guessing k[0] ... done
 Guessed k[0]= 73 (with freq. 9) *** OK ***
Gathering keystream first bytes for IV=04FFxx ... done
Guessing k[1] ... done
 Guessed k[1]= fc (with freq. 10) *** OK ***
Gathering keystream first bytes for IV=05FFxx ... done
Guessing k[2] ... done
 Guessed k[2]= 5e (with freq. 10) *** OK ***
Gathering keystream first bytes for IV=06FFxx ... done
Guessing k[3] ... done
 Guessed k[3]= 32 (with freq. 9) *** OK ***
Gathering keystream first bytes for IV=07FFxx ... done
Guessing k[4] ... done
 Guessed k[4]= 2c (with freq. 5) *** OK ***
Gathering keystream first bytes for IV=08FFxx ... done
Guessing k[5] ... done
 Guessed k[5]= 4b (with freq. 13) *** OK ***
Gathering keystream first bytes for IV=09FFxx ... done
Guessing k[6] ... done
 Guessed k[6]= 98 (with freq. 11) *** OK ***
Gathering keystream first bytes for IV=0AFFxx ... done
Guessing k[7] ... done
 Guessed k[7]= dd (with freq. 10) *** OK ***
Gathering keystream first bytes for IV=0BFFxx ... done
Guessing k[8] ... done
 Guessed k[8]= 95 (with freq. 13) *** OK ***
Gathering keystream first bytes for IV=0CFFxx ... done
Guessing k[9] ... done
 Guessed k[9]= a9 (with freq. 10) *** OK ***
Gathering keystream first bytes for IV=0DFFxx ... done
Guessing k[10] ... done
 Guessed k[10]= d8 (with freq. 9) *** OK ***
Gathering keystream first bytes for IV=0EFFxx ... done
Guessing k[11] ... done
 Guessed k[11]= 73 (with freq. 14) *** OK ***
Gathering keystream first bytes for IV=0FFFxx ... done
Guessing k[12] ... done
 Guessed k[12]= f5 (with freq. 14) *** OK ***
```

Testing random generated values

After generating the decryption of values (message and key) created with the *bash.sh* executable and comparing them with the originals, we have verified that the code is correct, and we have proceeded to decrypt the values of the original files.

The guessed values from the auxiliary official .dat files given from the subject material is the following:

```
ubuntu@ubuntu-VirtualBox:~/shared/DProtection$ python simulateattack.py
key is 44b44a85fa1f26dc60fa6abe56 and message is c2
Gathering keystream first bytes for IV=01FFxx ... done
Guessing m[0] ... done
 Guessed m[0]= c2 (with freq. 40) *** OK ***
Gathering keystream first bytes for IV=03FFxx ... done
Guessing k[0] ... done
 Guessed k[0]= 44 (with freq. 11) *** OK ***
Gathering keystream first bytes for IV=04FFxx ... done
Guessing k[1] ... done
 Guessed k[1]= b4 (with freq. 15) *** OK ***
Gathering keystream first bytes for IV=05FFxx ... done
Guessing k[2] ... done
 Guessed k[2]= 4a (with freq. 13) *** OK ***
Gathering keystream first bytes for IV=06FFxx ... done
Guessing k[3] ... done
 Guessed k[3]= 85 (with freq. 14) *** OK ***
Gathering keystream first bytes for IV=07FFxx ... done
Guessing k[4] ... done
 Guessed k[4]= fa (with freq. 15) *** OK ***
Gathering keystream first bytes for IV=08FFxx ... done
Guessing k[5] ... done
 Guessed k[5]= 1f (with freq. 10) *** OK ***
Gathering keystream first bytes for IV=09FFxx ... done
Guessing k[6] ... done
 Guessed k[6]= 26 (with freq. 13) *** OK ***
Gathering keystream first bytes for IV=0AFFxx ... done
Guessing k[7] ... done
 Guessed k[7]= dc (with freq. 22) *** OK ***
Gathering keystream first bytes for IV=0BFFxx ... done
Guessing k[8] ... done
 Guessed k[8]= 60 (with freq. 14) *** OK ***
Gathering keystream first bytes for IV=0CFFxx ... done
Guessing k[9] ... done
 Guessed k[9]= fa (with freq. 13) *** OK ***
Gathering keystream first bytes for IV=0DFFxx ... done
Guessing k[10] ... done
 Guessed k[10]= 6a (with freq. 12) *** OK ***
Gathering keystream first bytes for IV=0EFFxx ... done
Guessing k[11] ... done
 Guessed k[11]= be (with freq. 11) *** OK ***
Gathering keystream first bytes for IV=0FFFxx ... done
Guessing k[12] ... done
 Guessed k[12]= 56 (with freq. 11) *** OK ***
```

Official guessed values

Our guess of the message is: ***0xc2***
And the guess of the key is: ***0x44b44a85fa1f26dc60fa6abe56***

The output above also shows the frequency of the guessed message and each byte of the key.