Напад на Алгоритмот за Распределба на Клучеви кај RC4

Стефан Теофиловски
201161

1. Што е RC4?

RC4 претставува симетричен шифрувач развиен во 1987 година од страна на криптографот Роналд Ривест. Бил широко употребуван во многу апликации, интернет пребарувачи, безжични мрежи и VPNs. Причина за неговата широка употреба е тоа што алгоритмот е многу брз и едноставен за енкрипција и декрипција на податоци.

Овие функции (енкрипција и декрипција) се идентични, што го прави овој алгоритам симетричен. Се користи таен клуч за генерирање на псевдо-рандом битови кои подоцна се употребуваат за XOR операцијата со оригиналната порака или т.н. „plaintext“ со што се добива шифрираниот текст или „cyphertext“.

2. Каде и за што се користи RC4?

Алгоритмот RC4 користи бајт-по-бајт шифрување, користејќи клуч што се добива во текот на процесот на енкрипција. Ова го прави алгоритмот многу лесен за имплементација и во софтверот и во хардверот.

Апликации кои го користат RC4:

- WEP → The Wired Equivalent Privacy протокол, RC4 се користи на почетокот на развој на WI-FI мрежата.
- SSL/TLS протоколите → RC4 се користел за безбедност при интернет пребарување, емаил и некои интернет апликации.
- Microsoft Office → RC4 се користи за шифрирање на документи до пред појавување на Office 2010.
- Adobe PDF: RC4 → слично како и кај MO се користи за шифрирање на документите.

3. Имплементација на RC4

RC4 алгоритамот работи во 2 фази:

1. Алгоритам за дистрибуција на клучеви или KSA (Key Scheduling Algorithm)
2. Алгоритам за псевдо-рандом генерирање или PRGA (Pseudo-Random Generation Algorithmm)

KSA служи за креирање на т.н. „state-array“ од 256 бајти. Иницијалната вредност на оваа низа се броевите подредени од 0-255 која се разместува (shuffle). Со помош на овој „state-array“, PRGA генерира случајни клучеви. Клучевите може да варираат од 1-256 бајти (најчесто 64 и 128), така што колку поголем клуч толку побезбедно шифрување. Секој креиран клуч влијае на следното разместување на „state-array”. Добиените клучеви служат за правење на операцијата XOR со оргиналниот текст на пораката. На овој начин се добива шифрираниот текст.

```
def KSA():
        j = 0
        N = len(S) # S -> the initial state vector array
        for i in range(0, N):
            j = (j + S[i]+key_list[i]) % N # key_list -> decimal version of the key stream
            S[i], S[j] = S[j], S[i]
            print(i, " ", end ="")
            print(S)
```

Слика 1. Имплементација на KSA во Python

```
def PGRA():

    N = len(S)
    i = j = 0
    global key_stream
    key_stream = []

    for k in range(0, len(plaintext)):
        j = (j + S[i]) % N

        S[i], S[j] = S[j], S[i]
        print(k, " ", end ="")
        print(S)
        t = (S[i]+S[j]) % N
        key_stream.append(S[t])

    print("Key stream : ", key_stream)
    print(" ")
```

Слика 2. Имплементација на PRGA во Python

4. Колку е безбеден RC4?

RC4 бил широко употребуван се додека не биле откриени слабости на алгоритмот во 2003 и 2013 година.

Во 2003 е откриен недостаток во алгоритмот што му овозможува на напаѓачот да го открие клучот дури за помалку од минута. Овој недостаток придонесе за WEP протоколот за WI-FI мрежата да биде што побрзо заменет со WPA (Wi-Fi Protected Access) бидејќи во тој момент единствен протокол бил WEP.

Во 2013 е откриен уште еден недостаток во алгоритмот. Откриен е од група на истражувачи од „Royal Holloway, University of London" и „University of Illinois, Chicago" кои работеле на блоковски шифрувачи. Овој напад е познат како RC4 NOMORE напад. Се базира на тоа што се бара „жртва" која отвара веб страница што користи злонамерен JavaScript код што пробува да се најави на некоја сметка (на пример: Gmail). Ова му овозможува на напаѓачот да собере доволно податоци за да ја изврши потребната криптоанализа. Овој напад бара 16 777 216 пробувања (сесии), но подоцна се откриени многу попрактилни и поелегантни начини што дополнително го ослабуваат RC4.

Треба да се напомене дека веќе овој алгоритам скоро никаде не се користи и не се препорачува за користење во новите апликации.

5. Напад на алгоритмот за дистрибуција на клучеви

5.1 Видови напади на RC4

Постојат повеќе напади на алгоритмот RC4 меѓу кои најзначајни се издвојуваат:

- *FMS (Fluhrer-Mantin-Shamir Attack)* – Напад базиран на статистика и веројатности. Овозможува рекреирање на клучот врз основа на иницијален вектор. Се користи за напади на WEP протоколот.
- *Key Reconstruction Attack* – Напад базиран на откривање некои шеми во алгоритмот за дистрибуција на клучеви. Тежок за имплементација и бара големо познавање на полињата на криптографија и криптоанализа.
- *Invariance Weaknesses Attack* – Алгебарски напад фокусиран на непроменливите својства на алгоритмот за дистрибуција на клучеви. Својствен само за неколку клучеви во RC4 и не е практичен во споредба со другите напади.
- *Time-memory Tradeoff Attack* – Напад на груба-сила (brutte-force) што вклучува пре-дефинирани табели со пресметки за да се намали временската сложеност на нападот.
- *Bias Vulnerabillities Attack* – Статистички напад кој наоѓа не-случајност во алгоритмот за генерирање на случајни стримови со помош на многу оригинал/шифрувани парови пораки.

5.2 FMS напад на RC4

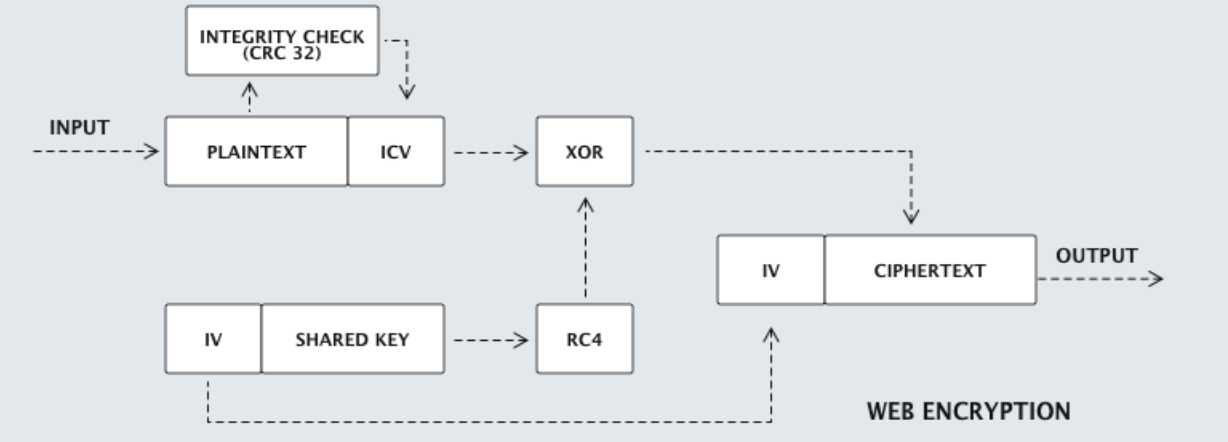
Во овој документ ќе се задржиме на нападот FMS. Овој напад е базиран на статистика и се извршува на WEP протоколот што пак се користи за безбедност на безжични мрежи. Првично е објавен од страна на Scott Fluhrer, Itsik Mantin и Adi Shamir во 2001 година.

За да успее овој напад потребно е да имаме пристап до неколку оригинални пораки и голем број на шифрирани пакети. Нападот претставува значајно достигнување во криптографијата и безбедноста на безжичните мрежи бидејќи поттикнува креирање на нови протоколи (WPA, WPA2) , кои се користат и ден денес.

5.3 Што е WEP протокол?

За да разбереме како работи FMS нападот прво треба да разгледаме како функционира овој протокол и неговата имплементација.

WEP (Wired Equivalent Privacy) – протокол дизајниран за обезбедување безбедност при безжичен транспорт на податоци. Се базира на RC4 алгоритмот што користи таен клуч кој е споделен на сите уреди поврзани на некоја безжична мрежа.



Слика 3. Како работи WEP протоколот

Протоколот користи иницијален вектор и клуч кои му се предаваат заедно на RC4 алгоритмот за да се креира уникатен клуч за шифрирање на податоците. Иницијалниот вектор се состои од 24 (48) бита додека тајниот клуч од 40 (80) што дава вкупно 64 (128) бита. За да може да биде дешифрирана оригиналната порака, иницијалниот вектор се праќа заедно со шифрираниот текст. Ова овозможува да се најдат слабости на протоколот и на RC4 алгоритмот, што дава можност за наоѓање и извршување напади врз истиот.

5.3 Имплементација на FMS

FMS нападот открива слабост во начинот на кој RC4 го користи „key-stream” кој е генериран со помош на клуч и иницијален вектор (IV). Овој „key-stream” потоа се користи за да се направи XOR операција со оригиналниот текст за да се генерира шифрираниот текст. Нападот се базира на статистичка анализа на „key-stream“-от за да може да се открие клучот кој е ист за сите пораки во една безжична мрежа. Така доколку се открие овој клуч можеме да ги прочитаме сите пораки пратени во таа мрежа.

За успешен напад, на напаѓачот му се потребни доволно оригинал-шифрирани текст парови шифрирани со истиот клуч и иницијален вектор. Иницијалниот вектор има само 24 бита, па така после 2^{24} пратени пакети, овој вектор со сигурност ќе се повтори (според „pigeonhole principle“). Ова повторување со помош на статистички анализи го ослабува шифрувањето. За да добиеме доволно шифрирани пакети може да напишеме скрипта со која ќе симулираме шифрирање на познат текст „aa“, IV = [3, 255, 0] и клуч од 40 бита.

```python
# Initial IV form.
iv = [3, 255, 0]
sessionKey = iv + key
PLAIN_SNAP = "aa"

# Clear out what is originally in the file.
WEP_OUTPUT_FILE = open("wep-output.csv", "w", encoding='utf-8').close()
# Append possible IV and keyStreamByte.
WEP_OUTPUT_FILE = open("wep-output.csv", "a", encoding='utf-8')

# A is the number of known key bytes, it starts from 0 to the length of key.
for A in range(len(key)):
    iv[0] = A + 3
    for thirdByte in range(256):
        iv[2] = thirdByte
        sessionKey = iv + key
        print("Session Key: " + str(sessionKey))
        box = list(range(256))
        ksa(box, sessionKey, 256)
        i = 0
        j = 0
        i = (i + 1) % 256
        j = (j + box[i]) % 256
        box[i], box[j] = box[j], box[i]
        keyStreamByte = box[(box[i] + box[j]) % 256]
        cipherByte = (int(PLAIN_SNAP, 16)) ^ keyStreamByte
        WEP_OUTPUT_FILE.write(str(iv[0]) + "," + str(iv[1]) + "," + str(iv[2]) + "," + str(cipherByte) + "\n")
```

Слика 4. Генерирање на шифрирани пакети со познат текст и иницијален вектор

Во кодот прво иницијализираме врендости за иницијалниот вектор, клучот и познатиот текст. Потоа отвараме датотека во која ќе ги запишуваме врендостите добиени од шифрувањето. Во циклусот симулираме шифрување со RC4 алгоритам така што за секој пакет користиме различни вредности за иницијалниот вектор (со цел вредностите да бидат како во вистинско сценарио). Се иницијализира S-box за секој пакет со вредности 0-255. Потоа се вршат функциите KSA, PRGA и XOR со што се добива шифриран бајт кој заедно со вредностите на иницијалниот вектор ги запишуваме во „wep-output.csv“ датотеката. Овие вредности подоцна ќе ни бидат потребни за извршување на нападот.

Следниот чекор е да ги анализираме добиените вредности и да се обидеме да го откриеме клучот со кој тие се добиени. Ова може да го изведеме така што за секој бајт во тајниот клуч иницијализираме листа на веројатности со почетна вредност од 0 (на секој возможен бајт му

даваме веројатност 0 дека ќе го погодиме). Потоа итерираме низ секоја редица од шифрираните пакети запишани во „wep-output.csv” датотеката и ги земаме првите три бајти што всушност ни го претставуваат иницијалниот вектор. Симулираме KSA функција за дистрибуција на клучеви и ги зачувуваме првите оригинални 2 бајта од S-box. За да го откриеме третиот бит го проверуваме следното:

$$z = s\_box[1]$$
$$\text{if } z + s\_box[1] = A+3$$

Доколку овој услов е исполнет, откриваме некоја вркса помеѓу битот кој го погодуваме и S-box. Овој услов статистички е докажан дека ја зголемува веројатноста вредноста на ‘z’ да била генерирана од страна на KSA алгоритмот. Нападот продолжува со проверка дали првите два бајта од S-box останале непроменети. Доколку ова не е исполнето, ја занемаруваме веројатноста да го погодиме бајтот на „key-stream” и продолжуваме на следната редица. Доколку пак условот е исполнет и првите 2 бајта не се променети се извршува операцијата XOR помеѓу тие 2 бајти и двата бајти на познатиот текст (SNAP = ‘aa’). Добиениот резултат од XOR операцијата го користиме за да го откриеме следниот бајт со помош на формулата:

$$keybyte = (keyStreamByte - j - s\_box[i]) \% 256$$

На добиениот „keybyte” му ја зголемуваме веројатноста дека сме го погодиле. Откако ќе изминеме низ сите редици од шифрираните пакети, на бајтот му ја доделуваме вредноста со најголема веројатност. Процесот на откривање на клучот завршува кога ќе го направиме ова за секој бајт од клучот, односно по завршување на надворешниот циклус:

for A in range(key_length)

```
for A in range(keyLength):
    prob = [0] * 256
    for row in rows:
        key[0] = int(row[0])
        key[1] = int(row[1])
        key[2] = int(row[2])

        j = 0
        s_box = list(range(256))

        # Simulate the S-Box after KSA initialization.
        for i in range(A + 3):
            j = (j + s_box[i] + key[i]) % 256
            s_box[i], s_box[j] = s_box[j], s_box[i]
            # Record the original box[0] and box[1] value.
            if i == 1:
                original0 = s_box[0]
                original1 = s_box[1]

        i = A + 3
        z = s_box[1]
        # if resolved condition is possibly met.
        if z + s_box[z] == A + 3:
            # If the value of box[0] and box[1] has changed, discard this possibility.
            if (original0 != s_box[0] or original1 != s_box[1]):
                continue
            keyStreamByte = int(row[3]) ^ int(SNAP, 16)
            keyByte = (keyStreamByte - j - s_box[i]) % 256
            prob[keyByte] += 1
        # Assume that the most hit is the correct password.
        higherPossibility = prob.index(max(prob))
    key.append(higherPossibility)
```

Слика 5. Процесот на напад на RC4 шифрувачот

Крајниот чекор е да го тестираме нападот врз оредена порака. Во овој случај пораката е: „FMS attack on the rc4 Key Scheduling Algorithm.“.

```python
from fms_attack import *
from rc4 import *

s_box = list(range(256))
n_bits = 8
n = 256

key = getKey()
print("OGT: " + str(key))
key = bin(int(key, base=16))[2:].zfill(64)
print("Binary key: " + str(key))
key_list = [key[i:i + n_bits] for i in range(0, len(key), n_bits)]
key_list = convert_to_decimal(key_list)

diff = int(n-len(key_list))
if diff != 0:
    for i in range(0, diff):
        key_list.append(key_list[i])

print("Key list: " + str(key_list))

hex_text = ''
with open('encrypted-text.txt', 'r', encoding='utf-8') as f:
    hex_text = f.read().strip()

encrypted_text = [byte for byte in bytes.fromhex(hex_text)]
print("DECIMAL LIST: " + str(encrypted_text))

# encrypted_text = encryption(s_box, key_list, n, n_bits, plaintext)

original_text = decryprtion(key_list, n, n_bits, encrypted_text)

with open("original-text.txt", "w", encoding='utf-8') as file:
    file.write(original_text)
```

Слика 6. Тест на нападот

Со кодот на слика 6 го земаме клучот добиен од нападот извршен на слика 5. Ја повикуваме функцијата за декрипција од rc4 и како параметри ги праќаме клучот, шифрираниот текст, големината на s-box и колку битови разгледуваме во даден момент. По извршување го добиваме следниот излез:

```
TEXT_ARG: [56, 145, 191, 63, 85, 130, 49, 183, 249, 251, 183, 28, 232, 22, 54, 22, 80, 186, 32, 210, 53, 213, 0, 118, 120, 145, 99, 243, 228, 252, 241, 165, 218, 156
, 62, 222, 3, 148, 56, 21, 149, 175, 157, 39, 123, 75, 185]
KEY_STREAM: [126, 220, 236, 31, 52, 246, 69, 214, 154, 144, 151, 115, 134, 54, 66, 126, 53, 154, 82, 177, 1, 245, 75, 19, 1, 177, 48, 144, 140, 153, 149, 208, 182, 2
45, 80, 185, 35, 213, 84, 114, 250, 221, 244, 83, 19, 38, 151]
Decrypted: 0100011001001101010100110010000001100001011101000111010001100001011000111011011010011000000011011110110111000100000011101000110100001100101001000000111001001
1000110011010000100000010010110110010101011110010010000001010011011000111011010000110010101100100011101010110111000110100101101110011001110010000001000001011011000110011
1011011110111001001101001011101000110100001101100101101
Original sentence: FMS attack on the rc4 Key Scheduling Algorithm.
```

Слика 7. Излез по извршување на тестот

Од сликата можеме да видиме шифрираниот текст (TEXT_ARG) и клучот (KEY_STREAM) претворени во листа од бајти врз кој се врши XOR и се добива „Decrypted” што претставува низа од битови 1 и 0. По претворање на низата битови во зборови, ја добиваме посакуваната реченица: „FMS attack on the rc4 Key Scheduling Algorithm.“. Заклучуваме дека нападот е успешен.

6. Заклучок

Протоколот WEP за шифрување на безжични мрежи е веќе сметан како несигурен и не се препорачува во ниедна апликација. Поради напади како што е Fluhrer-Mantin-Shamir нападот WEP протоколот било приморано да се замени со некои понапредни протоколи како што се WPA2 или WPA3 (Wi-Fi Protected Access). Овие протоколи користат многу посилни методи за шифрување како на пример AES алгоритмот.

7. Референци

- https://en.wikipedia.org/wiki/RC4
- https://www.theregister.com/2013/09/06/nsa_cryptobreaking_bullrun_analysis/
- https://www.geeksforgeeks.org/implementation-of-rc4-algorithm/
- https://www.techtarget.com/searchsecurity/definition/Wired-Equivalent-Privacy#:~:text=Wired%20Equivalent%20Privacy%20(WEP)%20is,expected%20of%20a%20wired%20LAN.
- Fluhrer, Mantin, and Shamir. "Weaknesses in the Key Scheduling Algorithm of RC4." Proceedings of the 4th Annual Workshop on Selected Areas in Cryptography, Springer-Verlag, 2001, pp. 1-24.