Cryptography

Hackers ahead of time

Gaspare Ferraro
ferraro@gaspa.re

March 21, 2019

Visit us!

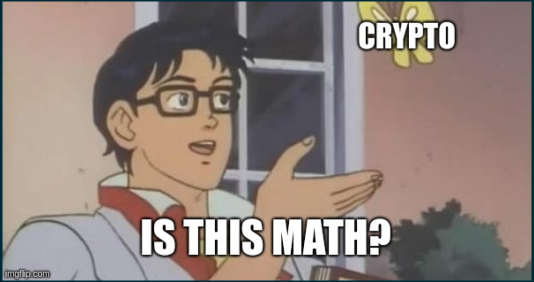
@GaspareG

Part I

Introduzione

Warning!

In questo incontro si fa uso della *matematica*!



Non è sempre stato così però...

La crittografia ieri

(a) Cifrario di Cesare

(b) Scitala

La crittografia oggi

Le necessità, così come le risorse a disposizione, si sono evolute ed oggi possiamo suddividere la crittografia in:

(EN|DE)CRYPTION	**ASYMMETRIC (RSA, ECC, ...)** **SYMMETRIC (DES, AES, ...)**
KEY EXCHANGE	**RSA, DH, ECDH, ...**
AUTHENTICATION	**RSA, DSA, ECDSA, ...**
HASHING	**MD5, SHA-1, SHA-256, ...**

Part II

Crittografia simmetrica

Crittografia simmetrica

I cifrari simmetrici sono quelli dove i messaggi m vengono cifrati e decifrati usando una stessa chiave k, che deve essere nota solo ed esclusivamente alle due parti.

$\mathcal{C}(m, k) = c$ (funzione di cifratura)
$\mathcal{D}(c, k) = m$ (funzione di decifratura)

Ovviamente deve valere che:
$\mathcal{D}(\mathcal{C}(m, k), k) = m$ (il messaggio originale non viene alterato durante lo scambio).

Per esempio nel cifrario di Cesare:
$\mathcal{C}(m, k) =$ ruota in avanti di k ogni singolo carattere.
$\mathcal{D}(c, k) =$ ruota indietro di k ogni singolo carattere.

I principi di Shannon

Come valutiamo se un cifrario è abbastanza robusto? (Dove con robustezza è intesa la sua possibilità di essere attaccato con successo).

Shannon definisce due concetti chiave:

- Confusione: la chiave deve essere ben distribuita nel cifrato (ogni bit del cifrato dovrebbe dipendere da ogni bit della chiave).
- Diffusione: il messaggio deve essere ben distribuito nel cifrato (ogni bit del cifrato dovrebbe dipendere da ogni bit del messaggio).

Nel caso del cifrario di Cesare non abbiamo nessun tipo di diffusione e una bassa confusione (perchè?).

XOR cipher

Consideriamo l'operazione XOR $\oplus$ (or esclusivo), valgono le seguenti proprietà:

- $0 \oplus 0 = 1 \oplus 1 = 0$
- $0 \oplus 1 = 0 \oplus 1 = 1$
- $x \oplus y \oplus y = x$

Definiamo lo XOR cipher come:
$\mathcal{C}(m, k) = m \oplus k$
$\mathcal{D}(c, k) = c \oplus k$

Problema: la chiave k potrebbe essere più corta del messaggio m.
Soluzione: usiamo ripetutamente la chiave: $k^{'} = k \cdot k \cdot \ldots \cdot k$ fino a raggiungere (o superare) la lunghezza di m.

Esempio:
`m = 01100011 01101001 01100001 01101111` (ciao in ASCII).
`k = 01111000 01111000 01111000 01111000` (x in ascii 4 volte)
`c = 00011011 00010001 00011001 00010111` (non stampabile, `GxEZFw==` in b64)

Crittoanalisi statistica

Spesso la vulnerabilità non è nell'algoritmo ma nella sua applicazione...

- La chiave è troppo corta rispetto al messaggio
- La chiave viene ripetuta svariate volte per cifrare diversi messaggi
- I messaggi usano un dizionario mal distribuito.
- Conosciamo il formato del messaggio (es: `flag{...}`)

In particolare parliamo di crittoanalisi statistica quando forziamo il cifrario non dal punto di vista algoritmico ma da quello statistico.
Ad esempio in italiano il 33% delle lettere usate è una tra le vocali `a`, `e`, `i` mentre solo con probabilità dello 0.5% si tratterà di una `z` o una `q`.

Crittoanalisi statistica

Strumento comodo per l'analisi statistica dei messaggi cifrati:

```
root@ddos:~/Desktop/xortool/xortool# xortool binary_xored
The most probable key lengths:
   1:   9.6%
   5:   15.0%
  10:   21.7%
  15:   9.3%
  20:   13.6%
  25:   6.0%
  30:   9.1%
  35:   4.2%
  40:   6.6%
  50:   5.0%
Key-length can be 5*n
Most possible char is needed to guess the key!
```

Conoscendo la parte iniziale si intravedono delle parole in chiaro:

```
This is clas**********l**
Do not share**********}
{FLG:ch3ck_e*****
```

Andando a tentativi si ricostruire la flag finale:

```
This is classified*********
Do not share the s*****
{FLG:ch3ck_em@il}
```

One-time Pad

Il problema dello XOR cipher è che cifrare usando ripetutamente la stessa chiave può far trapelare delle informazioni *statistiche* sul messaggio originale.

Parliamo quindi di Cifrario di Vernam (o one-time pad) quando la lunghezza della chiave è uguale a quella del messaggio, questo cifrario è chiamato *perfetto* perchè vale:

$$P(M = m | C = c) = P(M = m)$$

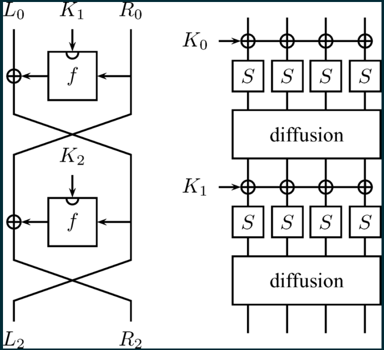
Ovvero la probabilità che M sia un certo messaggio sapendo che il cifrato è C è uguale alla probabilità che M sia un certo messaggio non sapendo il cifrato (tutti i messaggi sono equiprobabili, il messaggio cifrato non ci fornisce informazioni sulla chiave usata).

Bello in teoria, ma:

- La chiave deve essere scambiata usando un metodo sicuro (scambiarle *a mano*).
- La chiave deve essere generata casualmente e non riusata (altrimenti è possibile un many-time pad attack).

DES & AES

Data Encryption Standard (DES) e Advanced Encryption Standard (AES) si basano sul concetto della *S-Box* (scatola della sostituzione).



La confusione e la diffusione vengono implementate effettuando un (elevato) numero di operazioni invertibili.

Part III

Crittografia asimmetrica

Concetti base

La crittografia asimmetrica si basa sulle funzioni *one-way trapdoor* e sulla presenza di una coppia di chiavi (chiamate chiave pubblica e chiave privata).

Una funzione f si dice *trapdoor* se:

- Calcolare $y = f(x)$ è computazionalmente facile.
- Calcolare $x = f^{-1}(y)$ è computazionalmente difficile *(senza nessuna informazione aggiuntiva)*.

Un esempio è il problema della fattorizzazione:

- $m = f(\{p, q\}) = (p * q)$ (calcolo del prodotto tra i primi p e q).
- $\{p, q\} = f^{-1}(m) =$?? (scomposizione in fattori primi di n).

$f(\{49171, 61843\}) = 3040882153$ (facile quanto aprire una calcolatrice).
$f^{-1}(1841488427) =$?? (devo provare tutti i divisori da 2 a $\sqrt{n}$).
Il problema diventa banale se conosco uno dei due divisori:
$1841488427/58049 = 31723$

Aritmetica modulare

Diciamo che due interi a e b sono congrui modulo n, scritto $a \equiv b \pmod{n}$, se $(a \% n) = (b \% n)$ dove % è il resto della divisione intera (modulo).

Alcune proprietà matematiche:

- $a + k \equiv b + k \pmod{n}$, invariante per addizione.
- $k * a \equiv k * b \pmod{n}$, invariante per moltiplicazione.
- $a^k \equiv b^k \pmod{n}$, invariante per potenza.
- $\sqrt{a} \equiv b \pmod{n}$ se $a \equiv b^2 \pmod{n}$, radice quadrata.
- $a^{-1} \equiv b \pmod{n}$ se $ab \equiv 1 \pmod{n}$, inverso moltiplicativo.
- ...

RSA pt.1

L'algoritmo di cifratura asimmetrica più famoso è l'RSA (da Rivest Shamir Adleman) che si basa sul problema della fattorizzazione e sull'aritmetica modulare.

Il funzionamento di base è:

- Si scelgono due numeri primi a caso p e q (in modo *sicuro*).
- Si calcola il prodotto $n = p * q$ (che sarà il nostro modulo).
- Si calcola il toziente $\phi(n) = (p-1) * (q-1)$ (il numero dei coprimi con n).
- Si sceglie a caso un numero e, coprimo e minore di $\phi(n)$.
- Si calcola il numero d come inverso moltiplicativo di e, ovvero $ed \equiv 1 \pmod{\phi(n)}$.

Chiamiamo quindi:
$k_{pub} = (n, e)$ la chiave pubblica che distribuiamo.
$k_{priv} = (n, d)$ la chiave privata che teniamo segreta.

RSA pt.2

Una volta calcolata la chiave pubblica e quella privata possiamo cifrare e decifrare i messaggi in questo modo:

$\mathcal{C}(m, k_{pub}) = m^e \pmod{n}$
$\mathcal{D}(c, k_{priv}) = c^d \pmod{n}$

Si ma perchè funziona?

Dal teorema di Eulero sappiamo che $m^{\phi(n)} \equiv 1 \pmod{n}$.
I valori e e d sono calcolati in modo che $ed \equiv 1 \pmod{\phi(n)}$.

$\mathcal{D}(\mathcal{C}(m, k_{pub}), k_{priv}) = m^{ed} = m^{\phi(n)*t+1} = m^{\phi(n)^t} * m = 1 * m = m$

Un esempio per capire

Scegliamo i sequenti parametri:

- $p = 13$ e $q = 23$
- $n = p * q = 299$
- $\phi(n) = (13 - 1) * (23 - 1) = 264$
- $e = 7$, $\gcd(264, 7) = 1$
- $d = 151$, $7 * 151 \equiv 1 \pmod{264}$

Quindi $k_{pub} = (299, 7)$ e $k_{priv} = (299, 151)$. Vogliamo cifrare $m = 42$.

$$\mathcal{C}(m, k_{pub}) = m^e = 42^7 = 230539333248 = 107 \pmod{299}$$
$$\mathcal{D}(c, k_{priv}) = c^d = 107^{151} = 2743956545...948643 = 42 \pmod{299}$$

(Non tanto) a caso

"Si scelgono due numeri primi *a caso* p e q (in modo *sicuro*)"

- Scegliere p e q di almeno 1024 bit.
- Scegliere p e q non troppo vicini tra loro.
- Non riusare uno dei primi per altri moduli.

Con la potenza di calcolo attuale è possibile fattorizzare semiprimi fino a (circa) 768 bit, i moduli più grandi sono (per ora) resistenti agli attacchi bruteforce.

Se p e q sono vicini allora abbiamo che $n \simeq p^2 \simeq q^2$ e quindi anche $\sqrt{n}$ sarà vicino ai primi. Basterà quindi un attacco bruteforce che cerca i fattori vicino alla radice quadrata.

Se $n_1 = p * q'$ e $n_2 = p * q''$ allora $p = \gcd(n_1, n_2)$.

Part IV

Hashing

Creare confusione

Chiamiamo hash una funzione f *non invertibile* che associa una stringa ad una stringa di dimensione fissata. Con le seguenti proprietà:

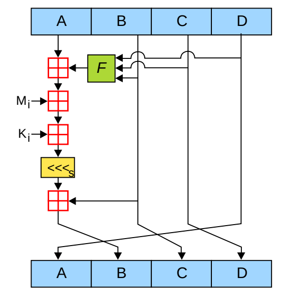
- Resistenza alla preimmagine: dato un hash h è difficile trovare m tale che $f(m) = h$.
- Resistenza alla seconda preimmagine: dato m_1 è difficile trovare m_2 tale che $f(m_1) = f(m_2)$.
- Resistenza alle collisioni: è difficile trovare coppie (m_1, m_2) tali che $f(m_1) = f(m_2)$.

Utilizzi pratici:

- Salvataggio delle password (hashate invece che in chiaro).
- Creazione di hashtable (struttura dati ad accesso diretto).
- Controllo di integrità (*md5sum*, *sha1sum*).
- Firma digitale (sull'hash invece che sul documento).

MD5

Inventato da Rivest nel 1991, divenne la funzione di hash standard fino al 2004-2006. Viene utilizzata ancora adesso per il controllo di integrità.



md5("password") = 5F4DCC3B5AA765D61D8327DEB882CF99
md5("passwore") = A826176C6495C5116189DB91770E20CE

Proof of Work

Spesso, come sistema di protezione per attacchi DOS o per altri motivi, ci viene chiesto di reversare (completamente o parzialmente) un hash.

L'accesso ad una risorsa ci verrà fornito solo dopo aver dimostrato (*Proof of Work*) la risoluzione ad un problema. Nel caso dell'hashing questa operazione viene chiamata hashcash (ed è alla base del mining delle criptovalute).

Esempio (VolgaCtf2017):

`Solve a puzzle: find an x such that 26 last bits of SHA1(x) are set, len(x)==29 and x[:24]=='58a5a7950d2ec81fae5c1c74'`

Trovare le collisioni pt.1

Come trovo una collisione / preimmagine?

- Database online (es. crackstation.net), immediato ma non completo.
- Bruteforce (es. John The Ripper, HashCat), lento ma completo.

Quanti tentativi dovrò fare per reversare un hash?

Paradosso dei compleanni:
se la funzione hash restituisce un output da n bit ($m = 2^n$ possibili hash) si avrà una collisione al 50% di probabilità dopo $2^{(n/2)}$ tentativi ($sqrt(m)$).

Trovare le collisioni pt.2

John The Ripper

```
root@kali: ~
File  Edit  View  Search  Terminal  Help
root@kali:~# john -format=LM /root/Desktop/hash.txt
Using default input encoding: UTF-8
Using default target encoding: CP850
Loaded 4 password hashes with no different salts (LM [DES 128/128 AVX-16])
Remaining 3 password hashes with no different salts
Press 'q' or Ctrl-C to abort, almost any other key for status
0g 0:00:01:21 0.03% 3/3 (ETA: 2017-05-16 00:22) 0g/s 29892Kp/s 29892Kc/s 90718KC
/s 085MSYP..085MSNY
0g 0:00:01:22 0.03% 3/3 (ETA: 2017-05-16 00:14) 0g/s 29957Kp/s 29957Kc/s 90903KC
/s FCUCDB7..FCUCDGT
0g 0:00:01:25 0.03% 3/3 (ETA: 2017-05-16 00:00) 0g/s 30073Kp/s 30073Kc/s 91218KC
/s NEWSAHC..NEWSB9B
0g 0:00:01:26 0.03% 3/3 (ETA: 2017-05-15 23:54) 0g/s 30114Kp/s 30114Kc/s 91330KC
/s VELH10..VELHLS
0g 0:00:01:27 0.04% 3/3 (ETA: 2017-05-15 23:50) 0g/s 30148Kp/s 30148Kc/s 91423KC
/s 4A93GP..4A902K
0g 0:00:01:28 0.04% 3/3 (ETA: 2017-05-15 23:44) 0g/s 30201Kp/s 30201Kc/s 91572KC
/s 08TS4DA..08TSFOA
0g 0:00:01:29 0.04% 3/3 (ETA: 2017-05-15 23:37) 0g/s 30254Kp/s 30254Kc/s 91720KC
/s IKPABAO..IKPABR6
0g 0:00:01:30 0.04% 3/3 (ETA: 2017-05-15 23:33) 0g/s 30283Kp/s 30283Kc/s 91799KC
/s 0JODGB..0JOD5C
0g 0:00:01:33 0.04% 3/3 (ETA: 2017-05-15 23:19) 0g/s 30401Kp/s 30401Kc/s 92124KC
/s H1OGW8W..H1O6CL1
```

Trovare le collisioni pt.3

Hashcat

```
Session..........: hashcat
Status...........: Running
Hash.Type........: iTunes backup >= 10.0
Hash.Target......: $itunes_backup$*10*50782427bcc454da54b51256351cb1de...c32eb7
Time.Started.....: Sun Jan 14 14:15:38 2018 (1 hour, 5 mins)
Time.Estimated...: Sun Jan 14 18:05:56 2018 (2 hours, 44 mins)
Guess.Mask.......: Oliver?a?a [8]
Guess.Queue......: 2/8 (25.00%)
Speed.Dev.#2.....:        1 H/s (1.84ms)
Recovered........: 0/1 (0.00%) Digests, 0/1 (0.00%) Salts
Progress.........: 2560/9025 (28.37%)
Rejected.........: 0/2560 (0.00%)
Restore.Point....: 2560/9025 (28.37%)
Candidates.#2....: Oliver>e -> OliverT9
HWMon.Dev.#2.....: N/A

[s]tatus [p]ause [r]esume [b]ypass [c]heckpoint [q]uit =>

Session..........: hashcat
Status...........: Running
Hash.Type........: iTunes backup >= 10.0
Hash.Target......: $itunes_backup$*10*50782427bcc454da54b51256351cb1de...c32eb7
Time.Started.....: Sun Jan 14 14:15:38 2018 (1 hour, 13 mins)
Time.Estimated...: Sun Jan 14 18:03:02 2018 (2 hours, 34 mins)
Guess.Mask.......: Oliver?a?a [8]
Guess.Queue......: 2/8 (25.00%)
Speed.Dev.#2.....:        1 H/s (1.84ms)
Recovered........: 0/1 (0.00%) Digests, 0/1 (0.00%) Salts
Progress.........: 2816/9025 (31.20%)
Rejected.........: 0/2816 (0.00%)
Restore.Point....: 2816/9025 (31.20%)
Candidates.#2....: OliverPh -> Oliverj6
HWMon.Dev.#2.....: N/A

[s]tatus [p]ause [r]esume [b]ypass [c]heckpoint [q]uit =>
```

Basta un poco di sale...

...e la password non va giù.

Abbiamo parlato di come *non salvare le password in chiaro* bensì salvare il loro hash.

Problema:
Se più persone usano le stessa password (tipico: *qwerty*, *123456*, ..., vedi SAW17/18) avrò hash duplicati nel database.

Soluzione:
Salvare f(password + salt) invece che f(password), dove *salt* è una stringa casuale generata per persona (possibilmente memorizzata in un luogo separato ai dati di login).

Un (pessimo) esempio...


TRENITALIA
GRUPPO FERROVIE DELLO STATO ITALIANE

Gentile GASPARE FERRARO,

come da tua richiesta, ecco i dati relativi al tuo Account

Riepilogo dati registrazione

USER-ID	GaspareG
PASSWORD	XZA[illegible]A

www.trenitalia.com

TRENITALIA
GRUPPO FERROVIE DELLO STATO ITALIANE

Fine