Algoritmi di Crittografia (2023/24)
Notebook 7

In [ ]:
```
from IPython.display import HTML
HTML('<style>{}</style>'.format(open('/home/mauro/.jupyter/custom/custom.css').read()))
```

Latex definitions

In [ ]:
```
# Import delle funzioni/moduli utilizzati nel notebook
from Crypto.Signature import pss
from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA
from Crypto.Util import number
```

Firma digitale

- La firma digitale (digital signature) è il metodo di autenticazione basato su crittografia a chiave pubblica.
- In questo contesto è, almeno in linea di principio, molto semplice da spiegare. Si tratta infatti di una "cifratura" effettuata con la chiave privata, anziché con quella pubblica.
- L'autenticità del mittente di un messaggio M deriva quindi dal fatto che solo con la corrispondente chiave pubblica M può essere "decifrato".
- In realtà poi, a seconda del sistema usato, i "dettagli" possono non essere proprio banali

Firma digitale con RSA

- Iniziamo dall'RSA perché, in questo caso, i dettagli matematici (di base) sono proprio gli stessi visti per la cifratura.
- L'osservazione fondamentale è il ruolo perfettamente simmetrico degli esponenti e e d
- Essi sono uno l'inverso dell'altro in $\mathbf{Z}_n^*$ e nella dimostrazione di correttezza del processo di decifrazione entrano in gioco solo nel prodotto $e \cdot d$.
- Ne consegue che, prescindendo da altre considerazioni, nel caso dell'RSA le virgolette di cui sopra potrebbero proprio essere eliminate: la firma digitale viene infatti apposta precisamente cifrando il messaggio con la chiave segreta anziché con quella pubblica, cioè utilizzando l'identico algoritmo con d al posto di e.

$$F = M^d \bmod n$$

- Naturalmente non si tratta di una cifratura (e infatti non abbiamo usato il simbolo C, bensì F, iniziale di firma) perché il messaggio così trasformato non è reso confidenziale, poiché può essere rimesso in chiaro mediante l'esponente e che è pubblicamente noto.
- D'altra parte, la confidenzialità non è qui l'obiettivo da raggiungere.
- Questo procedimento da solo non basta poi per essere certi dell'autenticità di F.
- Che cosa vuol dire qui, infatti, "rimettere in chiaro" F? visto che, matematicamente, è sempre possibile applicare la trasformazione:

$$F^e \bmod n$$

e dunque qualcosa si ottiene sempre!

- Per essere sicuri della provenienza dobbiamo quindi avere anche una copia del messaggio originale in modo da poter effettuare un confronto.
- La firma digitale di un messaggio M è dunque una coppia (M, F), dove $F = M^d \bmod n$
- La garanzia che la coppia sia da ricondurre al proprietario di e deriva dal fatto che, senza disporre di d, si ritiene computazionalmente intrattabile forgiare una coppia $(\bar{M}, \bar{F})$, dove $\bar{F} = \bar{M}^d \bmod n$.
- Possiamo dunque enucleare la basi del protocollo di autenticazione con firma digitale RSA.

Protocollo base

Firma del messaggio (Alice)

1. Calcola $F = M^d \bmod n$
2. invia la coppia (M, F) a Bob

Verifica della firma (Bob)

1. Si procura la chiave pubblica (e, n) di Alice
2. calcola $M' = F^e \bmod n$
3. accetta se e solo se $M' = M$

- In un'implementazione "reale" di qualsiasi protocollo di firma digitale, la coppia che viene inviata da Alice a Bob non include la firma diretta del messaggio M bensi di un suo fingerprint ottenuto mediante l'applicazione di una qualche funzione hash crittografica (es. SHA256)
- Questo ha il vantaggio di ridurre sensibilmente la dimensione dei numeri ai quali viene applicato il calcolo dell'esponenziale e consente anche la firma di messaggi arbitrariamente lunghi.
- Tutto questo senza perdita di sicurezza a patto che la funzione hash sia resistente alle collisioni
- Deve cioè essere computazionalmente proibitivo trovare, dato M, un altro messaggio $\bar{M}$ tale che $\text{hash}(\bar{M}) = \text{hash}(M)$.
- Se questo è il caso si dice che la funzione hash è second pre-image resistant

Protocollo base con hash

Firma del messaggio (Alice)

1. Calcola $H = \text{hash}(M)$
2. Calcola $F = H^d \bmod n$
3. invia la coppia (M, F) a Bob

Verifica della firma (Bob)

1. Si procura la chiave pubblica (e, n) di Alice
2. calcola $H' = F^e \bmod n$
3. calcola $H = \text{hash}(M)$
4. accetta se e solo se $H' = H$

Blinding attack al protocollo base

- Un attacco potenzialmente serio al protocollo base è quello chiamato blinding attack.
- L'obiettivo è di riuscire ad ottenere la firma del soggetto attaccato (supponiamo sia Alice) su un messaggio M che ella sicuramente non firmerebbe mai.
- Per avere successo l'attacco richiede almeno due condizioni (una ottenibile solo con ingegneria sociale).
- Una è l'equivalente della possibilità di scegliere il testo cifrato in un attacco alla confidenzialità.
- Dobbiamo supporre di poter ottenere che Alice firmi un messaggio $\bar{M}$ il cui contenuto ella condivide (poniamo, una petizione, una richiesta da parte di un gruppo a un qualche organismo pubblico, o simili) ma che (questa è la "condizione tecnica") nasconda in qualche modo il messaggio M pericoloso per Alice.
- Vediamo con ordine.

Blinding attack (Eva)

1. Cerca un numero R tale che $\bar{M} = R^e \cdot M \bmod n$ è un messaggio che, quando firmato da Alice, comporta dei vantaggi per quest'ultima;
2. "Convince" Alice a firmarlo, ottenendo $F = \bar{M}^d \bmod n$;
3. Poiché $F = \bar{M}^d \bmod n = (R^e \cdot M \bmod n)^d \bmod n = R^{ed} \cdot M^d \bmod n = R \cdot M^d \bmod n$, a Eva è sufficente calcolare $F \cdot R^{-1} \bmod n$ per ottenere proprio $M^d \bmod n$

- La parte più difficile è proprio il primo punto, ovvero nascondere il messaggio pericoloso "dentro" un messaggio apparentemente innocuo.
- Ma è tutto l'attacco a sfumare quando si utilizzi il valore hash del messaggio per eseguire la firma.
- In effetti non sono noti attacchi vincenti al protocollo base con hash.
- Per "buona misura", comunque, le possibilità di un attacco di avere successi si riducono ulteriormente con l'utilizzo di "padding" casuale.

Lo standard PSS (Probabilistic Signature Scheme).

- L'idea in se è semplice. Se si applica una funzione hash al messaggio, in generale (di fatto sempre) si ottiene una sequenza di bit che è di molto inferiore alla lunghezza massima dei messaggi consentiti dalla lunghezza degli attuali moduli RSA.
- Lo "spazio" rimanente può essere riempito (padded) con bit casuali e il calcolo RSA vero e proprio applicato all'hash del messaggio opportunamente "padded".
- I dettagli sono un pò più "tediosi" (stile RSA-OAEP) e li omettiamo.

Esempio di uso di RSA-PSS con il package Python Crypto

In [ ]:
```
from Crypto.Signature import pss
from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA
from Crypto.Random import get_random_bytes
```

Generazione ed esportazione delle chiavi

In [ ]:
```
key = RSA.generate(2048)
# Esportazione della chiave pubblica su file
with open('BobKey.pem','wb') as f:
    f.write(key.publickey().exportKey())
# Esportazione della chiave privata su file con password di protezione
privateKey = \
key.exportKey(passphrase='_AIV3ryII5tr0ngIIIPa55w0rd_')
with open('BobSecretKey.pem','wb') as f:
    f.write(privateKey)
```

Preparazione e firma del messaggio con RSA-PSS

In [ ]:
```
message = b'Try this, if you can!'
key = RSA.import_key(open('BobSecretKey.pem').read(),passphrase='_AIV3ryII5tr0ngIIIPa55w0rd_')
h = SHA256.new(message)
keylen = key.size_in_bytes()           # Lunghezza (in byte) della chiave
digestlen = h.digest_size              # Lunghezza dell'output della funzione hash
saltlen = keylen - digestlen -2        # Lunghezza massima del salt. La minima è 0 -> schema deter
ministico
signer = pss.new(key, salt_bytes = saltlen, rand_func = get_random_bytes)
signature = signer.sign(h)
# Invia la coppia (m,signature)
```

Verifica dell'autenticità del messaggio firmato

In [ ]:
```
# riceve la coppia (m,signature)
key = RSA.import_key(open('BobKey.pem').read())   # import della chiave
h = SHA256.new(message)                           # digest del messaggio
keylen = key.size_in_bytes()                      #
digestlen = h.digest_size                         # calcolo della lunghezza del salt (per il paddi
ng)
saltlen = keylen - digestlen -2                   #
verifier = pss.new(key, salt_bytes = saltlen, rand_func = get_random_bytes)
try:
    verifier.verify(h, signature)
    print("La firma è autentica.")
except (ValueError, TypeError):
    print("La firma non è autentica.")
```

Firma digitale con il protocollo di ElGamal

- Come per i messaggi cifrati, anche quelli firmati con il protocollo di Elgamal sono protetti dalla presumibile intrattabilità del problema del calcolo del logaritmo discreto in opportuni gruppi ciclici.
- Noi abbiamo studiato il protocollo di cifratura (e ora vedremo quello per la firma digitale) soprattutto per ragioni storico/didattiche
- Nella documentazione del package Python Crypto possiamo infatti leggere quanto segue.

> Even though ElGamal algorithms are in theory reasonably secure,
> in practice there are no real good reasons to prefer them to RSA instead

- Ci limitiamo quindi all'aspetto matematico.
- Diversamente dall'RSA, il protocollo per la firma non è identico a quello di cifratura.
- Le chiavi pubblica e privata (che, al solito, supporremo di Alice) sono ovviamente le stesse ma le ricordiamo per comodità di riferimento.

1. Chiave segreta. Oltre ai valori p e g resi disponibili come parte della chiave pubblica, un numero $a \in \{2, \ldots, p-2\}$, scelto uniformemente a caso da Alice.
2. Chiave pubblica. È composta da tre numeri (1) il modulo p che definisce il gruppo $\mathbf{Z}_p^*$; (2) una radice primitiva g del gruppo; (3) il valore $A = g^a \bmod p$, dove a è la chiave segreta.

Firma del messaggio (Alice)

1. Sceglie un numero k a caso ma tale che $\mathrm{MCD}(k, p-1) = 1$
2. calcola le due quantità $r = g^k \bmod p$ e $s = k^{-1}\,(M - a \cdot r) \bmod (p-1)$
3. invia a Bob il messaggio firmato $(M, (r, s))$

Verifica della firma (Bob)

1. Si procura la chiave pubblica (p, g, A) di Alice
2. calcola le due quantità $x_1 = A^r \cdot r^s \bmod p$ e $x_2 = g^M \bmod p$
3. accetta la firma come autentica se e solo se $x_1 = x_2$

Correttezza

- Vediamo dapprima a che cosa corrisponde la quantità $r^s \bmod p$

$$\begin{aligned}
r^s \bmod p &= \left(g^k \bmod p\right)^s \bmod p \\
&= \left(g^k\right)^{k^{-1}(M-a\cdot r) \bmod (p-1)} \bmod p \\
&= \left(g^k\right)^{k^{-1}(M-a\cdot r)-(p-1)h} \bmod p \\
&= \left(g^k\right)^{k^{-1}(M-a\cdot r)} \left(g^k\right)^{-(p-1)h} \bmod p \\
&= g^{M-a\cdot r} \left(g^{-kh}\right)^{p-1} \bmod p \\
&= g^{M-a\cdot r} \left(\left(g^{-kh}\right)^{p-1} \bmod p\right) \bmod p \\
&= g^M g^{-a\cdot r} \bmod p \\
&= g^M \left(g^{-a} \bmod p\right)^r \bmod p \\
&= g^M A^{-r} \bmod p
\end{aligned}$$

  dove h (nel terzo passaggio) è il quoziente della divisione di $k^{-1}\,(M - a \cdot r)$ per $p - 1$.

- A questo punto è evidente che, in mancanza di tentativi "forgiatura" (o errori non malevoli):

$$x_1 = A^r \cdot r^s \bmod p = A^r \cdot (r^s \bmod p) \bmod p = A^r \cdot \left(g^M A^{-r} \bmod p\right) \bmod p = g^M \bmod p$$

Sicurezza

- Qui consideriamo solo alcuni fra i più importanti accorgimenti necessari per la sicurezza.

Uso singolo di k

- Il valore segreto k deve essere utilizzato una sola volta.
- Si noti, preliminarmente che, se la firma è autentica, $x_1 = x_2$ e ovviamente vale anche

  $\log_g x_1 \bmod p = \log_g x_2 \bmod p = M \bmod p$. Da questo, con semplici passaggi

$$\begin{aligned}\log_g x_1 \bmod p &= \log_g A^r \cdot r^s \bmod p \\ &= (r \cdot \log_g A + s \cdot \log_g r) \bmod p \\ &= (r \cdot a + s \cdot k) \bmod p \\ &= M \bmod p\end{aligned}$$

- L'attaccante può dunque utilizzare due messaggi, M_1 e M_2, firmati con lo stesso valore k, e impostare il seguente

  sistema di due equazioni nelle due incognite a e k:

$$\begin{cases} a \cdot r_1 + k \cdot s_1 &= M_1 \pmod p \\ a \cdot r_2 + k \cdot s_2 &= M_2 \pmod p \end{cases}$$

- In questo modo, l'unica soluzione fornisce, oltre a k, anche il valore della chiave segreta a
- Se invece k è sempre diverso, qualsiasi sia il numero ℓ di equazioni considerate, il numero di incognite è sempre $\ell + 1$ e dunque il sistema è indeterminato

Valore hash del messaggio

- Il protocollo deve essere usato con hash del messaggio
- In caso contrario diviene possibile portare un message forgery attack, cioè produrre un messaggio correttamente firmato senza conoscere la chiave privata
- Si possono infatti determinare in anticipo i valori r ed s e costruire il messaggio che ha r ed s come firma
- L'attaccante sceglie inizialmente due numeri x e y, con $\text{MCD}(y, p-1) = 1$, dopodiché pone

$$r = g^x \cdot A^y \bmod p = g^x \cdot (g^a \bmod p)^y \bmod p = g^{x+a \cdot y} \bmod p$$

  e

$$s = -r \cdot y^{-1} \bmod (p-1)$$

  da cui può derivare il valore del messaggio M di cui (r, s) è la firma
- Eva sa infatti che Bob eseguirà il seguente controllo:

$$A^r \cdot r^s \bmod p \stackrel{?}{=} g^M \bmod p$$

  e dunque pone $M = x \cdot s \pmod{(p-1)}$.
- In questo modo risulta infatti:

$$\begin{aligned}A^r \cdot r^s \bmod p &= (g^a \bmod p)^r \cdot \left(g^{x+a \cdot y} \bmod p\right)^s \bmod p \\ &= g^{a \cdot r} \cdot g^{(x+a \cdot y) \cdot s} \bmod p \\ &= g^{a \cdot r + x \cdot s + a \cdot y \cdot s} \bmod p \\ &= g^{x \cdot s} \bmod p \\ &= g^{M + h(p-1)} \bmod p \\ &= g^M \bmod p\end{aligned}$$

- La correttezza del terz'ultimo passaggio è una conseguenza della scelta di s
- Infatti, poiché $s = -r \cdot y^{-1} \bmod (p-1)$ risulta $y \cdot s = -r + h' \cdot (p-1)$, con h' opportuno valore intero h', e dunque

$$g^{a \cdot r + a \cdot y \cdot s} \bmod p = g^{a \cdot r - a \cdot r + a \cdot h' \cdot (p-1)} \bmod p = 1$$

- Per "parare" questo attacco si usa firmare non il messaggio ma il valore hash di una quantità che include il messaggio
- Più precisamente, dopo avere scelto k e calcolato r, Alice firma $H(M||r)$, dove || indica la concatenazione
- Con questa scelta, Eva non può ovviamente porre $M = x \cdot s \bmod (p-1)$ come in precedenza, perché non è questo che Bob usa nella verifica.
- Eva dovrebbe trovare un messaggio M tale che $H(M||r) = x \cdot s \bmod (p-1)$, cosa che richiede che H non sia first pre-image resistant

Controllo sul valore di r

- È necessario che il protocollo di verifica della firma preveda il controllo che $r \in \{1, 2, \ldots, p-1\}$
- In caso contrario, diviene agevole per Eva produrre firme apparentemente autentiche su qualsiasi messaggio, nel momento in cui disponga della firma autentica (r, s) di un solo messaggio M
- Infatti, scelto il messaggio M' da far passare come autentico, Eva calcola preliminarmente il valore

$$x = M' \cdot M^{-1} \bmod (p-1)$$

  dopodiché sceglie i due valori r' ed s', che costituiranno la firma di M', nel modo indicato di seguito
- Nel caso di s' la scelta è semplice

$$s' \equiv s \cdot x \bmod (p-1)$$

- r' deve invece essere scelto in modo che soddisfi simultaneamente le due seguenti equazioni:

$$r' \equiv r \cdot x \bmod (p-1) \qquad \text{e} \qquad r' \bmod p = r$$

- Una soluzione con queste proprietà si può ottenere semplicemente ponendo $\alpha = r \cdot (x-1)$ e poi $r' = r + \alpha p$
- La verifica che $r' \bmod p = r$ è immediata
- Per la prima proprietà abbiamo

$$\begin{aligned} r' - rx &= r + \alpha p - rx \\ &= r + r(x-1)p - rx \\ &= -r(x-1) + r(x-1)p \\ &= r(x-1)(p-1) \end{aligned}$$

  e dunque $r' - rx$ è un multiplo di $p-1$, come richiesto
- La dimostrazione che M', con la firma (r', s') passa la verifica è lasciata come semplice esercizio di "manovra" dei moduli

- Per parare l'attacco è necessario controllare che la componente r della firma sia un numero minore del modulo p
- In una firma legale, infatti, r è definito come $g^k \bmod p$
- Al contrario, $r' = r + \alpha p$ è quasi sempre un numero maggiore di p e dunque il controllo di cui sopra è quasi sempre efficace
- Ci sono tuttavia un paio di casi speciali che vanno controllati
- Teniamo tuttavia presente che dal lato dell'attaccante è necessario che M abbia inverso modulo $p-1$ e dunque, in particolare, che sia strettamente minore di $p-1$
- Veniamo ai casi speciali:

1. Se $x = 1$ risulta $\alpha = 0$ e $r' = r$; tuttavia $x = 1$ richiede $M' = M \bmod (p-1)$ e dunque, poiché $M < p-1$, risulta propriamente $M' = M$ e l'attacco svanisce
2. Se $x = 0$ risulta $\alpha = -r$ e $r' = r - rp$. $x = 0$ implica che il messaggio "falso" sia un multiplo di $p-1$ e questo potrebbe effettivamente costituire una minaccia. Ora, è vero che un valore negativo di r fa fallire l'esponenziale modulare (presente nel calcolo di $A^r r^s \bmod p$); tuttavia Eva, che conosce molto bene l'aritmetica modulare, in questo caso non presenta r' bensì $r' \bmod p-1 = 0$. Questa modifica non può essere fatta sempre perché cambia anche l'altro fattore presente nel calcolo di $A^r r^s \bmod p$, e cioè $r^s \bmod p$. L'accorgimento di Eva funziona solo in questo caso, in quanto anche $s = 0$.

In [ ]:
```
from AClibrary.crypto import EGKey
from Crypto.Math.Numbers import Integer
```

In [ ]:
```
key = EGKey(256)
publickey = key.publickey()
```

In [ ]:
```
text = b'un messaggio'
```

In [ ]:
```
M,(r,s) = key.sign(text)
```

In [ ]:
```
M,(r,s)
```

In [ ]:
```
publickey.verify(M,r,s)
```

In [ ]:
```
ord('n')
```

In [ ]:
```
publickey.verify(Integer.from_bytes(b'uo messaggio'),r,s)
```

In [ ]:
```
def r_attack(key,Mprime,M,r,s):
    '''Attacco basato sulla mancata verifica del valore di r'''
    if Integer.gcd(M,key.p-1)>1:
        print("Impossibile portare l'attacco")
        return
    M1=Integer.inverse(M,key.p-1)  # inversione di M modulo p-1
    x = (Mprime*M1)%(key.p-1)      # calcolo del valore x
    if x==0:
        sp = Integer(0)
        rp = (r-r*key.p)%(key.p-1)
    else:
        sp=(s*x)%(key.p-1)
        alpha=r*(x-1)
        rp = r+alpha*key.p
    return rp,sp
```

In [ ]:
```
fakemsg = b'I owe Eve 10000 bucks'
Mprime = Integer.from_bytes(fakemsg)
```

In [ ]:
```
rp,sp = r_attack(publickey,Mprime,M,r,s)
```

In [ ]:
```
publickey.verify(Mprime,rp,sp)
```

- Osserviamo che cosa accade se $M' \bmod p - 1 = 0$

In [ ]:
```
Mprime = Integer(34)*(publickey.p-1)
```

In [ ]:
```
rp,sp = r_attack(publickey,Mprime,M,r,s)
```

In [ ]:
```
rp,sp
```

- In questo caso, come previsto

In [ ]:
```
publickey.verify(Mprime,rp,sp)
```

Firma digitale con DSA (Digital Signature Algorithm)

- L'algoritmo DSA è un metodo definito con precisione da NIST (National Institute of Standards and Technology)
- È stato adottato come standard nel 1994 (FIPS PUB 186)
- Versione attuale rilasciata nel luglio 2013 (FIPS PUB 186-4)
- Come al solito vediamo le tre componenti di generazione chiavi, firma e verifica
- A differenza di altri protocolli (di genesi accademica e dunque inizialmente presentati nella loro essenza matematico/algoritmica) DSA prevedeva "da subito" l'uso di una funzione hash sul messaggio da firmare
- Nello specifico, la funzione hash era $\texttt{SHA} - 1$

Protocollo originale

Generazione delle chiavi (Alice)

1. Genera un numero primo q di 160 bit
2. Genera un numero primo p di 1024 bit tale che q sia un divisore primo di $p-1$
3. Determina un elemento $g \in \mathbf{Z}_p^*$ di ordine q (in altri termini, un generatore del sottogruppo di $\mathbf{Z}_p^*$ di $q \approx 2^{160}$ elementi)
4. Sceglie a caso un numero $a \in \mathbf{Z}_q$
5. Calcola $A = g^a \bmod p$
6. Pubblica (p, q, g, A) e tiene riservato il numero a

Firma del messaggio (Alice)

1. Dato il messaggio M, calcola $m = \texttt{SHA}-1(M)$ (160 bit)
2. Sceglie k uniformemente a caso in $\mathbf{Z}_q^*$
3. Calcola $r = (g^k \bmod p) \bmod q$ e $s = k^{-1}(m + a \cdot r) \bmod q$
4. Se anche uno solo dei due valori (r o s) è 0, sceglie un diverso valore di k (passo 2)
5. Altrimenti invia il messaggio M insieme alla firma (r, s)

Verifica della firma (Bob)

1. Si procura la chiave pubblica (p, q, g, A) di Alice
2. Controlla che risulti $0 < r, s < q$
3. Calcola $x = \texttt{SHA}-1(M) \cdot (s^{-1} \bmod q)$
4. Calcola $y = r \cdot (s^{-1} \bmod q)$
5. Esegue il controllo $(g^x A^y \bmod p) \bmod q \stackrel{?}{=} r$ e accetta la firma come autentica in caso di uguaglianza

Un esempio illustrativo con numeri piccoli

In [ ]:
```
from ACLIB.utils import isPrime, modular_inverse
from random import randint
```

In [ ]:
```
##########################################################
##### Generazione delle chiavi privata e pubblica #####
##########################################################
while (q := int(input("Inserisci il valore di q: "))) and not isPrime(q):
    pass
# Determinazione di un primo p t.c. q divide p-1
r = 2*q+1  # Arbitrario valore di partenza
while (p:=2*q*r+1) and not (isPrime(r) and isPrime(p)):
    r += 2
# Determinazione di un generatore del sottogruppo di q elementi
h = 2
r2 = (p-1)//q
while (g:=h**r2 % p) and g == 1:
    h+=1
# Scelta (casuale) della chiave privata
a = randint(1,q-1)
# Calcolo dell'ultimo valore della chiave privata
A = g**a % p
print('Public key')
print(f'q={q}\tp={p}\tg={g}\tA={A}')
```

```
Inserisci il valore di q: 17
Public key
q=17    p=1259  g=316   A=83
```

In [ ]:
```
g**q%p
```

```
1
```

In [ ]:
```
h
```

```
2
```

In [ ]:
```
##########################################################
## Firma di un messaggio (visto come già "hash-ato") ##
##########################################################
while (m:=int(input(f'Inserisci il valore del messaggio (m<{q}): '))) and (m>=q or m<0):
    pass
while True:
    k = randint(1,q-1)
    r = (g**k % p) % q
    s = (modular_inverse(k,q)*(m+r*a))%q
    #if r%q != 0 and s%q != 0:
    if r%q == 0 or s%q == 0:
        break
print('Digital signature')
print(f'msg={m}\tr={r}\ts={s}')
```

```
Inserisci il valore del messaggio (m<17): 3
Digital signature
msg=3   r=5     s=0
```

In [ ]:
```
#########################################################
####### Verifica dell'autenticità del messaggio #######
#########################################################
m_rcvd,r_rcvd,s_rcvd = 16,r,s
x = m_rcvd*modular_inverse(s_rcvd,q)
y = r_rcvd*modular_inverse(s_rcvd,q)
rprime = ((g**x)*(A**y)% p) % q
if r_rcvd == rprime:
    print("Accept")
else:
    print("Reject")
```

```
Reject
```

Due semplici esercizi

1. Dimostrare che, se k (che pure è ephemeral) è stata compromessa, anche la chiave privata di Alice è compromessa
2. Dimostrare che, inviando un messaggio firmato con r o s uguali a 0, la chiave privata viene compromessa

In [ ]:
```
# Attacco quando k è noto e entrambi i valori r ed s sono non nulli
aprime = ((k*s-m)*modular_inverse(r,q))%q
aprime == a
```

In [ ]:
```
# Attacco quando r = 0 ma s è non nullo
kprime = (modular_inverse(s,q)*m)%q
kprime == k
```

```
True
```

In [ ]:
```
# Attacco quando r è non nullo e s=0
aprime = (q-m)*modular_inverse(r,q)%q
aprime==a
```

```
True
```

- Osservazione: il caso $(s \equiv 0) \bmod q$ implica comunque che la verifica non è possibile nel modo "standard". Se r è non nullo, la verifica può essere fatta da Alice recuperando prima a (nel modo appena visto) e poi verificando che r soddisfa l'equazione

$$m + r \cdot a = 0 \mod q$$

In [ ]:
```
(m+r*a)%q
```

- Se poi anche r è nullo, allora questo implica che $m = 0 \bmod q$.

Sicurezza

- La sicurezza del protocollo DSA dipende dalla difficoltà di calcolo del logaritmo discreto su un sottogruppo di q elementi
- Il protocollo orginale, in qui q viene scelto nell'intervallo $(2^{159}, 2^{160})$ fornisce quindi $\log(\sqrt{2^{160}}) = 80$ bit di sicurezza, sulla base delle attuali conoscenze riguardo la difficoltà di calcolo del logaritmo discreto

Specifiche del protocollo attuale

- La lunghezza del modulo p può essere uno dei seguenti valori: 1024, 2048, o 3072 bit
- La lunghezza in bit di q è uno dei valori 160, 224 o 256
- Come funzione hash può essere utilizzata qualsiasi funzione specificata nelle pubblicazioni FIPS 180

Un esempio con il package Crypto

In [ ]:
```
from Crypto.PublicKey import DSA
from Crypto.Hash import SHA256,SHA224
from Crypto.Signature import DSS
from math import log2,ceil
```

In [ ]:
```
key = DSA.generate(2048)
ceil(log2(key.q))
```

In [ ]:
```
key.__dict__  # Il valore di x è quello che nella descrizione del protocollo abbiamo chiamato a
```

In [ ]:
```
message = b"Hello"
m = SHA224.new(message)
```

In [ ]:
```
signer = DSS.new(key,mode='fips-186-3') # Il modo prevede la scelta randomizzata di k con RNG
signature = signer.sign(m)
```

- Alice invia la coppia message,signature
- Bob si procura la chiave pubblica di Alice

In [ ]:
```
pubkey = key.publickey()
```

In [ ]:
```
pubkey.__dict__
```

- A questo punto può ricalcolare l'hash del messaggio ricevuto ed effettuare la verifica

In [ ]:
```
m = SHA224.new(message)
#m =  SHA256.new(b'Helln')
```

In [ ]:
```
verifier = DSS.new(pubkey, 'fips-186-3')
```

In [ ]:
```
try:
    verifier.verify(m, signature)
    print("The message is authentic.")
except ValueError:
    print("The message is not authentic.")
```

Autenticità delle chiavi pubbliche

- Quando Bob "si procura" (abbiamo usato più volte questa locuzione ...) la chiave pubblica di Alice, vuoi per cifrare un messaggio, vuoi per verificare l'autenticità di un documento, egli deve essere certo che tale chiave pubblica sia effettivamente la chiave di Alice
- Ci sono almeno due approcci per ottenere questo risultato, esemplificati dall'uso che ne fanno le suite di protocolli TLS/OpenSSL e GnuPG/OpenPGP
- Il primo approccio necessita di un'infrastruttura, costituita dalle cosiddette autorità di certificazione (CA)

Approccio TLS

- Tale soluzione è bene illustrata nel caso di interazione fra browser (client) e web server
- L'interazione tipicamente parte per iniziativa del client che contatta il server, ad esempio di un negozio di vendita on-line
- Questo primo contatto è in chiaro e il rischio che in questa fase si verifichino i preliminari di un attacco (ad esempio mediante site-spoofing) è alquanto elevato
- Il server deve rispondere con un messaggio che include un certificato di autenticità
- Si tratta essenzialmente della chiave pubblica del server firmata da una qualche autorità riconosciuta in grado di fungere da garante per il server
- L'interazione prosegue poi in modi differenti a seconda che il client già possieda, o meno, un certificato del server precedentemente verificato e in corso di validita

1. Se possiede tale certificato, il client possiede anche la chiave pubblica verificata e l'interazione può procedere secondo il protocollo concordato
2. Se non possiede tale certificato ma possiede la chiave pubblica verificata dell'autorità che garantisce per il server, il client acquisisce come autentico il certificato memorizzandolo nel cosiddetto keyring per usi futuri. Anche in questo caso, l'interazione può procedere secondo il protocollo concordato.
3. Se non possiede neppure la chiave pubblica dell'autorità, il client deve prima autenticare quest'ultima mediante un'autorità di livello superiore che garantisce per essa.
   - Il problema assume cioè connotati ricorsivi.
   - Al riguardo va però notato che il certificato presentato dal server deve includere specifica della "catena" di certificazioni, che dal server arriva ad un'autorità top-level che garantisce per le subordinate.
   - E dunque fondamentale che le chiavi pubbliche delle autorità di massimo livello siano memorizzate nel keyring quando il sistema operativo viene installato (da fonte attendibile)

In [ ]:
```
!openssl s_client -connect www.google.it:443
```

GnuPG (II)

- All'approccio che caratterizza TLS (brevemente schematizzato sopra) si contrappone quello utilizzato da GnuPG/OpenPGP, la cui organizzazione è essenzialmente orizzontale (potremmo dire di tipo peer-to-peer)
- Si tratta di una soluzione molto più leggera che non potrebbe essere utilizzata per gli scopi in cui si applica TLS (commercio elettronico, home-banking, ...)
- In ambito GnuPG, l'associazione fra persone (e i loro indirizzi email) e chiavi pubbliche avviene tramite il cosiddetto web of trust, ovvero una rete di fiducia
- Vediamo un esempio concreto.
  1. Un docente (a caso...) firma i messaggi di posta elettronica destinati, fra gli altri, agli studenti dei suoi corsi, di modo che questi ultimi siano certi della loro provenienza
  2. Se uno studente vuole essere sicuro della fonte può accedere ad un server di chiavi PGP, ad esempio il MIT PGP Public Key Server e recuperare la chiave pubblica del docente in questione.
  3. Come può però egli/ella essere sicuro che il sito non sia stato "spoof-ato" o che la chiave sia stata modificata nel transito?
  4. La chiave ha però anche un fingerprint. Per altra via, ad esempio durante una lezione online, il docente comunica tale fingerprint e così gli studenti, ragionevolmente certi di stare interagendo con il docente, possono verificare che la chiave sia autentica.
  5. A quel punto la importano nel proprio keyring attribuendo ad essa un valore di confidenza elevato.
  6. Uno studente, non presente durante questa fase pubblica, nota che molti suoi compagni hanno attribuito alta confidenza a quella particolare chiave e decide che questo è sufficiente per "credere" che la chiave appartenga effettivamente a quel docente
  7. A questo punto anch'egli la importa, eventualmente attribuendole un grado di fiducia un poco inferiore a quello dei propri compagni

- Quello delineato sopra è solo un esempio di un possibile scenario in cui il web-of-trust si allarga
- OpenPGP include regole precise per attribuire grado di fiducia più o meno elevato ad una chiave sulla base del grado di fiducia a questa attribuito da altri

Importare una chiave nel keyring

- Per cercare una chiave

In [ ]:
```
!gpg --keyserver pgp.mit.edu --search leoncini@unimore.it
```

- Per importare una chiave

```
gpg --keyserver pgp.mit.edu --recv-keys 96C0819CD95DCEE8
```

- Per attribuire un grado di fiducia alla chiave importata

```
> gpg --edit-key leoncini@unimore.it
gpg (GnuPG) 2.2.27; Copyright (C) 2021 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Secret key is available.

sec  rsa4096/96C0819CD95DCEE8
     created: 2013-02-01  expires: never       usage: SC
     trust: ultimate      validity: ultimate
ssb  rsa4096/3626DF78751E14AC
     created: 2013-02-01  expires: never       usage: E
[ultimate] (1). Mauro Leoncini <leoncini@unimore.it>
[ultimate] (2)  Mauro Leoncini <mauro.leoncini@unimore.it>
[ revoked] (3)  mauro.leoncini (A strong key, 4096 bit long) <leoncini@unimore.it>

gpg> trust
sec  rsa4096/96C0819CD95DCEE8
     created: 2013-02-01  expires: never       usage: SC
     trust: ultimate      validity: ultimate
ssb  rsa4096/3626DF78751E14AC
     created: 2013-02-01  expires: never       usage: E
[ultimate] (1). Mauro Leoncini <leoncini@unimore.it>
[ultimate] (2)  Mauro Leoncini <mauro.leoncini@unimore.it>
[ revoked] (3)  mauro.leoncini (A strong key, 4096 bit long) <leoncini@unimore.it>

Please decide how far you trust this user to correctly verify other users' keys
(by looking at passports, checking fingerprints from different sources, etc.)

  1 = I don't know or won't say
  2 = I do NOT trust
  3 = I trust marginally
  4 = I trust fully
  5 = I trust ultimately
  m = back to the main menu

Your decision?
```

- Se abbiamo un'elevata fiducia che una data chiave appartenga alla persona individuata dall'indirizzo email, possiamo rendere noto (e utilizzabile) questo fatto firmando tale chiave (ovviamente con la nostra chiave privata)

```
gpg --sign-key D95DCEE8
```

- Dopodiché possiamo inviare la chiave firmata al server (o ai server), il quale fonderà la nostra firma con le altre disponibili per leoncini@unimore.it

```
gpg --keyserver pgp.mit.edu --send-keys D95DCEE8
```

In [ ]: