2. STRUMENTI CRITTOGRAFICI IN OpenSSL

OpenSSL è un progetto open source che fornisce implementazioni per:

- Funzioni Crittografiche (o Primitive);
- Protocolli quali Secure Sockets Layer (SSL) e Transport Layer Security (TLS).

Comprende comandi eseguibili per funzioni e protocolli crittografici e librerie contenenti API per sviluppare applicazioni crittografiche. Viene supportata anche crittografia basata su Curve Ellittiche (Elliptic Curve Cryptography).

OpenSSL si occupa di realizzare in concreto tutti i concetti che sono nel mondo della crittografia, quindi creare e gestire chiavi private, pubbliche e parametri. Permette di effettuare operazioni crittografiche a chiave pubblica etc..

Gli esempi che verranno mostrati sono stati sviluppati utilizzando Linux Ubuntu 18.04.4 LTS, OpenSSL Versione 1.1.1

2.1 CIFRATURA SIMMETRICA IN OpenSSL

OpenSSL fornisce numerosi cifrari simmetrici. Molti cifrari supportano varie modalità operative (comportamento di un cifrario simmetrico quando la dimensione dei dati da cifrare è maggiore di un singolo blocco): ECB (insicura), CBC, CFB, OFB, CTR. La modalità operativa di default è CBC, se nessun'altra è esplicitamente specificata.

Per conoscere i cifrari supportati da OpenSSL si digita il seguente comando:

openssl list --cipher-commands (l'opzione --cipher-commands permette di mostrare i cifrari supportati dal sistema che si usa)

```
aes-128-cbc        aes-128-ecb        aes-192-cbc        aes-192-ecb
aes-256-cbc        aes-256-ecb        aria-128-cbc       aria-128-cfb
aria-128-cfb1      aria-128-cfb8      aria-128-ctr       aria-128-ecb
aria-128-ofb       aria-192-cbc       aria-192-cfb       aria-192-cfb1
aria-192-cfb8      aria-192-ctr       aria-192-ecb       aria-192-ofb
aria-256-cbc       aria-256-cfb       aria-256-cfb1      aria-256-cfb8
aria-256-ctr       aria-256-ecb       aria-256-ofb       base64
bf                 bf-cbc             bf-cfb             bf-ecb
bf-ofb             camellia-128-cbc   camellia-128-ecb   camellia-192-cbc
camellia-192-ecb   camellia-256-cbc   camellia-256-ecb   cast
cast-cbc           cast5-cbc          cast5-cfb          cast5-ecb
cast5-ofb          des                des-cbc            des-cfb
des-ecb            des-ede            des-ede-cbc        des-ede-cfb
des-ede-ofb        des-ede3           des-ede3-cbc       des-ede3-cfb
des-ede3-ofb       des-ofb            des3               desx
rc2                rc2-40-cbc         rc2-64-cbc         rc2-cbc
rc2-cfb            rc2-ecb            rc2-ofb            rc4
rc4-40             seed               seed-cbc           seed-cfb
seed-ecb           seed-ofb           sm4-cbc            sm4-cfb
sm4-ctr            sm4-ecb            sm4-ofb
```

Ciascun elemento della lista che sono stati riportati prende il nome di **ciphername**. Un ciphername è costituito dalla concatenazione di 3 elementi che sono separati da un trattino e che sono:

- **Nome del cifrario**
- **Lunghezza della chiave**
- **Modalità operativa**

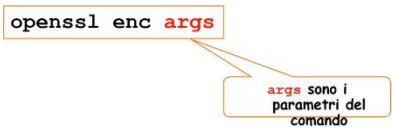
Ad esempio: aes-256-cbc

Nota bene che l'unica parte obbligatoria di un ciphername è il nome del cifrario, dopodiché sarà OpenSSL assegnerà di default i restanti parametri.

Se non viene specificato diversamente, i dati sono letti dallo standard di input e scritti sullo standard di output. Solo un singolo file alla volta può essere cifrato o decifrato e ciascun cifrario richiede una chiave per effettuare la cifratura o la decifratura (tale chiave deve essere nota solo al mittente ed ai destinatari)

Per utilizzare la cifratura simmetrica in OpenSSL, si fa tramite il comando *enc* (encrypt/encode) che permette di accedere ai cifrari simmetrici fornitida OpenSSL. Le opzioni principali del comando enc che permettono di personalizzare il comportamento del comando sono:

- ***-ciphername*** : tipo di cifrario, lunghezza della chiave e modalità operativa.
- ***-in filename*** : file di input.
- ***-out filename***: file di output.
- ***-e or -d*** : specifica se si tratta di cifratura o decifratura.
- ***-K key*** : chiave usata dal cifrario per cifrare o decifrare. Se non viene specificata, OpenSSL deriverà questa chiave da una password.
- ***-pass arg*** : sorgente della password: I valori possibili per ***arg*** sono ***pass:password*** o ***pass:filename***, dove ***password*** è la password e ***filename*** è il file contenente la password. Se non si usa questo parametro viene mostrato un prompt per inserire la password.
- ***-base64*** : applica la codifica Base64 prima o dopo le operazioni crittografiche.



Per ottenere la lista completa delle opzioni enc, è possibile utilizzare *man enc*.

Concentriamoci sulla codifica in Base64, che è molto usato perché permette di memorizzare flussi di dati binari mediante caratteri stampabili (chiavi crittografiche, certificati, allegati e-mail, etc…). Base64 è un sistema di sistema di decodifica/codifica per dati binari, che utilizza 64 simboli: A-Z, a-z, 0-9, +, /. La codifica in Base64 avviene nel seguente modo:

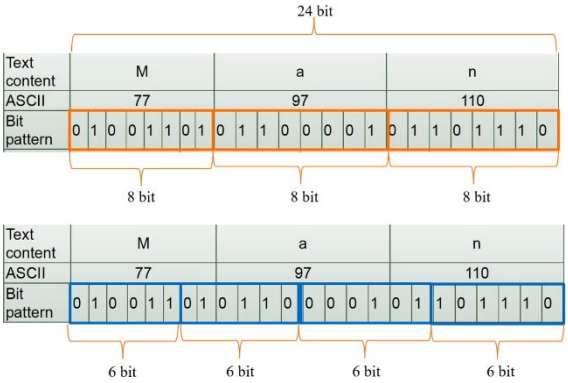
La figura al lato mostra i 64 caratteri stampabili. A ciascun carattere stampabile è associato un valore intero univoco compreso tra 0 e 63.

Per capire questo sistema di codifica come opera, vediamo un esempio:

valore	codifica	valore	codifica	valore	codifica	valore	codifica
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/

Tabella di conversione Base64

Supponiamo di voler codificare in Base64 il flusso di bit relativo alla parola “Man”. Assumendo che le lettere siano codificate in ASCII, ciascun simbolo può essere rappresentato da 8 bit. Ad esempio la lettera M può essere codificata con i bit 01001101 e così via per le altre due lettere.

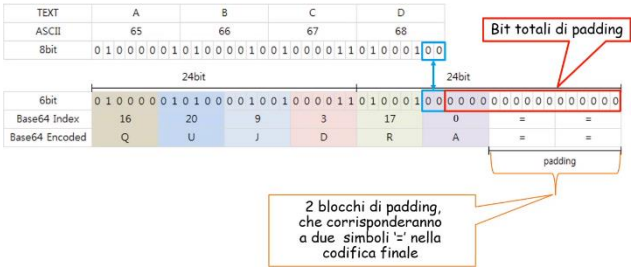


La codifica Base 64 procede nel seguente modo: il flusso preso in input viene processato in blocchi da 24 bit, in cui ciascun blocco è suddiviso in gruppi di 6 bit (a partire da sinistra), questo perché ogni simbolo in Base64 rappresenta ben 6 bit di dati. Successivamente, viene considerato il valore decimale di ciascun gruppo di bit, tale valore rappresenterà un indice nella tabella di codifica Base64 (valori da 0 a 63). Nell’esempio qui al lato, il primo blocco ha indice 19, secondo 22, terzo 5 e quarto 46. Una volta calcolati questi indici, vado a vedere nella tabella a che codifica corrispondono. Dopo aver fatto questa operazione posso affermare che:

La parola “Man”, in Base64 corrisponde a “TWFu”.

Abbiamo detto in precedenza che il flusso preso in input viene processato in blocchi di 24 bit. Se il numero totale di bit da processare non è un multiplo di 24 allora viene utilizzato il padding, cioè vengono inseriti bit nulli (0) alla fine. Nella codifica viene inserito il simbolo “=” per ogni gruppo di 6 bit che manca per creare un blocco da 24 bit. Ad esempio:

Se si vuole processare una stringa binaria la cui dimensione non è multipla di 24 bit. Infatti sono 32 bit. Per codificare questi 32 bit in Base 64, bisogna aggiungere dei bit di padding affinchè la lunghezza della stringa diventi un multiplo di 24. Aggiungo 16 bit di padding, di cui i primi 4 andranno a riempire l’ultimo blocco da 6 per formare il carattere “A” in Base64. I restanti 2 blocchi formati da 6 bit ciascuno, vengono codificati con il simbolo “=”. In questo modo ho ottenuto ciò che volevo.



La stringa ABCD, in base 64 corrisponde a “QUJDRA==”.

Vediamo ora un po’ di sintassi:

Per codificare un file in Base64 può essere usata la seguente sintassi:

openssl enc -base64 -in input-file -out output-file

Per decodificare un file codificato in Base64 può essere usata la seguente sintassi

openssl enc -base64 -d -in input-file -out output-file

Vediamo un primo esempio di cifratura:

FileInChiaro.rtf

Nel mezzo del cammin di nostra vita mi ritrovai per una selva oscura ché la diritta via era smarrita. Ahi quanto a dir qual era è cosa dura esta selva selvaggia e aspra e forte che nel pensier rinova la paura! Tant’è amara che poco è più morte; ma per trattar del ben ch’i’ vi trovai, dirò de l’altre cose ch’i’ v’ho scorte.

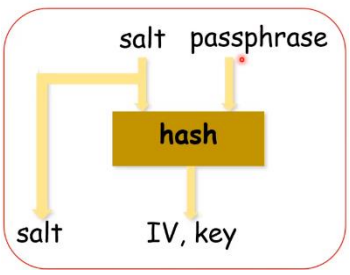
Si possiede il seguente file si da il seguente comando: ***openssl enc -aes-256-cbc -in FileInChiaro.rtf -out FileCifrato.rtf -e -pass pass: P1pp0B4ud0*** che genera il seguente file di output:

FileCifrato.rtf

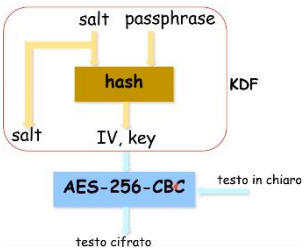
Viene cifrato un file utilizzato AES a 256 bit in modalità operativa CBC. In questo caso, come password viene utilizzata la stringa P1pp0B4ud0. Si ottiene il file cifrato. Se cercassi aprirlo, il contenuto è privo di significato e semantica.

Per la cifratura è stata utilizzata una password che non è l’unica componente che viene utilizzata per generare una password, infatti:

La password (passphrase) viene data in input ad una funzione Hash insieme ad un valore casuale salt. L’Hash genera un IV (vettore di inizializzazione) e la chiave che verrà utilizzata per cifrare/decifrare. Questo processo è detto **Key Derivation Function (KDF).**



Ritornando all’esempio descritto in precedenza (***openssl enc -aes-256-cbc -in FileInChiaro.rtf -out FileCifrato.rtf -e -pass pass: P1pp0B4ud0***) quello che succede è:



A partire dall’ultima versione di OpenSSL, la Key Derivation Function consigliata è PBKDF2, le cui caratteristiche sono state pubblicate come RFC2898. Rappresenta un’evoluzione di PBKDF1. La cosa importante da sapere è che usa HMAC più volte ed un salt per derivare una chiave di cifratura.

Mostriamo ora un esempio di cifratura simmetrica che però ora viene fatta in Base64:

Il testo in input (FileInChiaro.rtf) è lo stesso precedente per cui non viene riportato.

Diamo il seguente comando: ***openssl enc -aes-256-cbc -in FileInChiaro.rtf -out FileCifratoB64.rtf -e -base64 -pass pass:P1pp0B4ud0***

Il file di output generato è:

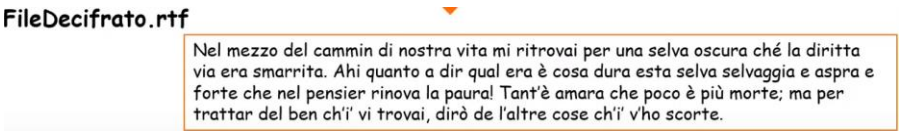
In questo modo, il file cifrato può essere mandato a qualsiasi persona poiché tutto il testo è visibile, a differenza dell'esempio fatto in precedenza.

Quando viene dato il comando ***openssl enc -aes-256-cbc -in FileInChiaro.rtf -out FileCifratoB64.rtf -e -base64 -pass pass:P1pp0B4ud0***, il sistema ci visualizza un messaggio di WARNING: "deprecated key derivation used. Using -iter or -pbkdf2 would be better", questo è un semplice dettaglio che è giusto mostrare.

Adesso passiamo al caso contrario: possediamo il FileCifratoB64.rtf e vogliamo restituire il suo testo in chiaro.

Eseguiamo il seguente comando: ***openssl enc -aes-256-cbc -in FileCifrato.rtf -out FileDecifrato.rtf -d -base64 -pass pass: P1pp0B4ud0***

Il risultato è il seguente:

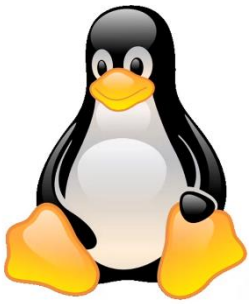


Mostriamo un altro esempio interessante:

Viene preso in input questo file tux.bmp che è un immagine bitmap. Adesso cifriamo il file usando AES con le modalità ECB e CBC, usando una chiave a 128 bit. I comandi rispettivi sono:

- (1) ***openssl enc -aes-128-ecb -in tux.bmp -out tux_aes_128_ecb.bmp -e -pass pass: P1pp0B4ud0***
- (2) ***openssl enc -aes-128-cbc -in tux.bmp -out tux_aes_128_cbc.bmp -e -pass pass: P1pp0B4ud0***

tux.bmp

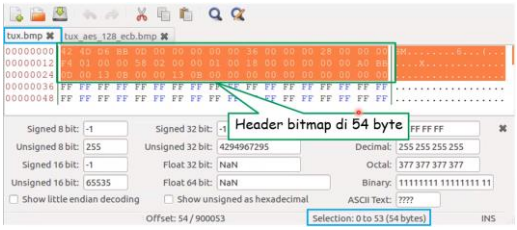


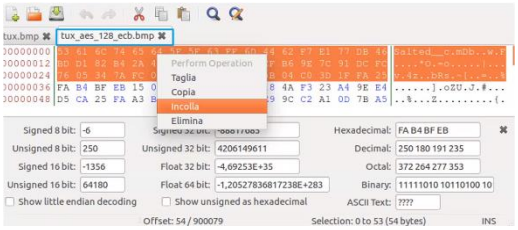
Abbiamo generato quindi 2 immagini cifrate distinte.

Per visualizzare il contenuto di un'immagine bitmap cifrata è necessario ripristinare il relativo header (metadati), contenuto nei primi 54 byte dell'immagine originaria (tux.bmp). Per farlo useremo l'editor esadecimale Bleass:

Siano *tux_aes_128_ecb.bmp* il file contenete la cifratura di tux.bmp con AES a 128 bit in modalità ECB e *tux_aes_128_cbc.bmp* il file contenente la cifratura di tux.bmp con AES a 128 bit in modalità CBC.

Usando Bless ripristiniamo l'header bitmap dell'immagine cifrata: apriamo sia il file tux.bmp che il file tux_aes_128_ecb.bmp. Copiamo i primi 54 byte del file tux.bmp in quelli corrispondenti del file tux_aes_128_ecb.bmp (La stessa operazione deve essere fatta per il file tux_aes_128_cbc.bmp)

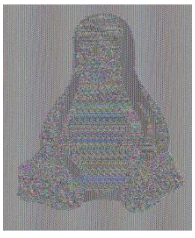




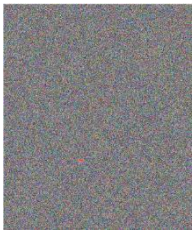
Assicurarsi che la selezione avvenga utilizzando l'indicizzazione in formato decimale. Ciò può essere ottenuto cliccando sul rettangolino in blu. Assicurarsi inoltre di aver selezionato i primi 54 bytes.

Dopo aver salvato i file, è stato ripristinato l'Header. Vediamo i risultati:

Per (1): Abbiamo detto in precedenza che ECB è una modalità insicura, e questo si riflette in questo risultato. Si nota che è un'immagine cifrata, però è abbastanza chiaro che si tratta di un pinguino, si notano le zampe, si nota che sta seduto e altre informazioni. Questo piccolo esempio ci fa capire che ECB non è una modalità sicura di cifratura.



Per (2): non si riesce a distinguere nulla, l'immagine è priva di semantica e riferimenti.



2.2 CIFRATURA ASIMMETRICA IN OpenSSL

Vedremo solo la cifratura RSA usando OpenSSL.

In generale, OpenSSL fornisce i comandi ***genrsa*** ed ***rsa*** per generare, esaminare, manipolare ed usare chiavi RSA. Il comando ***rsault*** per cifrare (e firmare) mediante RSA. I nomi dei comandi sono molto evocativi, infatti leggendo il comando si capisce subito a cosa serve.

Come abbiamo appena detto, la coppia di chiavi RSA può essere generata in OpenSSL mediante il comando ***genrsa***.

Le opzioni principali del comando ***genrsa*** sono:

openssl genrsa [options] [numbits]:

options:

- ***-des, -des3, -aes128, -aes192, -aes256***… : Cifra le chiavi generate, utilizzando DES o 3DES oppure utilizzando AES a 128, 192 o 256 bit, etc.
- ***-out file*** : Scrive in un file le chiavi generate.
- ***-passout arg*** : Password usata per la cifratura delle chiavi.
- ***-passin arg*** : Password usata per la decifratura delle chiavi.
- ***-F4, -3*** : esponente pubblico, in OpenSSL può essere 65537 (-F4) oppure 3 (-3). Il valore di default è 65537 e non sono ammessi altri valori.

numbits: numero di bit del modulo RSA, di default è 2048

Mediante il seguente comando è possibile generare una coppia di chiavi RSA a 1024 bit: tale coppia sarà cifrata attraverso AES a 128 bit, usando la stringa "P1pp0B4ud0" come password. Il risultato verrà scritto nel file rsaprivatekey.pem

Openssl genrsa -out rsaprivatekey.pem -passout pass: P1pp0B4ud0 -aes128 1024

Se la generazione andrà a buon fine, verrà mostrato il seguente messaggio:

```
Generating RSA private key, 1024 bit long modulus
....++++++
.............++++++
e is 65537 (0x10001)
```

È possibile visualizzare il contenuto del file *rsaprivatekey.pem* mediante il seguente comando:

openssl rsa -in rsaprivatekey.pem -text

NOTA BENE: Per visualizzare il contenuto *rsaprivatekey.pem* è necessario conoscere la sua password di cifratura (P1pp0B4ud0 nell'esempio precedente).

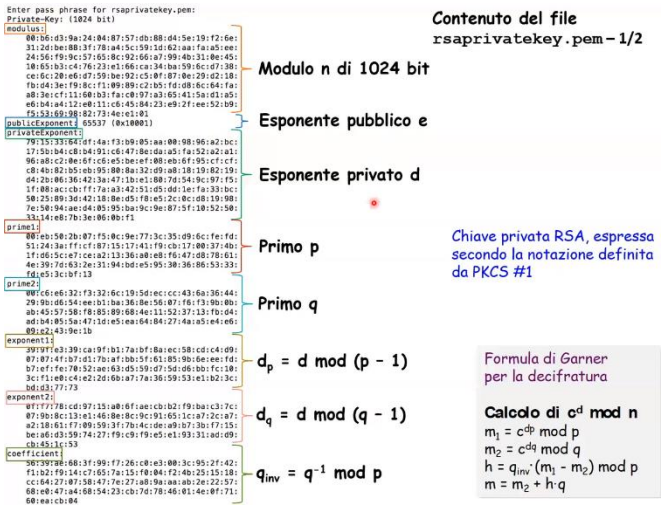
Le chiavi RSA in OpenSSL sono rappresentate secondo lo standard PKCS #1 (RFC3447), nessun dettaglio a riguardo.

La coppia di chiavi RSA viene rappresentata in OpenSSL in questo modo:

Ci sono tutti i parametri della coppia di chiavi RSA espressa nel formato PKCS #1. Oltre ai parametri tipici di RSA (n, e, d, p, q) ci sono altri parametri quali d_p, d_q, q_{inv} che sono parametri usati da OpenSSL per questioni di efficienza nella computazione. Questo file contiene al suo interno le componenti relative alla chiave pubblica e privata. C'è inoltre la codifica PEM della chiave privata RSA, che è una codifica in Base64:

```
writing RSA key
-----BEGIN RSA PRIVATE KEY-----
MIICWwIBAAKBgQC205okBIdX24jUXhnybjEtvog/eKRcWR1iqvql7iRW+ZxXZYyS
ZqeZSzEORRBls8R2I+FmyjS6WWzXOM5sIObXWb6SxQ+HDinSGPvUPvmM8QmJwrX9
2Gxk+qg+zxFgs/rAl6NlQVrRpea0pBLgEcZFhCPpL+5SufVTaZiCc07hAQIDAQAB
AoGAeRUzZN9K87kFqgCYlqK8F1u0yLSRxkeO2qX6UqKhljCDm/G5b7vCOtvlc/P
yEuCteuVgIoy2agYGYIZ1CsGNkI6RxvhgH1UnJf1Hwisy/96o0JR1d0e+jO8UCWJ
PUIYjtX45SwM2BmYflCUrtQFlbqcnodfEFJQMxToez4GC/ECQQDrUCsH9Qyedzw1
2Wz+/VEkOv/PhxUXQfnLFwA3Sx/WXOfOohM2oOj2R9h4YU45fWMuMZS95ZUwNoZT
M/3lPL8TAkEAxuYy8zJsGV3szENqNkQpm9ZU7rG6No5WB/bzmwurRVdY+IWJaE4R
UjcT+9SttAVaRx3l6mSEJ0qlSOYJ4kOeGwJAOZ/jOcqfsXq/iuxYzcTZBwdPt9F7
r7tfYYWbbu79t+/+cFKuY9VZ113W/wQPPHgxOIta6d6NllT4bI8vdN3cwJAD/d4
zZcVoG+uy7L5usN8B5uME+FGjoyckWUcpyynohhh9wlZP3tM3qm3O/cVvqbTWXQn
+cn55eGTMa3Zy0UcUwJAVjmuaD+Z9ybA4wA8lS9C8bL5FMdlehXwBPJLJRUYzGQn
B1hHfieomqqrLiJXaOBHpGhUI8t9eEYBTg9xYOrLBA==
-----END RSA PRIVATE KEY-----
```

Codifica PEM della chiave privata RSA



Alla fine si notano anche i bit di padding (==).

Mediante il seguente comando è possibile estrarre la chiave pubblica RSA memorizzata nel file *rsaprivatekey.pem* (ciò si fa perché in RSA c'è una netta separazione tra chiave pubblica e chiave privata e quindi vanno gestite in modo differente, poiché la chiave pubblica deve essere resa pubblica (scusate il gioco di parole cit.)):

Decifrando il contenuto di tale file attraverso la password "P1pp0B4ud0" e scrivendo la chiave pubblica nel file *rsapublickey.pem*

openssl rsa -in rsaprivatekey.pem -passin pass: P1pp0B4ud0 -pubout -out rsapublickey.pem

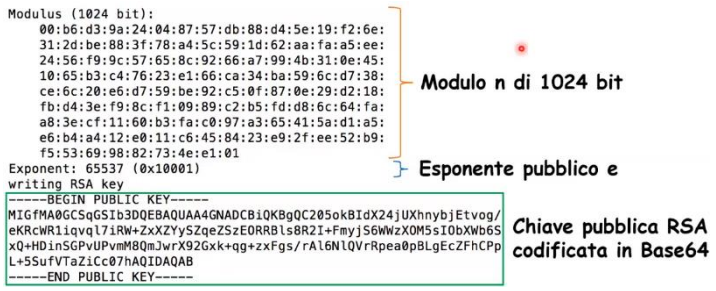
Viene preso in input il file contenente le due chiavi, lo apro con la password specificati e con il comando *-pubout* indico di inserire all'interno del file *rsapublickey.pem* la parte pubblica relativa alla coppia di chiavi.

Per visualizzare il contenuto del file *rsapublickey.pem* è possibile utilizzare il seguente comando:

openssl rsa -pubin -in rsapublickey.pem -text

E da notare che sto specificando che all'interno di questo file è presente una chiave pubblica. Questo è fatto col parametro *-pubin -in.*

Il contenuto del file *rsapublickey.pem* è il seguente:



Banalmente si osserva che manca l'Esponente privato d, mentre è presente tutta la parte relativa alla chiave pubblica.

Un'osservazione: il formato utilizzato di default da OpenSSL è *pem*. Di conseguenza OpenSSL si aspetta di ricevere tutti i file di input, output e codifica in formato *pem*. Questo formato è una codifica della rappresentazione delle chiavi che è fatta in PKCS #1. Oltre al formato *pem* esiste un altro formato che è quello *der* che è poco utilizzato poiché è un formato binario e quindi scomodo da usare (non si può usare per fare copia e incolla e varie operazioni descritte anche in precedenza).

Adesso possiamo esaminare l'ultimo comando OpenSSL che è coinvolto nelle operazioni di cifratura asimmetrica, che è il comando ***rsautl***: prima di passare alla spiegazione concreta del comando è necessario dire che le operazioni di cifratura e decifratura RSA sono onerose dal punto di vista computazionale, per cui queste operazioni non vanno utilizzate per cifrare grosse mole di dati (molto lente). Per tale scopo va invece usata la cifratura simmetrica (comando enc). Invece, usiamo le tecniche RSA per scambio chiavi, autenticazione, firma etc..

Il comando ***rsault*** consente di usare le chiavi RSA per la cifratura: permette di specificare opzioni per cifrare e decifrare i dati.

openssl rsault [options]

options:

- ***-in file*** : File di input
- ***-out file*** : File di output
- ***-encrypt*** : Cifra con la chiave pubblica
- ***-decrypt*** : Decifra con la chiave pubblica
- ***-sign*** : Firma con la chiave privata
- ***-verify*** : Verifica con la chiave pubblica
- ***-inkey*** : Chiave presa in input
- ***-passin arg*** : Sorgente da cui deve essere letta la password
- ***-pubin*** : Specifica che l'input è una chiave pubblica RSA
- ***-pkcs, -oaep, -ssl, -raw*** : Padding da usare: PKCS#1 (default), PKCS#1 OAEP (Optimal Asymmetric Encryption Padding), modalità speciale di padding usata in SSL v2, nessun padding, rispettivamente

Un'osservazione: in RSA, la cifratura avviene con la chiave pubblica e si decifra con la chiave privata. Per quanto riguarda la firma, invece, si firma con la chiave privata e si verifica la firma con la chiave pubblica.

Vediamo un esempio di Cifratura e Decifratura:

Si consideri il seguente testo in chiaro: ***testoInChiaro.txt*** : "Benvenuti al corso di Sicurezza, A.A. 2019/2020!".

Cifro nel seguente modo:

openssl rsautl -encrypt -pubin -inkey rsapublickey.pem -in testoInChiaro.txt -out testoCifrato.txt

Usando la chiave pubblica contenuta in ***rsapublickey.pem*** il contenuto di ***testoInChiaro.txt*** è cifrato e scritto in ***testoCifrato.txt***

Decifro nel seguente modo:

openssl rsautl -decrypt -inkey rsaprivatekey.pem -in testoCifrato.txt -out testoDecifrato.txt

Usando la chiave privata contenuta in ***rsaprivatekey.pem*** il contenuto di ***testoCifrato.txt*** è decifrato e scritto in ***testoDecifrato.txt***

Proviamo a rispondere a questo quesito: è possibile recuperare una chiave privata RSA a partire dalla relativa chiave pubblica? Prima di rispondere a questa domanda con un piccolo esempio di RSA Cracking è doveroso ricordare che la sicurezza di RSA si basa sull'intrattabilità dal punto di vista computazionale del problema della fattorizzazione. Questo problema su istanze di una certa dimensione non si può risolvere in tempo ragionevole. In concreto, a partire dal modulo n pubblico, ricavare i fattori che lo costituiscono p q è un'operazione che richiede tantissimo tempo.

Quello che faremo in questo esempio è di partire da n e ricavare p q che verranno codificate in maniera opportuna (conforme allo standard PKCS#1). Useremo un modulo piccolo.

Per concretizzare questo esempio si fa uso di vari strumenti e librerie Python che permettono di codificare in PKCS #1 p q, viene anche utilizzato uno strumento chiamato YAFU che permette di fattorizzare un numero dato in input. Verrà anche usato un file Python ***genPriv.py*** che è uno Script Python per la codifica di chiavi in formato PKCS1. Per installare le librerie necessarie bisogna dare il seguente comando su terminale:

```
sudo apt-get install python-pyasn1 python3-pyasn1
python-pyasn1-modules python3-pyasn1-modules
pypy-pyasn1
```

A monte della procedura di RSA Cracking viene generata una coppia di chiavi RSA con modulo 50 bit, estraiamo da essa la relativa chiave pubblica e ne stampiamo il contenuto:

opensll genrsa 50 > smallkey.pem [GENERO LA CHIAVE]

openssl rsa -pubout -in smallkey.pem > smallkey_pub.pem [ESTRAGGO LA PARTE PUBBLICA]

openssl rsa -in smallkey_pub.pem -pubin -text -modulus [VADO A MOSTRARLA NELLA FIGURA IN BASSO]

NOTA BENE: OpenSSL 1.1.1 consente di generare chiavi aventi lunghezza minima pari a 512 bit, di conseguenza come viene fatto in questo esempio, per generare chiavi di lunghezza inferiore è necessario utilizzare versioni precedenti di OpenSSL.

L'output dell'ultimo comando è:

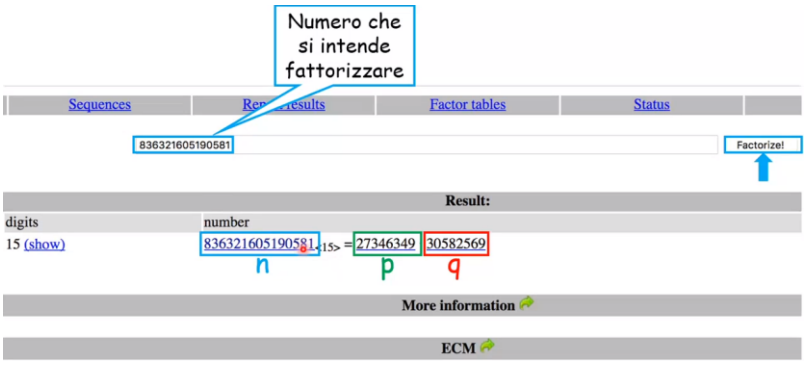
Questa è la chiave pubblica composta dall'esponente *e* e dal modulo *n* che è di 50 bit. Per mostrarli bisogna convertire il numero in base 10 presente in figura in binario e si ottiene la stringa da 50 bit. Adesso si vuole fattorizzare il modulo n (836321605190581) per trovare i fattori primi di p e q.

```
Modulus (50 bit): 836321605190581 (0x2f8a14c3203b5)
Exponent: 65537 (0x10001)
Modulus=2F8A14C3203B5
writing RSA key
-----BEGIN PUBLIC KEY-----
MCIwDQYJKoZIhvcNAQEBBQADEQAwDgIHAvihTDIDtQIDAQAB
-----END PUBLIC KEY-----
```

Considereremo due metodi alternativi per farlo:

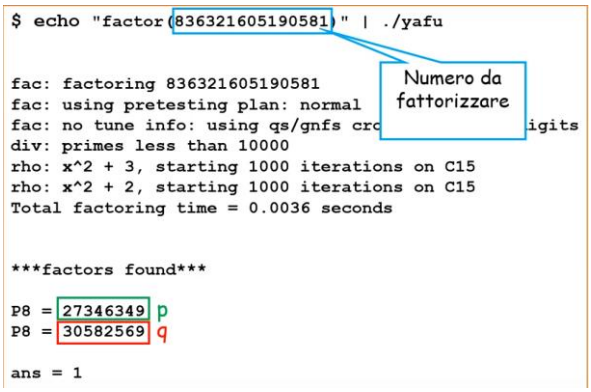
- Metodo online utilizzando http://factordb.com
- Metodo offline utilizzando YAFU

Iniziamo col metodo online, la procedura è molto semplice e viene riportato il risultato:



Dopo pochissimo tempo ci viene mostrato il risultato della fattorizzazione, i p q che sono i fattori di n.

Con il metodo offline, usando YAFU, non entreremo nei dettagli molto complessi che il programma utilizza per ottenere p q:



Con il comando echo, viene inviato in pipeline a YAFU il modulo n da fattorizzare.

Dopo aver ottenuto il valore dei numeri p e q, possiamo ricostruire la chiave privata e codificarla in PKCS#1 mediante il programma ***genPriv.py***

```
$ python2 genPriv.py
<<I valori di p, q ed e vanno inseriti in formato decimale>>
Inserisci p >> 27346349
Inserisci q >> 30582569
Inserisci e >> 65537
Inserire il nome del file dove memorizzare la chiave privata
>>  smallkey_priv.pem
```

2.3 ACCORDO SU CHIAVI OPENSSL

Vediamo come si può realizzare un **Accordo su Chiavi** in OpenSSL. Esistono vari metodi, tra cui quello di Diffie-Hellman (più famoso) e vedremo la realizzazione di questo.

La prima cosa da fare per utilizzare questo protocollo è quello di generare i parametri pubblici da utilizzare in tale protocollo, questi parametri devono essere condivisi tra le parti.

Il comando che OpenSSL mette a disposizione per la generazione di questi parametri è ***dhparam***. La sintassi è la seguente:

openssl dhparam [options] [numbits]

Dove **options**:

- **-inform arg** : formato di input, dove arg può essere DER o PEM.
- **-outform arg** : formato di output, dove arg può essere DER o PEM.
- **-in arg** : dove arg è il file di input.
- **-out arg** : dove arg è il file di output.
- **-text** : stampa i parametri Diffie-Hellman in formato testuale.
- **-2** : genera i parametri usando 2 come valore del generatore.
- **-5** : genera i parametri usando 5 come valore del generatore.

e **numbits:** numero di bit dei parametri da generare, di default sono 2048.

Le opzioni -2 e -5 sono le più importanti. Il valore di default è 2.

Mediante il seguente comando vengono generati i parametri pubblici per DH e vengono salvati nel file *dhparams.pem*:

openssl dhparam -out dhparams.pem -2 1024

Output del comando:

```
Generating DH parameters, 1024 bit long safe prime, generator 2
This is going to take a long time
.....+.....+.....+.....
.....+.....+.....+.....
.....+.....+.....+.....+..
.....
.....+.....+.....
....+.....+.....+.....+.....
.....+.+.....
.....
.....+.....+.
+.....+.....+.....+.....+.....
.....+.....
.....
.....
.....+.....
.....
.....+.....+.....+.....
+.....
.....+.....
.....+.....
.....+.....++*++++++*
```

Osservazione riguardante una domanda tipica orale: quando si chiede di simulare questo protocollo, molti studenti fanno generare i parametri Diffie-Hellman sia all'utente 1, sia all'utente 2. Questo è sbagliato è tutto quello che segue non ha più validità. Il comando dhparam va eseguito una sola volta da un partecipante e in qualche modo verrà passato all'altro partecipante.

Anche nel caso di Diffie-Hellman si ha la possibilità di visualizzare i parametri generati, memorizzati nel file dhparams.pem. Il comando è il seguente:

openssl dhparam -in dhparams.pem -text

```
Diffie-Hellman-Parameters: (1024 bit)
    prime:
        00:9e:e7:b3:91:2a:2c:6e:ca:38:cd:80:2d:c3:47:
        24:14:ed:64:a1:98:45:07:16:4d:e9:e2:2d:1e:84:
        15:80:d9:3c:fc:d6:63:db:ff:8f:33:31:36:ef:0d:
        e3:9d:7a:af:1c:a2:6a:ca:e7:9e:53:6f:a8:c6:a1:
        26:95:ce:17:ea:9d:cd:c8:11:69:21:e2:2c:bd:25:
        fe:20:dc:9f:49:c5:33:58:fd:8d:7c:07:57:6c:1c:
        5a:b3:73:45:22:4d:ef:d5:34:b3:ac:a6:5d:a3:04:
        81:13:2a:a4:a7:4e:34:60:1b:73:b6:0a:3b:a0:3d:
        4d:d7:81:4a:f3:39:6d:67:a3
    generator: 2 (0x2)
-----BEGIN DH PARAMETERS-----
MIGHAoGBAJ7ns5EqLG7KOM2ALcNHJBTtZKGYRQcWTeniLR6EFYDZPPzWY9v/jzMx
Nu8N4516rxyiasrnnlNvqMahJpXOF+qdzcgRaSHiLL0l/iDcn0nFM1j9jXwHV2wc
WrNzRSJN79U0s6ymXaMEgRMqpKdONGAbc7YKO6A9TdeBSvM5bWejAgEC
-----END DH PARAMETERS-----
```

p

g

Codifica PEM dei parametri DH

I parametri e le chiavi DH sono rappresentati e codificati secondo lo standard PKCS #3.

Torniamo al funzionamento effettivo del protocollo, fino ad ora abbiamo generato i parametri pubblici DH, che devono essere gli stessi per ambedue le parti. A questo punto ogni utente genera la propria coppia di chiavi e la memorizza in un file. Per la generazione della chiavi su utilizza un opportuno comando: **genpkey**:

Assumiamo che tale file sia **dhkey1.pem** (utente 1) e **dhkey2.pem** (utente 2):

UTENTE 1: **openssl genpkey -paramfile dhparams.pem -out dhkey1.pem**

UTENTE 2: **openssl genpkey -paramfile dhparams.pem -out dhkey2.pem**

È possibile visualizzare la struttura di un file contenente una coppia di chiavi mediante il seguente comando: **openssl pkey -in dhkey1.pem -text**

Questo comando appena mostrato stampa la struttura del seguente contenuto:

```
DH Private-Key: (1024 bit)
    private-key:
        40:51:79:87:a0:af:d4:28:48:e0:b4:64:61:a8:09:
        8d:26:8d:a7:0c:37:66:cc:ce:37:07:98:79:6f:46:
        ec:39:e1:1c:fc:3d:af:29:48:fc:7a:3b:34:ab:c2:
        e8:f3:fb:43:17:a8:7c:c7:16:ca:07:9a:54:27:06:
        34:06:78:03:32:c2:b0:71:33:de:af:17:1a:85:10:
        43:60:38:71:76:b2:f2:94:cc:8f:b7:4c:b9:9d:8a:
        66:9a:85:4c:c6:8f:8a:a7:11:bb:79:40:d0:ff:0d:
        02:dc:97:6c:92:09:6e:5a:9e:c8:3d:80:43:f3:a5:
        f6:03:12:47:16:4a:a4:60
    public-key:
        00:85:14:d3:dc:3b:ed:83:3d:46:09:c9:a7:14:f5:
        94:d2:c1:87:b5:10:0d:39:79:7a:c5:a6:9d:be:e0:
        94:c5:6d:4a:4e:ee:f9:09:69:32:dc:1b:79:4d:a0:
        d9:a3:5b:9e:c2:e8:bf:e6:4f:2d:a9:ea:79:ed:85:
        ce:64:d2:1f:a8:15:4b:f6:73:72:6f:fc:cc:bf:45:
        ea:0a:71:4d:e8:28:9f:cd:1d:21:90:a5:b3:54:d8:
        3b:87:68:09:89:aa:3d:5a:b0:ce:8b:40:ad:2d:23:
        b2:20:a2:6f:2a:a9:ae:9d:a2:80:1b:0d:bd:f0:7d:
        9c:80:62:30:7b:69:3c:8a:53
    prime:
        00:8c:39:39:d1:60:1a:c7:cb:cd:7a:1a:f7:8f:db:
        22:cc:97:76:d1:a4:c9:c7:d8:f4:98:a3:3f:4d:24:
        17:9f:97:a9:e1:9a:ed:e6:3c:3d:36:16:6e:2a:a8:
        00:04:09:9c:9d:13:32:c2:01:8e:7a:e5:eb:49:6c:
        0f:52:47:5b:51:d2:89:e3:9b:0d:f4:cf:fb:a4:a8:
        3f:64:ec:73:74:63:4b:b9:c9:2a:ec:ee:e1:cf:3c:
        67:3d:21:c4:f1:96:b9:09:85:39:34:bb:14:ce:a2:
        38:c7:33:59:37:11:c7:fe:0d:99:e0:7a:59:4e:7d:
        dc:bd:28:2a:01:16:f7:87:13
    generator: 2 (0x2)
```

Chiave Privata (x)

Chiave Pubblica (g^x mod p)

Numero Primo (p)

Generatore (g)

Secondo lo standard PKCS #3, una chiave privata DH è costituita da una chiave privata (x), la chiave pubblica, un numero primo p e dal generatore g (tutto ciò coerente alla teoria studiata).

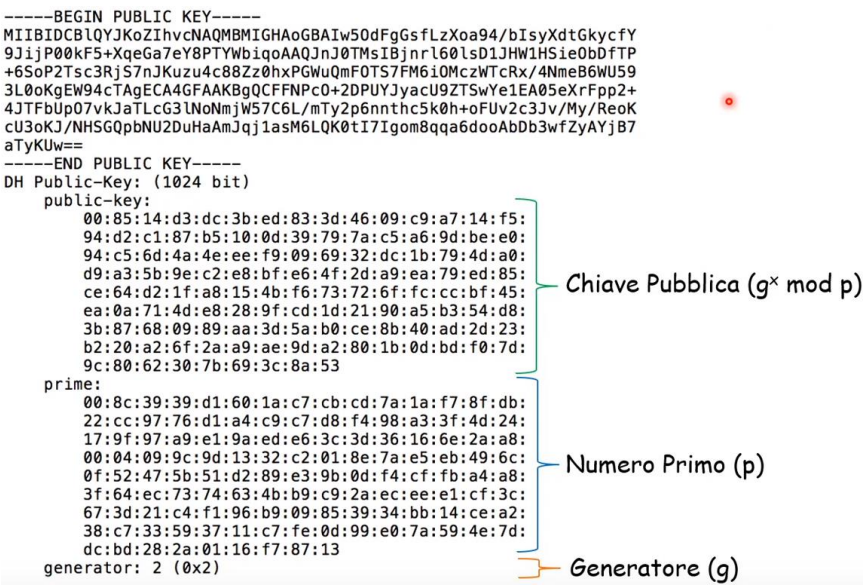
Proseguiamo col protocollo, il quale stabilisce che i 2 utenti debbano scambiarsi le rispettive chiavi pubbliche. Usiamo un procedimento molto simile a quello visto riguardante le coppie di chiavi RSA: essenzialmente per scambiarsi le chiavi pubbliche, ciascun utente estrae la propria chiave pubblica dalla coppia di chiavi e la memorizza in un file. Per fare questa operazione usiamo il comando **pkey**:

UTENTE 1: **openssl pkey -in dhkey1.pem -pubout -out dhpub1.pem**

UTENTE 2: **openssl pkey -in dhkey2.pem -pubout -out dhpub2.pem**

Per visualizzare la struttura del file dhpub.pem: **openssl pkey -pubin -in dhpub1.pem -text**

Otteniamo il seguente contenuto:



Si nota che manca la componente privata x, coerentemente all'idea di estrarre la parte pubblica.

Continuiamo il flusso di esecuzione: in questo momento le parti si sono accordate sui parametri, hanno generato la propria coppia di chiavi e hanno estratto la parte pubblica. L'ultimo passo riguarda l'idea di accordarsi su una chiave condivisa. Questa cosa è fatta attraverso il comando **pkeyutl** che richiede di specificare la chiave privata di un utente, la chiave pubblica dell'altro utente:

UTENTE 1: **openssl pkeyutl -derive -inkey dhkey1.pem -peerkey dhpub2.pem -out segreto1.bin**

UTENTE 2: **openssl pkeyutl -derive -inkey dhkey2.pem -peerkey dhpub1.pem -out segreto2.bin**

Se tutto va a buon fine, segreto1.bin e segreto2.bin sono identici. Ciò può essere verificato in vari modi, ad esempio tramite il comando **cmp** di Linux: **cmp -b segreto1.bin segreto2.bin** . Se il comando non restituisce niente allora i 2 file sono identici, se sono diversi il comando restituisce la posizione della prima riga in cui i 2 file differiscono.

Questo segreto generato può essere usato ad esempio come chiave per cifratura simmetrica, etc…

2.4 FIRME DIGITALI OPENSSL

Vedremo in particolare come implementare 2 tipologie di firme digitali con OpenSSL: vedremo come implementare la firma RSA e la firma DSA:

Firma RSA:

- FIRMA: **rsautl-sign**
- VERIFICA: **rsautl-verify**

Firma DSA:

- GENERAZIONE PARAMETRI: **dsaparam**
- GENERAZIONE CHIAVI: **gendsa**
- FIRMA: **dgst-sign**
- VERIFICA: **dgst-verify**

FIRMA RSA:

Il comando **rsautl** consente di usare le chiavi RSA per la firma: permette di specificare opzioni per firmare e verificare le firme. Nota bene: la firma è di solito apposta su un Hash di file (la dimensione dei dati da firmare è vincolata alla lunghezza della chiave). Firmare un Hash garantisce che il file sia autenticato e integrità (nessuna adulterazione del file). Vediamo il comando nel dettaglio:

openssl rsautl [options]

Dove **options:**

- **-in file** : file di input.
- **-out file** : file di output.
- -**encrypt** : cifra con la chiave pubblica.
- **-decrypt** : decifra con la chiave pubblica.
- **-sign** : firma con la chiave privata.
- **-verify** : verifica con la chiave pubblica.
- **-inkey**: chiave presa in input.
- **-passin arg**: sorgente da cui deve essere letta la password.
- **-pubin** : specifica che l'input è una chiave pubblica RSA
- **-pkcs, -raw** : padding da usare, PKCS#1 v1.5 (default) o nessun padding

Andremo a usare 2 parametri: **-sign** e **-verify**

Mediante il seguente comando è possibile firmare un file con RSA: **openssl rsautl -sign -inkey rsaprivatekey.pem -in testoInChiaro.txt -out rsasign.bin** In questo comando si osserva un file rsaprivatekey.pem (come generarlo è stato già trattato precedentemente). Si va quindi a firmare con RSA il file testoInChiaro.txt, per fare ciò si utilizza una chiave privata (specifico questa cosa con -inkey) e la firma la vado a memorizzare nel file rsasign.bin

Mediante il seguente comando è possibile verificare la firma RSA di un file: **openssl rsautl -verify -pubin -inkey rsapublickey.pem -in rsasign.bin -out testoInChiaro.txt** : per fare la verifica quindi uso il parametro **-verify**, la verifica è fatta attraverso una chiave pubblica dunque **-inkey rsapublickey.pem**, e specifico di verificare la firma del file **rsasign.bin**, creato precedentemente rispetto al file **testoInChiaro.txt**.

Se la verifica ha successo, il file **rsasign.bin** viene correttamente decifrato, altrimenti viene restituito un messaggio d'errore.

Mediante RSA è anche possibile firmare l'Hash di un file, quindi non firmo direttamente il file, ma firmo l'Hash del file: **openssl sha1 -sign rsaprivatekey.pem -out rsasign.bin testoInChiaro.txt**. Come si vede, si va a firmare l'Hash sha1 del file testoInChiaro.txt. La semantica è identica all'operazione precedente ma è firmato un Hash.

La verifica è del tutto speculare: **openssl sha1 -verify rsapublickey.pem -signature rsasign.bin testoInChiaro.txt**, si usa il comando **-verify**, si specifica la chiave pubblica, deve essere specificata la firma da verificare e si deve dire qual è il file di cui si vuole specificare la firma. Se la verifica va a buon fine viene mostrato "verified OK", altrimenti viene mostrato "verification failure".

Ai fini pratici è opportuno utilizzare l'approccio di firma e verifica con Hash.

FIRMA DSA:

Utilizzato in sistemi con hardware debole.

È possibile generare i parametri dello schema DSA mediante il comando **dsaparam**:

openssl dsaparam [options] [numbits]

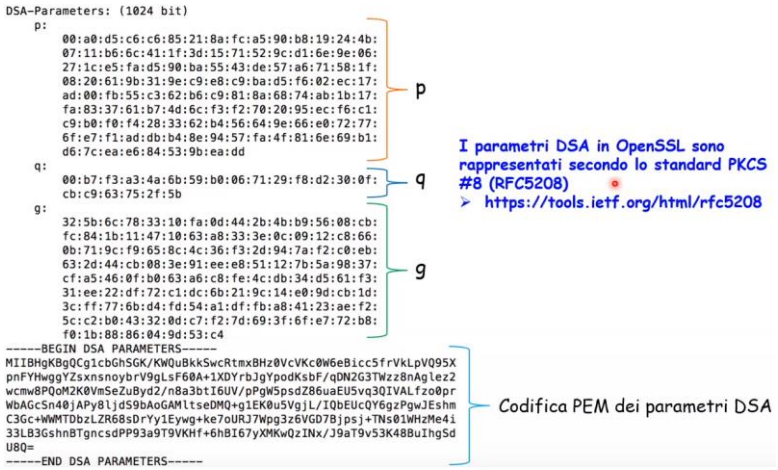
dove **options**:

- **-inform arg**: formato di input, dove **arg** può essere DER o PEM.
- **-outform arg**: formato di output, dove **arg** può essere DER o PEM.
- **-in arg**: dove **arg** è il file di input.
- **-out arg**: dove **arg** è il file di output.
- **-text**: stampa i parametri DSA in formato testuale.

E **numbits**: numero di bit da generare.

Mediante il seguente comando è possibile generare parametri DSA a 1024 bit: **openssl dsaparam -out dsaparams.pem 1024**

I parametri generati possono essere visualizzati mediante il seguente comando: **openssl dsaparam -in dsaparams.pem -text**. Il risultato del comando è il seguente:



Per poter usare lo schema DSA bisogna generare una coppia di chiavi a partire dai parametri dsa generato prima. Questa idea si concretizza con questo comando: **gendsa** che è molto simile al comando **genrsa**: genera a partire da dsaparams.pem una coppia di chiavi dsa che viene protetta con des3. Tutto ciò si concretizza col seguente comando: **openssl gendsa -out dsaprivatekey.pem -des3 dsaparams.pem**

Mediante il seguente comando è possibile estrarre la chiave pubblica dal file **dsaprivatekey.pem** : **openssl dsa -in dsaprivatekey.pem -pubout -out dsapublickey.pem**. Questo comando prende in input la coppia di chiavi e restituisce in output le componenti pubbliche delle coppie di chiavi

Adesso si può firmare e verificare tramite DSA. Per quanto riguarda la firma, con il seguente comando è possibile firmare l'Hash SHA1 di file.txt: **openssl dgst -sha1 -sign dsaprivatekey.pem -out dsasign.bin file.txt**, si specifica qual è la chiave privata utilizzata per la firma, il file che conterrà la firma e quale file si vorrà firmare.

Per quanto riguarda la verifica: **openssl dgst -sha1 -verify dsapublickey.pem -signature dsasign.bin file.txt**, in cui si specifica quale funzione Hash è stata utilizzata per firmare, usare il parametro -verify, specificare la chiave pubblica usata per la verifica, specificare quale file contiene la firma e rispetto a quale file deve essere controllata la firma (file.txt). Se la verifica ha esito positivo allora viene mostrato "Verified OK", altrimenti viene mostrato "Verification Failure".

FUNZIONI HASH CON OPENSSL:

OpenSSL fornisce numerose funzioni Hash (o chiamate anche Message Digest): MD4, MD5, SHA1, SHA3, RIPEMD-160, etc…

Mediante il seguente comando è possibile visualizzare le funzioni Hash fornite: **openssl list --digest -commands**. Si faccia attenzione che alcune funzioni fornite hanno problemi di sicurezza.

L'implementazione di funzioni Hash in OpenSSL è realizzato mediante il comando **dgst** che permette di accedere a tutte le funzioni Hash che la versione che si sta usando di OpenSSL fornisce. Questo comando opera su dati letti dallo standard input, oppure su uno o più file. Se alla funzione Hash viene passato più di un file, viene calcolato un Hash separato per ciascun file. L'Hash calcolato è scritto in formato esadecimale sullo standard output, a meno che non sia specificato un file di output.

Vediamo il comando **dgst:**

```
openssl dgst args file
```

Dove **args**:

- **-digest** : funzione Hash da usare per il calcolo del message digest. digest può essere uno degli algoritmi mostrati dal comando **openssl list –digest-commands**
- **-out filename** : file in cui scrivere l'output della funzione. Altrimenti l'output viene scritto sullo standard output.
- **-hmac key** : crea un hashed MAC (HMAC) usando una determinata chiave.

E **file**: file (uno o più) su cui deve essere applicata la funzione Hash.

Mediante il seguente comando è possibile calcolare la funzione Hash (SHA512 nell'esempio) di un file preso in input (file.txt): **openssl dgst -sha512 -out DigestOutput.txt file.txt**, il risultato viene messo in DigestOutput.txt.

Mediante il seguente domando è possibile calcolare l'HMAC di un file preso in input (file.txt), utilizzando la stringa P1pp0B4udo come chiave:

openssl dgst -sha512 -out HMACOutput.txt -hmac P1pp0B4ud0 file.txt, dove -sha512 sarà la funzione Hash usata all'interno dell'HMAC.

PSEUDOCASUALITA' IN OPENSSL

OpenSSL fornisce un comando per generare byte pseudocasuali, i quali possono essere mostrati sullo standard output e memorizzati su file. Per la generazione di questi byte OpenSSL usa dei Deterministic Random Bit Generator che hanno una buona qualità crittografica e che si basano su AES a 256 bit. Il seme usato da OpenSSL usa di default i random byte che sono collocati in cartelle definite dal sistema Unix.

Il comando per generare flussi pseudocasuali è **rand**. Vediamo in dettaglio:

```
openssl rand [options] [numbyte]
```

dove **options**:

- **-rand file(s) :** file usati come seme per il generatore.
- **-out file** : file su cui scrivere i dati generati, che altrimenti verrebbero scritti sullo standard output.
- **-base64**: i dati generati sono codificati in formato base64.
- **-hex** : i dati generati sono codificati in formato esadecimale

e **numbyte**: numero di byte che si intende generare.

Nel seguente esempio vengono generati 12 byte pseudocasuali e scritti nel file pseudorandom-data.bin 12:

openssl rand -out pseudorandom-data.bin 12 (se si decidesse di aprire il file generato sarebbe illeggibile e dunque poco utile ai fini pratici)

Mediante il seguente comando è possibile generare 6 random byte e codificarli in Base64: **openssl rand -base64 6** (in questo caso viene generata una stringa di caratteri pseudocasuali leggibile e utile ai fini pratici. Attenzione, se si contano i caratteri sono 8. Perché non sono 6? Perché si deve tener conto che la codifica che si effettua è in Base64, dunque valgono tutte le idee mostrate nelle pagine precedenti).

Naturalmente questo concetto può essere esteso ad un numero arbitrario di byte: **openssl rand -base64 12** produrrà una stringa di 16 caratteri.

La codifica Base64 pone delle limitazioni e cioè che quando i dati sono convertiti in Base64, la stringa prodotta avrà sempre una lunghezza multipla di quattro. Se si desidera una stringa pseudocasuale la cui lunghezza non sia multipla di quattro, è necessario "accorciare" tale stringa mediante altri strumenti, ad esempio:

mediante il seguente comando è possibile generare una stringa pseudocasuale composta da 39 caratteri stampabili:

openssl rand -base64 39 | cut -c1-39. Questa è una pipe in cui, l'output prodotto alla sinistra del simbolo | è preso in input dal comando posto dopo |, che è il comando cut.

Nel seguente esempio, utilizzando il contenuto del file .bash_history come seme per il comando rand, vengono generati 128 byte random, codificati in Base64: **openssl rand -rand .bash_history -base64 128**

Adesso vedremo come realizzare 2 concetti molto importanti, alla base di molte cose che utilizziamo e sono: **Public Key Infrastructure (PKI)** e **Time Stamping Authority (TSA)**

PUBLIC KEY INFRASTRUCTURE (PKI)

Si occupa di emettere certificati, revocarli quando necessario (maggiori informazioni sul file della teoria nel capitolo dedicato).

Per creare una PKI è innanzitutto necessario stabilire una root directory, dove risiederanno tutti i file file della *Certification Authority (CA)*:

```
mkdir CA
cd CA
```

Dopo aver creato la directory, al suo interno vanno creare due sottodirectory:

- **certs :** utilizzata per conservare una copia di tutti i certificati rilasciati dalla CA.
- **private** : utilizzata per mantenere una copia della chiave privata della CA. NOTA BENE, la chiave privata della CA deve essere protetta nel miglior modo possibile, tale chiave dovrebbe essere memorizzata a livello hardware o su una macchina non accessibile alla rete (una macchina detta "air gap".

Tutto ciò si traduce con i seguenti 2 comandi:

```
mkdir certs private
chmod g-rwx, o-rwx private
```

Col secondo comando andiamo a settare dei permessi alla cartella "private", cioè permettendo ad un utente non root di accedere alla cartella perché altrimenti per come funziona OpenSSL si potrebbero avere delle limitazioni di accesso a questa cartella (questa operazione ha senso per i fini didattici ma **NON** deve essere fatta in contesti reali).

Dopo aver creato le 2 cartelle all'interno di CA, bisogna creare anche alcuni file che sono necessari per il funzionamento della CA. In particolare i file che si creano sono 3 (sono testuali) e sono:

- **serial** : usato per tenere traccia dell'ultimo numero di serie utilizzato per il rilascio di un certificato. Nota bene che due certificati emessi dalla stessa CA devono sempre avere numeri di serie differenti. Il file serial sarà inizializzato a contenere il numero 01
- **certindex.txt** : usato per memorizzare le informazioni sui certificati emessi dalla CA (OpenSSL richiede che tale file esista, anche se vuoto, contiene informazioni come Common Name (CN), Nazione (C) , etc..).
- **crlnumber** : usato per tenere traccia dei certificati revocati. Il file crlnumber sarà inizializzato a contenere il numero 01.

Anche in questo caso traduciamo le idee in comandi pratici:

```
echo '01' > serial
touch certindex.txt
echo '01' > crlnumber
```

Abbiamo creato tutta la parte infrastrutturale con questa serie di operazioni. A questo punto, deve avvenire la fase di configurazione e il file con cui la CA si configura è un file chiamato **openssl.cnf**. Questo file si trova all'interno della cartella itcssl (si può anche eseguire il comando da terminale: **locate openssl.cnf**, vediamo agevolmente dove il file è collocato). La cosa importante è che non verrà utilizzata la configurazione system wide di questo file, bensì una sua versione locale: quindi copieremo questo file nella root directory della CA. Questo viene fatto per vari motivi: innanzitutto possiamo usare una versione ad hoc del file di configurazione in base a quello che ci serve. Un altro motivo è che se erroneamente si va a creare qualche grave danno all'interno del file di configurazione, avendone una fatta una copia locale, il danno è inutile in quanto nella peggiore ipotesi mi basterà eliminare il file locale e riprendere quello system wide.

Una volta copiato questo file, lo si apre con un opportuno editor di testo e al suo interno occorre fare un'operazione preliminare: bisogna trovare la seguente opzione:

```
RANDFILE                = $ENV : : HOME/.rnd
```

Poiché per i nostri scopi non serve questa linea di comando, la si mette a commento inserendo #:

```
#RANDFILE               = $ENV : : HOME/.rnd
```

A questo punto bisogna configurare ulteriori opzioni all'interno del file openssl.cnf:

- Bisogna verificare che **dir** sia eguagliato alla Root directory della CA (poiché si suppone che si siano seguiti i passi precedenti e che quindi il file openssl.cnf si trova all'interno della cartella CA, lo farò puntare alla cartella in cui si trova, quindi basterà semplicemente inserire un punto → . (si ricorda che la directory punto indica la directory corrente nei sistemi Unix)).
- Bisogna verificare che **serial** sia eguagliato al file serial che abbiamo creato precedentemente: quindi bisogna inserire: **$dir/serial**.
- Bisogna verificare che **database** sia eguagliato al file certindex.txt che abbiamo creato precedentemente: quindi bisogna inserire **$dir/certindex.txt**.
- Bisogna verificare che **new_certs_dir** sia eguagliato alla directory certs che abbiamo creato precedentemente: quindi bisogna inserire **$dir/certs**.
- Inserire per **certificate**: **$dir/cacert.pem**
- Inserire per **private_key**: **$dir/private/cakey.pem**
- Bisogna verificare che **default_days** (durata del certificato emesso) sia eguagliato a 365, eventualmente aggiungerlo manualmente.
- Bisogna verificare che **default_crl_days** (numero di giorni dopo i quali è emessa una nuova CRL, Certificate Revocation List) sia eguagliato a 30, eventualmente aggiungerlo manualmente.
- Bisogna verificare che **default_md** (algoritmo usato per il calcolo del Message Digest) sia eguagliato a sha1, eventualmente aggiungerlo manualmente.

A questo punto ci manca 1 cosa: bisogna creare il certificato (self-signed root certificate, cioè rilasciato da se stesso) della CA con il seguente comando:

```
openssl req -new -x509 -extensions v3_ca -keyout private/cakey.pem -out cacert.pem -days 365 -config ./openssl.cnf
```

OpenSSL richiede una password per cifrare la chiave privata. La sicurezza dell'intera CA si basa sulla chiave privata. Se questa chiave è compromessa, viene compromessa l'integrità della CA. Tutti i certificati emessi dalla CA, sia prima che dopo la compromissione, non possono più essere considerati attendibili.

Adesso bisogna simulare una richiesta di un utente che ha bisogno di avere un certificato. Per fare ciò è innanzitutto necessario creare la directory in cui risiederà la chiave privata dell'utente:

```
mkdir private
chmod g-rwx, o-rwx private
```

Mediante il seguente comando un utente effettua la richiesta di certificato:

```
openssl req -new -out utente-req.pem -keyout private/utente1-key.pem -days 365 -config ./openssl.cnf
```

Questo comando crea due file:

- **utente1-req.pem** : contenente la richiesta di certificato.
- **utente1-key.pem** : contenente le chiavi associate alla richiesta di certificato.

Nota bene che verrà richiesta una password, che sarà usata per cifrare la chiave privata.
Poiché stiamo simulando una richiesta, bisogna inviare alla CA il file utente1-req.pem (dunque copio questo file nella Root directory della CA).
Assumiamo che CA abbia ricevuto la richiesta, la quale è pronta a rilasciare il certificato a seguito della richiesta. Questa operazione sono realizzate mediante comando **ca** che permette di implementare tutte le operazioni che una CA fornisce. La sua struttura è la seguente:

```
openssl ca args
```

Dove **args**:

- **-in file** : file di input contenente la richiesta di certificato.
- **-out file** : file di output.
- **-config file** : file di configurazione.
- **-gencrl** : genera una nuova CRL.
- **-revoke file** : revoca un certificato passato come file.
- **-updatedb** : rimuove dal database dei certificati scaduti.

La CA, ricevuta la richiesta, genera il certificato per l'utente mediante il seguente comando:

openssl ca -out utente1-cert.pem -config ./openssl-cnf -in utente1-req.pem

Se il comando va a buon fine:

- Viene aggiunta una nuova entry nel file certindex.txt
- Viene incrementato il numero di serie nel file serial.
- Vengono generati due nuovi file:
  - **utente1-cert.pem** : certificato che dovrà essere inviato all'utente.
  - Copia di tale certificato, che sarà automaticamente memorizzata nella directory certs della CA (il nome di tale file è dato dal suo numero seriale seguito dall'estensione .pem).

La revoca di un certificato richiede una copia del certificato da revocare (la CA possiede una copia di tutti i certificati che ha emesso). Mediante il seguente comando la CA può revocare un certificato emesso:

openssl ca -revoke utente1-cert.pem -config ./openssl.cnf

Osservazioni:

Su un certificato revocato non avviene alcuna modifica. L'unico cambiamento è nel database della CA certindex.txt: l'entry corrispondente al certificato revocato inizia con la lettera R, mentre quella corrispondente ad un certificato valido inizia con la lettera V.

Una volta che un certificato è stato rilasciato non può essere più modificato: non è più possibile aggiornare tutte le copie di un dato certificato emesso.

Per rendere pubbliche le informazioni presenti in certindex.txt , sia utilizza una *Certificate Revocation List (CRL)*.

Mediante il seguente comando è possibile generare la CRL:

openssl ca -gencrl -out exampleca.crl -config ./openssl.cnf

Se il comando è eseguito senza errori, non viene mostrato alcun output, altrimenti, viene mostrato un messaggio d'errore.

Mediante il seguente comando è possibile visualizzare il contenuto della CRL:

openssl crl -in example.crl -text

Un esempio di CRL è il seguente:

exampleca.crl

```
Certificate Revocation List (CRL):
        Version 1 (0x0)
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: /O=University of Salerno/L=Fisciano/ST=Italy/C=IT
        Last Update: Feb 20 21:17:00 2017 GMT
        Next Update: Mar 22 21:17:00 2017 GMT
Revoked Certificates:
    Serial Number: 01
        Revocation Date: Feb 20 20:33:51 2017 GMT
    Serial Number: 02
        Revocation Date: Feb 20 21:11:00 2017 GMT
    Serial Number: 03
        Revocation Date: Feb 20 21:15:26 2017 GMT
    Signature Algorithm: sha1WithRSAEncryption
         66:77:28:17:7f:d5:38:1e:e9:74:cc:e0:6d:e0:78:79:5a:
         62:e0:fc:53:29:d9:0a:ea:9e:df:69:f4:e8:03:76:12:d6:
         96:1c:c4:fb:68:af:e4:b4:12:d1:ee:d8:68:ae:4b:81:
         30:51:fb:16:5b:42:fc:55:c3:
         2f:3d:72:48:14:9a:1f:a6:8c:
         08:0f:61:17:be:63:2a:26:24:
         03:e2:b6:38:b8:6b:ec:c1:7b:
         5f:17
-----BEGIN X509 CRL-----
MIIBUzCBvTANBgkqhkiG9w0BAQUFADBQMR4wHAYDVQQKExVVbml2ZXJzaXR5IG9m
IFNhbGVybm8xETAPBgNVBAcTCEZpc2NpYW5vMQ4wDAYDVQQIEwVJdGFseTELMAkG
A1UEBhMCSVQXDTE3MDIyMDIxMTcwMFoXDTE3MDMyIxMTcwMFowPDASAgEBFw0x
NzAyMjAyMDMzNTFaMBICAQIXDTE3MDIyMDIxMTEwMFowEgIBAxcNMTcwMjIwMjEx
NTI2WjANBgkqhkiG9w0BAQUFAAOBgQBmdygXf9U4HFrpdMzgbeB4eVpi4PxTKdkK
6p6F32n06AN2EtaWHMT7aK/ktBLflr3u2GiuS4EwUfsWW0L8VcFEyhriuErPQVEv
PXJIFJofpo1ll4aBPAWiYhEID2EXvmMqJiQYdXl+fX3sXiED4rY4uGvswXu9Qpn3
qUhPlJ1fFw==
-----END X509 CRL-----
```

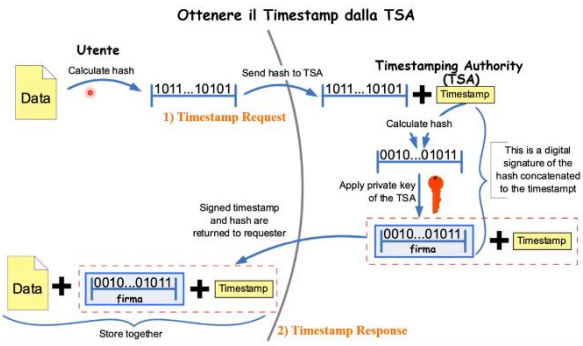
Numero seriale dei certificati revocati e relativa data di revoca

TIME STAMPING AUTHORITY (TSA)

Si occupa di certificare il tempo, cioè TSA certifica che l'esistenza di un certo file a partire da un certo istante temporale. OpenSSL permette di realizzare una specifica tipologia di TSA, poiché ce ne sono varie. Quella che OpenSSL utilizza è conforme ad RFC 3161 ed è basata sul modello Client/Server.

Il protocollo alla base di questa TSA funziona nel seguente modo:

Un utente vuole che gli venga certificato il tempo relativo ad un determinato file. L'utente calcola un valore Hash per il file di suo interesse e questo valore Hash lo invia alla TSA (Timestamp Request), la quale riceve il valore Hash, concatena a questo valore Hash un Timestamp e su questa concatenazione ne calcola l'Hash. La TSA poi firma (RSA pura) questo Hash e invia il Timestamp e la firma dell'Hash all'utente che l'ha richiesto (Timestamp Response).



Chiunque può verificare il Timestamp di un determinato file, quindi non solo l'utente che ne ha fatto richiesta. Per effettuare la verifica bisogna calcolare l'Hash del file originario e si concatena al Timestamp emesso dalla TSA, calcolo l'Hash di questa concatenazione e se questo risultato è uguale alla decifratura della firma che viene generata da TSA allora il Timestamp e il file non sono stati alterati e il Timestamp è stato emesso dalla TSA ed è valido. Se la verifica non va a buon file allora almeno una delle due condizioni appena descritte non è più valida.

D'ora in avanti assumeremo che:

- **cacert.pem** sia il certificato della CA.
- **tsacert.pem** sia il certificato di firma della TSA rilasciato dalla CA.
- **tsakey.pem** sia la chiave privata della TSA.

Il processo di creazione della struttura e della configurazione di una TSA è molto simile a quello visto per una CA. Per implementare una TSA è necessario:

- Creare la sua struttura (Directory, file, etc.).
- Creare il suo certificato di firma, tale certificato deve contenere particolari estensioni. Nota bene, andranno specificati alcuni parametri nel file openssl.cnf sia della TSA che della CA che dovrà rilasciare il certificato di firma.

Per creare una TSA è innanzitutto necessario creare una struttura simile a quella fatta per la CA:

```
mkdir TSA
cd TSA
mkdir private
chmod g-rwx, o-rwx private
echo '01' > tsaserial
```

Bisogna poi aggiungere la seguente estensione al file openssl.cnf sia della CA che della TSA (riporto l'immagine per evitare errori nella trascrizione)::

```
[v3_tsa]
basicConstraints=CA:FALSE
subjectKeyIdentifier=hash
authorityKeyIdentifier=keyid,issuer
keyUsage = nonRepudiation,digitalSignature
extendedKeyUsage = critical,timeStamping
```

Bisogna aggiungere i seguenti parametri al file openssl.cnf della TSA (riporto l'immagine per evitare errori nella trascrizione):

```
[ tsa ]

default_tsa = tsa_config1

[ tsa_config1 ]
dir        = .
serial         = $dir/tsaserial
signer_cert    = $dir/tsacert.pem
signer_key     = $dir/private/tsakey.pem
digests        = sha1, sha256, sha384, sha512
```

Bisogna fare attenzione a controllare che i path definiti in questo file di configurazione siano consistenti con i path dell'infrastruttura creata.

Abbiamo dunque creata la struttura della TSA ed è stata configurata.

A questo punto bisogna creare il certificato: la TSA genera la richiesta di certificato e la invia alla CA:

```
openssl req -new -out tsa-req.pem -keyout private/tsakey.pem -days 365 -extensions v3_tsa -config ./openssl.cnf
```

Dove -extensions v3 tsa indica che si sta effettuando una richiesta per un tipo specifico di certificato.

La CA genera il certificato per la TSA e lo invia a quest'ultima:

```
openssl ca -out tsacert.pem -extensions v3_tsa -config ./openssl.cnf -in tsa-req.pem
```

I comandi coinvolti nel funzionamento di una TSA sono:

- Generazione di una Timestamp Request relativa ad un file: **openssl ts -query**
  La struttura generale del comando **ts -query** è:
  openssl ts -query args
  dove args:
  - **-data file_to_hash** : file per il quale deve essere creata la richiesta di timestamp.
  - **-config file :** file di configurazione.
  - **-digest digest_bytes** : permette di usare direttamente il digest, senza specificare il file coi dati.
  - **-cert** : richiede alla TSA di includere nella risposta il relativo certificato di firma.
  - **-out request.tsq** : file di output in cui verrà memorizzata la richiesta.
  - **-text** : restituisce l'output in formato human-readable.
- Generazione di una Timestamp Response in base ad una richiesta: **openssl ts -reply**
  La struttura generale del comando **ts -reply** è:
  openssl ts -reply args
  dove args:
  - **-queryfile request.tsq** : file contenente la richiesta di timestamp in formato DER.
  - **-signer tsa_cert.pem** : certificato della TSA in formato PEM.
  - **-inkey private.pem** : chiave privata della TSA in formato PEM.
  - **-in response.tsr** : specifica un response timestamp o un timestamp token precedentemente creato in formato DER.
  - **-out response.tsr** : la risposta della TSA è scritta in questo file.
- Verifica della corrispondenza tra una risposta ed una particolare richiesta o un determinato file: **openssl ts -verify**
  La struttura generale del comando **ts -verify** è:
  openssl ts -verify args
  dove args:
  - **-data file** : file di cui si vuole verificate il timestamp.
  - **-in response** : response timestamp, in formato DER, che deve essere verificato.
  - **-CAfile trusted_certs.pem** : file contenente un insieme di certificati trusted e self-signed, in formato PEM.
  - **-untrusted cert_file.pem** : file contenente un insieme di certificati non trusted, in formato PEM, che possono essere necessari alla costruzione della certificate chain relativa al certificato di firma della TSA.
  - **-queryfile request.tsq** : richiesta di timestamp in formato DER.

Mediante il seguente comando un utente crea una richiesta di Timestamp per il file1.txt:

```
openssl ts -query -data file1.txt -out file1.tsq
```

Mediante il seguente comando è possibile visualizzare la precedente richiesta, contenuta in file1.tsq:

```
openssl ts -query -in file1.tsq -text
```

Mediante il seguente comando la TSA genera una risposta (timestamp response) ad una richiesta di timestamp:

```
openssl ts -reply -config ./openssl.cnf -queryfile file1.tsq -inkey private/tsakey.pem -signer tsacert.pem -out file1.tsr
```

Mediante il seguente comando è possibile visualizzare il contenuto di una timestamp response:

```
openssl ts -reply -in file1.tsr -text
```

Mediante il seguente comando è possibile verificare una timestamp response relativa ad una richiesta:

```
openssl ts -verify -queryfile file1.tsq -in file.tsr -Cafile cacert.pem -untrusted tsacert.pem
```

Mediante il seguente comando è possibile verificare una timestamp response rispetto al file originario:

```
openssl ts -verify -data file1.txt -in file1.tsr -Cafile cacert.pem -untrusted tsacert.pem
```