UNIVERSITÀ DI PARMA

DIPARTIMENTO DI SCIENZE MATEMATICHE, FISICHE ED INFORMATICHE
Corso di Laurea in Informatica

Sicurezza delle reti

Parte II: Crittografia applicata e OpenSSL

RETI DI CALCOLATORI - a.a. 2022/2023

Roberto Alfieri

La sicurezza delle reti: sommario

PARTE A

- I servizi di sicurezza
- Metodi e strumenti di attacco
- Metodi e strumenti di difesa

PARTE B

- Crittografia applicata e OpenSSL

PARTE C

- Protocolli di autenticazione
- IPsec
- VPN
- Sicurezza delle reti WiFi

Crittografia

La crittografia è lo studio dei metodi per rendere un messaggio “ofuscato” in modo da renderlo comprensibile solo a persone a cui il messaggio è destinato (servizio di sicurezza **Confidenzialità**).

L'algoritmo che esegue la cifratura o decifratura è detto **Cipher.**

La **cifratura E** (Encryption) si applica ad un **messaggio in chiaro P (Plaintext)** per ottenere un **testo cifrato C (Ciphertext). C=E(P)**

La **decifratura D** (decryption) si applica ad un **messaggio cifrato C** per ottenere il **testo in chiaro P. P=D(C)**

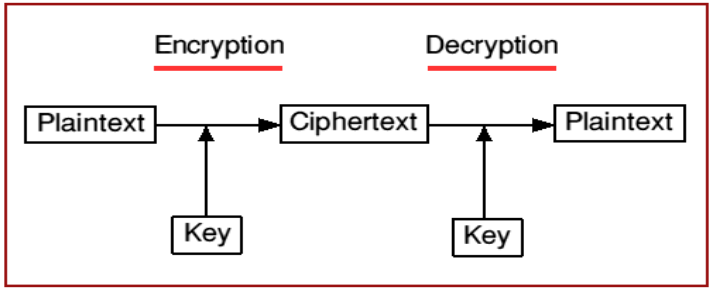
La **crittanalisi** è lo studio dei metodi per ottenere il significato di un testo cifrato violando la tecnica di cifratura del Cipher per verificarne la robustezza.

La robustezza di un Cipher non si basa sulla segretezza degli algoritmi di cifratura, ma sull'utilizzo di **chiavi segrete k** utilizzate nella fase di cifratura e decifratura.

Se gli algoritmi utilizzano la stessa chiave per cifrare e decifrare sono detti a **chiave simmetrica**, altrimenti sono detti a **chiave pubblica.**

A chiave simmetrica C=E(P,k) P=D(C,k)

A chiave pubblica C=E(P,k1) P=D(C,k2)



SSL/TLS

SSL (Secure Socket Layer) protocollo sviluppato originariamente dalla Netscape per fornire autenticazione e privacy in una connessione tra un client e un server mediante la l'uso di diverse tecnologie crittografiche.

Le principali tecnologie sono:

- Algoritmi crittografici (a chiave simmetrica o a chiave pubblica)
- Message Digest e firma digitale
- Certificati e Certification Authority
- Security Socket Layer (SSL) e Transport Layer Security (TLS)

Versioni di SSL e TLS:

- SSL v2.0 (1995): prima versione implementata da Netscape (deprecata dal 2011)
- SSL v3.0 (1996): revisione della precedente (deprecata dal 2015)
- TLS 1.0 (1999): RFC 2246. Revisione di SSL 3.0 (deprecata dal 2021)
- TLS 1.1 (2006): RFC 4346. (deprecata dal 2021)
- **TLS 1.2** (2008): RFC 5246.
- **TLS 1.3** (2018): RFC 8446 (attuale)

Vedi “SSL/TLS Strong Encryption: An Introduction” https://httpd.apache.org/docs/2.4/ssl/ssl_intro.html

Implementazioni del protocollo SSL/TLS

OPENSSL (http://www.openssl.org/)

E' una libreria Open Source basata sulla libreria SSLeay sviluppata da Eric Young e Tim Hudson.

Fino alla versione openSSL1.1 si applica la licenza originale SSLeay più restrittiva e incompatibile rispetto a GPL.

Dalla versione openSSL3.0 si applica la Apache License v2 (https://www.openssl.org/source/license)

Fornisce:

- Un ToolKit per l'utilizzo a linea di comando delle API.

- La Libreria SSL per la programmazione di canali cifrati con SSL/TLS.

- La Libreria Crypto per la programmazione di diversi algoritmi crittografici tra cui la cifratura a chiave simmetrica, a chiave pubblica, certificati e funzioni di Hash.

GnuTLS (http://www.gnutls.org/)

Creato per consentire alle applicazioni del Progetto GNU di usare una libreria compatibile con la GPL

Licenza GPL e LGPL

Fornisce API di programmazione (C/C++ Python PHP) e alcune utilities di supporto.

OpenSSL ToolKit

Il Toolkit consente l'utilizzo a linea di comando di tutte le operazioni crittografiche della libreria.
Sintassi:

```
openssl comando [opzioni dei comandi]
```

Ad esempio per vedere la versione di openssl in uso:

```
openssl version
```

Documentazione:

man <comando>	https://www.openssl.org/docs/manpages.html
wiki	https://wiki.openssl.org/index.php/Main_Page
OpenSSL Command-Line HOWTO:	http://www.madboa.com/geek/openssl/

Principali Comandi:

Cipher simmetrici: *enc (base64 bf des des3 idea rc4 rc5)*

Crittografia a chiave pubblica: *rsa rsautl gendsa genrsa*

Certificati X.509: *x.509 (gestione dati dei certificati), ca (per utilizzare le funzioni di Ceritification Authority), req (richiesta di certificati), verify (verifica di certificati), crl (Certificate Revocation List), crl2pkcs7 pkcs7(conversione tra diversi formati di certificati),*

Message digest *: dgst (md2, md5, rmd160, sha, sha1)*

Altro*: ciphers -v (elenco ciphers supportati), speed (benchmarking), prime (numeri primi) e rand, password, smime, s_client, s_server*

Comandi OpenSSL: base64

Viene utilizzata per codificare una sequenza binaria in una sequenza di caratteri ASCII.

La sequenza binaria è suddivisa in blocchi di 6 bit ciascuno dei quali viene trasformato in un carattere ASCII mediante un alfabeto dei 64 caratteri [a-zA-Z0-9./]

Vedi https://wiki.openssl.org/index.php/Base64

```
$ openssl   base64 -e  -in toencrypt.txt –out toencrypt.b64 (-e default)
$ openssl   base64 -d  -in toencrypt.b64 -out toencrypt.txt

$echo "encode me" | openssl base64 > encryped.b64
$openssl   base64 -d -in encrypted.b64
```

Comandi OpenSSL: prime e rand

OpenSSL contiene routine per trattare numeri primi e numeri random
(poiché questi servono per le tecniche crittografiche).

Vedi: https://wiki.openssl.org/index.php/Random_Numbers

Esempi:

```
# Testare se uno o più numeri sono primi:
$ openssl prime 11905475924546O753
> 1A6F7AC39A53511 is not prime
$ for N in $(seq 100 300); do openssl prime $N; done

# Scrive un numero random di 128 byte su stdout codificato in base64
$ openssl rand -base64 128

# Scrive un numero random di 32 bits (4 bytes) su un file
$ openssl rand -out random-data.bin 4
```

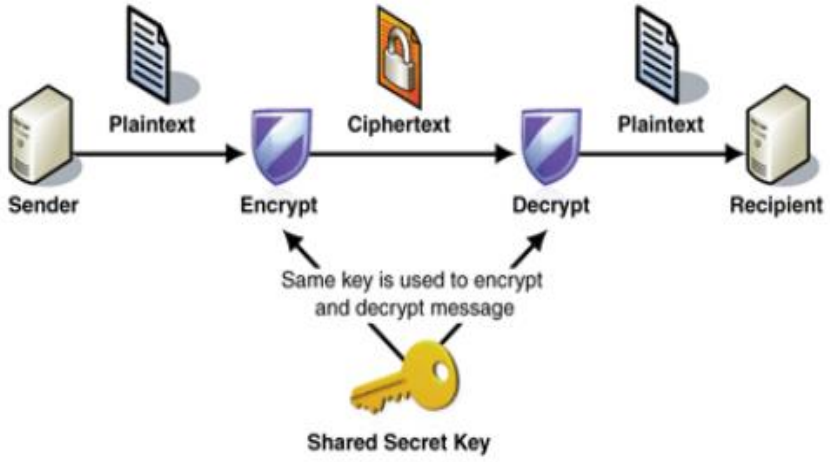
Esempio: programma in prime.c che usa le funzioni BN (BigNum) della libreria Crypto per generare numeri primi random da 1024 bit

http://didattica-linux.unipr.it/~roberto.alfieri@unipr.it/matdid/RETI/security/prime.c

Crittografia a chiave Simmetrica

E' una tecnica crittografica a chiave condivisa che consente di ottenere la cifratura di un messaggio P $C=E_K(P)$ in modo che la stessa chiave possa essere utilizzata per decifrarlo $P=D_K(C)$

Gli algoritmi E e D (Cipher) sono noti. La complessità della cifratura è data da K la cui lunghezza determina l'ampiezza dello spazio delle possibili codifiche di P e di conseguenza la robustezza della cifratura.

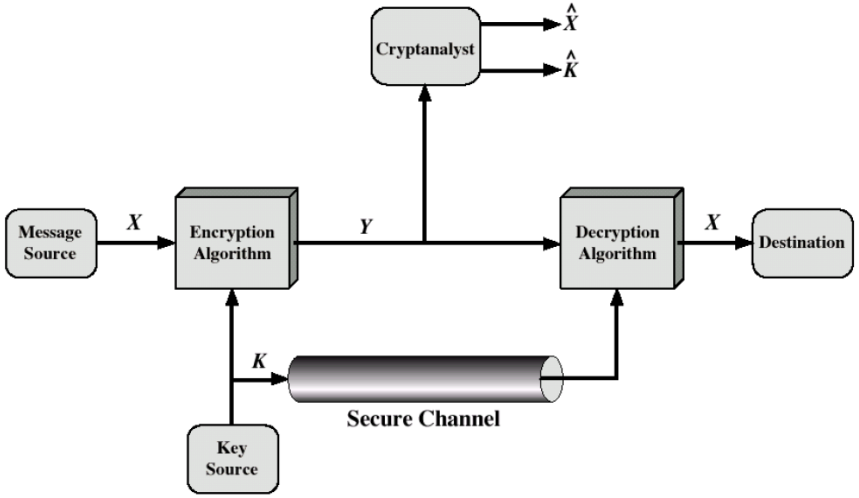


Algoritmi di cifratura molto performanti e semplici da implementare.

Crittografia a chiave Simmetrica: criticità

1) E' necessario un canale (o un metodo) sicuro per **distribuire la chiave tra le due parti.**

Generalmente lo scambio avviene attraverso algoritmi a chiave pubblica, più complessi sia da implementare che da eseguire, ma che permettono questo scambio in modo sicuro.



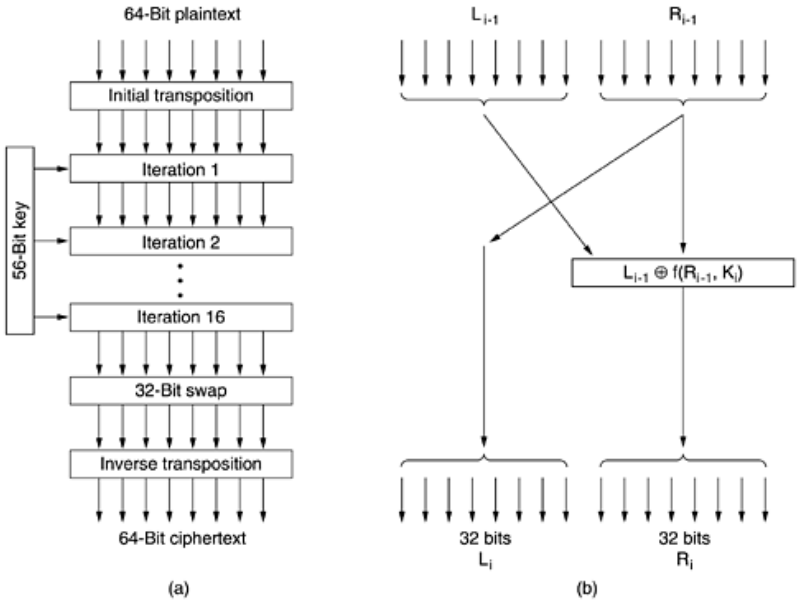
2) Il cipher può comunque essere compromesso con **un attacco a forza bruta**, ovvero testando tutte le possibili chiavi del cipher.

La protezione da questo tipo di attacco si ottiene aumentando la lunghezza del **numero di bit della chiave** e conseguentemente lo spazio delle possibili chiavi da testare, in modo che il costo per forzare il cipher ecceda il valore del dato cifrato.

Cipher Simmetrici

OpenSSL supporta tutti i principali Cipher Simmetrici, tra cui:

des: (Data Encryption Standard) http://it.wikipedia.org/wiki/Data_Encryption_Standard
sviluppato dall'IBM e adottato dal governo USA nel 1977. Lavora **su blocchi** del messaggio di 64 bit, con chiave di 64 bit di cui 8 sono di parità dispari, pertanto la chiave è di 56 bit.

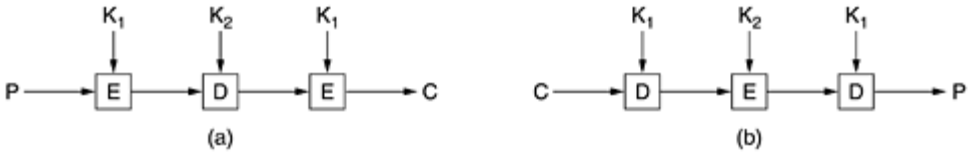


(a) struttura generale (b) dettaglio di una iterazione

La lunghezza della chiave è troppo breve e l'algoritmo può essere forzato in poche ore.

Cipher Simmetrici

TripleDES: E' l'algoritmo DES applicato 3 volte. Può utilizzare 1, 2 o 3 chiavi DES. Esempio con 2 chiavi EDE (K1 e K2, totale 112 bit):



AES: Nel 1997 il NIST (National Institute of Standard and Technology) promosse un concorso per la realizzazione di un nuovo Cipher Simmetrico che sarebbe diventato uno standard del Governo USA con il nome di AES (Advanced Encryption Standard).

L'algoritmo vincitore fu Rijndael (V. Rijmen e J. Daemen).

L'algoritmo supporta chiavi da 128, 192 o 256 bit su blocchi da 128 bit.

RC4: (Rivest 1987) è un algoritmo utilizzato ampiamente in protocolli quali il WEP (WiFi) e SSL.

Comandi OpenSSL: enc

Questo comando serve per cifrare/decifrare utilizzando uno tra i cipher simmetrici supportati che sono circa 50, tra cui

-des -des3 (3 chiavi EDE) -aes128 -aes192 -aes256 -rc4

Vedi la pagina Wiki: https://wiki.openssl.org/index.php/Enc

Esempio: DES3

```
$ openssl enc -e -a -des3 -in myfile      -out myfile.des3
$ openssl enc -d -a -des3 -in myfile.des3 -out myfile
( -a    determina l'output codificato BASE64 )
```

Message Digest

Il **Message Digest** (MD)

è una sequenza di bit di lunghezza limitata e fissa associata ad un messaggio (P) e ne rappresenta la “firma” (o “impronta”).

Il MD **non è invertibile,** ovvero non è possibile risalire al messaggio originale.

Se il massaggio cambia anche di un solo bit il MD diventa completamente diverso.

Il MD viene calcolato applicando al messaggio una **funzione di Hash: MD=H(P)**

L'algoritmo deve essere “**Collision Free**” , ovvero deve evitare (o minimizzare) la possibilità che 2 messaggi generino lo stesso MD. Per questo il MD non può essere troppo breve.

Applicazioni Principali:

- Verificare l'integrità di messaggi o file. Il messaggio P viene spedito assieme al MD; chi li riceve ricalcola i MD e lo compara con quello ricevuto.

- Verifica della password. Viene memorizzato il MD della password in chiaro MD=H(clear_passw). Per verificare la password bisogna confrontare H(proposed_clear_passw) con MD.

Comandi OpenSSL: dgst

Principali Digest:

MD5 è un algoritmo di Hashing a 128 bit realizzato da R. Rivest nel 1991 (RFC1321).

RIPEMD è una famiglia di algoritmi sviluppati come alternativa Europea a MD5
Il più importante e' **ripemd160** http://en.wikipedia.org/wiki/RIPEMD

SHA (Sechure Hash Algorithm) indica una famiglia di 5 diverse funzioni sviluppate da NSA come standard Federale del governo USA:

- sha1 (160 bit)
- sha2 (**sha224** (224 bit) – **sha256** (256bit) – **sha384** (384 bit) **- sha512** (512bit))

Il comando **dgst** serve ad applicare una tra le funzioni di Hash supportate (man dgst):

Esempi:

```
openssl dgst -md5 -hex myfile  (-md5 default, vedi anche md5sum)
openssl sha1 -hex myfile
```

HMAC

Se 2 parti A e B condividono una chiave simmetrica possono autenticare i messaggi che si scambiano utilizzando lo schema HMAC (Hashed Message Authentication Code) che è una funzione di Hash applicata al Messaggio e alla Chiave condivisa:

Firma= HMAC (K,M)

- Il mittente che deve inviare M calcola HMAC(K,M) che invia assieme ad M
- Il destinatario riceve M e HMAC(K,M), quindi verifica la firma ricalcolando l'HMAC

Questo schema è utilizzato in diversi protocolli crittografici, tra cui IPsec.

```
echo "foo" | openssl dgst  -sha256 -hmac 123
#-hmac key : create a hashed MAC using “key”.
```

Password

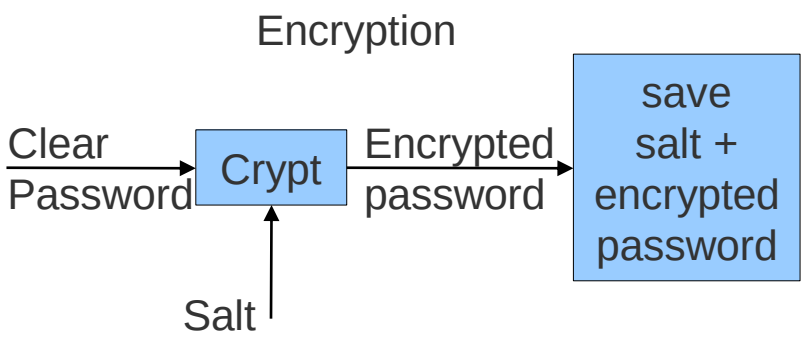
La password è una sequenza alfanumerica che l'utente deve inserire per accedere ad una risorsa protetta. E' quindi un segreto condiviso tra utente e risorsa, che normalmente la risorsa memorizza in formato cifrato.

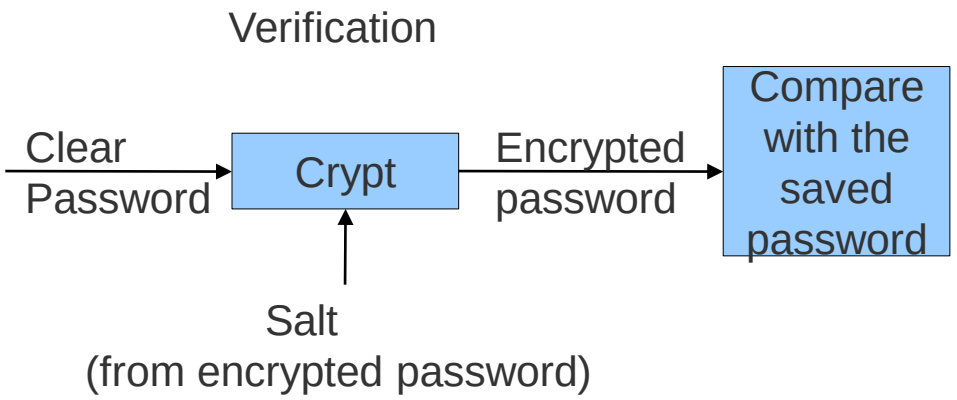
La cifratura non e' invertibile, quindi per la verifica occorre cifrare la password da verificare e confrontarla con la password cifrata memorizzata dalla risorsa.

La cifratura avviene utilizzando cipher simmetrici, o funzioni di Hash.

Spesso si utilizza usa stinga casuale, detta «salt», per complicare gli attacchi al dizionario.

Il comando **passwd** per generare le password Unix/Linux si basa sulla funzione **crypt**().





Password con crypt()

crypt() in glibc:

Crypt itera 25 volte l'algoritmo DES per cifrare un messaggio costante (tipicamente una sequenza di 0) utilizzando una chiave simmetrica derivata dalla password in chiaro inserita dall'utente + una stringa casuale noti di 12 bit “salt”. Il salt viene scritto in chiaro all'inizio della password cifrata con codifica base64 (2 caratteri).

La modifica di DES (25 iterazioni + salt) rende la cifratura non invertibile.

La verifica della password consiste nel confronto tra i messaggi codificati:

crypt viene ripetuto con la password da verificare e il salt prelevato dalla password cifrata.

Il fatto che la cifratura non è invertibile consente di utilizzare gli algoritmi di Hashing come tecniche alternative di codifica, implementate in glibc2.

crypt in glibc2: aggiunge la possibilità di cifrare con MD5.

L'hash MD5 è di 128 bit

L'output è una stringa composta al più da 34 byte di cui:

-la prima parte è il salt: 1<max 8 char>$

-la seconda parte è una sequenza di 22 caratteri (128 bit di MD5 / 6 di BASE64) contenenti MD5(<salt><clear_password>)

Comandi OpenSSL: passwd

Con openssl è possibile generare le password utilizzando il crypt di glibc (default) oppure un Message Digest:
MD5 con l'opzione -1, SHA-256 opzione -5, SHA-512 opzione -6

Esempi:

```
$ openssl passwd mysecret
QvpTKPjqpBD9.
(i primi 2 caratteri Qv sono il salt generato random e utilizzato
   assieme a mysecret per derivare la chiave di cifratura)

$ openssl passwd -salt Qv mysecret (riproduce la stessa cifratura)

$ openssl passwd -1 mysecret
$1$NvOCDeMO$5keqaA/5i/O7kpEXArm0L/
( NvOCDeMO è il salt casuale, le restanti 22 cifre BASE64 sono l'hash
   MD5 di 128 bit)
```

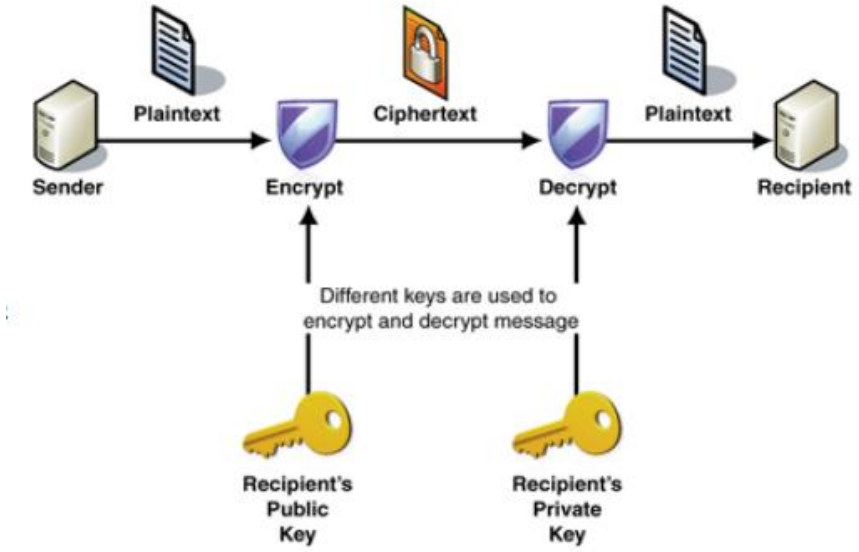
Le password di Apache:
https://httpd.apache.org/docs/2.4/misc/password_encryptions.html

Algoritmi a chiave pubblica

La cifratura a chiave simmetrica ha una grave debolezza nella condivisione della chiave: il trasferimento della chiave la espone ad intercettazione.

Diffie ed Hellmann (Stanford) nel 1976 proposero una tecnica nuova di crittografia, basata sull'aritmetica modulare, in cui vengono utilizzate due chiavi K_e e K_d distinte per la codifica $C=K_e(P)$ la decodifica $P= K_d(C)$.

Assegnando ad una chiave il ruolo di “chiave privata” e all'altra il ruolo di “chiave pubblica” si supera la debolezza della chiave condivisa.



Algoritmi a chiave pubblica

Una importante proprietà di questo algoritmo è: $K_d(K_e (P)) = K_e(K_d (P)) = P$

Questo consente di poter applicare le chiave in 2 modi diversi ottenendo 2 diverse funzioni:

1) Applicando prima la chiave pubblica si ottiene la **Privacy (crypt/decrypt).**

A deve inviare un messaggio P riservato a B su di un canale insicuro.

B possiede una coppia di chiave asimmetriche B_e (pubblica) e B_d (privata).

A cifra P con la chiave pubblica di B: $C=B_e(P)$. Solo B può decifrarlo $P= B_d(C)$

2) Applicando prima la chiave privata si ottiene **l'Autenticazione e Integrità (sign/verify)**

A deve inviare un messaggio P attraverso un canale insicuro. Tutti lo possono leggere, ma chi lo riceve deve essere sicuro che è stato inviato da A.

A possiede una coppia di chiave asimmetriche A_e (pubblica) e A_d (privata).

A cifra P con la propria chiave privata: $C= A_d (P)$. Chiunque può applicare la chiave pubblica di A: $P=A_e(C)$. La decifratura funziona solo se P è stato cifrato da A.

RSA

Rivest, Shamir e Adleman del MIT implementarono nel 1978 l'algoritmo che ha preso il loro nome (RSA) e che è attualmente il più utilizzato nelle applicazioni crittografiche a chiave pubblica.

1) si scelgono 2 numeri primi casuali p e q, sufficientemente grandi.
2) si calcolano m=pq (chiamato modulo) z=(p-1)(q-1)
3) si sceglie un numero e<(p-1)(q-1) coprimo con (p-1)(q-1) #(senza divisori comuni)
4) si calcola un numero d tale che e*d = 1 mod z (il resto della divisione (e*d) /z deve essere 1)

$C = P^e \bmod m$ (e,m) è la chiave pubblica

$P = C^d \bmod m$ (d,m) è la chiave privata

Le 2 chiavi hanno una parte comune **m** (M**odulo**) che è tipicamente di 1024 o 2048 bit e una parte specifica **e, d (Esponenti)** di circa 20 bit.

Per violare la chiave privata occorre determinare **d**. Questo può essere fatto solo per "forza bruta" fattorizzando **m** che è il prodotto di 2 numeri primi.

L' operazione richiederebbe un tempo enormemente grande anche sul più veloce dei computer.

La cifratura $C = P^e \bmod m$ limita la dimensione massima di P (deve essere $P < m$)

Comandi OpenSSL: RSA

OpenSSL 1.x supporta RSA con i seguenti comandi: genrsa, rsa e rsautl.
Nota: In openSSL 3.x i comandi sono genpkey, pkey e pkeyutl

genrsa (man genrsa) consente di creare una coppia di chiavi RSA:

```
openssl genrsa -out rsa_key.pem 2048
# aggiungere -des3 per cifrare la chiave privata con un pass-phrase
```

rsa (man rsa) consente di processare le chiavi rsa. Esempi:

`openssl rsa -in rsa_key.pem -text` (mostra il contenuto)

`openssl rsa -in rsa_key.pem -pubout -out rsa_pub.pem` (estrae la chiave pubblica)

rsautl viene utilizzato per cifrare/decifrare firmare/verificare con chiavi RSA.

Esempio encrypt/dectypt:

```
openssl rsautl -encrypt -pubin -inkey rsa_pub.pem -in text.txt     -out
   encrypted.txt
openssl rsautl -decrypt        -inkey rsa_key.pem -in encrypted.txt -out text.txt
```

Esempio sign/verify:

```
openssl rsautl -sign           -inkey rsa_key.pem -in text.txt    -out signed.txt
openssl rsautl -verify -pubin  -inkey rsa_pub.pem -in signed.txt  -out text.txt
```

Comandi OpenSSL: DIGEST firmato con RSA

Le chiavi RSA possono essere utilizzate per firmare il Digest di un messaggio:

Il seguente comando crea il digest di file.txt utilizzando SHA1, quindi lo firma con la chiave privata:

```
openssl dgst -sha1 -sign rsa_key.pem -out file.sha1_sign   file.txt
```

Per la verifica occorre il messaggio originale, il digest firmato e la chiave pubblica di chi ha firmato:

```
openssl dgst -sha1 -verify rsa_pub.pem -signature file.sha1_sign file.txt
> Verified OK
```

Algoritmi a chiave pubblica: Certificazione dell'identità

Se io genero una coppia di chiavi, uso la chiave privata per cifrare un messaggio e pubblico il messaggio cifrato, chiunque può verificare con la mia chiave pubblica che il messaggio l'ho cifrato io, ma questo non garantisce nulla riguardo la mia identità.

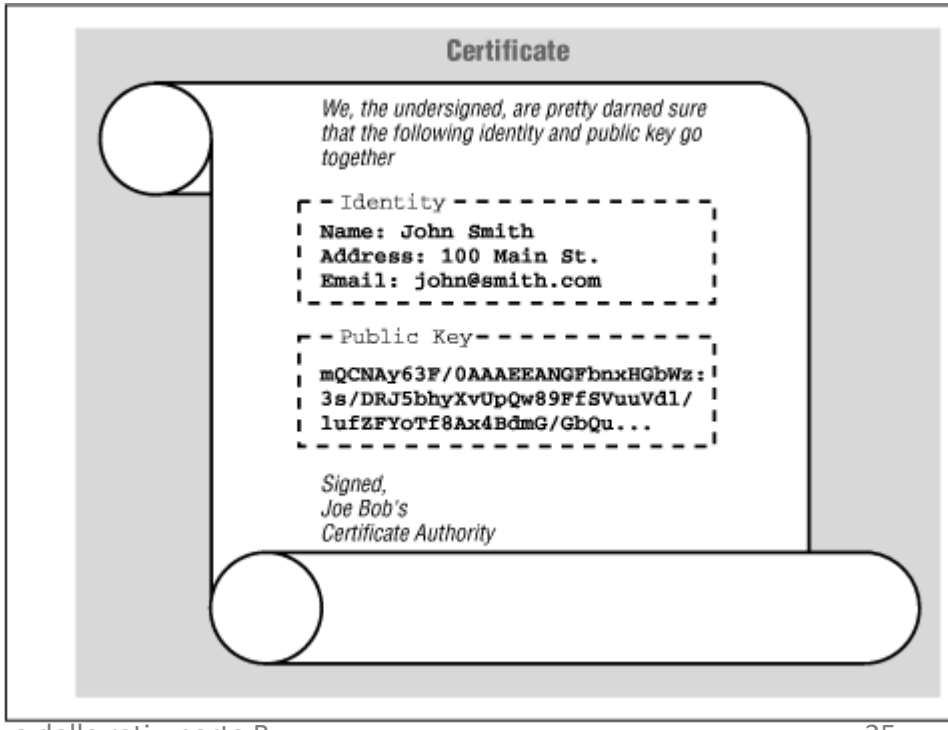
Per associare in modo certo una chiave pubblica a una persona (o host o software) si utilizza il Certificato, ovvero l'insieme della chiave pubblica e dei dati del proprietario firmati (cifrati con la chiave privata) da una **Autorità di Certificazione** che garantisce l'autenticità dei dati contenuti nel **Certificato.**

Applicazioni principali**:**

Posta elettronica (identità del mittente)

Connessioni Web (identità del server e del client)

Software (identità dello sviluppatore)



Certificati X.509

X.509 è lo standard Internazionale emanato da ITU per il formato dei Certificati.

La versione v.3 di X.509 è utilizzata da SSL/TLS.

Questo standard stabilisce quali informazioni possono comporre un certificato; i principali campi sono:

Version: numero della versione di X.509 (v1, v2 o v3)

Serial Number: numero univoco di emissione da parte della CA

Signature Algorithm: Algoritmo usato per firmare il certificato

Issuer: Distinguished Name DN della CA che ha emesso il certificato

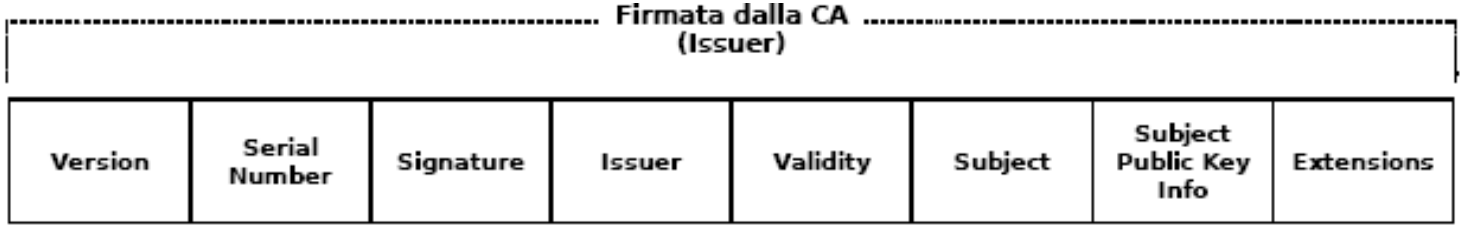
Validity: Inizio e Fine del periodo di Validità.

Subject: Distinguished Name DN del proprietario del Certificato.

Subject Public Key Info : Chiave pubblica (Modulo + Esponente) e algoritmo utilizzato

X509v3 Extensions: Estensioni opzionali (solo v3).

Signature: Firma da parte della CA (MD del Certificato, cifrato con la chiave privata della CA).



La Certification Authority

La CA è un ente che firma le richieste di certificato da parte di una comunità di utenti/host/software garantendone l'identità.

La CA possiede una propria coppia di chiavi e autofirma la propria richiesta (self-signed).

Per identificare univocamente i certificati esiste un name-space gerarchico di certificati, in cui ogni nodo ha un attributo e un valore.

I principali attributi sono: O (Organization), OU (Org. Unit), C (Country), CN (CommonName) , ecc

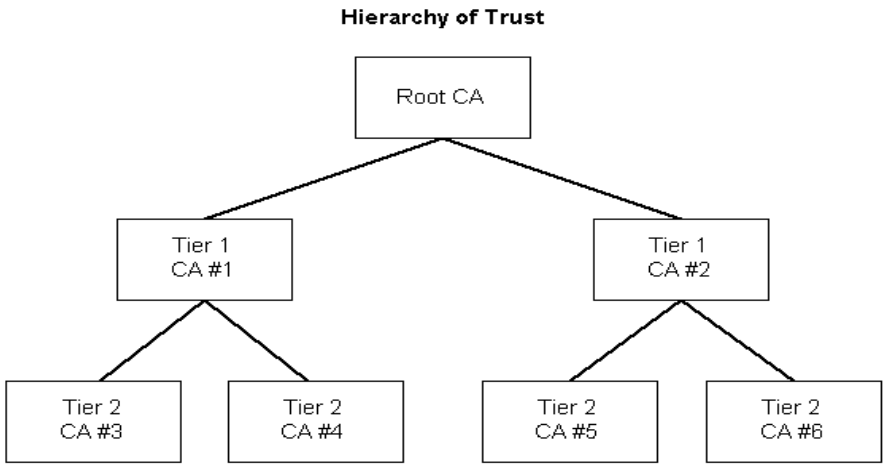
Ogni certificato possiede quindi un FQDN (Subject). Esempio:

C=IT, O=UniprScienze, OU=Staff, CN=roberto alfieri/emailAddress=roberto.alfieri@unipr.it

La CA possiede un BaseDN ed è vincolata ad emettere Certificati all'interno del suo BaseDN.

Una CA può firmare certificati anche per altre CA, poiché il namespace gerarchico consente di organizzare diverse CA all'interno dello stesso albero.

Ogni CA deve gestire la lista dei Certificati Revocati (CRL) che va aggiornata regolarmente.



Istanze di Certification Authority

Alcune CA rilasciano certificati a pagamento e sono già inserite e riconosciute dai più diffusi client Web e SMTP, come ad esempio Comodo, Entrust, GeoTrust, GlobalSign, Cybertrust, Verisign e DigiCert (vedi le impostazioni del Web Browser).

Generalmente una Organizzazione/Ente/Impresa può creare una CA per le certificazioni riconosciute al suo interno

Il GARR fornisce un Certification Service per tutti gli enti afferenti: https://www.servizi.garr.it/cs

GARR partecipa al Trusted Certificate Service (TCS) promosso da Géant a favore delle reti della ricerca europee.

Tramite questo servizio offre gratuitamente alla propria comunità certificati digitali x.509 emessi da una delle maggiori Certification Authority commerciali: Sectigo Limited, riconosciuta automaticamente dalla quasi totalità dei browser web esistenti.

Comandi OpenSSL: ca

OpenSSL consente di creare una propria Certification Authority:

Questo comando genera chiave e certificato autofirmato:

```
openssl req -config openssl.cnf -newkey rsa:512 -days 1000 –nodes   \
                -keyout cakey.pem  -out cacert.pem -x509 -new
```

Con il comando **ca** vengono gestite le operazioni della Certification Authority.
Esempi:

- Firma di una richiesta di Certificato:

```
  openssl ca -in req.pem -out newcert.pem
                        -config /var/www/html/myCA/openssl.cnf
```

- Generazione della Certificate Revocation List:

```
     openssl ca   -gencrl   -out crl.pem
```

Esempio di Certificato della CA (MyCA)

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 0 (0x0)
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=IT, O=UniprScienze, OU=UniprScienzeCA, CN=UniprScienze/emailAddress=roberto.alfieri@fis.unipr.it
        Validity
            Not Before: Jun  4 20:21:12 2010 GMT
            Not After : Jun  3 20:21:12 2015 GMT
        Subject: C=IT, O=UniprScienze, OU=UniprScienzeCA, CN=UniprScienze/emailAddress=roberto.alfieri@fis.unipr.it
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (512 bit)
                Modulus (512 bit):
                    00:cc:89:ba:73:31:b2:b3:e8:74:d9:30:b8:93:02:
                    51:b1:12:8f:f5:e7:f9:2f:96:68:15:e6:4a:19:d8:
                    66:6a:e9:74:66:3e:9f:6f:02:25:ef:3e:5f:09:c3:
                    63:70:e7:40:63:53:a8:75:3c:b7:a8:cb:68:de:4e:
                    dd:c2:89:c9:6d
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                7A:4A:AE:FA:83:9D:C8:DC:E7:74:0A:B9:17:3A:CB:38:3A:31:CA:A7
            X509v3 Authority Key Identifier:
                keyid:7A:4A:AE:FA:83:9D:C8:DC:E7:74:0A:B9:17:3A:CB:38:3A:31:CA:A7
```

Comandi OpenSSL: req

Questo comando serve per generare una richiesta di certificato.

La chiave privata può essere fornita (-key) o può essere generata dal comando.

Con l'opzione -nodes non viene cifrata la chiave privata. Questo serve per i certificati host per i quali non è possibile digitare una Passphrase.

Il file di configurazione (openssl.cnf) contiene le informazioni relative al proprietario del certificato, alcune delle quali vengono richieste interattivamente.

Esempi:

Generazione della coppia di chiavi (chiave privata e richiesta di certificato):

```
openssl req -new -nodes -out hostreq.pem -keyout hostkey.pem -config
  openssl.cnf
```

Verifica il contenuto della richiesta:

```
openssl req -in req.pem -text
```

Certificati self-signed

Con l'opzione -x509 non viene generata una richiesta, ma un certificato self-signed

Se il file di configurazione (-config) non viene fornito i dati del certificato vengono richiesti interattivamente.

Generazione del certificato self-signed per un server:

```
openssl req -new -nodes -out hostcert.pem -keyout hostkey.pem -x509
   Generating a 1024 bit RSA private key  ......
   writing new private key to 'keyfile.pem'
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
Country Name (2 letter code) [GB]: IT
State or Province Name (full name) [Berkshire]: Parma
Locality Name (eg, city) [Newbury]: Parma
Organization Name (eg, company) [My Company Ltd]: UNIPR
Organizational Unit Name (eg, section) []: LPR
Common Name (eg, your name or your server's hostname) []: lpr1.fis.unipr.it
Email Address []: roberto.alfieri@unipr.it
```

Formati dei certificati X.509 : DER e PEM

I certificati X.509 possono essere rappresentati in diversi possibili formati.
I principali sono: DER, PEM e PKCS12.

DER è un formato binario utilizzato in ambiente Windows e Java con estensioni .DER o .CER.

PEM è testuale (base64) ed è utilizzato prevalentemente in ambiente Unix.
Può contenere Certificati, richieste di Certificati, chiavi private.
Gli oggetti contenuti sono delimitati da stringhe del tipo:

----- BEGIN CERTIFICATE --- ---END CERTIFICATE ----
---BEGIN RSA PRIVATE KEY--- ---END RSA PRIVATE KEY ---
-----BEGIN CERTIFICATE REQUEST--- ---END CERTIFICATE REQUEST--

Comandi OpenSSL: x509

Visualizzare i campi e cambiare formato (tra PEM e DER) di un certificato:

Esempi:

```
openssl x509 -in cert.pem -noout -text
openssl x509 -in cert.pem -noout -serial
openssl x509 -in cert.pem -noout -subject
openssl x509 -in cert.pem -inform PEM -out cert.der -outform DER
```

https://www.openssl.org/docs/man1.1.1/man1/x509.html

Formati dei certificati X.509 : PKCS

PKCS (Public Key Cryptography Standards) è un gruppo si Standard creati da “RSA Data Security” con lo scopo di creare formati di interoperabilità.

Sono stati creati 15 standard (da PKCS#1 a PKCS#15) ma i formati prevalentemente utilizzati sono:

PKCS#12: Personal Information Exchange Syntax Standard.

Nato come evoluzione di PFX, definisce un formato per immagazzinare chiave privata e certificato in un file protetto con password (cifratura con chiave simmetrica). L'estensione è p12.

PKCS#7: Cryptographic Message Syntax Information.

E' un formato per rappresentare messaggi cifrati o firmati ed è utilizzato da S/MIME per l'invio di e-mail cifrate e/o firmate. Recepito da IETF (RFC 2986).

PKCS#11: Cryptographic Token Interface

Definisce le API per utilizzare i Token Crittografici, come le SmartCard e le chiavi USB

Comandi OpenSSL: pkcs12

Questo comando serve per gestire i file in formato PKCS12.
https://www.openssl.org/docs/manmaster/man1/openssl-pkcs12.html

Esempi:
Genera il file PKCS12 dai file PEM:

```
openssl pkcs12 -export -out cert.p12 -inkey userkey.pem -in usercert.pem
```

Dal formato PKCS12 a PEM (separatamente chiave e certificato):

```
#estrai la chiave da cert.p12 (aggiungere -nodes per i server)
openssl pkcs12 -nocerts -in cert.p12 -out userkey.pem
#estrai il certificato da cert.p12
openssl pkcs12 -clcerts -nokeys -in cert.p12 -out usercert.pem
#visualizza il certificato in chiaro
openssl x509 -in usercert.pem -text
```

Dal formato PEM a PKCS12 (con cypher AES256-CBC):

```
openssl pkcs12 -export -inkey userkey.pem -in usercert.pem -out test.p12
   -certpbe AES-256-CBC -keypbe AES-256-CBC
```

Comandi OpenSSL: smime

MIME (Multipurpose Internet Mail Extensions)

è una estensione del protocollo di posta Elettronica per poter includere in un unico messaggio più documenti (Allegati), codificati in ASCII Standard (vedi la Posta Elettronica).

MIME introduce un header in cui è possibile inserire campi che descrivono il contenuto, tra cui:

Content-Type : Tipo di dato contenuto (esempio image/JPEG)

Content-transfer-encoding: Codifica del contenuto (esempio BASE64)

S/MIME (Secure/MIME)

include in questo schema la possibilità di

cifrare/decifrare il contenuto, utilizzando un algoritmo a chiave simmetrica.

La chiave simmetrica viene cifrata con la chiave pubblica del destinatario e inviata insieme al messaggio stesso.

(Content-Type del messaggio cifrato: application/x-pkcs7-mime)

firmare/verificare un messaggio allegando la Firma

(Content-Type della firma: application/x-pkcs7-signature)

Consultare le opzioni: man smime

https://www.openssl.org/docs/manmaster/man1/openssl-smime.html

Cifrare/Decifrare con smime

Per inviare un messaggio cifrato al destinatario la chiave simmetrica viene cifrata con la chiave pubblica del destinatario e inviata assieme al messaggio stesso. Il destinatario legge la chiave simmetrica utilizzando la propria chiave privata, quindi decripta il messaggio con la chiave simmetrica.

Il Mittente codifica il messaggio con seguente comando

```
openssl smime -encrypt -text -in message.txt  \
          -out encrypted-message.txt  destination-user-certificate.pem
```

viene generato un file Mime con una intestazione del tipo:

MIME-Version: 1.0

Content-Type: application/x-pkcs7-mime; name="smime.p7m"

Content-Transfer-Encoding: base64

Il Destinatario lo decodifica con il seguente comando:

```
openssl  smime  -decrypt -text  -in encrypted-message.txt  \
                    -out decrypted-message.txt  -inkey userkey.pem
```

Firmare/Verificare con smime

La firma consiste nel cifrare con la chiave privata del mittente il Digest del Messaggio, che verrà inviato insieme al messaggio stesso.

- **Garanzia dell'identità del Signer**: Consiste nella decifratura della Firma con la chiave pubblica del Signer.
- **Garanzia di Integrità del Messaggio**: Il messaggio è integro se il Digest decifrato nella firma e il Digest ricalcolato sono uguali.

Per firmare un messaggio:

```
openssl smime -sign -text -in message.txt  -out signed-message.txt \
                          -signer usercert.pem  -inkey userkey.pem
```

Il certificato del Signer è incluso nel messaggio.

Con questo comando il Destinatario estrae il Certificato del Signer:

```
openssl smime -pk7out -in signed-message.txt | openssl pkcs7 -print_certs
```

Con questo comando il destinatario verifica il certificato del Signer tra le CA di cui si fida (-CAfile o -CApath) , quindi usa il Certificato per decifrare la firma ed estrarre il MD, infine confronta il MD ricevuto con quello calcolato.

```
openssl  smime  -verify  -text -in signed-message.txt  -CAfile CAcert.pem
```

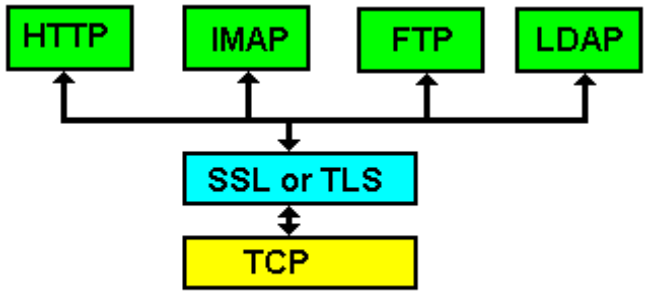
Crittografia nella comunicazione

Gli algoritmi crittografici simmetrici e a chiave pubblica vengono utilizzati per costruire protocolli di rete con l'obiettivo di cifrare la comunicazione, fornendo diversi servizi di sicurezza quali la **Confidenzialità, l'Autenticazione e il Non Ripudio.**

Il principale protocollo di questo tipo è **SSL** (Secure Socket Layer), poi diventato **TLS** (Transport Security Layer).

SSL /TLS è strutturato come un layer che si pone tra il trasporto (TCP) e l'applicazione.

Molte applicazioni di rete (ad esempio HTTP, IMAP, LDAP) sono state adattate per appoggiarsi su SSL (anziché TCP) per usufruire dei servizi di sicurezza del protocollo.



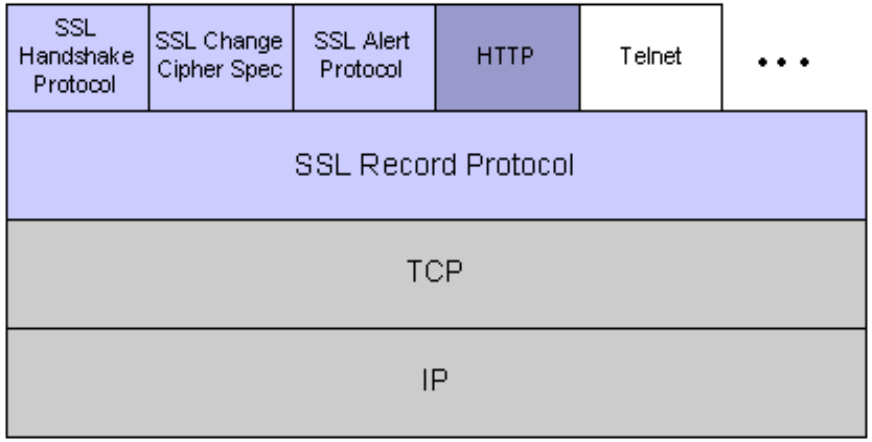
SSL/TLS

Il protocollo SSL mette in sicurezza una connessione Client-Server introducendo un **Layer di Sicurezza al livello di Trasporto (TLS)**.

In questo modo l'applicativo non vede più le API di TCP e deve quindi riscrivere l'interfaccia con il livello di Trasporto utilizzando le API SSL/TLS.

Vengono utilizzati 2 protocolli principali:

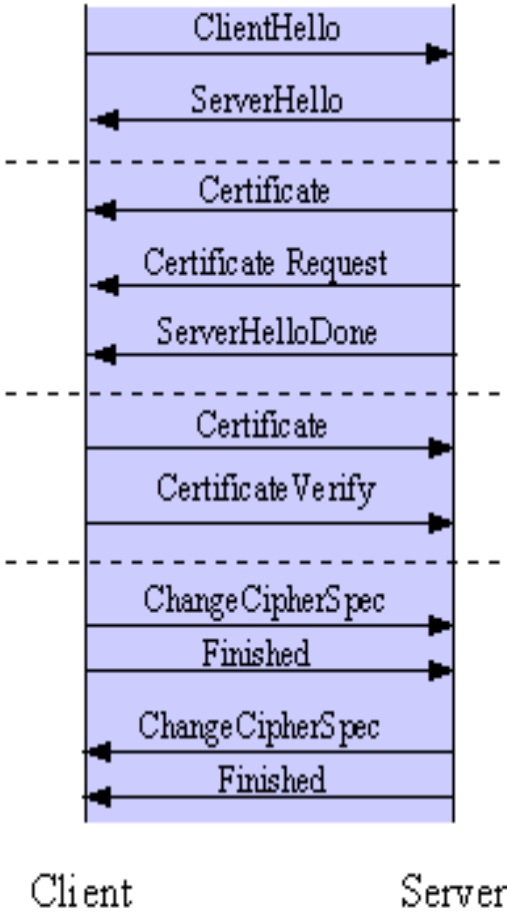
- Il Record Layer Protocol è utilizzato per trasmettere e ricevere dati (cifratura e decifratura) mediante un Cipher Simmetrico.
- L'Handshake Protocol interviene solo all'inizio (o per ripristinare una sessione interrotta). Negozia i parametri per lo scambio dei dati (scambio dei Certificati e condivisione di una chiave di Sessione per il Cipher Simmetrico).



SSL Handshake Protocol

1. Il client manda SSL version e il Cipher Suite da utilizzare
2. Il server manda il proprio SSL versione e il cipher suite
3. Il server manda il proprio certificato
4. Il server opzionalmente richiede il certificato del client
5. Il client verifica il certificato del server e procede solo se è ok
6. Il client genera il pre-master secret
7. Il client cripta la chiave con il cert. del server e la invia
8. Se il sever ha richiesto l'autenticazione del client,
   Il client cifra un challenge e lo invia al server
9. Se il server non riesce a decifrare termina la sessione.
10. Client e server usano il pre-master secret per
    Generare la chiave di sessione condivisa.
11. Il client manda un messaggio di conclusione dell'handshake
12. Il server manda un messaggio di conclusione dell'handshake

In qualsiasi momento entrambi possono **Renegoziare** la connessione, nel qual caso la procedura è ripetuta.

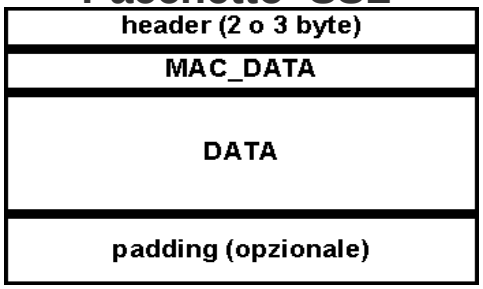


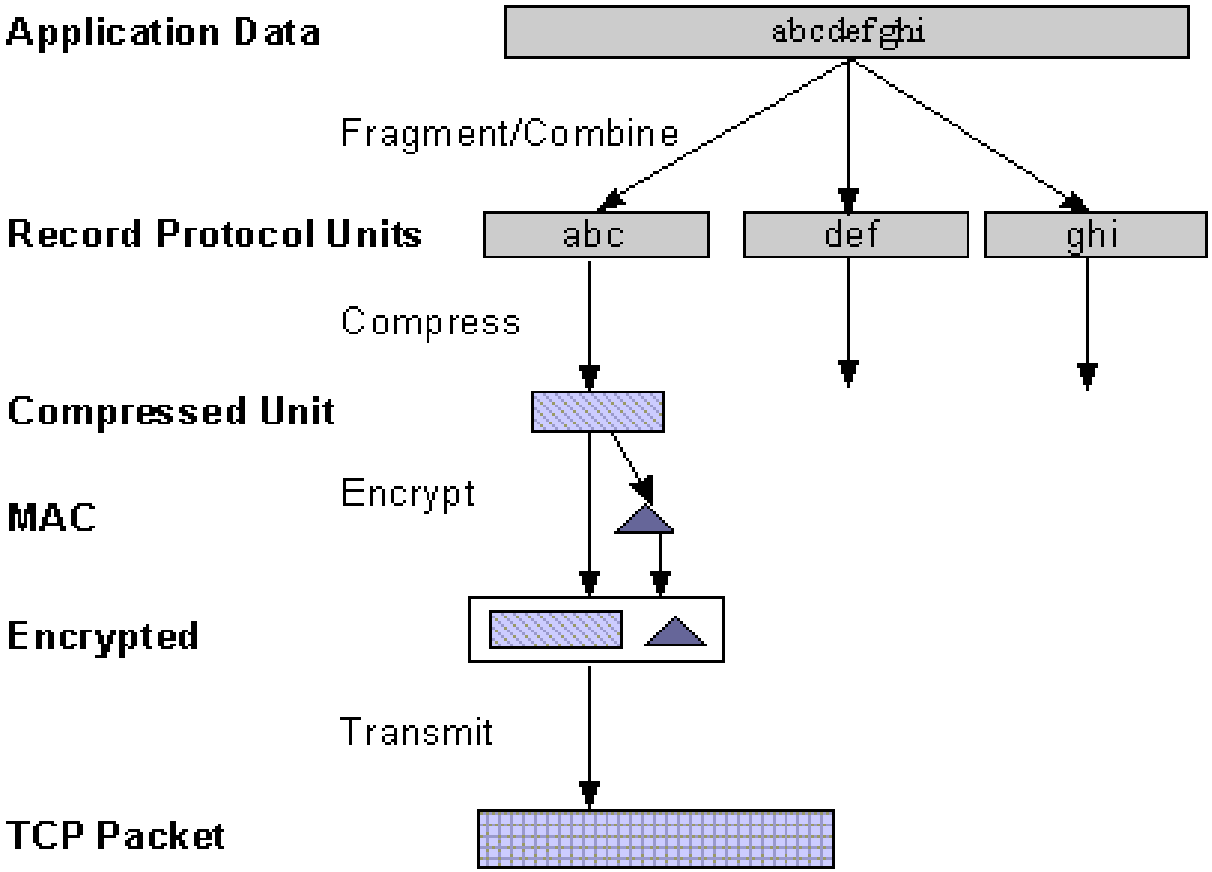
SSL Record Protocol

Al termine dell'Handshake Protocol inizia l'SSL Record Protocol:

1. Frammentazione del flusso
2. Compressione dei frammenti
3. Calcolo del Digest
4. Cifratura

Pacchetto_SSL





SSL Cipher suites

Una sessione SSL/TLS richiede l'utilizzo di un set di strumenti crittografici per

- Scambio chiavi (Kx)
- Autenticazione (Au)
- Cifratura del canale (Enc)
- Digest (Mac)

Un cipher suite è un insieme di strumenti che implementano le funzioni necessarie.
https://www.ssl.com/it/guida/tls-conformit%C3%A0-agli-standard/

Il seguente comando elenca le cipher suite supportate in openssl:

```
openssl ciphers -v
```

Applicazioni SSL e StartTLS

Alcune versioni di applicazioni comuni incorporano SSL/TLS attivando connessioni cifrate in ascolto su porte diverse e che possono quindi coesistere con le versioni non cifrate. Le principali applicazioni SSL sono:

- **https** (HTTP over SSL) 443/tcp
- **pop3s** (POP3 over SSL) 995/tcp
- **imaps** (IMAP over SSL) 993/tcp
- **smtps** (SMTP over SSL) 465/tcp
- **ldaps** (LDAP over SSL) 636/tcp

STARTTLS è una estensione della comunicazione in chiaro, che offre l'opzione di passare ad una connessione cifrata (SSL o TLS) anziché passare ad una connessione cifrata separata.

IMAP, SMTP , POP e LDAP hanno implementazioni con supporto StartTLS.

Comandi OpenSSL: s_client s_server

s_client (info: man s_client) è un client in grado di attivare una connessione con un server ssl/tls. Visualizza i dati del protocollo ssl handshake (protocollo tls utilizzato, cipher, la chiave di sessione, il certificato del server, subject, ecc)

Esempio: **`openssl s_client -connect www.unipr.it:443`**

Il carattere R forza la rinegoziazione, Q forza la disconnessione

Una volta connessi è possibile digitare manualmente i comandi da inviare al server (ad esempio, se il server è www : "GET / HTTP/1.0")

s_server (info: man s_server) svolge le stesse funzioni lato server. Esempio:

`openssl s_server -accept 443 -www -cert hostcert.pem -key hostkey.pem`

Con l'opzione www viene emulato un web server che risponde al browser con i dati salienti della connessione SSL.

Programmazione openSSL

La libreria **SSL** fornisce le API necessarie per la programmazione in C di canali cifrati.

https://developer.ibm.com/tutorials/l-openssl/

La cifratura avviene grazie all'astrazione di I/O, denominata BIO (Basic I/O abstraction), che consente di creare un canale cifrato o in chiaro

https://www.openssl.org/docs/manmaster/man7/bio.html

Esempio in C:

TLS_client.c https://wiki.openssl.org/index.php/SSL/TLS_Client

Simple_TLS_Server.c https://wiki.openssl.org/index.php/Simple_TLS_Server

Python consente di creare connessioni SSL/TLS, sia client che server, attraverso diverse librerie che implementano un wrapper di TCP basato su openSSL. Le principali sono:

- **TLS/SSL wrapper for socket objects** https://docs.python.org/2/library/ssl.html
- **pyOpenSSL** https://pyopenssl.org/en/stable/

Il modulo httplib supporta sia HTTP, con httplib.HTTPConnection(), che HTTPS, grazie al metodo httplib.HTTPSConnection()