ISTITUTO NAZIONALE DI FISICA NUCLEARE

Sezione di Firenze

INFN/TC-02/14
23 Maggio 2002

OpenSSL API MINI–TUTORIAL

Riccardo Veraldi
INFN, Sezione di Firenze

Abstract

Dopo una breve descrizione del protocollo SSL vengono presentate le principali API in C della libreria OpenSSL, toolkit Open Source che implementa SSL v2/v3 e TLS v1. Viene anche analizzato un esempio di semplice applicazione client/server da cui potere iniziare per sviluppare proprie applicazioni SSL.

PACS: 07.05.Bx, 89.20.Ff

Pubblicata dal ***SIS–Pubblicazioni***
Laboratori Nazionali di Frascati

1 Cos'è SSL e TLS

SSL (Secure Socket Layer) è un protocollo di comunicazione originariamente sviluppato (versione 2.0) da Netscape Development Corporation è stato accettato universalmente all'interno del World Wide Web come protocollo per l'autenticazione e la comunicazione criptata tra client e server. La successiva versione 3.0 è stata sviluppata pubblicamente con una collaborazione estesa fra molti enti. TLS (Transport Layer Security) può essere considerata come una versione 3.1 di SSL, sviluppato dalla IETF rappresenta il tentativo di standardizzare definitivamente SSL (rfc2246 - The TLS Protocol Version 1.0 2 Agosto 1999)

2 Architettura di SSL

La TCP/IP protocol suite governa il trasporto e l'instradamento dei dati sulla rete. Altri protocolli come HTTP, LDAP o IMAP stanno al di sopra di TCP/IP nel senso che utilizzano TCP/IP come base per supportare specifici task relativi alle applicazioni come visualizzare una pagina web, un messaggio di posta elettronica etc. SSL invece risiede al di sopra di TCP/IP ma al di sotto dei protocolli di applicazione di alto livello come IMAP o HTTP, collocandosi al livello OSI 5 e 6 di Presentation e Session layer. SSL utilizza TCP/IP per conto dei protocolli di applicazione di più alto livello e permette ad un server SSL di autenticarsi nei confronti di un client SSL e vice-versa consentendo a entrambi gli host di stabilire una connessione criptata e sicura:

SSL server authentication: permette ad un utente di verificare l'identità del server. Il client SSL usa tecniche standard di crittografia a chiave pubblica per controllare che il certificato di un server sia valido e sia stato rilasciato da un'opportuna CA presente nella lista delle Trusted CA del client stesso.

SSL client authentiation: Consente ad un server di confermare l'identità

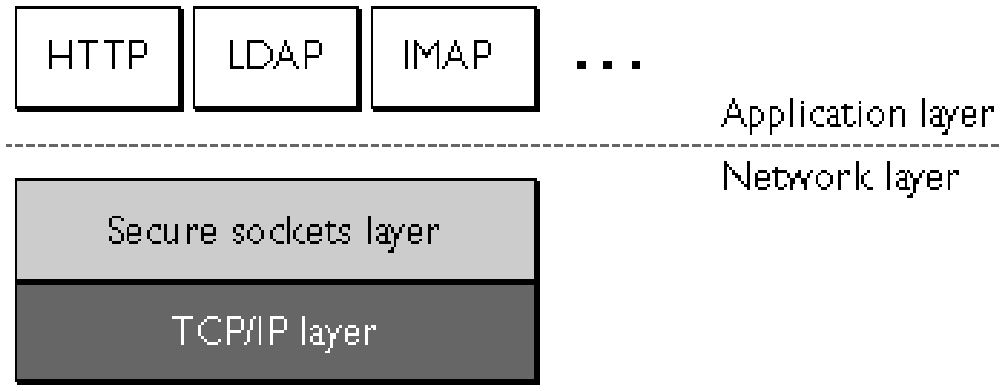

Figura 1: SSL sta al di sopra del livello di trasporto e al di sotto del livello di applicazione

di un utente. Con le stesse tecniche di crittografia a chiave pubblica il server SSL può controllare che il certificato del client sia valido e rilasciato da una Trusted CA.

Connessione SSL crittografata: richiede che le informazioni trasmesse tra server e client siano criptate dal software che invia i dati e decriptate dal software ricevente, fornendo in tal modo un alto grado di confidenzialità alla connessione. Oltre a questo tutti i dati inviati attraverso una connessione SSL criptata sono protetti da eventuali tentativi di tampering dei dati determinando pertanto se i dati possano essere stati alterati durante il transito.

Il protocollo SSL include due tipi di sotto-protocolli: *SSL record* e *SSL handshake*. *SSL record* definisce il formato utilizzato per trasmettere i dati. *SSL handshake* implica l'utilizzo di *SSL record* per scambiare una serie di messaggi tra un server SSL e un client SSL quando questi per la prima volta stabiliscono una connessione SSL. Questo scambio di messaggi comprende le seguenti azioni:

- autenticazione del server verso il client;

- server e client negoziano gli algoritmi di cifratura supportati da entrambi;
- il client si autentica nei confronti del server (opzionale);
- utilizzo della crittografia a chiave pubblica per generare shared-secrets;
- Inizio della connessione SSL criptata vera e propria.

Di seguito vi è un elenco degli algoritmi crittografici e/o di hashing utilizzati da SSL:

DES: Data Encryption Standard.

DSA: Digital Signature Algorithm.

KEA: Key Exchange Algorithm.

MD5: Message Digest algorithm sviluppato da Rivest.

RC2 e RC4: algorimi di cifratura sviluppati per RSA Data Security (Rivest).

RSA: algoritmo a chiave pubblica (Rivest, Shamir, Adleman).

RSA key exchange: key exchange: algoritmo per lo scambio delle chiavi durante la negoziazione SSL basato sull'algoritmo RSA.

SHA-1: Secure Hash Algorithm.

3DES: DES applicata 3 volte.

Gli algoritmi di key-exchange come RSA sono utilizzati per consentire al server ed al client di creare la chiave simmetrica di cifratura che sarà utilizzata da entrambi durante la sessione SSL.
Di seguito sono elencate alcune delle combinazione di algoritmi di cifratura supportati da SSL che utilizzano RSA Key-Exchange sono:

- 3DES (chiave a 168 bit) + SHA-1

- AES128 + HMAC-MD5
- RC4 (128 bit) + MD5
- RC2 (128bit) + MD5
- DES (56 bit) + SHA-1
- RC4 (40bit) + MD5
- RC2 (40 bit) + MD5
- no encryption + MD5

3 SSL Handshake

Il protocollo SSL utilizza una combinazione di sistema di criptazione a chiave privata e pubblica. Gli algoritmi di criptazione a chiave simmetrica sono più veloci degli algoritmi a chiave pubblica ma questi ultimi hanno il vantaggio di fornire un meccanismo di autenticazione efficace. Ogni sessione SSL incomincia sempre con uno scambio di messaggi chiamato SSL Handshake che permette al server di autenticarsi nei confronti del client usando la crittografia a chiave pubblica. Il passo successivo consiste nella cooperazione fra client e server per la creazione di chiavi di cifratura simmetriche utilizzate poi per criptare e decriptare i dati inviati nel canale SSL e per controllare che durante la sessione SSL le informazioni in transito non possano essere state modificate da un entità esterna a chi invia e riceve (tampering detection). Opzionalmente durante la fase di hanshake il client può autenticarsi nei confronti del Server.
Le varie fasi relative all'SSL Handshake possono essere in breve riassunte come segue:

1. Il client invia al server il numero di versione SSL supportato, i settaggi relativi agli algoritmi di cifratura supportati, e una serie di dati generati in modo random.

2. Il server invia al client il numero identificativo della versione del protocollo SSL supportato, settaggi relativi agli algoritmi di criptazione supportati, dati generati in modo random. Il server inoltre invia il proprio certificato e se il client sta richiedendo al server una risorsa che richieda autenticazione da parte del client stesso, il server richiede al client il certificato.

3. Il client procede all'autenticazione del server esaminando il certificato fornito. Se il server non può essere autenticato, allora non può essere stabilita una connessione SSL criptata, se invece l'autenticazione ha successo si passa al punto successivo.

4. Utilizzando i dati generati fino ad ora nella fase di Handshake il client crea un premaster secret per l'attuale sessione SSL, lo cripta con la chiave pubblica del server (ottenuta del certificato del server) e invia il premaster secret criptato al server.

5. Se il server ha richiesto a sua volta autenticazione da parte del client (passo opzionale) il client invia una parte di dati unici all'interno di questo handshake in atto e firma questi dati in modo digitale e invia il proprio certificato assieme al premaster secret criptato.

6. Se il server ha richiesto l'autenticazione da parte del client, il server tenta di autenticare il client stesso. Se L'autenticazione non ha sucesso la sessione viene terminata. Se l'autenticazione ha successo il server usa la propria chiave privata per decriptare il premaster secret e poi inizia una serie di operazioni (che anche il client opera partendo dal medesimo premaster secret) per generare il master secret.

7. Entrambi il client ed il server utilizzano il master secret per generare le session keys che sono chiavi simmetriche utilizzate per criptare e decriptare le informazioni che viaggiano all'interno della sessione SSL e per verificare l'integrità dei dati stessi ossia per determinare se i dati

in transito siano stati modificati in qualche modo all'interno del canale SSL.

8. Il server invia un messaggio al client informandolo che i dati futuri inviati saranno criptati con la session key stabilita precedentemente. Il server invia un messaggio criptato indicando la fine del server side handshake.

9. La fase di handshake termina e ha inizio la vera e propria sessione SSL. Il client ed il server utilizzano le session key per criptare e decriptare i dati che si inviano a vicenda e per convalidarne l'integrità.

4 SSL OpenSSL API

La libreria **OpenSSL** implementa i protocolli Secure Sockets Layer (SSL v2/v3) e Transport Layer Security (TLS v1). È un'implementazione open source e fornice un ricco sistema di API delle quali ora verranno analizzate le più comunemente utilizzate per potere scrivere codice per semplici applicazioni che supportino SSL/TLS. Per una più completa documentazione al riguardo si consiglia di visitare il sito *http://www.openssl.org* e di consultare le man pages relative ad **OpenSSL**.

D'ora in poi citando SSL ci si vuole riferire alla specifica implementazione dei Secure Socket Layer di OpenSSL e non al protocollo generico SSL quale RFC. Descriviamo brevemente quale deve essere la struttura generica di un'applicazione SSL, esaminando la situazione dal punto di vista di un software developer che voglia scrivere un programma SSL compliant. I Caratteri in **grassetto** si riferiscono a funzioni, strutture ed elementi specifici delle API OpenSSL. Verrà anche utilizzato in questa sede il termine "oggetto" quale sinonimo di "struttura" *C* senza però volere indicare alcuna relazione con elementi sintattici o paradigmi dei linguaggi ad oggetti (*C++*). Le API analizzate in questa sede si riferiscono ad una interfaccia privilegiata con il linguaggio *C*.

Ogni applicazione OpenSSL deve essere inizializzata tramite la chiamata alla funzione **SSL_library_init**(). Inoltre è necessario definire un contesto SSL (**SSL_CTX**) cioè il framework necessario per potere stabilire connessioni TLS/SSL, per fare questo si utilizza la funzione **SSL_CTX_new**(). All'interno di questo oggetto **CTX** si possono definire varie opzioni relative ai certificati, gli algoritmi da utilizzare etc. Per potere scrivere un'applicazione SSL è prima di tutto necessario creare una connessione di rete tramite l'utilizzo dei socket, che a sua volta potrà essere assegnata ad un oggetto di tipo **SSL**. Attraverso la chiamata alla funzione **SSL_new**() viene allocata e creata in memoria la struttura **SSL**. A questo punto le funzioni **SSL_set_fd**() e **SSL_set_bio**() possono essere utilizzate per associare il socket di rete alla struttura **SSL**. Successivamente, una volta portata a termine la fase di TLS/SSL handshake tramite **SSL_accept**() oppure **SSL_connect**() (a seconda del tipo di software client o server) si possono chiamare le funzioni **SSL_read**() e **SSL_write** per leggere e scrivere i dati nel tunnel TLS/SSL appena creato. La connessione SSL termina con una chiamata ad **SSL_shutdown**().

4.1 Strutture di Dati

La libreria **OpenSSL** correntemente definisce al suo interno le seguenti strutture di dati:

SSL_METHOD (SSL Method): è una struttura di dati che definisce i metodi e le funzioni interne alla libreria SSL, le quali a loro volta implementano le diverse versioni di protocollo (SSLv2, SSLv3 e TLSv1). Questa struttura è necessaria anche per potere creare un oggetto di tipo **SSL_CTX**.

SSL_CIPHER (SSL Cipher): è una struttura che contiene informazioni relative agli algoritmi di cifratura che sono parte integrante del nucleo del protocollo SSL/TLS. Gli algoritmi disponibili dipendono dal conte-

sto **SSL_CTX** e quelli utilizzati dipendono dalla sessione SSL per una determinata connessione (**SSL_SESSION**).

SSL_CTX (SSL Context): è una struttura globale che definishe il contesto SSL. È creata da un server o da un client una volta per ogni programma. Questa struttura contiene valori di default per tutte le altre strutture **ssl** che vengono poi create nel corso del programma.

SSL_SESSION (SSL Session): questa è una struttura che contiene i dettagli della sessione TLS/SSL corrente: le varie **SSL_CIPHER**, i certificati client e server, le chiavi, etc.

SSL (SSL Connection): è la struttura principale TLS/SSL che viene creata dal client o server per ogni connessione che viene stabilita. Rappresenta la struttura core di tutta l'API SSL. In fase di run-time l'applicazione ha a che fare con questa struttura che comunque è collegata a quasi tutte le altre strutture **ssl**.

4.2 Funzioni C delle API SSL

Attualmente la libreria **OpenSSL** continene 214 funzioni relative alle API. Queste possono essere divise in almeno 4 parti principali:

1. Funzioni relative ai metodi di protocollo SSL/TLS (cioè SSLv2, SSLv3, TLSv1) e che ritornano una struttura **SSL_METHOD**;

2. Funzioni relative agli algoritmi di criptazione, sono definite nella struttura **SSL_CIPHER**;

3. Funzioni relative al framework TLS/SSL che sono definite dalla struttura **SSL_CTX**, SSL protocol context.

4. Funzioni relative alla gestione della sessione SSL/TLS, sono definite dalla struttora **SSL_SESSION**.

5. Funzioni relative alla gestione della connessione SSL vera e propria in fase di RUN-TIME, sono definite dalla struttura **SSL**.

Data l'innumerevole quantità di funzioni presenti nella libreria OpenSSL analizziamo ora un po' più in dettaglio alcune delle principali funzioni **C** delle API OpenSSL, quelle più strettamente necessarie per scrivere applicazioni semplici.

4.2.1 SSL_load_error_strings()

```
#include <openssl/ssl.h>
void SSL_load_error_strings(void);
```

Carica le stringhe di errore per tutte le funzioni relative a **libcrypto** e **libssl**.

Valori di ritorno
Non ritorna alcun tipo di dato.

4.2.2 SSL_library_init()

```
#include <openssl/ssl.h>
int SSL_library_init(void);
#define OpenSSL_add_ssl_algorithms()    SSL_library_init()
#define SSLeay_add_ssl_algorithms()     SSL_library_init()
```

Inizializza la libreria SSL registrando gli algoritmi (cipher e digest). Deve essere chiamata all'interno del programma prima di ogni altra funzione SSL.

Valori di ritorno
SSL_library_init(void) ritorna sempre 1 come valore di ritorno della chiamata alla funzione.

4.2.3 SSLv3_method()

```
typedef struct ssl_method_st {
        int version;
        int (*ssl_new)(SSL *s);
        void (*ssl_clear)(SSL *s);
        void (*ssl_free)(SSL *s);
        int (*ssl_accept)(SSL *s);
        int (*ssl_connect)(SSL *s);
        int (*ssl_read)(SSL *s,void *buf,int len);
        int (*ssl_peek)(SSL *s,void *buf,int len);
        int (*ssl_write)(SSL *s,const void *buf,int len);
        int (*ssl_shutdown)(SSL *s);
        int (*ssl_renegotiate)(SSL *s);
        int (*ssl_renegotiate_check)(SSL *s);
        long (*ssl_ctrl)(SSL *s,int cmd,long larg,char *parg);
        long (*ssl_ctx_ctrl)(SSL_CTX *ctx,int cmd,long larg,char
*parg);
        SSL_CIPHER *(*get_cipher_by_char)(const unsigned char
*ptr);
        int (*put_cipher_by_char)(const SSL_CIPHER *cipher,unsigned
char *ptr);
        int (*ssl_pending)(SSL *s);
        int (*num_ciphers)(void);
        SSL_CIPHER *(*get_cipher)(unsigned ncipher);
        struct ssl_method_st *(*get_ssl_method)(int version);
        long (*get_timeout)(void);
        struct ssl3_enc_method *ssl3_enc; /* Extra SSLv3/TLS stuff
*/
        int (*ssl_version)();
        long (*ssl_callback_ctrl)(SSL *s, int cb_id, void (*fp)());
        long (*ssl_ctx_callback_ctrl)(SSL_CTX *s, int cb_id, void
(*fp)());
} SSL_METHOD;
```

```
SSL_METHOD *SSLv2_method(void);          /* SSLv2 */
SSL_METHOD *SSLv2_server_method(void);   /* SSLv2 */
SSL_METHOD *SSLv2_client_method(void);   /* SSLv2 */
SSL_METHOD *SSLv3_method(void);          /* SSLv3 */
SSL_METHOD *SSLv3_server_method(void);   /* SSLv3 */
SSL_METHOD *SSLv3_client_method(void);   /* SSLv3 */
SSL_METHOD *SSLv23_method(void);         /* SSLv3 but can rollback
to v2 */
SSL_METHOD *SSLv23_server_method(void); /* SSLv3 but can rollback
to v2 */
SSL_METHOD *SSLv23_client_method(void); /* SSLv3 but can rollback
to v2 */
SSL_METHOD *TLSv1_method(void);          /* TLSv1.0 */
SSL_METHOD *TLSv1_server_method(void);   /* TLSv1.0 */
SSL_METHOD *TLSv1_client_method(void);   /* TLSv1.0 */
SSL_METHOD *SSL_get_ssl_method(SSL *s);
```

Sono molte le funzioni API OpenSSL che gestiscono i metodi di protocollo SSL/TLS e sono tutte definite nella struttura **SSL_METHOD**. Ad esempio **SSLv3_method()** è la funzione costruttore per il tipo di protocollo SSLv3 per utilizzo combinato client/server. Tutti questi tipi di funzione hanno prototipi simili fra loro e servono per inizializzare all'interno del proprio programma il tipo di struttura dati specifica a seconda del tipo di protocollo che si vuole utilizzare, SSLv2, SSLv3 oppure TLSv1 e del tipo di applicazione che deve gestire il flusso di dati sia esso un server oppure un client.

4.2.4 SSL_CTX_new()

```
#include <openssl/ssl.h>
SSL_CTX *SSL_CTX_new(SSL_METHOD *method);
```

Crea un oggetto di tipo **SSL_CTX** inizializzando il framework necessario per stabilire connessioni SSL/TLS. **SSL_CTX_new()** inizializza la lista dei

cipher, le callback, le chiavi ed i certificati. La funzione ha un unico argomento che è un puntatore ∗**SSL_METHOD**. L'argomento della funzione serve per identificare il metodo di protocollo di connessione e può essere una delle seguenti combinazioni:

SSLv2_method(void), SSLv2_server_method(void), SSLv2_client_method(void)
Una connessione SSL/TLS stabilita con questi tipi di metodi, sarà in grado di utilizzare soltanto il protocollo SSLv2. Un determinato client invierà una messaggio di handshake indicando che il protocollo da lui supportato è SSLv2. D'altra parte un server che utilizza questo metodo sarà in grado di interpretare messaggi di handshake solo di tipo SSLv2.

SSLv3_method(void), SSLv3_server_method(void), SSLv3_client_method(void)
Una connessione SSL/TLS stabilita con questi tipi di metodi, sarà in grado di utilizzare soltanto il protocollo SSLv3. Da parte del client, questo invierà un messaggio di handshake indicando che il protocollo da lui supportato è SSLv3. Per quanto riguarda il server questo sarà in grado di interpretare soltanto messaggi di handshake di tipo SSLv3. Questo significa che per entrambe le tipologie di software client o server che utilizzano questi metodi non sarà possibile capire i messaggi di hanshake del protocollo SSLv2 e tanto meno TLSv1.

TLSv1_method(void), TLSv1_server_method(void), TLSv1_client_method(void)
Una connessione SSL/TLS stabilita con questi tipi di metodi è compatibile con il protocollo TLSv1 ma non con i protocolli SSLv2 ed SSLv3. Quindi i client invieranno in fase di connessione un messaggio di handshake indicando che il protocollo da loro supportato è TLSv1, ed i server saranno in grado di interpretare soltanto i messaggi di handshake da parte dei client di tipo TLSv1.

SSLv23_method(void), SSLv23_server_method(void), SSLv23_client_method(void)
Una connessione SSL/TLS stabilita con questi tipi di metodi è compatibile con i protocolli di tipo SSLv2 SSLv3 e TLSv1. In particolare un client che usa questo tipo di API invierà in fase di connessione un messaggio di handshake SSLv2 nel quale specifica la compatibilità con e SSLv3 e TLSv1. Dal lato server vi è il supporto completo in fase di hanshake con il client per tutti e 3 i tipi di protocollo SSLv2, SSLv3 e TLSv1. Se la propria applicazoine ha come target la compatibilità software con altre e diverse implementazioni di SSL, allora questo tipo di API rappresenta la scelta migliore.

Valori di ritorno

NULL: Significa che la creazione di un oggetto di tipo **SSL_CTX** non ha avuto successo. È opportuno controllare il codice di errore per sapere il motivo della mancata inizializzazione del contesto SSL.

puntatore ad un oggetto SSL_CTX: Il valore di ritorno punta ad una struttura **SSL_CTX** allocata in memoria.

La struttura **SSL_CTX** è abbastanza complicata e contiene al suo interno tutto il framework necessario per una connessione SSL/TLS compresa la struttura **SSL_METHOD** che abbiamo analizzato in precedenza.

4.2.5 SSL_CTX_use_certificate_file()

```
#include <openssl/ssl.h>
int SSL_CTX_use_certificate_file(SSL_CTX *ctx, const char *file, int
type);
```

SSL_CTX_use_certificate_file(), fa parte di un gruppo di funzioni che hanno il compito di caricare i certificati e le chiavi private all'interno del

contesto del framework SSL. L'insieme di funzioni SSL_CTX_* carica i certificati e le chiavi all'interno di una struttura **ctx** di tipo SSL_CTX. L'informazione viene trasmessa a oggetti di tipo **SSL** creati a partire da **ctx** tramite la funzione **SSL_new()** per copia, così che cambiamenti apportati al contesto **ctx** non si propaghino ad oggetti SSL già esistenti. In particolare **SSL_CTX_use_certificate_file()** carica all'interno di **ctx** il certificato che si trova in **file**. Il tipo di formato di certificato può essere di due tipi, **SSL_FILETYPE_PEM** oppure **SSL_FILETYPE_ASN1**.

Valori di ritorno
Se ha successo la funzione ritorna 1.

4.2.6 SSL_CTX_use_PrivateKey_file()

```
#include <openssl/ssl.h>
int SSL_CTX_use_PrivateKey_file(SSL_CTX *ctx,
                                const char *file, int type);
```

Aggiunge a **ctx** la chiave privata che si trova in **file**. Il tipo di formato del certificato va specificato attraverso i parametri **SSL_FILETYPE_PEM** oppure **SSL_FILETYPE_ASN1**.

Valori di ritorno
Se ha successo la funzione ritorna 1.

4.2.7 SSL_CTX_check_private_key()

```
#include <openssl/ssl.h>
int SSL_CTX_check_private_key(SSL_CTX *ctx);
```

Controlla che la chiave privata e il corrispondente certificato caricato nel contesto SSL **ctx** siano consistenti tra loro. Se è stata installata più di una coppia chiave/certificato (RSA/DSA), verrà controllato l'ultimo elemento in-

stallato. È da notare infatti che il sistema di storage interno a OpenSSL per i certificati può contenere due coppie chiave/certificato alla volta, una di tipo RSA e l'altra di tipo DSA.

Valori di ritorno
Se ha successo la funzione ritorna 1.

4.2.8 SSL_CTX_free()

```
#include <openssl/ssl.h>
void SSL_CTX_free(SSL_CTX *ctx);
```

Decrementa il numero di referenze a **ctx** e rimuove la struttura **SSL_CTX** allocata in memoria puntata da **ctx** liberando la memoria allocata se il reference count diventa 0. Non ha valori di ritorno.

4.2.9 SSL_new()

```
#include <openssl/ssl.h>
SSL *SSL_new(SSL_CTX *ctx);
```

SSL_new() crea una nuova struttura SSL che è necessaria per gestire tutti i dati relativi a una connessione TLS/SSL. La nuova struttura eredita le proprietà del contesto **ctx** che è il layer sottostante su cui si basa SSL. Queste proprietà sono il tipo di metodo di protocollo utilizzato (cioè SSLv2, SSLv3, TLSv1), varie opzioni, cipher, timeout setting etc.

Valori di ritorno
Ritorna i seguenti valori:

NULL: La creazione di una nuova struttura SSL non ha avuto successo.

Puntatore a una struttura SSL: Il valore di ritorno punta ad una struttura SSL allocata in memoria.

4.2.10 SSL_free()

```
#include <openssl/ssl.h>
void SSL_free(SSL *ssl);
```

Decrementa il contatore ai reference di **ssl** e rimuove la struttura SSL puntata da **ssl** liberando la memoria allocata se il contatore ai reference ha raggiunto il valore 0.

Valori di ritorno
Non ritorna alcun valore.

4.2.11 SSL_get_cipher()

```
#include <openssl/ssl.h>
SSL_CIPHER *SSL_get_current_cipher(SSL *ssl);
#define SSL_get_cipher(s)
        SSL_CIPHER_get_name(SSL_get_current_cipher(s))
```

SSL_get_current_cipher() ritorna un puntatore ad una struttura di tipo **SSL_CIPHER** contenente la descrizione dell'algoritmo di cifratura correntemente in uso relativamente ad una connessione stabilita tramite l'oggetto **ssl**. **SSL_get_cipher()** è una macro di **SSL_get_current_cipher()** che ritorna il nome dell'algoritmo di cifratura correntemente in uso, ritorna cioè un puntatore al nome del cipher utilizzato cioè un **char***.

Valori di ritorno
Ritorna il nome del cipher utilizzato oppure ritorna **NULL** se non è stata stabilita alcuna sessione SSL.

4.2.12 SSL_CIPHER_get_name()

```
#include <openssl/ssl.h>
```

```
const char *SSL_CIPHER_get_name(SSL_CIPHER *cipher);
```

Ritorna un puntatore al nome del cipher utilizzato. Se l'argomento della funzione è **NULL**, la stringa ritornata è "NONE".

4.2.13 SSL_accept()

```
#include <openssl/ssl.h>
int SSL_accept(SSL *ssl);
```

Aspetta che un determinato client TLS/SSL dia inizio all'handshake di connessione TLS/SSL. Il canale di comunicazione però (socket) deve già essere stato creato ed assegnato a **ssl** utilizzando **SSL_set_fd()**. Il comportamento di **SSL_accept()** dipende dallo strato *BIO* sottostante. Il BIO è un'astrazione di I/O, nasconde molti dei dettagli dell'I/O relativi ad un'applicazione e fornisce al programma un'interfaccia trasparente, coerente e semplificata dal punto di vista della programmazione. In generale se un'applicazione utilizza un BIO per fare I/O allora può gestire in modo trasparente connessioni SSL, connessioni di rete non criptate e file I/O.

Tornando ad SSL se il BIO sottostante l'applicazione è di tipo **blocking**, **SSL_accept()** ritornerà un valore solo quando l'handshake è terminato oppure se si verifica una situazione di errore. Se il BIO sottostante è di tipo **non-blocking**, allora **SSL_accept()** può ritornare un valore anche se il BIO al layer sottostante l'applicazione non può soddisfare le richieste di **SSL_accept()** di continuare l'handshake. In questo caso chiamando la funzione **SSL_get_error()** con il valore di ritorno di **SSL_accept()** ritornerà **SSL_ERROR_WANT_READ** oppure **SSL_ERROR_WANT_WRITE**. Il processo chiamante deve ripetere la chiamata dopo avere preso delle contromisure appropriate per soddisfare le richieste della funzione **SSL_accept()**.

Valori di ritorno

Possono presentarsi i seguenti casi:

1: L'handshake TLS/SSL è stato completato con successo ed è stata stabilita una connessione TLS/SSL.

0: L'handshake TLS/SSL non ha avuto successo ma è stato terminato in modo controllato seguendo le specifiche del protocollo TLS/SSL. Occorre chiamare **SSL_get_error()** per avere ulteriori informazioni.

<0: L'handshake TLS/SSL non ha avuto successo perchè è avvenuto un errore irreversibile a livello di protocollo oppure a livello di connessione. Lo shutdown non è stato inoltrato in modo "pulito". Per continuare L'I/O tramite SSL/TLS occorre eventulmente fare il flush dell' I/O buffer. Occorre chiamare **SSL_get_error()** per avere ulteriori informazioni.

4.2.14 SSL_set_fd()

```
#include <openssl/ssl.h>
int SSL_set_fd(SSL *ssl, int fd);
```

Associa l'oggetto SSL ad un file descriptor. Setta il file descriptor **fd** come interfaccia di input/output per la parte TLS/SSL di **ssl**. Tipicamente **fd** è il descrittore di file socket di una connessione di rete. Quando viene chiamata **SSL_set_fd()** automaticamente un BIO socket viene creato per interfacciare **ssl** e **fd** cioè per interfacciare la struttura SSL con il socket descriptor della connessione di rete, Il BIO e di conseguenza il motore SSL eredita il comportamento del socket descriptor **fd**.

Valori di ritorno

Possono presentarsi i seguenti casi:

0: La chiamata è fallita.

1: L'operazione ha avuto successo.

4.2.15 SSL_connect()

```
#include <openssl/ssl.h>
int SSL_connect(SSL *ssl);
```

Con questa chiamata un client inizia la procedura di handshake con un server TLS/SSL. Il canale di comunicazione deve essere già stato aperto ed assegnato a **ssl**. La funzione ha lo stesso comportamento di **SSL_accept()** per quanto riguarda le interazioni con l'I/O del sistema operativo ed i valori di ritorno della funzione.

4.2.16 SSL_read()

```
#include <openssl/ssl.h>
int SSL_read(SSL *ssl, void *buf, int n);
```

Legge **n** byte da **ssl** e li copia nel buffer **buf**. Pertanto questa funzione serve per leggere un certo numero di byte da una connessione SSL. Se necessario **SSL_read()** può negoziare una sessione TLS/SSL se non è stata esplicitamente configurata tramite **SSL_accept()** o **SSL_connect()**. Il comportamento di **SSL_read()** dipende dal BIO sottostante. Perchè una negoziazione trasparente abbia successo **ssl** deve essere inizializzata in modalità client o server. Se questo non è stato fatto esplicitamente configurando opportunamente il contesto **SSL_CTX** e chiamando **SSL_accept()** o **SSL_connect()** bisogna utilizzare le funzioni **SSL_set_connect_state()** o **SSL_set_accept_state()**:

```
#include <openssl/ssl.h>
void_SSL_set_connect_state(SSL_*ssl);
void_SSL_set_accept_state(SSL_*ssl);
```

Queste funzioni servono per settare la modalità **ssl** client oppure server. **SSL_read()** legge unità di dati che sono dei record TLS/SSL. I dati sono ricevuti in record di dimensione massima di 16KB per SSLv3/TLSv1. Solo quando un determinato record è completamente ricevuto può essere pro-

cessato (decryption e intrgrity check). Pertando può accadere che ci siano dati che non sono stati ancora processati dopo una chiamata a **SSL_read()**, ma rimangono nel buffer interno e saranno quindi processati alla successiva chiamata della funzione. Se **n** è maggiore del numero di byte nel buffer, **SSL_read()** ritorna il numero di byte nel buffer. Se non ci sono più dati nel buffer, **SSL_read()** inizierà a processare i dati del record successivo. Solo quando un determinato record è stato ricevuto e processato per intero allora **SSL_read()** ritorna un valore con successo. Siccome la dimensione di un record SSL/TLS può essere maggiore della massima dimensione del pacchetto del protocollo di trasporto del layer sottostante a SSL, potrebbe eessre necessario leggere più pacchetti dal transport layer prima che il record SSL sia completo e **SSL_read()** possa avere un valore di ritorno che indica successo. Se il BIO sottostante è di tipo blocking, **SSL_read()** ritornerà un valore solo quando l'operazione di lettura non sia finita oppure sia successo un errore. Se il BIO sottostante è di tipo non-blocking allora **SSL_read()** può ritornare anche quando il BIO non può soddisfare le richieste di **SSL_read()** per continuare nell'operazione di lettura. In questo caso una chiamata alla funzione **SSL_get_error()** darà come risultato **SSL_ERROR_WANT_READ** oppure **SSL_ERROR_WANT_WRITE**. Per default il comportamento del BIO sottostante ad SSL è di tipo **blocking**

Valori di ritorno

Ha i seguenti valori di ritorno:

>0: L'operazione ha avuto successo. Il valore di ritorno è il numero di byte letti dalla connessione TLS/SSL.

0: L'operazione di lettura non ha avuto successo. Il motivo può essere uno schutdown della connessione dovuto a un "close notify" mandato dall' host peer. È anche possibile che il peer remoto abbia eseguito uno shutdown a livello di socket o a livello di layer di trasporto abbia chiuso la connessione, per cui lo shutdown SSL risulta essere incompleto. Chiamando **SSL_get_error()** è possibile conoscere il tipo di

errore che ha causato lo shutdown della connessoine SSL. Nel caso sia stato fatto un *clean shutdown* dall'host peer il valore ritonato sarà **SSL_ERROR_ZERO_RETURN**. SSLv2 non supporta un protocollo di *shutdown alert* per cui in questo caso si può solo sapere se è stata chiusa la connessione a livello di transport layer ma non si può sapere se sia stata volutamente chiusa dal peer host o per altri motivi.

<0: L'operazione di lettura è terminata con un errore, oppure una determinata azione deve essere presa dal processo chiamante.

4.2.17 SSL_write()

```
#include <openssl/ssl.h>
int SSL_write(SSL *ssl, const void *buf, int n);
```

Scrive un numero **n** di byte dal buffer **buf** nel canale **ssl**. Quindi scrive un certo numero di byte all'interno di una connessione TLS/SSL. Ha le stesse caratteristiche di **SSL_read()** e gli stessi valori di ritorno.

4.2.18 SSL_shutdown()

```
#include <openssl/ssl.h>
int SSL_shutdown(SSL *ssl);
```

Effettua lo shut-down di una connessione TLS/SSL attiva. Invia un messaggio di *close notify* al progamma dall'altra parte della connessione SSL.

Valori di ritorno
Possono presentarsi i seguenti casi:

1: La procedura di shutdown SSL ha avuto successo.

0: La procedutra di shutdown non è completamente avvenuta a seconda dei casi potrebbe essere necessario chiamare nuovamente **SSL_shutdown()**

se per il tipo di connessione è richiesto uno shutdown di tipo bidirezionale.

-1: La procedura di shutdown non ha avuto successo a causa di errori avvenuti a livello di protocollo SSL o a livello di trasporto.

4.2.19 SSL_get_error()

```
#include <openssl/ssl.h>
int SSL_get_error(SSL *ssl, int ret);
```

Ritorna un valore successivamente ad un'operazione di I/O TLS/SSL. Ritorna un codice di errore conseguente ad una precedente chiamata alle funzioni **SSL_accept()**, **SSL_connect()**, **SSL_read()**, **SSL_write()** etc. che hanno come argomento un puntatore alla struttura **SSL**. Inoltre **SSL_get_error()** guarda all'interno della coda del thread correntemente in uso dalla sessione, per questo motivo **SSL_get_error()** deve essere utlizzata all'interno dello stesso thread che ha effettuato l'operazione di I/O SSL/TLS e non deve essere utilizzata nessun'altra funzione tra le chiamate alle API di I/O ed **SSL_get_error()**. La error queue relativamente al thread di I/O corrente deve essere vuota prima di effettuare una chiamata ad una funzione di I/O TLS/SSL oppure l'utlizzo successivo di **SSL_get_error()** non potrà essere pienamente funzionale. Sono molti i possibili valori di ritorno e non li elencheremo in questa sede per brevità. Precedentemente sono stati riportati alcuni dei valori di ritorno più comuni quali **SSL_ERROR_WANT_READ**, **SSL_ERROR_WANT_WRITE**, **SSL_ERROR_ZERO_RETURN**.

4.2.20 ERR_print_errors_fp()

```
#include <openssl/ssl.h>
#include <openssl/err.h>
void ERR_print_errors_fp(FILE *fp);
```

Utilizza il file pointer **fp** per scrivere una stringa di errore relativa alla connessione SSL in corso. Nel caso una delle funzioni SSL abbia riportato un errore, chiamando **ERR_print_errors_fp(stderr)** viene riportato sullo standard error il tipo di errore.

4.2.21 X509_get_issuer_name()

```
#include <openssl/ssl.h>
#include <openssl/x509.h>
X509_NAME *X509_get_issuer_name(X509 *a);
X509_NAME *X509_get_subject_name(X509 *a);
```

Sono funzioni che ritornano una struttura di tipo **X509_NAME**, e servono per maneggiare i dati relativi ai certificati. Purtroppo sono funzioni non ancora del tutto documentate all'interno di OpenSSL.

5 SSL server e client

Viene riportato ora un esempio di applicazione SSL client-server.
Analizziamo innanzituto la parte server:

```
 1      #include        <sys/types.h>
 2      #include        <sys/socket.h>
 3      #include        <sys/time.h>
 4      #include        <sys/param.h>
 5      #include        <time.h>
 6      #include        <netinet/in.h>
 7      #include        <arpa/inet.h>
 8      #include        <errno.h>
 9      #include        <err.h>
10      #include        <fcntl.h>
11      #include        <netdb.h>
12      #include        <signal.h>
```

```
13      #include        <stdio.h>
14      #include        <stdlib.h>
15      #include        <string.h>
16      #include        <sys/stat.h>
17      #include        <unistd.h>
18      #include        <sys/wait.h>
19      #include        <strings.h>
20
21
22      #include <openssl/rsa.h>
23      #include <openssl/crypto.h>
24      #include <openssl/ssl.h>
25      #include <openssl/err.h>
26      #include <openssl/x509.h>
27      #include <openssl/pem.h>
28
29      #define PORT 8822
30
31      #define CERTF  ''newcert.pem''
32      #define KEYF   ''server.key''
33
34
35      SSL_CTX*    ctx;
36      SSL*        ssl;
37      SSL_METHOD  *meth;
38
39      #define CHK_SSL(err) if ((err)==-1) {
ERR_print_errors_fp(stderr); exit(2); }
40
41
42
43      int main(int argc, char **argv)
44      {
45        int listenfd,connfd,err;
```

```
 46        const int on = 1;
 47        struct sockaddr_in servaddr;
 48        struct sockaddr_in cliaddr;
 49        socklen_t  clilen;
 50        char       buf[4096];
 51
 52        SSL_load_error_strings();
 53        SSL_library_init();
 54
 55        meth = SSLv3_server_method();
 56        ctx = SSL_CTX_new (meth);
 57        if (!ctx) {
 58          ERR_print_errors_fp(stderr);
 59          exit(2);
 60        }
 61
 62        if (SSL_CTX_use_certificate_file(ctx, CERTF,
SSL_FILETYPE_PEM) <= 0) {
 63          ERR_print_errors_fp(stderr);
 64          exit(3);
 65        }
 66        if (SSL_CTX_use_PrivateKey_file(ctx, KEYF,
SSL_FILETYPE_PEM) <= 0) {
 67          ERR_print_errors_fp(stderr);
 68          exit(4);
 69        }
 70        if (!SSL_CTX_check_private_key(ctx)) {
 71          fprintf(stderr,''Private key does not match the certificate
public key\n'');
 72          exit(5);
 73        }
 74
 75        if((listenfd = socket(AF_INET, SOCK_STREAM, 0))<0) {
 76          perror(''socket()'');
```

```
 77          exit(1);
 78        }
 79        setsockopt(listenfd,SOL_SOCKET, SO_REUSEADDR, &on,
sizeof(on));
 80        bzero(&servaddr, sizeof(servaddr));
 81        servaddr.sin_family      = AF_INET;
 82        servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
 83        servaddr.sin_port        = htons(PORT);
 84        if(bind(listenfd, (struct sockaddr *) &servaddr,
sizeof(servaddr))<0) {
 85          perror(''bind()'');
 86          exit(1);
 87        }
 88        if(listen(listenfd, 5)<0) {
 89          perror(''listen()'');
 90          exit(1);
 91        }
 92        clilen = sizeof(cliaddr);
 93        if ( (connfd = accept(listenfd, (struct sockaddr *)
&cliaddr, &clilen)) < 0){
 94          perror(''accept()'');
 95          exit(1);
 96        }
 97        close(listenfd);
 98        fprintf(stderr,''Connection from:
%s\n'',inet_ntoa(cliaddr.sin_addr));
 99        ssl = SSL_new(ctx);
100        if(!ssl) exit(1);
101        SSL_set_fd(ssl,connfd);
102        err = SSL_accept(ssl);  CHK_SSL(err);
103        fprintf(stderr,''Started SSL connection using %s\n'',
SSL_get_cipher (ssl));
104
105        err = SSL_read (ssl, buf, sizeof(buf) - 1);
```

```
CHK_SSL(err);
106         buf[err] = '\0';
107         printf(''%s\n'',buf);
108         err = SSL_write (ssl, ''I hear you'', strlen(''I hear
you''));  CHK_SSL(err);
109         SSL_free (ssl);
110         close(connfd);
111         SSL_CTX_free (ctx);
112         return 0;
113       }
```

Analizziamo le righe di codice passo passo:

1-19: File di include generici del programma compresi i file di include necessari per il framework TCP.

22-27: File di include specifici del framework SSL.

29-32: Viene definita la porta TCP sulla quale si mette in ascolto il demone SSL e vengono definiti i file con il certificato del server SSL e la chiave privata.

35-39: Vengono dichiarati i puntatori del contesto del framework SSL e viene definita una macro shortcut che richiama la funzione SSL che stampa su standard error gli errori.

45-50: Siamo all'interno del main del programma vengono dichiarati i socket descriptor e la strutture per manipolare i dati relativi ai socket.

52-53: Viene inizializzata la libreria OpenSSL registrando i cipher e digest disponibili.

55-60: Viene inizializzato il framework SSL con la chiamata alla funzione **SSLv3_server_method()** specificando che il progamma usa un metodo server e protocollo SSLv3. Viene poi creato un nuovo contesto SSL_CTX partendo dal metodo di protocollo appena definito, il

contesto creato sarà quello della connessione SSL che fra poco verrà inizializzata.

62-73: In questa fase vengono caricati dall'engine SSL i file con il certificato rilasciato da una CA e la chiave privata. Viene poi chiamata la funzione **SSL_CTX_check_private_key()** per controllare che certificato e private key siano consistenti fra loro.

75-97: Queste righe di codice comprendono tutta la parte che inizializza una connessione TCP server side. **listen()** crea un socket IPv4/TCP. Con la funzione **setsockopt()** si vanno a modificare quelli che sono i parametri di default del socket e in questo caso viene settato il parametro **SO_REUSEADDR** in modo da potere riutilizzare la porta TCP del server SSL associata al socket pair precedentemente definito senza dovere aspettare il **TIME_WAIT** state del socket TCP. In questo caso la medesima porta potrà così essere riutilizzata da una nuova connessione, in un nuovo socket pair. Viene poi riempita la struttura **sockaddr_in** con l'indirizzo IP del server SSL e la porta TCP, in questo caso con il parametro **INADDR_ANY** specifichiamo di ascoltare su tutte le interfacce di rete disponibili nel caso ve ne sia più di una. Le funzioni **htons()** e **htonl()** si occupano di allineare l'host byte order (che è diverso per architetture hardware e sistemi operativi diversi) con il network byte order IP che è invece di tipo *b*ig-endian. Con **bind()** viene assegnato un socket specifico all'indirizzo locale di protocollo, precedentemente definito riempendo la struttura **sockaddr_in**. A questo punto dopo avere chiamalo la funzione **accept()**, il socket diventa completamente operativo e può rispondere a richieste di connessione. Se non ci sono connessioni in coda il programma si mette in attesa (default blocking I/O). Appena si riempie la coda prende la prima connessione e crea un nuovo socket **connfd** con le stesse proprietà del socket chiamante e alloca un nuovo file descriptor per il socket.

99-103: Passiamo ora alla connessione SSL vera e propria. Dal punto di vita

delle API viene trattata in modo trasparente senza doversi preoccupare del tansport layer che sta al livello sottostante. Viene creata una nuova struttura SSL con **SSL_new()** che serve per gestire i dati relativi alla nuova connessione SSL, la quale eredita tutte le proprietà del contesto **SSL_CTX** precedentemente inizializzato, ovvero il tipo di protocollo che in questo caso sarà SSLv3, i setting relativi al timeout, il framework relativo ai certificati etc. Con **SSL_set_fd()** viene associato all'oggetto SSL appena creato un file descriptor che servirà per le operazioni di I/O SSL. Questo file descriptor è il socket descriptor **connfd** precedentemente creato durante l' inizializzazione della connessione TCP. **SSL_accept()** mette il progamma in attesa di una connessione SSL da parte di un client basandosi sul canale di comunicazione socket TCP precedentemente inizializzato.

105-108: A questo punto è possibile leggere e scrivere dati nel tunnel SSL da e verso un client che abbia stabilito una connessione. Per scrievere e leggere dati all'interno della connessione SSL si utilizzano le funzioni **SSL_read()** e **SSL_write()**.

109-113: Viene liberata la memoria allocata dalla struttura SSL, viene chiuso il socket e viene liberata la memoria allocata per il contesto e tutto il framework SSL. Con questa procedura il server chiude la connessione dopo avere ricevuto una richiesta di *connection shutdown* da parte del client.

Analizziamo ora invece il programma client SSL riportando prima il codice e poi analizzandolo brevemente tenendo presente i concetti già analizzati per la parte server:

```
1       #include <sys/types.h>
2       #include <sys/socket.h>
3       #include <sys/time.h>
4       #include <time.h>
5       #include <netinet/in.h>
```

```
 6      #include <arpa/inet.h>
 7      #include <errno.h>
 8      #include <fcntl.h>
 9      #include <netdb.h>
10      #include <signal.h>
11      #include <stdio.h>
12      #include <stdlib.h>
13      #include <string.h>
14      #include <sys/stat.h>
15      #include <unistd.h>
16      #include <sys/wait.h>
17      #include <strings.h>
18      #include <errno.h>
19
20
21      #ifndef INADDR_NONE
22      #define INADDR_NONE     0xffffffff      /* should be in
<netinet/in.h> */
23      #endif
24
25
26      #include <openssl/rsa.h>
27      #include <openssl/crypto.h>
28      #include <openssl/ssl.h>
29      #include <openssl/err.h>
30      #include <openssl/x509.h>
31      #include <openssl/pem.h>
32
33      #define VALIDUSERCHAR
''abcdefghijklmnopqrstuvwxyz1234567890_-''
34
35      #define PORT 8822
36
37      #define CHK_NULL(x) if ((x)==NULL) exit (1)
```

```
 38      #define CHK_ERR(err,s) if ((err)==-1) { perror(s);
exit(1); }
 39      #define CHK_SSL(err) if ((err)==-1) {
ERR_print_errors_fp(stderr); exit(2); }
 40
 41
 42      SSL_CTX*    ctx;
 43      SSL*        ssl;
 44      SSL_METHOD  *meth;
 45      X509*        server_cert;
 46
 47
 48      struct sockaddr_in      tcp_srv_addr;   /* server's
Internet socket addr */
 49      struct servent          tcp_serv_info;  /* from
getservbyname() */
 50      struct hostent          tcp_host_info;  /* from
gethostbyname() */
 51
 52      void cert(void);
 53      int tcp4open(char *, char *,int );
 54
 55
 56      int main(int argc, char **argv)
 57      {
 58
 59        int socket,err;
 60        char *server=NULL;
 61        char buf[4096];
 62
 63        if(argv[1])
 64          server=argv[1];
 65        else {
```

```
 66          fprintf(stderr,''%s remote SSL server
host\n'',argv[0]);
 67          exit(1);
 68        }
 69
 70
 71        SSL_load_error_strings();
 72        SSL_library_init();
 73        meth = SSLv3_client_method();
 74        ctx = SSL_CTX_new (meth);
 75        if (!ctx) {
 76          ERR_print_errors_fp(stderr);
 77          exit(2);
 78        }
 79
 80        socket=tcp4open(server,NULL,PORT);
 81
 82        ssl = SSL_new (ctx); CHK_NULL(ssl);
 83        SSL_set_fd (ssl, socket);
 84        err=SSL_connect(ssl); CHK_SSL(err);
 85        fprintf(stderr,''SSL connection using %s\n'',
SSL_get_cipher (ssl));
 86        cert();
 87
 88        err = SSL_write(ssl, ''Hello World'', strlen(''Hello
World'')); CHK_SSL(err);
 89        err=SSL_read (ssl, buf, sizeof(buf) - 1); CHK_SSL(err);
 90        buf[err]='\0';
 91        printf(''%s\n'',buf);
 92        SSL_shutdown (ssl);
 93        close (socket);
 94        SSL_free (ssl);
 95        SSL_CTX_free (ctx);
 96        return 0;
```

```
 97      }
 98
 99      void cert(void)
100      {
101        char *str;
102
103
104        server_cert = SSL_get_peer_certificate (ssl);
CHK_NULL(server_cert);
105        str = X509_NAME_oneline (X509_get_subject_name
(server_cert),0,0);
106        CHK_NULL(str);
107
108
109        printf (''\t subject: %s\n'', str);
110        free (str);
111        str = X509_NAME_oneline (X509_get_issuer_name
(server_cert),0,0);
112        CHK_NULL(str);
113        printf (''\t issuer: %s\n'', str);
114        free (str);
115        X509_free (server_cert);
116        return;
117      }
118
119
120
121
122      int  tcp4open(char *host, char *service,int port)
123      {
124        int    fd,resvport;
125        unsigned long  inaddr;
126        struct servent  *sp;
127        struct hostent  *hp;
```

```
128
129         /*
130          * Initialize the server's Internet address structure.
131          * We'll store the actual 4-byte Internet address and
the
132          * 2-byte port# below.
133          */
134
135         bzero((char *) &tcp_srv_addr, sizeof(tcp_srv_addr));
136         tcp_srv_addr.sin_family = AF_INET;
137
138         if (service != NULL) {
139           if ( (sp = getservbyname(service, ''tcp'')) == NULL) {
140             printf(''tcp_open: unknown service: %s/tcp'',
service);
141             return(-1);
142           }
143           tcp_serv_info = *sp;      /* structure copy */
144           if (port > 0)
145             tcp_srv_addr.sin_port = htons(port);
146           /* caller's value */
147           else
148             tcp_srv_addr.sin_port = sp->s_port;
149           /* service's value */
150         } else {
151           if (port <= 0) {
152             perror(''tcp_open: must specify either service or
port'');
153             return(-1);
154           }
155           tcp_srv_addr.sin_port = htons(port);
156         }
157
158         /*
```

```
159         * First try to convert the host name as a
dotted-decimal number.
160         * Only if that fails do we call gethostbyname().
161         */
162
163        if ( (inaddr = inet_addr(host)) != INADDR_NONE) {
164          /* it's dotted-decimal */
165          bcopy((char *) &inaddr, (char *)
&tcp_srv_addr.sin_addr,
166          sizeof(inaddr));
167          tcp_host_info.h_name = NULL;
168
169        } else {
170          if ( (hp = gethostbyname(host)) == NULL) {
171            perror(''tcp_open: host name error'');
172            return(-1);
173          }
174          tcp_host_info = *hp;  /* found it by name, structure
copy */
175          bcopy(hp->h_addr, (char *) &tcp_srv_addr.sin_addr,
176          hp->h_length);
177        }
178
179        if (port >= 0) {
180          if ( (fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
181            perror(''tcp_open: can't create TCP socket'');
182            return(-1);
183          }
184
185        } else if (port < 0) {
186          resvport = IPPORT_RESERVED - 1;
187          if ( (fd = rresvport(&resvport)) < 0) {
188            perror(''tcp_open: can't get a reserved TCP port'');
189            return(-1);
```

```
190           }
191         }
192
193         /*
194          * Connect to the server.
195          */
196
197         if (connect(fd, (struct sockaddr *) &tcp_srv_addr,
198               sizeof(tcp_srv_addr)) < 0) {
199           perror(''tcp_open: can't connect to server'');
200           close(fd);
201           return(-1);
202         }
203
204         return(fd);  /* all OK */
205       }
```

La struttura del programma dal punto di vista delle API SSL è del tutto analoga alla versione server:

1-18: File di include generici necessari per il progamma.

26-31: File di include necessari per le API OpenSSL.

33: È la definizione di una stringa di caratteri (non più usata poi nel seguito del progamma) in cui si definiscono i caratteri leciti che possono attraversare la connessione TLS/SSL, questo per evitare particolari tecniche di buffer overflow che si basano su sequenze particolari di caratteri **char**.

42-45: Vengono dichiarate le strutture SSL più importanti. A differenza del server in questo casa abbiamo anche la definizione di un puntatore a una struttura di tipo **X509** che serve per la gestione dei certificati.

48-53: Viene utilizzata nel programma client una funzione **tcp4open()** che

apre una connessione tcp con il server. In questo esempio non ci interessano i particolari della funzione che peraltro utilizza dei concetti già enunciati precedentemente trattando della connessione TCP relativa al progamma server. La funzione **tcp4open()** apre una connessione TCP prendendo come argomenti un hostname e un servizio TCP oppure un hostname ed una porta TCP.

59-68: Siamo quindi all'interno del main del programma, vengono dichiarati i file descriptor relativi ai socket. In questa parte di codice il progamma utilizza il secondo argomento passato sulla riga di comando per decidere il nome dell'host remoto a cui collegarsi per stabilire una sessione SSL.

71-77: Come per la parte server viene inizializzato il framework SSL, dalla parte client però si utilizza il metodo SSLv3_client_method() che specifica al framework SSL che si sta trattando di un client e non di un server. Il metodo di protocollo utilizzato è SSLv3 ovviamente il medesimo utilizzato del server.

80: Viene aperta una connessione TCP con il server e viene ritornato un file descriptor per il socket.

82-85: Viene aperta la connessione SSL inizializzando l'handshake con il server tramite la chiamata ad **SSL_connect()**, mentre il server, come già analizzato in precedenza, invoca **SSL_accept()**.

86: La funzione **cert()** effettua le chiamate **X509_get_issuer_name()** e **X509_get_issuer_name()**. In questa parte del programma si possono scrivere linee di codice specifiche per effettuare operazioni di controllo sui certificati, (controllo sull'issuer di una CA, controllo sulla data di scadenza del certificato etc.).

88-91: Ora può iniziare la comunicazione sul canale SSL, il client scrive al server una stringa "Hello World", ed il server risponde con la stringa "I hear you".

92-96: A questo punto ultimata la comunicazione nel tunnel SSL, il client manda un alert di "close notify" al server chiamando **SSL_shutdown()**, viene chiuso il socket, e viene rilasciata la memoria allocata per la struttura SSL e per il contesto di tutto il framework necessario alla connessione (**SSL_free()**, **SSL_CTX_free()**).

Vediamo ora il client e il server SSL in azione:

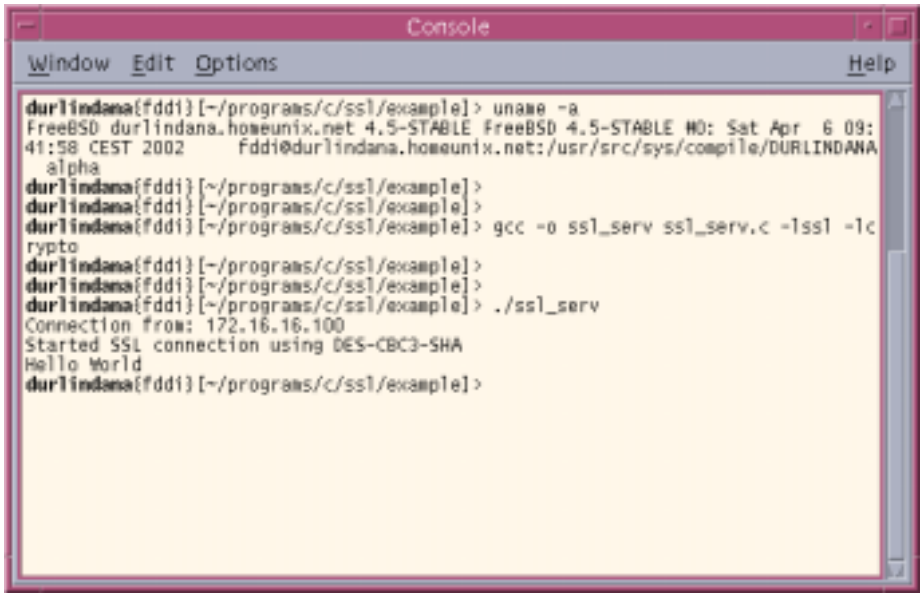

Figura 2: Server SSL dell'esempio analizzato in esecuzione

6 Conclusioni

L'insieme di tutte le funzioni che formano l'API di OpenSSL sono innumerevoli e ne sono state analizzate solamente una minima parte necessaria per sviluppare applicazioni SSL di base. Per quanto riguarda OpenSSL esiste poi una parte specifica di API, che in questa sede non è stata analizzata, che riguarda la libreria **crypto** fornita con OpenSSL. Questa libreria raccoglie

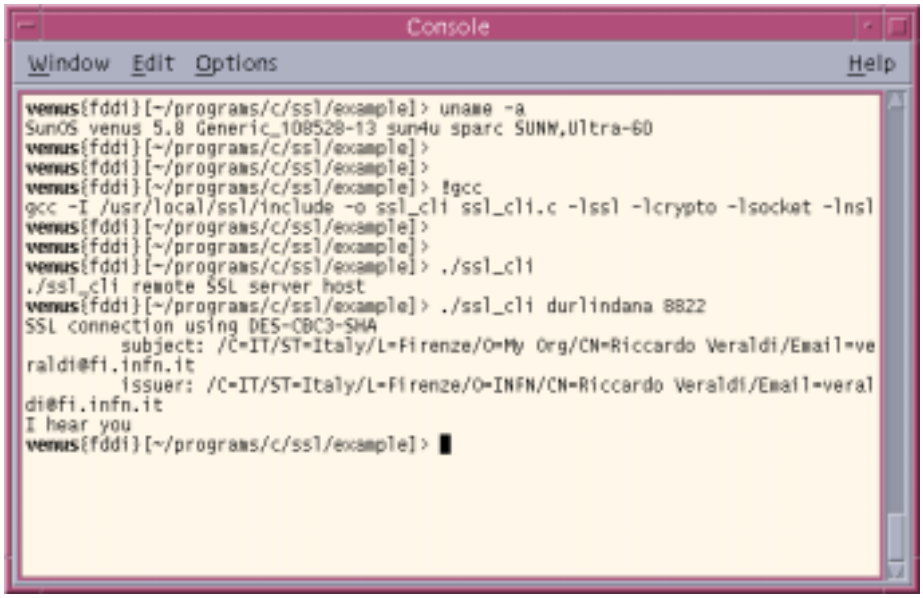

Figura 3: Client SSL dell'esempio analizzato in esecuzione

una vasta gamma di algoritmi di crittografia che sono serviti all'implementazione OpenSSL di SSL, TLS e S/MIME, e che sono stati utilizzati anche per implementare OpenSSH, OpenPGP ed altri standard crittografici.

Per approfondimenti ed integrazioni riguardanti le API OpenSSL si rimanda al sito *http://www.openssl.org* e alla documentazione ed ai progammi di esempio nonchè allo stesso codice sorgente relativi alla distribuzione sorgente di OpenSSL scaricabile dal medesimo sito. Esiste anche una mailing list molto attiva *openssl-users@openssl.org* che tratta dei vari argomenti legati ad OpenSSL, dall'utilizzo e creazione dei certificati alle API stesse che sono state brevemente trattate in questo mini-tutorial.