Università degli studi di Catania

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Informatica Applicata

Sicurezza dei Sistemi Informatici 1

Prof. Giuseppe Scollo

Anno Accademico 2006/2007

PROTOCOLLO SSL

VOTADORO ANDREA

A40/000144

SOMMARIO:

1. EVOLUZIONE DI SSL

SSL(secure socket layer protocol) è un protocollo aperto e non proprietario; è stato sottoposto da Netscape Communications all'Engineering Task Force per la sua standardizzazione, anche se di fatto è stato accettato come uno standard da tutta la comunità di internet ancor prima del verdetto del'IETF. La versione 3.0 del protocollo rilasciata nel novembre del 1996, è un evoluzione della precedente versione del 1994 la SSL v2.0, è rappresentata al momento una delle soluzioni più utilizzata per lo scambio delle informazioni cifrate. Tale evoluzione introduce un livello di sicurezza superiore rispetto alla precedente grazie a una maggiore attenzione nella fase di autenticazione tra client e server. Infine IETF propose il TLS (Trasport Layer Security) che contiene le migliorie delle versioni precedenti.

2. INTRODUZIONE

Il protocollo SSL è nato al fine di garantire la privacy delle comunicazioni su Internet, infatti permette alle applicazioni client/server di comunicare in modo da prevenire le intrusioni, le manomissioni e le falsificazioni dei messaggi.Viene accoppiato molto spesso ad altri protocolli che lavorato a livello applicativo per fornire sicurezza quando è necessario:

- HTTP: che combinato con SSL/TLS ci da il protocollo HTTPS, usato per proteggere i dati sensibili inviati da e per i server web. Usa una porta specifica (443) e un prefisso per gli URL di tipo https:// .
- SMTP: spesso usato con TLS per proteggere il contenuto delle e-mail inviate; FTP, NNTP, XMPP....

Il protocollo SSL garantisce la sicurezza del collegamento mediante tre funzionalità fondamentali:

- **Privatezza del Collegamento:** Per assicurare un collegamento sicuro tra due utenti coinvolti in una comunicazione, i dati vengono protetti utilizzando algoritmi di crittografia a chiave simmetrica (ad es. DES, RC4, ecc.);
- **Autenticazione:** L'autenticazione dell'identità nelle connessioni può essere eseguita usando la crittografia a chiave pubblica (per es. RSA, DSS ecc.). In questo modo i client sono sicuri di comunicare con il server corretto, prevenendo eventuali interposizioni. Inoltre è prevista la certificazione sia del server che del client;
- **Affidabilità:** Il livello di trasporto include un controllo sull'integrità del messaggio basato su un apposito MAC (Message Authentication Code) che utilizza funzioni hash sicure (per es. SHA, MD5 ecc.). In tal modo si verifica che i dati spediti tra client e server non siano stati alterati durante la trasmissione.

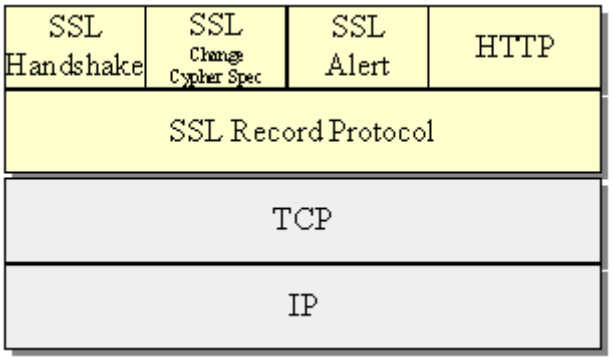
3. OBIETTIVI

Gli scopi del Protocollo SSL, in ordine di priorità, sono:

- **Sicurezza del collegamento:** SSL stabilisce un collegamento sicuro tra due sistemi;
- **Interoperabilità:** Programmatori di diverse organizzazioni dovrebbero essere in grado di sviluppare applicazioni utilizzando SSL, accordandosi sui parametri utilizzati dagli algoritmi di crittografia senza necessità di conoscere il codice l'uno dell'altro;
- **Ampliamento:** SSL cerca di fornire una struttura dentro la quale i futuri metodi di crittografia a chiave pubblica e chiave simmetrica possano essere incorporati senza dover per questo creare un nuovo protocollo;
- **Efficienza:** Le operazioni di crittografia tendono a essere molto laboriose per la CPU, particolarmente le operazioni con le chiavi pubbliche. Per questa ragione l'SSL ha incorporato uno schema di session caching opzionale per ridurre carico computazionale quanto possibile (riciclo delle sessioni, uso di cifratura simmetrica). Particolare attenzione è stata posta nel ridurre l'attività sulla rete.

4. ARCHITETTURA PER UN WEB SICURO

Il protocollo SSL è composto da:

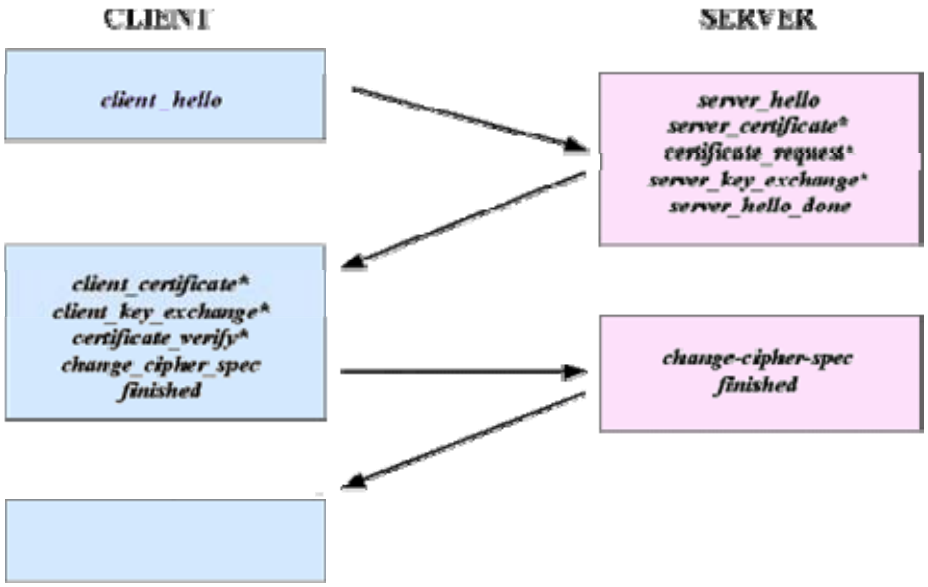


- **SSL Record:** Permette lo scambio di record come unità elementare di trasmissione;
- **SSL Handshake:** Permette di contrattare i parametri relativi al meccanismo di sicurezza;
- **SSL Change Cipher Spec:** Innesca il cambio di cifrario usato, in particolare determina l'inizio delle comunicazioni cifrate ;
- **SSL Alert:** Invia segnalazioni di errori e di situazioni di warning;

4.1 PROTOCOLLO SSL HANDSHAKE

Il protocollo SSL usa una combinazione di chiavi pubbliche e chiavi simmetriche. La cifratura a chiave simmetrica è molto più veloce della cifratura a chiave pubblica, anche se quest'ultima provvede ad una tecnica di autenticazione migliore. Una sessione SSL inizia sempre con uno scambio di messaggi chiamati di *SSL handshake*. L'handshake consente al server di autenticarsi al client usando una tecnica a chiave pubblica, quindi permette al client ed al server di cooperare per la creazione delle chiavi simmetriche usate per una veloce cifratura, decifratura e controllo delle intrusione durante la sessione avviata. Eventualmente, l'handshake permette anche al client di autenticarsi al server.Nel protocollo SSL Handshake l'avvio di una nuova connessione può avvenire o da parte del client o da parte del server. Se è il client ad iniziare, allora questo invierà un messaggio di *client hello*, iniziando così la *fase di Hello*, e si porrà in attesa della risposta del server che avviene con un messaggio di *server hello*. Nel caso in cui sia il server ad iniziare la connessione, questo invierà un messaggio di *hello request* per richiedere al client di iniziare la *fase di Hello*. Con lo scambio di questi messaggi, il client ed il server si accorderanno sugli algoritmi da usare per la generazione delle chiavi; in particolare il client ne proporrà una lista, quindi sarà il server a decidere quale di essi dovrà essere utilizzato. A questo punto può iniziare o meno, a seconda del metodo di autenticazione impiegato (autenticazione di entrambe le parti, autenticazione del server con client non autenticato, totale anonimato), uno scambio di certificati tra client e server. L'autenticazione è un controllo che si può effettuare per provare l'identità di un client o di un server. Un client abilitato può controllare che il certificato del server sia valido e che sia stato firmato da un'autorità fidata (CA). Questa conferma può essere utile, per esempio, se un utente invia il numero di carta di credito e vuole controllare l'identità del ricevente.Allo stesso modo un client può essere autenticato, questo potrebbe essere utile se il server è ad esempio una banca che deve inviare dati finanziari confidenziali ad un suo cliente che necessita quindi di essere autenticato.
E' importante notare che sia l'autenticazione del client che quella del server implica la cifratura di alcuni dati condivisi con una chiave pubblica o privata e la decifratura con la chiave corrispondente. Nel caso dell'autenticazione del server, il client deve cifrare dei dati segreti con la chiave pubblica del server. Solo la corrispondente chiave privata può correttamente decifrare il segreto, così il client ha delle assicurazioni sulla reale identità del server poichè solo lui può possedere tale chiave. Altrimenti il server non potrà generare le chiavi simmetriche richieste per la sessione, che verrà così terminata. In caso di autenticazione del client, questi deve cifrare alcuni valori casuali condivisi con la sua chiave privata, creando in pratica una firma. La chiave pubblica nel certificato del client può facilmente convalidare tale firma se il certificato è autentico, in caso contario la sessione verrà terminata. Avvenuta l'autenticazione si procede con la generazione delle chiavi per la cifratura e per l'autenticazione dei dati provenienti dal livello di applicazione, tutto questo attraverso i messaggi di *server key exchange* e *client key exchange*. Terminata questa operazione, il server annuncerà la fine della *fase di Hello* al client, con l'invio di un messaggio di *server hello done*, dopodichè con l'invio di un messaggio di *change CipherSpec* entrambi controlleranno la correttezza dei dati ricevuti e se tutto è avvenuto in maniera corretta, da questo punto in poi useranno gli algoritmi di sicurezza concordati. La fase di handshake terminerà con l'invio, da entrambe le parti, di un messaggio di *finished* che sarà il primo dato ad essere cifrato.

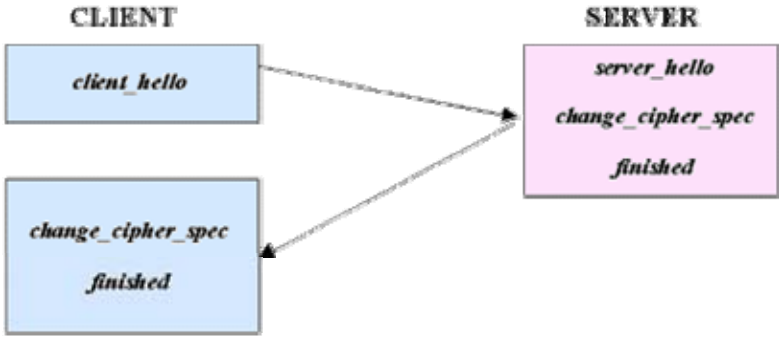
Le fasi che caratterizzano l'handshake sono rappresentate nella figura seguente:



Schema di Handshake

I messaggi contrassegnati con "*" non sono obbligatori ed il messaggio *change CipherSpec* non fa parte del protocollo handshake ma viene inviato in quella posizione.

Il client ed il server possono decidere di riabilitare una precedente sessione o di duplicarne una esistente invece di negoziare di nuovo i parametri di sicurezza; in questo caso si parla di riesumazione di sessioni:



Schema di Handshake con riesumazione

Il client manda un *client hello* usando il Session ID della sessione da riesumare; il server allora controlla la sua session cache per trovare una corrispondenza: se questa è trovata il server manda un *server hello* con lo stesso Session ID. Adesso sia il client che il server devono mandare un messaggio di *change CipherSpec* e procedere direttamente fino al messaggio di *finished*. Una volta che il ripristino è completo, il client ed il server possono iniziare. Se il server non trova una corrispondenza di Session ID nella propria session cache, allora genererà un nuovo Session ID, e verrà eseguito un handshake completo.

4.1.1 MESSAGGI DI HANDSHAKE

I messaggi usati dal protocollo SSL Handshake sono descritti nei seguenti paragrafi, nell'ordine in cui devono essere spediti.

➢ 4.1.1.1 MESSAGGI DI HELLO

I messaggi di Hello vengono utilizzati nella fase iniziale del protocollo SSL Handshake per avviare una nuova connessione.
L'inizio della connessione tra il client ed il server può avvenire in due modi:

- con l'invio da parte del server del messaggio *hello request*. Questo messaggio può essere inviato dal server in ogni momento, ma viene ignorato dal client se il protocollo handshake è già iniziato;
- con l'invio del messaggio *client hello*, da parte del client, al quale il server deve rispondere con un messaggio di *server hello*. Qualora il server non dovesse rispondere si verifica un errore fatale e la connessione fallisce.

Hello request

Il messaggio di hello request è una richiesta di comunicazione che il server invia al client. Se il client risponde alla richiesta del server, con il messaggio *client hello*, si avvia la fase di negoziazione. Dopo aver mandato un *hello request* il server non deve ripeterlo finché la fase di handshake non sarà terminata.

Client hello

Il messaggio di *client hello* è utilizzato da un client per:

- avviare una nuova connessione con un server;
- rispondere ad un hello request del server;
- rinegoziare i parametri di sicurezza di una connessione esistente.

Il messaggio *client hello* ha i seguenti campi:

- **Protocol version:** sono due byte utilizzati per indicare la versione di SSL in uso;
- **Random byte:** sono dei byte casuali generati dal client che vengono memorizzati nella struttura seguente:

  ```
  struct {
      uint32 gmt_unix_time;
      opaque random_byte[28]
  } Random;
  ```

  dove gmt_unix_time è la data e l'ora corrente nel formato UNIX standard a 32-bit e random_byte è una sequenza di 28 byte prodotti da un generatore di numeri casuali.

- **Session identifier:** sono 32 byte contenenti l'identificativo di una precedente sessione di una connessione che potrebbe essere riesumata. Se sono tutti zero, indicano che si tratta di una nuova sessione.
- **Lista delle CipherSuite (cipher_suites):** è la lista contenente le combinazioni di algoritmi di crittografia supportati dal client, ordinata secondo le sue preferenze. Il server effettuerà una scelta tra gli algoritmi presenti nella lista, se nessuno di questi è supportato verrà restituito un messaggio di errore e la connessione verrà chiusa.
- **Lista di compression method (compression_methods):** è una lista di algoritmi di compressione, in ordine di preferenza, supportati dal client. Se il server non supporta nessuno di quelli specificati dal client, la sessione fallisce.

La struttura complessiva del messaggio di *client hello* è quindi la seguente:

```
struct {
    Protocol version client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites;
    CompressionMethod compression_methods;
} ClientHello;
```

Dopo aver mandato un *client hello*, il client aspetta un messaggio di server hello, qualsiasi altro messaggio di handshake provocherebbe un errore fatale, tranne un hello request che sarebbe ignorato.

Server hello

Il server processa il messaggio di **client hello** e risponde con un **server hello** o con un **alert** per il fallimento dell'handshake. La struttura di questo messaggio è simile a quella del **client hello**:

```
struct {
        ProtocolVersion server_version;
        Random random;
        SessionID session_id;
        CipherSuite cipher_suite;
        CompressionMethod compression_method;
} ServerHello;
```

ove:

- **Protocol version**: si tratta di una coppia di byte che rappresenta la versione del protocollo scelto. Questo valore corrisponde al minimo tra la versione del protocollo proposta dal client e la massima supportata dal server;
- **Random byte**: sono dei byte casuali generati dal server che vengono memorizzati nella struttura **Random**, la cui definizione è uguale a quella usata per memorizzare i byte casuali generati dal client;
- **Session identifier**: identifica la sessione che deve essere avviata. Se l'identificativo di sessione inviato dal client è diverso dal valore zero, il server cercherà nella sua session cache per trovare un valore di riscontro: se lo trova, risponderà con lo stesso valore indicato nel *client hello* e la vecchia sessione verrà riesumata. Se non lo trova, il server ritornerà un identificativo vuoto per indicare che la sessione non era memorizzata e quindi non può essere ripresa. Nel caso in cui l'identificativo nel *client hello* è pari a zero il server setterà un nuovo identificativo per la sessione da avviare.
- **CipherSuite**: questa coppia di byte rappresenta la famiglia di algoritmi scelta dal server tra quelli proposti dal client.
- **Compression method**: rappresenta il metodo di compressione scelto dal server tra quelli proposti dal client.

➢ 4.1.1.2 SERVER CERTIFICATE

Se il server deve essere autenticato, come in genere accade, esso manda il suo certificato immediatamente di seguito al **server hello,** con il messaggio **server certificate.** Un certificato è una struttura dati composta da dati in chiaro come: una chiave pubblica, una stringa identificativa e dalla firma di un autorità che lega chiave ed identità. Il tipo di certificato deve essere appropriato all'algoritmo per lo scambio di chiavi selezionato nella CipherSuite. Quindi nel certificato ci saranno le informazioni utili al funzionamento dell'algoritmo. Generalmente viene utilizzato il certificato *X.509.v3*.

➢ 4.1.1.3 CERTIFICATE REQUEST

Il messaggio *certificate request* è inviato dal server al client per richiederne il certificato utilizzato per fornire le informazioni necessarie all'algoritmo di scambio di chiavi. In particolare il messaggio *certificate request* contiene una lista di tipi di certificati, ordinati secondo le preferenze del server, ed una lista di nomi di autorità fidate che emettono certificati. Se un server anonimo richiede l'identificazione del client si verifica un errore fatale.

➢ 4.1.1.4 MESSAGGIO DI SERVER KEY EXCHANGE

Se il messaggio *server certificate* non deve essere inviato o se non contiene una chiave pubblica di cifratura, ma solo una per la firma, o infine se il server non ha certificato, quest'ultimo invia il messaggio *server key exchange* che contiene o i parametri per l'accordo di chiavi Diffie-Hellman o i parametri per una cifratura RSA oppure i parametri per Fortezza. La scelta di uno di questi tre dipende dalla CipherSuite selezionata; se viene inviato uno dei primi due allora ci sarà anche una firma nel caso in cui il server abbia presentato un certificato.

➢ 4.1.1.5 SERVER HELLO DONE

Il messaggio di *server hello done* è inviato dal server al client per indicare che la *fase di Hello* dell'handshake è completata. Dopo l'invio del messaggio il server si mette in attesa di una risposta dal client, il quale verificherà che il server abbia una valida certificazione, se richiesta, e controlla che i parametri del *server hello* siano accettabili.

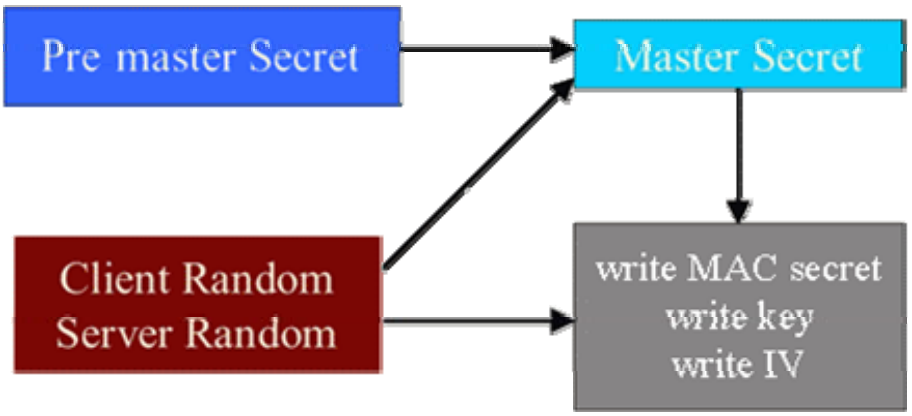
➢ 4.1.1.6 CLIENT CERTIFICATE

Il messaggio *client certificate* è il primo messaggio che il client può mandare dopo aver ricevuto il messaggio *server hello done* e la richiesta di un certificato da parte del server. Se non è disponibile un certificato, il client lo segnalerà per mezzo di un messaggio di alert, al quale il server potrebbe rispondere con un errore fatale se è necessaria l'autenticazione del client.

➢ 4.1.1.7 MESSAGGIO DI CLIENT KEY EXCHANGE

Il messaggio *client key exchange* deve essere inviato obbligatoriamente dal client al fine di fissare il pre master secret. Il pre master secret è utilizzato per calcolare il master secret, cioè un valore condiviso da client e server necessario alla generazione di:

- **Chiavi MAC:**
  - **server_write_MAC_secret:** chiave segreta che sarà utilizzata per calcolare il MAC sui blocchi generati nel livello inferiore del protocollo e inviati dal server;
  - **client_write_MAC_secret:** chiave segreta che sarà utilizzata per calcolare il MAC sui blocchi generati nel livello inferiore del protocollo e inviati dal client;
- **Chiavi simmetriche:**
  - **server_write_key:** chiave simmetrica di sessione per i dati cifrati dal server e messi in chiaro dal client;
  - **client_write_key:** chiave simmetrica di sessione per i dati cifrati dal client e messi in chiaro dal server;
- **Vettori di inizializzazione:**
  - **server_write_IV:** sequenza di byte utilizzata per le cifrature in modalità CBC del server;
  - **client_write_IV:** sequenza di byte utilizzata per le cifrature in modalità CBC del client;



Il contenuto del messaggio di *client key exchange* dipende dall'algoritmo per lo scambio di chiavi specificato nella CipherSuite:

- **RSA:** il client genera il pre master secret di 48 byte, lo cifra con la chiave pubblica presa dal certificato del server o con una chiave temporanea RSA presa dal messaggio *server key exchange* e spedisce il risultato in un messaggio cifrato. Il server userà la sua chiave privata per mettere in chiaro il pre master secret
- **Fortezza KEA:** il client deriva una chiave simmetrica, denominata Token Encryption Key (TEK), usando Fortezza Key Exchange Algorithm (KEA). In tali calcoli sono usate sia la chiave pubblica del server, ricavata dal suo certificato, sia i parametri privati del client. Il client genera la chiave di sessione, la protegge usando il TEK e la spedisce al server, insieme al vettore di inizializzazione (IV) usato nel CBC; quindi genera un pre master secret di 48 byte, lo cifra usando il TEK e lo spedisce. Il server metterà in chiaro il pre master secret e lo convertirà in master secret; che in questo caso verrà usato solo per i calcoli del MAC.
- **Diffie-Hellman:** è usato un convenzionale algoritmo Diffie-Hellman per negoziare una chiave che sarà usata come pre master secret.

➢ 4.1.1.8 CERTIFICATE VERIFY

Se il client in precedenza ha inviato un certificato, ora col messaggio *certificate verify* invierà un hash dei messaggi di handshake, scambiati a partire da *client hello* fino a questo punto, e della master secret, in modo che il server possa verificarne l'autenticità.

➢ 4.1.1.9 CHANGE CIPHERSPEC

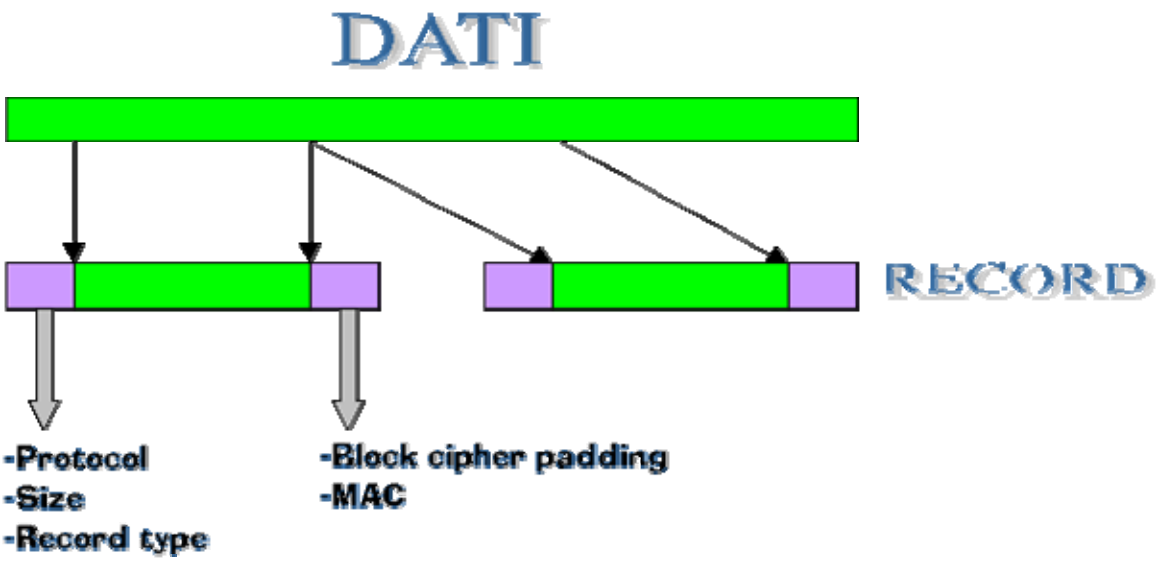
Questo messaggio ha lo scopo di segnalare i cambiamenti delle strategie di cifratura. Esso è inviato sia dal client che dal server per comunicare alla parte ricevente che i dati seguenti saranno protetti dalle chiavi e dal CipherSpec appena negoziato. Il client invia un messaggio di *change CipherSpec* dopo l'invio dei messaggi di *key exchange* e *certificate verify* della fase di handshake, ed il server ne invia uno dopo che il messaggio di *key exchange* è stato ricevuto dal client con successo. Quando si riprende una vecchia sessione il messaggio di *change CipherSpec* è inviato dopo i messaggi di Hello.

- **4.1.1.10 FINISHED**

Questo messaggio è sempre mandato dopo un messaggio *change CipherSpec* per verificare che i processi di scambio di chiavi e autenticazione abbiano avuto successo, se non è preceduto da questo si verifica un errore fatale. Questo è il primo messaggio protetto con l'algoritmo negoziato. Adesso la fase di handshake è completata ed il client ed il server possono cominciare a scambiare i dati del livello applicazione.

4.2 PROTOCOLLO SSL RECORD

In linea generale il protocollo SSL prende i messaggi che devono essere trasmessi, li frammenta in blocchi di dati (record di 214 byte o meno), opzionalmente li comprime, applica un MAC (Message Authentication Code che viene utilizzato per il protocollo dell'integrità dei dati), li cifra, e trasmette il risultato. I dati ricevuti vengono decifrati, verificati, decompressi, e riassemblati, quindi vengono trasmessi al livello più alto.



4.2.1 SSLPLAINTEXT RECORD

I blocchi creati dal Protocollo SSL Record sono organizzati in record SSLPlaintext la cui struttura è illustrata di seguito.

```
struct {
        ContentType type;
        ProtocolVersion version;
        uint16 length;
        opaque fragment[SSLPlaintext.length];
} SSLPlaintext;
```

I campi della struttura SSLPlaintext hanno il seguente significato:

- **type:** indica quale tra i protocolli di livello più alto, cioè SSL Handshake o HTTP, è usato per processare il frammento.
- **version:** la versione del protocollo impiegato.
- **length:** la lunghezza in byte del frammento SSLPlaintext che segue. Non può eccedere 2^{14} byte.
- **fragment:** i dati provenienti dal livello applicazione. Questi dati sono trasparenti e considerati come un blocco indipendente per essere poi trattati dal protocollo di livello più alto specificato nel campo type.

4.2.2 COMPRESSIONE E DECOMPRESSIONE DEL SSLPLAINTEXT RECORD

Tutti gli SSLPlaintext record sono compressi. L'algoritmo di compressione trasforma un SSLPlaintext record in un SSLCompressed record definito di seguito:

```
struct {
        ContentType type; /* come SSLPlaintext.type */
        ProtocolVersion version; /* come SSLPlaintext.version*/
        uint16 length;
        opaque fragment[SSLCompressed.length];
} SSLCompressed;
```

Significato dei campi:

- **length:** la lunghezza in byte del frammento SSLCompressed. La lunghezza non deve eccedere $2^{14} + 1024$.
- **fragment:** la forma compressa del SSLPlaintext.fragment.

La compressione deve essere senza perdita di dati e non può aumentare la lunghezza del contenuto più di 1024 byte. Se la decompressione del campo fragment relativo al SSLCompressed record fosse lungo più di 2^{14} byte, verrà segnalato un errore irreversibile.

4.2.3 PROTEZIONE DEGLI SSLCOMPRESSED RECORD

Tutti i record sono protetti usando gli algoritmi di crittografia e MAC definiti nel corrente CipherSpec. Le tecniche usate per le operazioni di crittografia e di MAC trasformano una struttura SSLCompressed in una SSLCiphertext. La decifratura inverte tale trasformazione. Le trasmissioni includono anche un numero di sequenza, in modo che i messaggi persi, alterati o intrusi siano rilevabili.

La struttura SSLCiphertext è la seguente:

```
struct {
        ContentType type;
        ProtocolVersion version;
        uint16 length;
        select (CipherSpec.cipher_type) {
            case stream: GenericStreamCipher;
            case block: GenericBlockCipher;
        } fragment;
} SSLCiphertext;
```

Il significato dei campi è il seguente:

- **type:** come il SSLCompressed.type .
- **version:** come il SSLCompressed.version .
- **length:** la lunghezza in byte del SSLCiphertext.fragment non può eccedere 2^{14} + 2048.
- **fragment:** la forma cifrata del SSLCompressed.fragment, incluso il MAC. Vengono selezionate per tale campo due strutture diverse a seconda che sia stato scelto l'uso di un algoritmo di crittografia che lavora su tutto lo stream di bit dei dati, GenericStreamCipher, o su dei blocchi, GenericBlockCipher.

4.3 PROTOCOLLO SSL CHANGE CIPHER SPEC

Questo protocollo è finalizzato a segnalare le transizioni nelle strategie di crittografia. Il protocollo consiste di un singolo messaggio, che è cifrato e compresso secondo il `Current` (e non il `pending`) `CipherSpec`.Il messaggio di **change cipher spec** è mandato sia dal client che dal server per notificare all'altro che da quel momento in poi i records saranno protetti usando le nuove, appena negoziate CipherSpec e chiavi.

4.4 PROTOCOLLO SSL ALERT

I messaggi di alert sono utilizzati per notificare eccezioni che possono avvenire nella comunicazione. Essi contengono il livello di severità e una descrizione dell'evento occorso. Ad esempio un messaggio con livello "fatal" si risolve con l'immediata terminazione della sessione. Come gli altri messaggi anche quelli di alert sono cifrati e compressi.

Le strutture usate sono le seguenti:

```
enum {
      warning(1), fatal(2), (255)
} AlertLevel;

enum {
      close_notify(0), unexpected_message(10), bad_record_mac(20), decompression_failure(30),
       handshake_failure(40), no_certificate(41), bad_certificate(42), unsupported_certificate(43),
       certificate_revoked(44), certificate_expired(45), certificate_unknown(46), illegal_parameter(47), (255)
} AlertDescription;


struct {
      AlertLevel level;
      AlertDescription description;
} Alert;
```

I messaggi di alert definiti nella struttura AlertDescription sono descritti di seguito:

- **close_notify:** notifica al ricevente che il mittente non trasmetterà più su quella connessione. **unexpected_message:** un messaggio inappropriato è stato ricevuto. Questo tipo di allerta è sempre fatale.
- **bad_record_mac:** questa allerta è spedita quando un record è ricevuto con un errato MAC. Questo tipo di allerta è sempre fatale.
- **decompression_failure:** la funzione di decompressione ha ricevuto un errato input, ad esempio dati che si espandono oltre la lunghezza loro assegnata. Questo tipo di allerta è sempre fatale.
- **handshake_failure:** indica che il mittente è incapace di negoziare un set accettabile di parametri di sicurezza tra quelli possibili. Questo tipo di allerta è sempre fatale.
- **no_certificate:** può essere mandato in risposta ad un *certificate request* se non è disponibile una certificazione appropriata.
- **bad_certificate:** in caso di certificazione errata (ad esempio se contiene una firma che non è verificata correttamente).
- **unsupported_certicate:** se una certificazione è di un tipo non supportato.
- **certificate_revoked:** una certificazione è stata revocata dal suo firmatario.
- **certificate_expired:** una certificazione è scaduta o attualmente non valida.
- **certificate_unknown:** qualche altro non specificato problema è sorto nel processare la certificazione.
- **illegal_parameter:** un campo del handshake è di dimensione errata o inconsistente con altri campi. Questo tipo di allerta è sempre fatale.

4.4.1 ALLERTA DI CHIUSURA

Per chiudere una connessione, il client ed il server devono avviare lo scambio dei messaggi di chiusura. In particolare il messaggio *close notify* notifica al ricevente che il mittente non trasmetterà più su quella connessione. La sessione diviene non riesumabile se la connessione è terminata senza il dovuto messaggio close_notify con livello warning.

5. CONSIDERAZIONI FINALI E DEBOLEZZE DI SSL

SSL esamina esplicitamente un numero di attacchi, incluso l'attacco forza bruta, crittoanalisi, replay e l'uomo in mezzo. SSL è progettato per agire a livello di rete: ciò significa che non protegge da attacchi agli host, per i quali sarebbe consigliabile proteggersi con un package del tipo Tripwire. I Tripwire usano funzioni hash sicure per assicurare che i documenti non siano cambiati rispetto ad una versione di riferimento protetta in scrittura.Ci sono diversi punti di SSL che, pur non essendo critici, potrebbero offrire ulteriore sicurezza. Ad esempio nel protocollo *SSL Handshake* alcuni dati spediti con il messaggio *Client hello* potrebbero essere spediti in un secondo momento, crittografati. Nel protocollo *SSL Record*, un errato *MAC* non dovrebbe far terminare la connessione ma causare una richiesta di ripetizione del messaggio: infatti ci sono pochi attacchi che possono trarre vantaggio dalla duplice spedizione di dati, mentre terminando la connessione si apre la strada agli attacchi basati sul servizio negato. E' importante notare tuttavia che questo approccio alla problematica della sicurezza sembra concepito in modo da permettere facili aggiornamenti ed il supporto a nuovi algoritmi di crittografia, senza stravolgere la struttura di SSL.

6. RIFERIMENTI

1. http://telemat.die.unifi.it/book/Internet/Security/elab3.htm
2. http://it.wikipedia.org/wiki/Secure_Sockets_Layer
3. http://telemat.die.unifi.it/book/corso_telematica/lez_170/grp_6.html
4. http://www.dia.unisa.it/professori/masucci/sicurezza/ssl.pdf
5. http://www.esiaonline.it/sicurezza/oSSL.asp
6. http://www.alground.com/site/modules/sections/index.php?op=viewarticle&artid=20