

SSL/TLS — Protocollo Handshake

1 Protocollo Handshake

Il protocollo Handshake, che viene eseguito prima di inviare qualunque dato applicativo, è la parte più complessa di TLS, poiché deve garantire interoperabilità tra client e server che non si conoscono a priori e potrebbero supportare schemi crittografici diversi. Infatti, esso permette al client e al server di:

- Negoziare la **cipher suite**, l'insieme degli algoritmi crittografici che verranno utilizzati nella sessione, ovvero:
 - il metodo per lo scambio delle chiavi;
 - l'algoritmo di cifratura per il protocollo Record;
 - l'algoritmo MAC per il protocollo Record.
- Eseguire l'autenticazione tramite lo scambio di certificati digitali X.509. Client e server possono *autenticarsi a vicenda*, oppure si può *autenticare solo il server* (questo è ciò che tipicamente avviene in ambito Web) o infine si può svolgere una *sessione anonima* senza autenticazione (il che non è però consigliato).
- Scambiare le informazioni necessarie per la generazione delle chiavi segrete.

2 Generazione delle chiavi

Come già detto, SSL/TLS richiede quattro diverse chiavi segrete condivise:

- una *server write key* per la cifratura nella direzione del traffico dal server al client;
- una *server write MAC key* per il calcolo e la verifica dei MAC nella direzione dal server al client;
- una *client write key* per la cifratura nella direzione dal client al server;
- una *client write MAC key* per il calcolo e la verifica dei MAC nella direzione dal client al server.

Inoltre, la cifratura (tipicamente in modalità CBC) richiede due vettori di inizializzazione (per le due direzioni del flusso di dati).

Questi 6 valori formano insieme il **keyblock**, che viene visto come un “blocco unico” (di dimensioni variabili in base agli algoritmi impiegati) e generato a partire da dei valori scambiati nell’handshake: un **pre-master secret** e due **valori random** (nonce).

3 Struttura dell’handshake

Il protocollo Handshake è composto da un insieme di messaggi che vengono scambiati nel corso di quattro fasi:

1. negoziazione degli algoritmi di sicurezza che verranno utilizzati nella sessione;
2. autenticazione del server (opzionale ma consigliata) e scambio delle informazioni per la generazione delle chiavi;
3. autenticazione del client (opzionale) e scambio delle informazioni per la generazione delle chiavi;
4. fine.

Il seguente schema mostra la struttura generale del protocollo, ma gli specifici messaggi inviati nelle fasi 2, 3 e 4 variano in funzione del metodo di scambio delle chiavi stabilito nella fase 1 (i messaggi non sempre presenti sono qui indicati tra parentesi quadre):

- (1) $C \rightarrow S$: ClientHello
 $S \rightarrow C$: ServerHello
- (2) $S \rightarrow C$: [Certificate]
 $S \rightarrow C$: [ServerKeyExchange]
 $S \rightarrow C$: [CertificateRequest]
 $S \rightarrow C$: ServerHelloDone
- (3) $C \rightarrow S$: [Certificate]
 $C \rightarrow S$: ClientKeyExchange
 $C \rightarrow S$: [CertificateVerify]
- (4) $C \rightarrow S$: ChangeCipherSpec
 $C \rightarrow S$: Finished
 $S \rightarrow C$: ChangeCipherSpec
 $S \rightarrow C$: Finished

Si noti che l’handshake è iniziato sempre dal client (perché nel modello client-server è il client a fare una richiesta al server).

4 Fase 1: negoziazione della cipher suite

La fase 1 dell'handshake comprende i messaggi ClientHello e ServerHello, che servono principalmente per negoziare la cipher suite da usare nella sessione.

- (1) $C \rightarrow S$: ClientHello
 $S \rightarrow C$: ServerHello

Il messaggio **ClientHello** indica, *in plaintext*:

- la più alta versione di SSL/TLS supportata dal client;
- un numero random (nonce) di 32 byte generato dal client, che verrà utilizzato nella generazione del keyblock;
- opzionalmente, un ID di una sessione già negoziata che si vuole riattivare;
- una lista di cipher suite supportate, dove ciascuna cipher suite è una terna di metodo di scambio delle chiavi, algoritmo di cifratura e algoritmo per il calcolo dei MAC;
- opzionalmente, una lista di metodi di compressione supportati.

Il messaggio **ServerHello** con cui il server risponde ha una struttura simile. Infatti, i suoi contenuti più importanti sono:

- la versione di SSL/TLS selezionata, che è la più alta supportata sia dal client che dal server;
- la cipher suite selezionata, che è la più sicura tra quelle supportate sia dal client che dal server;
- un valore random di 32 byte generato dal server, anch'esso da utilizzare nella generazione del keyblock.

5 Metodi di scambio delle chiavi

SSL/TLS supporta diversi metodi di scambio delle chiavi, ma tutti questi metodi hanno lo scopo di far condividere al client e al server un segreto chiamato **pre-master secret**, che successivamente verrà utilizzato, insieme ai numeri random scambiati nella fase 1, per la generazione del keyblock.

I metodi per lo scambio delle chiavi si suddividono in quelli basati su RSA e quelli basati su Diffie-Hellman:

- nel caso di RSA il client genera il pre-master secret e lo condivide inviandolo cifrato con la chiave pubblica del server;

- nel caso di Diffie-Hellman il client e il server calcolano localmente il pre-master secret come risultato appunto dell'algoritmo Diffie-Hellman.

Esistono poi diversi metodi di scambio delle chiavi basati su ciascuno di questi due algoritmi, perché tali algoritmi possono essere combinati in vari modi con l'autenticazione basata sui certificati digitali. In particolare, TLS supporta cinque metodi di scambio delle chiavi,

- *RSA*,
- *RSA temp*,
- *fixed Diffie-Hellman*,
- *ephemeral Diffie-Hellman*,
- *anonymous Diffie-Hellman*,

che adesso verranno presentati nel dettaglio.

5.1 RSA

Con il metodo **RSA** il pre-master secret viene generato dal client e inviato al server cifrandolo con la chiave pubblica KP_S del server stesso.

Questo metodo prevede che il server invii al client il certificato contenente la propria chiave pubblica, o eventualmente la catena di certificati che servono al client per riconoscere il certificato del server. Così, verificando che il certificato del server sia emesso da una CA riconosciuta (eventualmente tramite la catena di certificati) e non sia scaduto o revocato, il client autentica il server. Inoltre, dal certificato il client ottiene la chiave KP_S che deve usare per inviare il pre-master secret.

I messaggi scambiati nelle fasi 2 e 3 con questo metodo sono:

- (2) $S \rightarrow C$: Certificate = $\langle KP_S, S \rangle_{CA}$
 $S \rightarrow C$: [CertificateRequest]
 $S \rightarrow C$: ServerHelloDone
- (3) $C \rightarrow S$: [Certificate]
 $C \rightarrow S$: ClientKeyExchange = {pre-master} $_{KP_S}$
 $C \rightarrow S$: [CertificateVerify]

Tra questi, i più importanti sono:

- il messaggio Certificate con il quale il server invia al client il proprio certificato, qui rappresentato usando la notazione $\langle KP_S, S \rangle_{CA}$ (e non la notazione $CA \langle S \rangle$ adottata in precedenza) per mettere in evidenza la chiave pubblica che esso (come ogni certificato) contiene;

- il messaggio ClientKeyExchange tramite cui il client invia al server il pre-master secret generato, cifrandolo con la chiave KP_S ottenuta dal certificato.

Il messaggio CertificateRequest del server e i messaggi Certificate e CertificateVerify del client vengono invece scambiati solo se si desidera autenticare il client, come si vedrà più avanti.

5.2 RSA temp

Potrebbe essere che la coppia di chiavi KP_S e KR_S associate al certificato del server siano valide solo per la firma, e non per la cifratura. In questo caso il metodo RSA non si può applicare, e bisogna invece usare **RSA temp**, che prevede i seguenti passi:

1. Il server genera localmente una coppia di chiavi RSA temporanee, KP_T e KR_T , usate solo per questa sessione.
2. Il server invia al client la chiave pubblica temporanea KP_T , firmata con la chiave privata RSA KR_S valida per la firma, e allega il certificato della corrispondente chiave pubblica KP_S .
3. Il client verifica il certificato per autenticare il server, poi usa la chiave pubblica presente nel certificato per verificare la firma della chiave pubblica temporanea.
4. Il client genera il pre-master secret e lo invia al server cifrandolo con la nuova chiave pubblica KP_T .

Dal punto di vista dei messaggi scambiati, RSA temp funziona in questo modo:

- (2) $S \rightarrow C$: Certificate = $\langle KP_S, S \rangle_{CA}$
 $S \rightarrow C$: ServerKeyExchange = $KP_T, \{KP_T\}_{KR_S}$
 $S \rightarrow C$: [CertificateRequest]
 $S \rightarrow C$: ServerHelloDone
- (3) $C \rightarrow S$: [Certificate]
 $C \rightarrow S$: ClientKeyExchange = {pre-master} $_{KP_T}$
 $C \rightarrow S$: [CertificateVerify]

- Nel messaggio Certificate il server invia il certificato della propria chiave pubblica KP_S valida solo per la firma.
- Nel messaggio ServerKeyExchange il server invia la chiave pubblica temporanea KP_T e la firma $\{KP_T\}_{KR_S}$ di tale chiave generata con la chiave certificata per la firma.
- Nel messaggio ClientKeyExchange il client invia al server il pre-master secret cifrato con la chiave pubblica temporanea (invece che direttamente con la chiave pubblica certificata).

Per il resto, i messaggi sono uguali allo scambio di chiavi RSA.

È importante che la chiave KP_T sia temporanea, ogni volta nuova, per evitare attacchi a replay, e che sia firmata per evitare che l'attaccante possa sostituirla.

5.3 Fixed Diffie-Hellman

Il metodo **fixed Diffie-Hellman** genera il pre-master secret utilizzando l'algoritmo Diffie-Hellman, con parametri pubblici g , q e Y_S del server fissi e contenuti in un certificato rilasciato da una CA, $\langle g, q, Y_S, S \rangle_{CA}$. Il server invia tale certificato al client nel messaggio Certificate, e il client risponde comunicando nel messaggio ClientKeyExchange il proprio parametro pubblico Y_C , eventualmente preceduto dal messaggio Certificate contenente il certificato di tale parametro, se è richiesta l'autenticazione del client.

- (2) $S \rightarrow C$: Certificate = $\langle g, q, Y_S, S \rangle_{CA}$
 $S \rightarrow C$: [CertificateRequest]
 $S \rightarrow C$: ServerHelloDone
- (3) $C \rightarrow S$: [Certificate]
 $C \rightarrow S$: ClientKeyExchange = Y_C
 $C \rightarrow S$: [CertificateVerify]

Nel caso in cui è richiesta l'autenticazione del client, siccome il parametro Y_C deve essere certificato il client non può generarlo al momento in base ai parametri del server, ma allora deve conoscere a priori il server, sapere prima di iniziare la sessione i valori dei parametri g e q da cui Y_C dipende — in sostanza, deve ottenere un certificato specifico per la comunicazione con un determinato server, o con un insieme di server che condividono gli stessi parametri g e q . Inoltre, per fissare tutti i parametri pubblici bisogna fissare anche i parametri privati X_S e X_C (se questi ultimi variassero si otterrebbero valori di Y_S e Y_C diversi da quelli certificati), dunque il pre-master secret generato è sempre uguale in tutte le sessioni tra una determinata coppia di client e server. Grazie ai valori random scambiati nella fase 1, però, da pre-master secret uguali si ottengono comunque keyblock (e quindi chiavi di sessione) diversi.

In pratica, lo scambio di chiavi fixed Diffie-Hellman non è molto utilizzato.

5.4 Ephemeral Diffie-Hellman

Il metodo **ephemeral Diffie-Hellman**, a differenza di fixed Diffie-Hellman, applica l'algoritmo Diffie-Hellman con parametri ogni volta diversi, generati al momento (i parametri g e q possono anche essere riutilizzati, l'importante è che cambino i parametri privati X_S e X_C e di conseguenza i parametri pubblici Y_S e Y_C). Per assicurarne l'autenticità, i parametri pubblici del server sono firmati con la chiave privata del server stesso, e se è richiesta l'autenticazione del client allora anche il parametro pubblico del client è firmato con la chiave privata del client.

La validazione delle firme dei parametri richiede lo scambio dei certificati, dunque il server deve innanzitutto inviare un messaggio Certificate contenente il certificato della propria chiave pubblica, seguito dal messaggio ServerKeyExchange contenente i parametri pubblici del server e la loro firma.

- (2) $S \rightarrow C$: Certificate = $\langle KP_S, S \rangle_{CA}$
- $S \rightarrow C$: ServerKeyExchange = $g, q, Y_S, \{g, q, Y_S\}_{KR_S}$
- $S \rightarrow C$: [CertificateRequest]
- $S \rightarrow C$: ServerHelloDone

Il client verifica l'identità del server mediante il certificato, usa la chiave pubblica contenuta nel certificato per verificare la firma dei parametri, poi genera il proprio parametro Y_C e lo invia al server nel messaggio ClientKeyExchange, eventualmente firmato con la propria chiave privata e preceduto da un messaggio Certificate contenente il proprio certificato, se il server ha richiesto l'autenticazione del client.

- (3) $C \rightarrow S$: [Certificate]
- $C \rightarrow S$: ClientKeyExchange = Y_C
- $C \rightarrow S$: [CertificateVerify]

5.5 Anonymous Diffie-Hellman

Il metodo **anonymous Diffie-Hellman** è simile all'ephemeral Diffie-Hellman, ma non prevede nessun tipo di autenticazione né del server né del client, cioè omette lo scambio dei certificati e le firme dei parametri dell'algoritmo Diffie-Hellman:

- (2) $S \rightarrow C$: ServerKeyExchange = g, q, Y_S
- $S \rightarrow C$: ServerHelloDone
- (3) $C \rightarrow S$: ClientKeyExchange = Y_C

Il fatto che i parametri non siano firmati significa che un attaccante potrebbe modificarli, portando i due host comunicanti a generare keyblock diversi, "sbagliati". Tuttavia, come si vedrà a breve, questo non è un grave problema, perché alla fine dell'handshake client e server si scambiano dei messaggi Finished che servono proprio a verificare di aver entrambi generato le stesse chiavi, dunque un attacco di questo tipo sarebbe rilevato immediatamente, prima della trasmissione dei dati applicativi.

Il vero problema dello scambio di chiavi anonymous Diffie-Hellman è appunto l'assenza di autenticazione, il fatto che il client si fidi dell'identità del server senza validarla. Questo è infatti l'unico metodo nel quale non si richiede l'autenticazione neanche del server (mentre l'autenticazione del client può essere omessa anche negli altri metodi).

6 Fase 2: autenticazione del server e scambio delle chiavi

Dopo aver presentato i diversi metodi di scambio delle chiavi si possono analizzare più nel dettaglio i messaggi inviati nelle fasi 2, 3 e 4.

Nella fase 2 il server invia una serie di messaggi al client:

- (2) $S \rightarrow C$: [Certificate]
- $S \rightarrow C$: [ServerKeyExchange]
- $S \rightarrow C$: [CertificateRequest]
- $S \rightarrow C$: ServerHelloDone

Il primo messaggio che il server invia nella fase 2, per i metodi di scambio delle chiavi che lo richiedono, è **Certificate**.

- Nel caso di RSA, RSA temp ed ephemeral Diffie-Hellman esso contiene il certificato X.509 v3 (o la catena di certificati) della chiave pubblica del server, la quale:
 - per RSA è una chiave pubblica RSA valida per la cifratura;
 - per RSA temp è una chiave pubblica RSA valida per la firma;
 - per ephemeral Diffie-Hellman è una chiave pubblica RSA o DSS/DSA per la firma dei parametri Diffie-Hellman.
- Nel caso di fixed Diffie-Hellman esso contiene il certificato dei parametri pubblici Diffie-Hellman associati al server.
- Nel caso di anonymous Diffie-Hellman esso non viene spedito.

Il messaggio successivo è **ServerKeyExchange**, anch'esso dipendente dal metodo di scambio delle chiavi.

- Nel caso di RSA e fixed Diffie-Hellman esso non viene inviato (perché la chiave o i parametri necessari sono già contenuti nel messaggio Certificate).
- Nel caso di RSA temp esso contiene la chiave pubblica RSA temporanea, firmata con la chiave privata RSA corrispondente al certificato.
- Nel caso di ephemeral Diffie-Hellman esso contiene i parametri pubblici Diffie-Hellman del server, firmati con la chiave privata RSA o DSS/DSA corrispondente al certificato.
- Nel caso di anonymous Diffie-Hellman esso contiene i parametri pubblici del server, senza alcuna firma.

Per impedire attacchi a replay, la firma dei parametri di sicurezza nel messaggio `ServerKeyExchange` (la chiave pubblica temporanea o i parametri Diffie-Hellman) è generata calcolando il valore di hash non solo su tali parametri, ma anche sui valori random scambiati nella fase 1:

$$H(\text{ClientHello.random} \parallel \text{ServerHello.random} \parallel \text{parametri})$$

Dopo il messaggio `ServerKeyExchange`, il server può opzionalmente richiedere l'autenticazione al client, inviando un messaggio **CertificateRequest** nel quale sono specificati:

- il tipo di certificato che il client deve inviare (cioè se esso deve contenere una chiave RSA o DSS per la firma oppure i parametri per fixed o ephemeral Diffie-Hellman, e con quali algoritmi può essere firmato il certificato stesso, ad esempio ancora RSA o DSS);
- le autorità di certificazione accettate.

Infine, il server segnala il termine della fase 2 inviando il messaggio **ServerHelloDone**.

7 Fase 3: autenticazione del client e scambio delle chiavi

Nella fase 3 il client verifica innanzitutto se il certificato e i parametri eventualmente ricevuti dal server sono validi, inviando un alert in caso di errore, altrimenti invia al server una serie di messaggi per completare l'autenticazione e lo scambio delle chiavi:

- (3) $C \rightarrow S$: [Certificate]
 $C \rightarrow S$: ClientKeyExchange
 $C \rightarrow S$: [CertificateVerify]

Se il server ha richiesto l'autenticazione del client allora il client invia in risposta un messaggio **Certificate** contenente il proprio certificato (oppure invia l'alert `no_certificate` se non dispone di un certificato del tipo richiesto).

Il messaggio successivo che il client spedisce è **ClientKeyExchange**, il cui contenuto dipende dal tipo di scambio della chiave.

- Nel caso di RSA e RSA temp contiene il pre-master secret, cifrato con la chiave pubblica estratta dal certificato (RSA) oppure con quella temporanea (RSA temp).
- Nel caso di fixed, ephemeral e anonymous Diffie-Hellman contiene il parametro pubblico Y_C . In particolare, per ephemeral Diffie-Hellman quando è richiesta l'autenticazione del client tale parametro è firmato con la chiave privata corrispondente al certificato del client.

Infine, se il client ha inviato il proprio certificato deve mandare un messaggio **CertificateVerify** per dare la prova di essere il proprietario di tale certificato. Questo messaggio contiene una firma digitale computata su tutti i precedenti messaggi scambiati durante l'handshake: si calcola il valore di hash della concatenazione dei messaggi e lo si cifra con la chiave privata del client corrispondente alla chiave pubblica contenuta nel certificato. Così, verificando la firma con la chiave pubblica contenuta nel certificato, il server ottiene la prova che il client con cui sta comunicando possiede la corrispondente chiave privata, ovvero che esso è effettivamente il proprietario del certificato. Invece, un attaccante che provasse a usare fraudolentemente il certificato del client non riuscirebbe a generare il messaggio CertificateVerify corretto, non essendo a conoscenza della chiave privata. Siccome la firma è calcolata sui messaggi precedenti essa risulta diversa in ogni sessione, dunque non sono possibili attacchi a replay.

Si osservi che il protocollo handshake prevede il messaggio CertificateVerify per la verifica del certificato del client, ma non un analogo messaggio per la verifica del certificato del server. Il motivo è che la verifica del certificato del server avviene già tramite il processo di generazione delle chiavi, nel quale (a seconda del metodo di scambio delle chiavi) si usano

- i parametri contenuti nel certificato del server, oppure
- i parametri contenuti nel ServerKeyExchange, che sono firmati con la chiave privata corrispondente a tale certificato,

dunque il server riesce a generare le chiavi corrette o a firmare il ServerKeyExchange solo se è in possesso dei parametri privati corrispondenti al certificato. Al contrario, le informazioni contenute nel certificato del client non servono per la generazione delle chiavi (anche perché questo certificato è facoltativo — le chiavi devono poter essere generate anche senza), dunque è necessario un apposito messaggio di verifica.

8 Fase 4: fine

Nella fase 4, che completa l'esecuzione del protocollo Handshake, ciascuno dei due host (prima il client e poi il server) invia un messaggio ChangeCipherSpec seguito da un messaggio Finished:

- (4) $C \rightarrow S$: ChangeCipherSpec
- $C \rightarrow S$: Finished
- $S \rightarrow C$: ChangeCipherSpec
- $S \rightarrow C$: Finished

I messaggi **ChangeCipherSpec**, definiti dall'omonimo protocollo, servono a sincronizzare tra il lato client e il lato server la chiusura dell'handshake e l'inizio della comunicazione protetta con gli algoritmi crittografici e le chiavi appena negoziati.

Invece, i messaggi **Finished** sono i primi messaggi cifrati e autenticati inviato da ciascuno dei due host e gli ultimi messaggi inviati prima di iniziare lo scambio dei dati applicativi. Essi servono a dare una prova che il server e il client abbiano generato o condiviso correttamente il pre-master secret, ricavato da esso lo stesso keyblock (dunque lo stesso insieme di chiavi) e selezionato gli stessi algoritmi crittografici. Infatti, ciascuno dei messaggi Finished contiene il MAC calcolato su tutti i precedenti messaggi scambiati durante l'handshake: se chi lo riceve riesce a decifrarlo e verificare il MAC, ottiene la prova di correttezza degli algoritmi e delle chiavi impiegati per cifratura e MAC (in una direzione del flusso di traffico, quindi con i due messaggi Finished si verificano entrambe le direzioni).

9 Derivazione del keyblock

Per derivare il keyblock dal pre-master secret e dai due valori random (il che viene fatto in seguito alla fase di scambio delle chiavi e prima dell'invio dei messaggi Finished), TLS 1.2 usa una **funzione pseudo-casuale** (*PRF*, *PseudoRandom Function*). In particolare, tale funzione viene applicata due volte:

1. la concatenazione del pre-master secret con i numeri random viene data come input (seed) alla PRF, ottenendo così un valore che prende il nome di **master key** (o master secret);
2. la master key viene passata nuovamente in input alla PRF, ancora insieme ai valori random, per computare il keyblock.

Infine, dal keyblock si estraggono le 4 chiavi e i 2 vettori di inizializzazione necessari per gli algoritmi di cifratura e di calcolo dei MAC.