Il Protocollo Handshake di TLS (fino alla versione 1.2)

Basato sulla presentazione "TLS Handshake Protocol"

Indice

Il meccanismo di Handshake di **Transport Layer Security (TLS)**, nelle sue versioni precedenti alla 1.3, è il fondamento che permette di stabilire una comunicazione cifrata sicura tra due parti. È bene sottolineare che **TLSv1.3 è significativamente diverso** e semplifica notevolmente i complessi passaggi qui descritti.

1 Meccanismo e Contesto dell'Handshake

L'handshake è un processo attivato in diverse circostanze, ma il suo verificarsi è legato a **ogni volta che viene stabilita una nuova Connessione TCP** che necessita di protezione TLS.

Il caso più intensivo è la **Negoziazione Iniziale Completa**, dove il client si connette per la prima volta. Tuttavia, per ottimizzare l'efficienza, se client e server hanno già interagito, possono utilizzare la **Ripresa della Sessione** (*Session Resumption*), un handshake abbreviato che evita di ripetere l'intero scambio di autenticazione. Un meccanismo correlato è il **Re-keying**, usato per "rinfrescare" le chiavi di sessione attive dopo un certo periodo o quantità di dati scambiati, pur restando all'interno della stessa sessione logica.

1.1 Sessione e Rotazione delle Chiavi

La sicurezza TLS si basa sulla distinzione tra la **Sessione TLS**, che è il contesto di sicurezza logico e duraturo (inglobando algoritmi concordati, ecc.), e le singole **Connessioni TCP** che ne fanno parte.

Solo la **prima Connessione TCP** richiede l'Handshake completo. Nelle connessioni successive, si sfrutta il *re-keying*: pur riutilizzando il contesto di sicurezza, vengono generate **nuove chiavi segrete simmetriche** specifiche per ogni Connessione TCP. Questo garantisce la **Forward Secrecy** e limita i danni in caso di compromissione di una singola chiave.

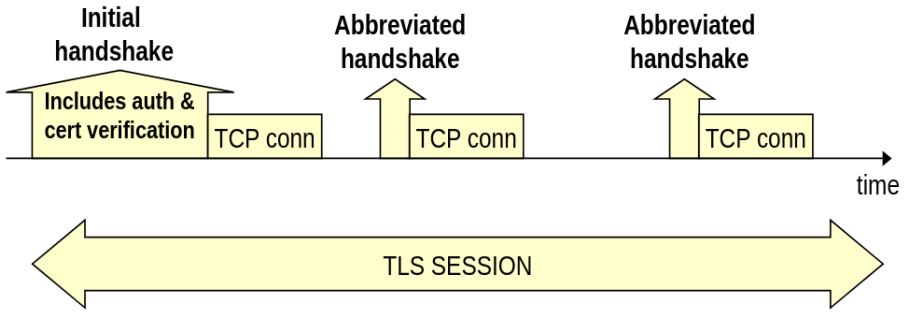

Figura 1: Handshake durante una sessione TLS

1.2 Fasi e Obiettivi del Primo Handshake Completo

Il primo handshake è il più oneroso e ha il compito di eseguire quattro azioni fondamentali che costruiscono la fiducia e stabiliscono i segreti crittografici :

1. **Autenticazione**: Il processo di Handshake TLS stabilisce innanzitutto un tunnel sicuro, ma la responsabilità dell'autenticazione non è sempre reciproca. L'obiettivo primario e quasi universale è l'**Autenticazione del Server** da parte del Client, che avviene tramite la verifica del certificato digitale del server, un passaggio imprescindibile per garantire all'utente (o alla macchina) che sta comunicando con l'entità prevista.

   Al contrario, l'Autenticazione del Client a livello TLS, nota come mTLS (Mutual TLS), è meno comune. Mentre è rara nei siti web tradizionali, è considerata essenziale nelle

moderne architetture a servizi, specialmente nel cloud, dove il client da autenticare è tipicamente una macchina o un servizio e non un utente umano.

Infatti, in un'applicazione rivolta al pubblico, come l'accesso a una piattaforma bancaria, si preferisce un approccio ibrido: Client e Server creano prima il tunnel TLS nel quale il Client autentica il Server (livello di trasporto); successivamente, il Client si autentica al Server a un livello superiore, quello **applicativo**, fornendo credenziali (ad esempio, tramite un protocollo come PAP). Altri contesti, come le reti wireless, utilizzano invece protocolli come l'EAP-TTLS (Tunneled TLS), dimostrando come il tunnel TLS possa essere impiegato per incapsulare e proteggere metodi di autenticazione successivi.

2. **Negoziazione degli Algoritmi** (*Cipher Suite*): È la fase critica in cui Client e Server si accordano sull'insieme di algoritmi crittografici da impiegare. Una negoziazione robusta è essenziale per prevenire i **Downgrade Attack**, dove un attaccante tenta di forzare l'uso di algoritmi obsoleti e vulnerabili.
3. **Scambio di Valori Casuali** (*Nonces*): Vengono scambiati valori **random** generati da entrambe le parti, utilizzati per garantire l'unicità della chiave di sessione e la prevenzione di attacchi di *replay*.
4. **Scambio Sicuro di Chiavi Segrete**: Sfruttando la **crittografia asimmetrica**, le parti scambiano le informazioni necessarie per **derivare la chiave segreta simmetrica** che verrà usata per cifrare i dati. Le chiavi di sessione non vengono mai scambiate in chiaro.

L'intero processo è intrinsecamente progettato per garantire la **Negoziazione sicura dei segreti condivisi**, l'**Autenticazione** e, soprattutto, la **Robustezza agli attacchi Man-in-the-Middle (MITM)**, resistendo al *tampering* che mira ad alterare l'esito della negoziazione (attacchi di **downgrade**).

2 Flusso dei Messaggi dell'Handshake (Schema Completo)

Nella Figura 2, è indicato il flusso di messaggi che si scambiano Client e Server durante la fase di Handshake.

1. **Negoziazione Iniziale (Messaggi 1-2):**
   - Il *Client* avvia la comunicazione con `Client Hello`, proponendo versioni TLS supportate, algoritmi crittografici (*cipher list*) e valori casuali (*nonces*).
   - Il *Server* risponde con `Server Hello`, accettando la versione TLS e la *cipher suite* più sicura comune ad entrambi.
2. **Autenticazione e Scambio Chiavi (Messaggi 3-8):**
   - Il *Server* invia il `Server Certificate` (Messaggio 3) per la propria autenticazione.
   - Messaggi opzionali (`Server Key Exchange`, `Server Certificate Request`) gestiscono ulteriori scambi di parametri e l'eventuale richiesta di autenticazione del Client.
   - Il *Client* (Messaggio 7) genera un segreto (*premaster secret*) e lo invia cifrato usando la chiave pubblica del Server (ottenuta dal certificato). Questo permette la derivazione della chiave simmetrica di sessione.
3. **Finalizzazione e Passaggio alla Cifratura (Messaggi 9-11):**
   - Le parti inviano il messaggio `Cipher Specifications` (Messaggi 8, 10), che segnala l'intenzione di iniziare a usare la cifratura con le chiavi appena derivate.

- Il messaggio `Finished` (Messaggi 9, 11) è l'ultimo messaggio dell'Handshake e viene **cifrato** con le nuove chiavi. Serve come prova che entrambe le parti sono riuscite a completare l'Handshake e a derivare correttamente le chiavi, confermando che nessuno ha manomesso i messaggi precedenti.

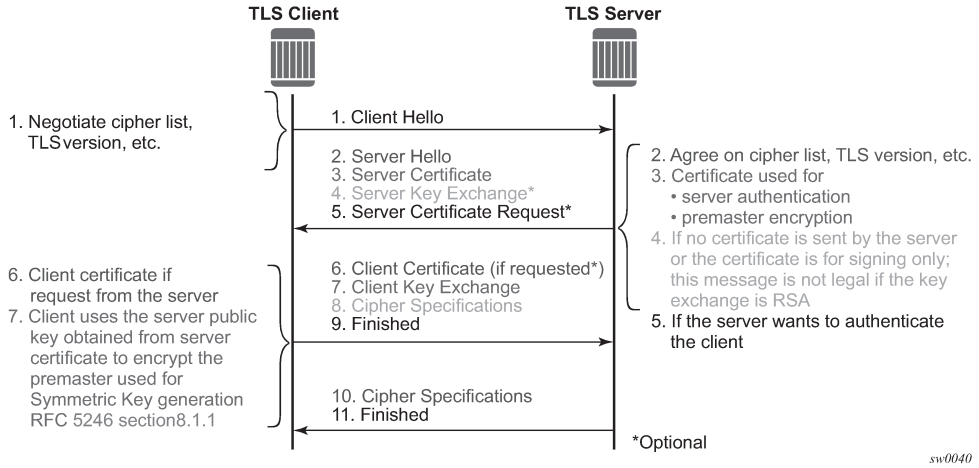

Figura 2: Flusso dei messaggi handshake

2.1 Nota sulla Segmentazione TCP

Quando il server o il client invia messaggi durante l'Handshake TLS (ad esempio, il certificato), questi non vengono trasmessi direttamente sulla rete, ma seguono una precisa gerarchia di protocolli.

Il processo inizia a livello logico (**Livello Handshake**), dove vengono generati i messaggi di controllo come il *Server Hello* o il *Certificate*. Questi dati passano al **Livello Record TLS**, il quale svolge un ruolo cruciale: aggiunge un'intestazione per identificare il tipo di contenuto e la versione TLS, e gestisce la **frammentazione** o il raggruppamento dei dati in unità logiche chiamate **Record TLS**.

Successivamente, i Record TLS vengono incapsulati dal **Livello TCP** (il Livello di Trasporto), dove avviene l'ultima fase di preparazione per la trasmissione fisica.

Il punto fondamentale risiede nell'assenza di una corrispondenza rigida tra la dimensione dei Record TLS e i segmenti TCP. Un singolo messaggio logico (come un certificato voluminoso) può essere troppo grande per un singolo Record TLS o per un singolo segmento TCP.

Per questo, un Record TLS può essere a sua volta **frammentato in più pacchetti TCP** se supera la dimensione massima di trasmissione (MTU) della rete. Viceversa, più Record TLS di piccole dimensioni possono essere **accorpati** in un unico segmento TCP per ottimizzare l'efficienza.

In sintesi, i dati scorrono dall'alto (messaggi logici) verso il basso (pacchetti fisici), e ogni livello ha la libertà di segmentare o aggregare i dati per garantire la massima efficacia e affidabilità della trasmissione sulla rete.

2.2 Formato dei Messaggi Handshake

Ricordiamo che l'handshake TLS è un protocollo, e di conseguenza i messaggi e il loro formato è ben definito. Il messaggio di handshake non viaggia da solo, ma viene **incapsulato** all'interno di un **TLS Record** (Figura 3).

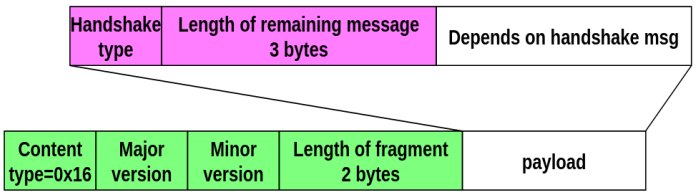

Figura 3: Record TlS con all'interno il messaggio Handshake

Nello specifico, come possiamo vedere nella Figura 3, un Record TLS contenente un messaggio di handshake è strutturato nel seguente modo:

- **Livello Esterno: TLS Record Header (Riquadro Verde)** Questa è l'intestazione del protocollo di trasporto TLS.
  - **Content Type:** `0x16` (Questo valore esadecimale identifica specificamente che il pacchetto contiene un messaggio di Handshake).
  - **Major Version:** Versione maggiore del protocollo.
  - **Minor Version:** Versione minore del protocollo.
  - **Length of fragment:** 2 bytes (indica la dimensione del payload che segue).
  - **Payload:** Contiene il messaggio di Handshake vero e proprio.
- **Livello Interno: Handshake Header (Riquadro Rosa/Bianco)**Questa parte si trova dentro il payload del livello esterno.
  - **Handshake Type:** 1 byte (Indica l'ID del messaggio, ad esempio 1 per *client_hello*).
  - **Length of remaining message:** 3 bytes (Indica la lunghezza dei dati successivi).
  - **Body:** Contenuto variabile ("Depends on handshake msg"). I dati qui dentro cambiano a seconda del tipo di handshake (es. i parametri crittografici, i certificati, etc.).

Nella Tabella 1, si possono vedere le tipologie di messaggi Handshake.

ID	Handshake Type
0	hello_request
1	client_hello
2	server_hello
11	certificate
12	server_key_exchange
13	certificate_request
14	server_hello_done
15	certificate_verify
16	client_key_exchange
20	finished

Tabella 1: Tipi di messaggi Handshake TLS

3 Fase 1: Negoziazione Iniziale (Hello)

La fase di negoziazione è il primo passo assoluto del protocollo TLS. Possiamo immaginarla come una stretta di mano "diplomatica" in cui Client e Server, prima ancora di scambiarsi dati segreti, devono mettersi d'accordo su come parlare (quale "lingua" usare e quali regole di sicurezza applicare).

Questa fase è composta da due messaggi principali: il Client Hello e il Server Hello.

3.1 Client Hello

Trattandosi del primo messaggio, viene inviato in **plaintext** (in chiaro) e può essere inteso come l'offerta iniziale delle opzioni supportate dal client. Sta poi al server decidere quali opzioni scegliere tra quelle offerte da cliente. Un tipico messaggio `client_hello` contiene:

- **Versione**: La versione TLS/SSL più alta supportata dal client.
- **Random (32 byte)**: Un nonce composto da 4 byte di timestamp e 28 byte casuali.
- **Session ID**: Se il client desidera riprendere una sessione precedente, include qui il vecchio ID. Altrimenti è vuoto.
- **Cipher Suites**: Una **lista** di tutte le combinazioni di algoritmi crittografici supportate dal client, in ordine di preferenza.
- **Algoritmo di Compressione**: Oggi sempre nullo (la compressione TLS si è rivelata vulnerabile ad attacchi come CRIME).

Il "menù" offerto dal client, in realtà consiste nella lista di **cipher suites**, ossia una lista di algoritmi crittografici, dove ciascuna opzione è descritta come segue (fino a TLS 1.2):
`TLS_AAAAAAA_WITH_BBBBBBBB_CCCCCCCC`

- **AAAAAAA** (Algoritmo di Scambio Chiave): Definisce come il server si autentica e come viene scambiata la chiave (es. `RSA`, `DHE_DSS`). In TLS 1.3 non viene più utilizzato `RSA` ma solo `DHE_RSA`, perciò non viene più specificato l'algoritmo di scambio di chiavi.
- **BBBBBBBB** (Algoritmo Simmetrico): L'algoritmo di cifratura per i dati (es. `3DES_EDE_CBC`, `AES_128_GCM`).
- **CCCCCCCC** (Algoritmo Hash): Usato per il Message Authentication Code (MAC) (es. `SHA`, `SHA256`).

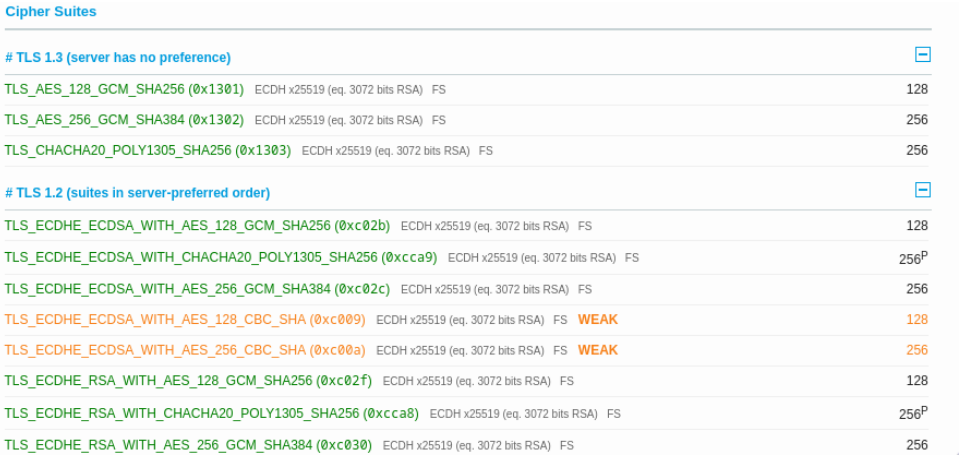

Figura 4: Esempio di lista cipher suites

Alla fine di questa dispensa sarai in grado di leggere un record di questo tipo :) .

Nota su TLS 1.0 e 1.1

Nonostante da gennaio 2022 le versioni TLS 1.0 e 1.1 siano state ufficialmente deprecate a favore delle più sicure 1.2 e 1.3, la loro completa dismissione risulta difficile per motivi di retrocompatibilità. Se un server supportasse esclusivamente i nuovi standard, rifiuterebbe le connessioni dai client più datati, rendendo il servizio inaccessibile. Di conseguenza, molti amministratori scelgono di mantenere attivi i vecchi protocolli: si accetta il compromesso di una sicurezza inferiore pur di garantire la continuità del servizio, poiché per l'utente medio la priorità rimane la possibilità di navigare senza blocchi.

3.2 Server Hello

Il Server Hello è la risposta diretta al Client Hello. Se il Client Hello era la "proposta", il Server Hello è la decisione finale.

In questa fase, il server seleziona i parametri che verranno effettivamente utilizzati per la connessione, basandosi su ciò che il client ha offerto e su ciò che il server stesso supporta. Ovviamente anch'esso in **plaintext**. Contiene:

- **Versione**: La versione **più alta supportata da entrambi**. (Se il Client offre 1.2 e il Server supporta solo 1.1, il Server risponderà 1.1).
- **Random (32 byte)**: Un *nuovo* valore casuale generato dal server, diverso da quello del client.
- **Session ID**: Se il server accetta la ripresa della sessione, conferma il Session ID, altrimenti ne genera uno nuovo.
- **Cipher Suite**: **Una singola** cipher suite, scelta dal server dalla lista offerta dal client.
- **Algoritmo di Compressione**: Selezionato dalla lista del client (oggi, nullo).

SSL Labs

Per verificare la sicurezza di un server, si può usare il tool https://www.ssllabs.com/ssltest/ .

4 Uso della Crittografia Asimmetrica In TLS

Per comprendere l'architettura di TLS, dobbiamo prima affrontare un problema fondamentale della crittografia. La crittografia simmetrica (quella in cui si usa la stessa chiave sia per cifrare che per decifrare) è estremamente efficiente e veloce. Tuttavia, porta con sé un enorme difetto logistico noto come il problema della distribuzione delle chiavi: come faccio a far avere la chiave segreta al mio interlocutore (es. un server dall'altra parte del mondo) senza che nessuno la intercetti durante il tragitto? Se provassimo a scriverla nel codice di un'app (hardcoding), chiunque potrebbe estrarla, creando un disastro di sicurezza.

La risposta a questo problema risiede nella crittografia asimmetrica. Invece di una sola chiave, questo sistema ne utilizza una coppia matematicamente correlata:

- Chiave Pubblica (K_{ENC}): È nota a tutti e serve solo per cifrare (chiudere il messaggio).
- Chiave Privata (K_{DEC}): È segreta, posseduta solo dal destinatario, e serve per decifrare (aprire il messaggio).

Il principio di sicurezza si basa sull'asimmetria matematica: i dati cifrati con la chiave pubblica possono essere letti solo dalla chiave privata corrispondente. Inoltre, pur conoscendo la chiave pubblica, è computazionalmente impossibile calcolare la chiave privata inversa.

Il problema della crittografia asimmetrica consiste nell'essere estremamente pesante per il processore: è circa 4-5 ordini di grandezza più lenta rispetto a quella simmetrica. Usarla per cifrare l'intero traffico di dati (es. lo streaming di un video) renderebbe la navigazione impossibile.

Il protocollo TLS adotta un approccio intelligente che unisce il meglio dei due mondi, chiamato approccio ibrido:

1. **Fase Iniziale (Handshake):** Si utilizza la lenta crittografia asimmetrica solo per il breve tempo necessario a scambiare o concordare una chiave temporanea condivisa (detta Master Secret).
2. **Fase Operativa (Record Protocol):** Una volta che entrambe le parti possiedono questa chiave condivisa, si passa alla veloce crittografia simmetrica per cifrare tutto il flusso di dati successivo.

4.1 Approcci allo Scambio Chiavi (fino a TLS 1.2)

Esistono due filosofie principali per ottenere la chiave comune durante l'handshake: il *trasporto della chiave* (dove uno decide e invia) e *l'accordo sulla chiave* (dove entrambi contribuiscono matematicamente).

1. Key Transport (es. RSA)

Una parte genera il segreto e lo invia cifrato all'altra.

1. Il **Server** invia il suo certificato contenente la propria **Chiave Pubblica**.
2. Il **Client** genera un segreto casuale (detto *pre-master secret*).
3. Il Client **cifra** il pre-master secret utilizzando la Chiave Pubblica del server e invia il risultato.
4. Solo il Server, possedendo la corrispondente **Chiave Privata**, può decifrare il messaggio e recuperare il segreto.
5. *Risultato:* Entrambi possiedono ora lo stesso segreto (generato dal Client).

2. Key Agreement (es. Diffie-Hellman)

Il segreto è frutto di un calcolo matematico congiunto.

1. Client e Server generano chiavi segrete (x e y) e scambiano **solo** le pubbliche

$$A = g^x \pmod{p} \quad \text{e} \quad B = g^y \pmod{p}$$

2. Entrambi calcolano **indipendentemente** lo stesso segreto combinando la propria privata con la pubblica altrui:

$$S = g^{xy} \pmod{p}$$

3. *Vantaggio:* Il segreto non viaggia mai sulla rete, neppure cifrato.

Una delle novità più radicali introdotte da TLS 1.3 è la **rimozione definitiva** del vecchio metodo di scambio chiavi statico (RSA Key Transport). Il protocollo impone ora l'obbligo della **Perfect Forward Secrecy (PFS)**, costringendo server e client a utilizzare esclusivamente chiavi effimere tramite l'algoritmo **(EC)DHE - (Eliptic Curve) Diffie-Hellman Ephemeral**.

In questo nuovo scenario, le chiavi a lungo termine presenti nei certificati (come RSA o ECDSA) perdono la funzione di cifratura del segreto e vengono rilegate al solo compito di **autenticazione**

tramite firma digitale. Questo cambiamento architetturale è cruciale: garantisce che, anche qualora la chiave privata del server venisse rubata in futuro, tutte le sessioni passate rimarrebbero blindate e indecifrabili.

5 Attacchi di Downgrade

Vediamo adesso il downgrade attack, una tecnica utilizzata in contesti MITM.

Immaginiamo due persone che sanno parlare perfettamente una lingua complessa e sicura (es. Italiano moderno), ma sanno anche un dialetto antico e facile da decifrare. Un Downgrade Attack è quando un intruso si mette in mezzo e fa credere a entrambi che l'altro sappia parlare solo il dialetto antico. Di conseguenza, le due vittime inizieranno a comunicare in modo debole, permettendo all'intruso di capire tutto.

Nel contesto TLS: Questo attacco è molto pericoloso. L'attaccante intercetta i messaggi di negoziazione (che viaggiano in chiaro) e modifica le richieste:

- Downgrade della Versione: Cambia la richiesta da TLS 1.2/1.3 a versioni obsolete (es. SSL 2.0 o 3.0).
- Downgrade delle Cipher Suite: Rimuove gli algoritmi forti dalla lista del client, lasciando solo quelli deboli o vulnerabili.

Perché funziona? Funziona perché nella fase iniziale di Handshake non c'è ancora autenticazione: client e server non hanno ancora un segreto condiviso per firmare i messaggi e garantirne l'integrità. Come vedremo, TLS risolve questo problema alla fine dell'handshake (tramite il messaggio *Finished*) per rilevare se c'è stata manipolazione.

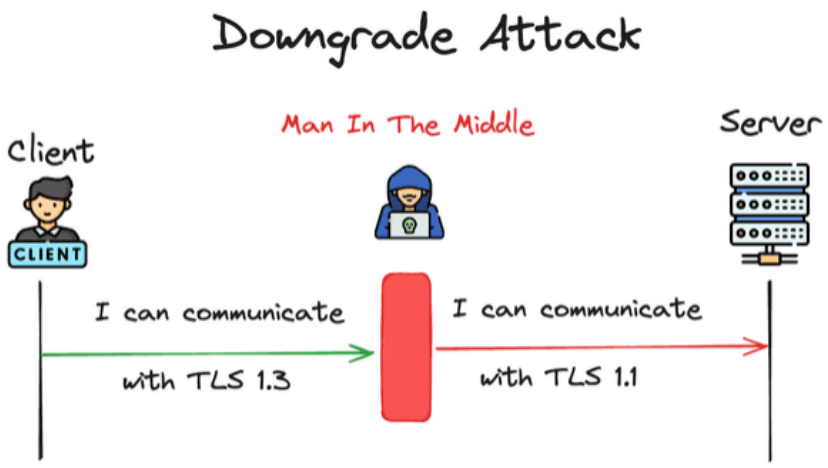

Figura 5: Downgrade Attack

1. Un attaccante Man-in-the-Middle (MITM) intercetta il `Client Hello`. Poiché il Client Hello viaggia in plaintext (in chiaro), l'attaccante può leggerlo e modificarlo (ad esempio, propone "TLS 1.2" e cipher suite robuste).
2. L'attaccante modifica il messaggio prima di inoltrarlo al server, sostituendo la versione con "SSL 2.0" (una versione vecchia e rotta) e rimuovendo tutte le cipher suite robuste.
3. Il Server riceve il `Client Hello` manomesso che richiede SSL v2.0. Il server pensa: "Ok, questo client è vecchio e supporta solo SSL 2.0. Per garantire la compatibilità (retro-compatibilità), accetto di usare questa versione".

4. Il Server risponde con un `Server Hello` che accetta SSL 2.0.
5. L'attaccante inoltra questa risposta al client. Client e Server sono stati ingannati e negoziano una connessione debole che l'attaccante può rompere.

Di conseguenza, il Client crede che il server sia vecchio e supporti solo SSL 2.0, il Server crede che il client sia vecchio e supporti solo SSL 2.0, quindi la connessione viene stabilita con una crittografia debole che l'attaccante può rompere facilmente. Nella fase 2 e 3 vedremo come questo è possibile prevenirlo.

6 Fase 2: Autenticazione del Server

Come illustrato nella Figura 1, dopo aver ricevuto il messaggio `client_hello`, il server deve selezionare gli algoritmi per lo scambio delle chiavi. Fino alla versione TLS 1.2, era possibile utilizzare RSA come algoritmo asimmetrico per questa fase, mentre in TLS 1.3 l'unico meccanismo consentito è Diffie-Hellman (DH). Da questo punto in poi, il server deve procedere sia alla propria autenticazione che allo scambio dei parametri crittografici.

6.1 Key Transport - RSA

Analizziamo il caso in cui venga negoziato l'algoritmo **RSA**. In questo scenario, il server invia il messaggio `Certificate`, contenente il proprio certificato X.509 (argomento che verrà approfondito successivamente) all'interno del quale risiede la sua chiave pubblica RSA. Poiché la chiave necessaria per cifrare è già presente nel certificato, il messaggio `Server Key Exchange` **non viene inviato**, in quanto ridondante.

Il client dispone dunque di tutte le informazioni necessarie per generare e cifrare il *Pre-Master Secret* (anch'esso oggetto di approfondimento futuro). L'autenticazione avviene in modo *implicito*: se il server riesce a decifrare il messaggio del client e a derivare le corrette chiavi di sessione, dimostra matematicamente di possedere la Chiave Privata corrispondente al certificato inviato.

Questo schema offre anche una protezione intrinseca contro i *downgrade attack* sulla versione del protocollo, come mostrato in Figura 6.

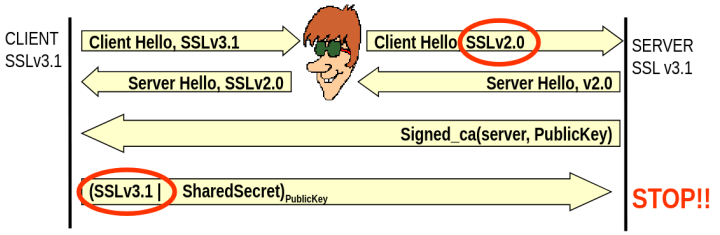

Figura 6: Protezione contro il downgrade attack nello scambio RSA

Il meccanismo di difesa funziona nel seguente modo: quando il client invia il `Client Key Exchange`, inserisce all'interno del pacchetto cifrato non solo il segreto, ma anche i primi due byte che indicano la **versione TLS nativa** con cui ha iniziato la connessione (es. "3.1" per TLS 1.0). Poiché questo dato è cifrato con la chiave pubblica del server, un attaccante MITM non può né leggerlo né modificarlo. Il server, una volta decifrato il pacchetto, confronta due valori:

- La versione presente all'interno del pacchetto cifrato (reale intenzione del client).
- La versione negoziata in chiaro nel `Server Hello` (potenzialmente manipolata).

Se i due valori non coincidono, il server rileva il tentativo di attacco e interrompe immediatamente la connessione.

6.2 Key Agreement - Diffie-Hellman (DH)

Analizziamo ora lo scenario in cui si sceglie Diffie-Hellman come algoritmo per lo scambio delle chiavi. Il protocollo DH può essere utilizzato in TLS in tre modalità distinte, che differiscono per la gestione dell'autenticazione e della durata delle chiavi:

Anonymous DH (ADH) Le coppie di chiavi DH (x, g^x) e (y, g^y) sono effimere (generate al momento) ma **non sono autenticate**. Il server invia i suoi parametri nel messaggio `Server Key Exchange`, ma non invia alcun certificato digitale (o ne invia uno non verificabile). Di conseguenza, chiunque può interporsi nella comunicazione: questa modalità è totalmente vulnerabile agli attacchi MITM e oggi è disabilitata quasi ovunque.

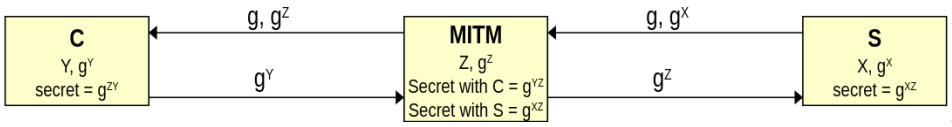

Figura 7: Trivial MITM

Fixed DH I parametri pubblici del server (g^x) sono **statici** e sono contenuti direttamente nel suo certificato X.509 (firmato dalla CA). In questo caso:

- Il server invia solo il messaggio `Certificate`.
- Il messaggio `Server Key Exchange` **non viene inviato** (i parametri sono già nel certificato).

Sebbene non sia vulnerabile al MITM (c'è il certificato), questa modalità ha un grave difetto: **manca la Forward Secrecy**. Se la chiave privata del server (che è fissa e longeva) viene compromessa, tutto il traffico passato può essere decifrato.

Ephemeral DH (DHE / ECDHE) Questa è la modalità standard nel TLS moderno (obbligatoria in TLS 1.3). Le chiavi DH sono **effimere** (cambiano a ogni sessione), garantendo la *Perfect Forward Secrecy*. Il funzionamento prevede l'uso combinato dei messaggi:

- `Certificate`: Il server invia il suo certificato contenente una chiave pubblica di firma (es. RSA o ECDSA).
- `Server Key Exchange`: Il server genera i parametri DH casuali per la sessione corrente, li **firma** digitalmente usando la chiave privata del certificato e li invia.

Il client usa la chiave nel certificato per verificare la firma e i parametri nel `Key Exchange` per calcolare il segreto condiviso.

La conclusione della Fase 2 prevede, opzionalmente, l'invio del messaggio `Certificate Request` (Type 13) qualora il server richieda l'autenticazione del client (scenario mTLS). Infine, il server trasmette il messaggio `Server Hello Done` (Type 14): si tratta di un pacchetto privo di payload (vuoto) che ha l'unico scopo di segnalare al client che il server ha terminato l'invio dei propri parametri e attende ora una risposta.

7 Fase 3: Autenticazione del Client

Normalmente l'autenticazione è monodirezionale (solo il Server). Se il server necessita di autenticare il client (*Mutual TLS*), invia il messaggio **Certificate Request** (Type 13).

Se questo messaggio non viene inviato, il client assume che il server non richieda autenticazione (caso standard del web: tu navighi su Google senza mostrare la tua carta d'identità digitale).

Struttura del Certificate Request

Il messaggio serve a "filtrare" i certificati ammissibili. Contiene:

1. **Certificate Types:** Tipi di chiave accettati (es. `rsa_sign`, `ecdsa_sign`).
2. **Signature Algorithms:** Hash e algoritmi di firma supportati (es. SHA256).
3. **Distinguished Names (DNs):** La lista delle **Certification Authority (CA)** fidate dal server.

Funzione: Il client controlla il proprio store di certificati e invia un certificato solo se è stato emesso da una delle CA elencate in questo campo.

Conseguenza: Il client dovrà rispondere obbligatoriamente con il messaggio `Certificate` seguito dal messaggio `Certificate Verify` (per provare il possesso della chiave privata).

Dopo aver ricevuto il `Server Hello Done`, il client procede con l'invio dei seguenti messaggi:

1. **Certificate (Opzionale)** Inviato solo se il server ha spedito una `Certificate Request`. Contiene il certificato digitale del client.
2. **Client Key Exchange (Obbligatorio)** Trasmette le informazioni crittografiche necessarie al server per generare la chiave comune:
   - **In RSA:** Invia il *Pre-Master Secret* cifrato con la chiave pubblica del server insieme alla versione di TLS che si sta utilizzando, per prevenire il downgrade attack sulla versione.
   - **In DH:** Invia i parametri pubblici del client (Y_{client}).
3. **Certificate Verify (Condizionale)** Inviato solo se il client si sta autenticando (ha inviato il certificato). È fondamentale per la sicurezza.

Il Ruolo cruciale del Certificate Verify

Questo messaggio contiene la **firma digitale** del client applicata su **tutti i messaggi di handshake scambiati fino a quel momento**. Ha una duplice funzione:

1. **Proof of Possession:** Dimostra che il client possiede la Chiave Privata associata al certificato inviato (altrimenti chiunque potrebbe impersonarlo copiando il certificato pubblico).
2. **Integrity Check:** Poiché la firma copre l'intero storico dei messaggi, permette di rilevare precocemente eventuali manipolazioni (tampering) avvenute nelle fasi precedenti (es. Downgrade Attacks sulle Cipher Suite).

8 Fase 4: Conclusione e Attivazione Cifrari

Questa è la fase conclusiva, simmetrica per Client e Server, in cui la teoria diventa pratica. Fino a questo momento, le parti hanno eseguito solo negoziazione e calcoli matematici su dati scambiati in chiaro o parzialmente cifrati. Adesso è necessario "premere l'interruttore" e rendere operativo il canale sicuro.

Questa fase ha due scopi fondamentali:

1. **Sincronizzazione:** Attivare i nuovi parametri di sicurezza negoziati (passaggio da *Pending State* a *Current State*).
2. **Verifica di Integrità:** Confermare che l'handshake non sia stato manomesso da un MITM e che entrambe le parti abbiano derivato le stesse chiavi.

8.1 Change Cipher Spec (CCS)

Sebbene appaia nel flusso dell'handshake, il `Change Cipher Spec` è tecnicamente un protocollo separato (Content Type `0x14`). È un messaggio di un singolo byte (valore `0x01`) che funge da segnale di commutazione.

Il meccanismo degli Stati (Record Layer)

Il livello TLS Record mantiene in memoria due stati per ogni direzione di comunicazione:

- **Current State:** Lo stato attivo, usato per cifrare i messaggi attuali (inizialmente è *No Encryption*).
- **Pending State:** Lo stato "in attesa", dove vengono accumulati i nuovi algoritmi e chiavi appena negoziati.

Il messaggio CCS ordina al Record Layer di: **copiare il Pending State nel Current State**. Da quell'istante preciso, ogni bit successivo sarà processato con la nuova suite crittografica.

8.2 Finished

Il messaggio `Finished` è cruciale per due motivi: è il **primo messaggio protetto** (cifrato e autenticato con MAC) con le nuove chiavi, ed è il sigillo di garanzia sull'intero processo.

Calcolo del Verify Data

Il contenuto del messaggio non è casuale, ma è il risultato di una Funzione Pseudo-Casuale (PRF) applicata a:

- Il **Master Secret** (la chiave suprema derivata).
- Una **Etichetta** specifica ("client finished" per il client, "server finished" per il server).
- L'**Hash di tutti i messaggi di handshake** scambiati fino a quel momento.

$$\text{VerifyData} = \text{PRF}(\text{MasterSecret}, \text{"client finished"}, \text{Hash}(\text{HandshakeMessages}))$$

Il caso particolare delle suite AEAD (es. GCM)

Se la Cipher Suite è di tipo **AEAD** (Authenticated Encryption with Associated Data), come ad esempio:

`TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`

succede una cosa controintuitiva:

- La parte **GCM** gestisce *sia* la cifratura che l'integrità (non serve un HMAC separato per i pacchetti).
- Allora a cosa serve il suffisso **SHA256**?
- Serve **esclusivamente** per la PRF (derivazione chiavi) e per il calcolo dell'hash nel messaggio `Finished`. **Non** viene usato per controllare l'integrità dei singoli pacchetti dati (ci pensa già il GCM).

Perché le etichette sono diverse?

L'uso di stringhe diverse ("client finished" vs "server finished") è una difesa contro gli **attacchi di riflessione**. Se il contenuto fosse identico, un attaccante potrebbe intercettare il `Finished` del client e rispedirlo indietro fingendosi il server. Differenziando le etichette, i due messaggi risultano matematicamente distinti.

Verifica Finale

Quando il Server riceve il `Finished` del Client:

1. Lo decifra con la chiave simmetrica appena attivata (se fallisce → errore chiavi).
2. Ricalcola autonomamente l'hash dei messaggi visti.
3. Confronta il suo calcolo con quello ricevuto.

Se un attaccante (MITM) avesse modificato anche un solo bit in precedenza (es. cambiando la lista delle cipher suite nel `Client Hello` per forzare un downgrade), gli hash divergerebbero. Il server non riuscirebbe a validare il messaggio e la connessione verrebbe immediatamente abbattuta ("Fatal Alert: Bad Record MAC").

9 Derivazione delle Chiavi - HKDF

In TLS, le chiavi utilizzate per cifrare i dati non vengono scambiate direttamente. Esiste invece una **gerarchia di derivazione** che permette di trasformare un segreto condiviso grezzo (scambiato asimmetricamente) in un set di chiavi simmetriche operative sicure.

Questa struttura a livelli garantisce che la compromissione di una chiave di basso livello (es. una chiave di cifratura) non comprometta necessariamente i livelli superiori o le sessioni future.

Flusso di Derivazione

$$\text{Pre-Master Secret} \xrightarrow{+\ \text{Randoms}} \text{Master Secret} \xrightarrow{+\ \text{Randoms}} \text{Connection Keys}$$

9.1 Pre-Master Secret (Il Livello Base)

È il segreto "grezzo" concordato durante la fase di Handshake. La sua natura dipende dall'algoritmo di scambio chiavi scelto:

- **RSA:** Viene generato dal *Client*, cifrato con la chiave pubblica del server e inviato.
- **Diffie-Hellman (DH):** È il risultato del calcolo matematico $g^{xy} \pmod{p}$.

Caratteristiche critiche:

- È la base di tutta la sicurezza.
- **Rischio Fixed DH:** Nel caso di Diffie-Hellman statico (Fixed DH), il Pre-Master Secret potrebbe essere *sempre lo stesso* per una data coppia Client-Server. Se usato direttamente come chiave di cifratura, renderebbe la connessione vulnerabile ad attacchi di replay o crittoanalisi. Ecco perché è necessario il passaggio successivo.

9.2 Master Secret (Il Perno della Sessione)

Il Master Secret è un valore di lunghezza fissa (solitamente 48 byte) derivato dal Pre-Master Secret. Serve a introdurre **entropia fresca** (casualità) nella sessione. L'operazione prende il nome di **EXTRACT**.

$$\text{Master Secret} = \text{PRF}(\text{Pre-Master}, \text{"master secret"}, R_C + R_S)$$

Dove:

- **PRF:** Pseudo-Random Function (funzione che mischia i dati in modo irreversibile).
- R_C **(Nonce C):** Valore casuale generato dal Client nel `ClientHello`.
- R_S **(Nonce S):** Valore casuale generato dal Server nel `ServerHello`.

Nota sui Nonce (Random Values): Questi valori includono spesso un *Timestamp* (per evitare replay attack basati sul tempo) + byte puramente casuali.

Ruolo nella Session Resumption: Se la sessione viene ripresa (Resumption), il Master Secret può essere ricalcolato o recuperato dalla cache, ma l'introduzione dei nuovi *Random Values* (che cambiano a ogni handshake, anche in quelli abbreviati) garantirà che le chiavi finali siano diverse.

9.3 Connection State Keys (Le Chiavi Operative)

Dal Master Secret vengono derivate le chiavi effettive utilizzate per proteggere i record TLS. Poiché TLS è un protocollo bidirezionale (Full Duplex) e richiede sia confidenzialità che integrità, vengono generate fino a **6 chiavi distinte** (Key Block). L'operazione prende il nome di **EXPAND**.

$$\text{Key Block} = \text{PRF}(\text{Master Secret}, \text{"key expansion"}, R_S + R_C)$$

Dal *Key Block* si estraggono in sequenza:

1. **Client Write MAC Key:** Per l'integrità dei dati inviati dal client.
2. **Server Write MAC Key:** Per l'integrità dei dati inviati dal server.
3. **Client Write Encryption Key:** Per cifrare i dati inviati dal client.
4. **Server Write Encryption Key:** Per cifrare i dati inviati dal server.
5. **Client Write IV:** Vettore di inizializzazione (se richiesto dal cifrario, es. CBC).
6. **Server Write IV:** Vettore di inizializzazione (se richiesto dal cifrario).

Perché non utilizzare un'unica chiave $K_{session}$ per entrambe le direzioni (come si fa in semplici implementazioni AES)? La motivazione è la sicurezza contro i **Reflection Attacks**.

Scenario di Attacco (Se usassimo una sola chiave)

Immaginiamo che esista solo una chiave K.

1. Il Client invia un messaggio cifrato $C = E_K(\text{"Ordina Bonifico"})$.
2. Un attaccante (MITM) intercetta C.
3. L'attaccante rispedisce C indietro al Client (riflessione).
4. Il Client riceve C, lo decifra con K e legge "Ordina Bonifico".
5. Il Client potrebbe credere che sia un comando valido inviato dal Server ed eseguirlo nuovamente.

Soluzione TLS: Con chiavi distinte, se il Client riceve un messaggio cifrato con la propria chiave ($K_{ClientWrite}$), lo scarta immediatamente perché si aspetta messaggi cifrati solo con $K_{ServerWrite}$.

È fondamentale comprendere che **non** avviene uno scambio fisico di queste chiavi finali. Poiché la derivazione matematica è deterministica e parte dallo stesso *Master Secret*, alla fine dell'handshake:

- Il **Client** calcola e possiede: $\{K_{ClientWrite}, K_{ServerWrite}\}$
- Il **Server** calcola e possiede: $\{K_{ClientWrite}, K_{ServerWrite}\}$

Entrambe le parti possiedono l'intero "mazzo" di chiavi. La differenza sta esclusivamente nel **ruolo** che queste chiavi assumono per ciascuna parte.

Flusso Operativo

Direzione Dati	Operazione Mittente	Operazione Destinatario
Client → Server	Il Client **Cifra** usando: $\boldsymbol{K_{ClientWrite}}$	Il Server **Decifra** usando: $\boldsymbol{K_{ClientWrite}}$
Server → Client	Il Server **Cifra** usando: $\boldsymbol{K_{ServerWrite}}$	Il Client **Decifra** usando: $\boldsymbol{K_{ServerWrite}}$

Quindi, per ottenere le 6 chiavi:

1. Client e Server si scambiano numeri casuali (**Nonces**) in chiaro.
2. Si accordano sul **Pre-Master Secret** (in modo sicuro).
3. Entrambi combinano *Pre-Master + Nonces* per ottenere lo stesso **Master Secret**.
4. Entrambi espandono il *Master Secret + Nonces* per generare le **6 chiavi di connessione**.
5. Ora possono comunicare cifrato.

9.4 Evoluzione delle Funzioni Pseudo-Casuali (PRF)

La PRF è il motore matematico di TLS: prende in input un segreto di lunghezza finita (es. il *Pre-Master Secret*), un'etichetta (*Label*) e un seme casuale (*Seed*), e produce in output una stringa di bit di lunghezza arbitraria (il *Key Block*) da cui si "tagliano" le chiavi di sessione.

L'implementazione di questa funzione è cambiata radicalmente nelle varie versioni del protocollo.

9.4.1 TLS 1.0 e 1.1: La PRF "Ibrida" (Hard-Coded)

Nelle prime versioni, la PRF non era negoziabile ma **hard-coded**. I progettisti non si fidavano completamente di un singolo algoritmo di hash, quindi idearono un meccanismo per combinarne due: **MD5** e **SHA-1**.

- **Split del Segreto:** Il segreto iniziale (S) viene diviso a metà.

$$S \to S_1 \text{ (prima metà)} \quad e \quad S_2 \text{ (seconda metà)}$$

- **Doppio Calcolo:**
  - La prima metà (S_1) viene elaborata con **MD5**.
  - La seconda metà (S_2) viene elaborata con **SHA-1**.
- **Combinazione XOR:** I risultati vengono combinati tramite operazione XOR.

$$\text{PRF}(S, \text{label}, \text{seed}) = P_{\text{MD5}}(S_1, \text{label}|\text{seed}) \oplus P_{\text{SHA-1}}(S_2, \text{label}|\text{seed})$$

Obiettivo: La robustezza. Per rompere questa PRF, un attaccante avrebbe dovuto trovare vulnerabilità simultanee sia in MD5 che in SHA-1.

9.4.2 Meccanismo di Espansione (P_{hash})

Sia in TLS 1.0 che 1.2, il cuore della generazione è la funzione di espansione P_{hash}, che usa iterativamente l'HMAC per generare dati all'infinito. Definito $A_0 = \text{seed}$, la sequenza viene calcolata come:

$$\begin{aligned} A_1 &= \text{HMAC}_{hash}(\text{secret}, A_0) \\ A_2 &= \text{HMAC}_{hash}(\text{secret}, A_1) \\ A_3 &= \text{HMAC}_{hash}(\text{secret}, A_2) \\ &\dots \end{aligned}$$

L'output finale è la concatenazione degli HMAC calcolati su questi valori intermedi:

$$P_{hash} = \text{HMAC}(A_1|\text{seed}) \parallel \text{HMAC}(A_2|\text{seed}) \parallel \dots$$

9.4.3 TLS 1.2: Standardizzazione e Negoziabilità

Con TLS 1.2 si abbandona l'approccio ibrido MD5/SHA-1 (ormai considerati deboli).

- **Singolo Hash:** La PRF utilizza una singola funzione hash, di default **SHA-256**.
- **Negoziabilità:** La PRF non è più fissa. Può essere negoziata tramite la Cipher Suite.
- **Semplificazione:** La formula diventa semplicemente:

$$\text{PRF}(\text{secret}, \text{label}, \text{seed}) = P_{\text{SHA-256}}(\text{secret}, \text{label}|\text{seed})$$

9.4.4 TLS 1.3: HKDF (RFC 5869)

In TLS 1.3 si adotta lo stato dell'arte: **HKDF** (HMAC-based Key Derivation Function). A differenza delle PRF precedenti "fatte in casa", HKDF è uno standard progettato da H. Krawczyk (2010) e *provably secure*.

HKDF divide il processo in due fasi logiche distinte (Extract & Expand), ma nel contesto dell'espansione delle chiavi TLS, utilizza una variante robusta basata su HMAC:

$$\text{HKDF-Expand-Label}(\text{Secret}, \text{Label}, \text{Context}, \text{Length})$$

Dove il "Context" include i Nonces e altri parametri unici della sessione, garantendo che le chiavi derivate siano crittograficamente indipendenti e sicure.

10 Come Leggere una Cipher Suite

Ricordiamo che una cipher suite, almeno fino al TLS 1.2 è descritta nel seguente modo:

`TLS_AAAAAAA_WITH_BBBBBBB_CCCCCCC`

dove:

- **AAAAAAA** (Algoritmo di Scambio Chiave): Definisce come il server si autentica e come viene scambiata la chiave (es. `RSA`, `DHE_DSS`). In TLS 1.3 non viene più utilizzato `RSA` ma solo `DHE_RSA`, perciò non viene più specificato l'algoritmo di scambio di chiavi.
- **BBBBBBB** (Algoritmo Simmetrico): L'algoritmo di cifratura per i dati (es. `3DES_EDE_CBC`, `AES_128_GCM`).
- **CCCCCCC** (Algoritmo Hash): Usato per il Message Authentication Code (MAC) (es. `SHA`, `SHA256`).

Vediamo adesso degli esempi di cipher suite e come leggerli. Consideriamo la figura Figura 8

Figura 8: Report www.overleaf.com

1. `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`

   Questa cipher suite descrive una configurazione moderna che garantisce la *Forward Secrecy*. La stringa può essere scomposta come segue:

   - **Protocollo:** `TLS` (Transport Layer Security).

- **Key Exchange (Scambio Chiavi):** `ECDHE` (*Elliptic Curve Diffie-Hellman Ephemeral*). L'uso di chiavi effimere (*Ephemeral*) implica che viene generata una nuova coppia di chiavi per ogni sessione. Questo garantisce la **Perfect Forward Secrecy**: se la chiave privata del server venisse compromessa in futuro, il traffico passato non potrebbe essere decifrato.
- **Autenticazione:** `RSA`. Poiché il protocollo Diffie-Hellman è anonimo, il server firma i parametri dello scambio chiavi usando la propria chiave privata RSA. Il client verifica la firma tramite il certificato del server per autenticarlo.
- **Cifratura Simmetrica:** `AES_128_GCM`. Viene utilizzato l'algoritmo AES con chiavi a 128 bit in modalità **GCM** (*Galois/Counter Mode*). GCM è una modalità **AEAD** (*Authenticated Encryption with Associated Data*), il che significa che garantisce contemporaneamente sia la riservatezza (cifratura) che l'integrità dei dati.
- **MAC / PRF:** `SHA256`. Poiché l'integrità dei record è già garantita dalla modalità GCM, l'algoritmo di hash SHA256 **non** viene usato per calcolare l'HMAC di ogni pacchetto. Esso viene invece utilizzato come base per la **PRF** (*Pseudo-Random Function*) all'interno dell'algoritmo HKDF, necessario per derivare le chiavi di sessione (Master Secret, Key Block) durante l'handshake.

2. `TLS_RSA_WITH_AES_128_GCM_SHA256`

   Questa cipher suite utilizza un approccio più tradizionale e **non** garantisce la Forward Secrecy:

   - **Key Exchange e Autenticazione:** `RSA`. In questo caso, RSA svolge un doppio ruolo:
     - *Scambio Chiavi:* Il client cifra il *Pre-Master Secret* con la chiave pubblica del server (estratta dal certificato) e lo invia al server. Solo il server può decifrarlo.
     - *Autenticazione:* L'autenticazione è **implicita**. Non avviene una firma digitale separata; il fatto che il server sia in grado di decifrare il messaggio e completare l'handshake dimostra il possesso della chiave privata corretta.

     Nota: Senza l'uso di chiavi effimere (come in ECDHE), se la chiave privata RSA del server viene rubata, tutto il traffico passato registrato può essere decifrato.
   - **Cifratura Simmetrica:** `AES_128_GCM`. Analogamente al caso precedente, si usa AES in modalità GCM (AEAD), coprendo cifratura e integrità.
   - **MAC / PRF:** `SHA256`. Anche qui, l'hash SHA256 è utilizzato esclusivamente per la funzione pseudo-casuale (PRF) per la generazione delle chiavi, e non per l'integrità dei messaggi (gestita da GCM).

3. `TLS_RSA_WITH_AES_128_CBC_SHA`

   Questa cipher suite utilizza una modalità di cifratura a blocchi classica (CBC) invece di una modalità autenticata (AEAD), il che cambia radicalmente l'uso dell'algoritmo di hashing:

   - **Key Exchange e Autenticazione:** `RSA`. Analogamente al caso precedente, RSA gestisce lo scambio chiavi (tramite cifratura del *Pre-Master Secret*) e fornisce autenticazione implicita.

- **Cifratura Simmetrica:** `AES_128_CBC`. Viene usato AES in modalità **CBC** (*Cipher Block Chaining*). A differenza della modalità GCM, la modalità CBC garantisce *solo* la riservatezza (confidenzialità) dei dati, ma **non** la loro integrità.
- **MAC / PRF:** `SHA` (SHA-1). A causa dei limiti del CBC, l'algoritmo di hash qui svolge due funzioni distinte e fondamentali:
  - *Integrità (HMAC):* Poiché CBC non autentica i dati, SHA viene utilizzato per calcolare un **HMAC** (*Hash-based Message Authentication Code*) per **ogni record TLS**. Questo hash viene appeso al messaggio cifrato per permettere al ricevente di verificare che i dati non siano stati alterati durante il transito.
  - *Derivazione Chiavi (PRF):* Viene utilizzato, come di consueto, come base per la funzione pseudo-casuale (PRF) per generare le chiavi di sessione.

  Nota: L'uso di `SHA` (che implica SHA-1) e della modalità CBC è oggi spesso considerato obsoleto o meno sicuro rispetto alle suite GCM con SHA256/384, a causa di vulnerabilità storiche (es. attacchi di tipo *Padding Oracle* o collisioni su SHA-1).

Nota: Cambiamenti in TLS 1.3

In **TLS 1.3**, la nomenclatura delle Cipher Suite è stata semplificata (es. `TLS_AES_128_GCM_SHA256`).
L'algoritmo asimmetrico di scambio chiavi (Key Exchange) e di autenticazione **non viene più esplicitato** nella stringa per un motivo fondamentale:

- Il supporto allo scambio chiavi **RSA statico** (senza Forward Secrecy) è stato **rimosso**.
- È obbligatorio utilizzare esclusivamente scambi di chiavi effimeri (Perfect Forward Secrecy), ovvero **ECDHE** o **DHE**.

Di conseguenza, in TLS 1.3 l'algoritmo di scambio chiavi e il tipo di certificato (RSA o ECDSA) vengono negoziati separatamente tramite estensioni del protocollo e non sono più vincolati alla scelta della cipher suite.