

# Gestione della Connessione TLS e Supporto alle Applicazioni

## Indice

<b>1 Protocollo di Alert (Alert Protocol)</b>	<b>2</b>
1.1 Struttura del Pacchetto e Formato . . . . .	2
1.2 Esempi di Alert e Significato (Sample Alerts) . . . . .	2
1.3 Gestione degli Errori Fatali . . . . .	3
1.4 Limiti del Protocollo: DoS e Integrazione Legacy . . . . .	3
<b>2 Truncation Attack</b>	<b>4</b>
2.1 Truncation Attack (Attacco di Troncamento) . . . . .	4
2.2 Soluzione al Truncation Attack: Il "Close Notify" . . . . .	4
<b>3 Rinegoziazione (Renegotiation)</b>	<b>6</b>
3.1 Vulnerabilità e Superficie di Attacco . . . . .	7
<b>4 Renegotiation Attack</b>	<b>8</b>
4.1 Come funziona l'attacco . . . . .	9
4.2 Esempio Pratico . . . . .	10
4.3 Esempio: Attacco Twitter di Kurmus . . . . .	10
4.4 Prevenire l'Attacco di Rinegoziazione TLS . . . . .	11
4.4.1 Soluzione Immediata: Disabilitazione . . . . .	11
4.4.2 La Patch Standardizzata: RFC 5746 . . . . .	11
4.4.3 La Soluzione Definitiva: TLS 1.3 . . . . .	12
<b>5 DTLS (Datagram TLS)</b>	<b>12</b>
5.1 Panoramica e Obiettivi (RFC 4347) . . . . .	12
5.2 Differenze Chiave: Adattare TLS a un mondo inaffidabile . . . . .	12
<b>6 Opossum Attack: Application Layer Desynchronization using Opportunistic TLS</b>	<b>13</b>

# 1 Protocollo di Alert (Alert Protocol)

TLS definisce dei messaggi speciali per veicolare informazioni di “alert” (avviso) tra le parti coinvolte. Come indicato nella **RFC2246**, questi messaggi sono fondamentali per la gestione dello stato della connessione.

- I messaggi dell’Alert Protocol sono incapsulati in **Record TLS**.
- Di conseguenza, essi vengono cifrati e autenticati (se inviati dopo che l’handshake è stato completato); in caso contrario (durante le prime fasi dell’handshake), viaggiano in chiaro.

## 1.1 Struttura del Pacchetto e Formato

Come mostrato nello schema del protocollo, un Alert è composto da un header standard TLS seguito da un payload di soli 2 byte.

Content Type	Major Ver	Minor Ver	Length	Payload
0x15	Es. 0x03	Es. 0x03	0x0002	<i>Alert (2 Byte)</i>

Tabella 1: Incapsulamento dell’Alert nel Record TLS

Il payload dell’Alert (i 2 byte finali) è strutturato come segue:

### 1. Primo Byte: Alert Level (Livello)

- **warning(1)**: Avviso, la connessione può proseguire (es. `close_notify`).
- **fatal(2)**: Errore critico, la connessione deve terminare.

### 2. Secondo Byte: Alert Description

- Specifica il tipo esatto di errore (ci sono circa 23 alert definiti nello standard base).

## 1.2 Esempi di Alert e Significato (Sample Alerts)

Di seguito sono riportati gli alert più comuni con le relative conseguenze tecniche:

### `unexpected_message`

#### *Livello: Fatal*

Indica che è stato ricevuto un messaggio inappropriato per lo stato attuale della connessione. Non dovrebbe mai essere osservato in comunicazioni tra implementazioni corrette.

### `bad_record_mac`

#### *Livello: Fatal*

Indica che il record è stato ricevuto con un **MAC errato**. Questo implica che il messaggio è stato corrotto durante il transito o che c’è un disallineamento nelle chiavi crittografiche.

### `record_overflow`

#### *Livello: Fatal*

La lunghezza del payload cifrato eccede il limite consentito. Nello specifico, il limite è definito come:

$$2^{14} + 2048 \text{ bytes}$$

(ovvero la lunghezza massima del plaintext  $2^{14}$  più l’espansione permessa per la compressione e il padding cifrato).

### `handshake_failure`

#### *Livello: Fatal*

Il mittente non è stato in grado di negoziare un set accettabile di parametri di sicurezza (es. nessuna cipher suite in comune) date le opzioni disponibili.

### `Certificate Alerts`

`(bad_certificate, unsupported_certificate, certificate_revoked, certificate_expired, certificate_unknown)`

#### *Livello: Warning o Fatal* (dipende dall’implementazione)

Riguardano vari problemi con la validazione del certificato (corrotto, firma invalida, revocato, scaduto, ecc.).

### 1.3 Gestione degli Errori Fatali

#### Comportamento su Alert FATAL

Se il livello dell'alert è **Fatal**, il protocollo impone regole severe:

- La connessione deve essere **terminata immediatamente**.
- **Non è permesso** il *session resumption*: l'ID di sessione deve essere invalidato.
- Tutto lo stato di sicurezza associato (chiavi, segreti) deve essere cancellato (*clear all!*).

#### Nota

**Nota sull'utilità dell'Alert Protocol:** Oltre alla gestione degli errori, l'alert protocol è fondamentale per evitare attacchi di *truncation* (che vedremo più avanti). Il messaggio di alert `close_notify` assicura che la chiusura della connessione sia intenzionale e non causata da un attaccante che taglia la comunicazione (TCP FIN) prematuramente.

### 1.4 Limiti del Protocollo: DoS e Integrazione Legacy

**Nota importante: TLS NON protegge il livello TCP!**

- TLS opera a livello applicativo/sessione, ma si appoggia su TCP per il trasporto. Di conseguenza, è esposto agli attacchi mirati al livello sottostante.
- **TCP Reset Attack:** Chiunque sia in grado di spoofare un pacchetto (falsificare l'indirizzo IP) può terminare arbitrariamente una connessione cifrata inviando un flag `RST` (Reset). Il sistema operativo ricevente chiuderà la connessione TCP, interrompendo di conseguenza anche il tunnel TLS, senza che TLS possa farci nulla.

**Come si possono proteggere applicazioni vecchie che usano TCP in chiaro e non supportano nativamente SSL/TLS?**

Una soluzione comune, ma problematica, è l'uso di strumenti come **stunnel** ([www.stunnel.org](http://www.stunnel.org)).

- **Meccanismo:** Si crea un tunnel *TCP over TLS over TCP*. L'applicazione crede di parlare in TCP semplice, ma stunnel intercetta il traffico, lo cifra in TLS e lo invia sopra una connessione TCP reale. È una soluzione non ottimale, infatti viene usata come ultima soluzione. Inoltre, porta a gravi errori prestazionali. Incapsulare TCP dentro un altro tunnel TCP causa un grave degrado delle performance.
- **Alternativa Migliore:** Utilizzare tunnel basati su **DTLS** (Datagram TLS), che opera su UDP, o VPN moderne (es. WireGuard/OpenVPN in modalità UDP).

#### Nota

**Perché "TCP over TCP" è una pessima idea?**

La slide cita i *"Conflicting congestion control loops"* (conflitti nei cicli di controllo della congestione). Ecco cosa succede tecnicamente:

1. Quando un pacchetto viene perso sulla rete fisica (il tunnel esterno), il TCP esterno (livello trasporto) se ne accorge e riduce la velocità di trasmissione per gestire la congestione, provando a ritrasmettere il pacchetto perso.
2. Nel frattempo, il TCP interno (quello dell'applicazione incapsulata) non riceve l'ACK e interpreta questo ritardo come una perdita di pacchetto.
3. Di conseguenza, anche il TCP interno innesca il suo meccanismo di ritrasmissione e riduce la sua finestra di congestione.

**Il risultato:** Si generano ritrasmissioni inutili (il TCP interno ritrasmette dati che il TCP esterno sta già ritrasmettendo) che intasano ulteriormente il canale proprio quando dovrebbe scaricarsi. Questo fenomeno, noto come **TCP Meltdown**, porta a latenze enormi e stalli della connessione.

Per questo si consiglia **DTLS** (che usa UDP): UDP non ha controllo di congestione o ritrasmissione, quindi lascia che se ne occupi solo il TCP interno dell'applicazione, evitando il conflitto.

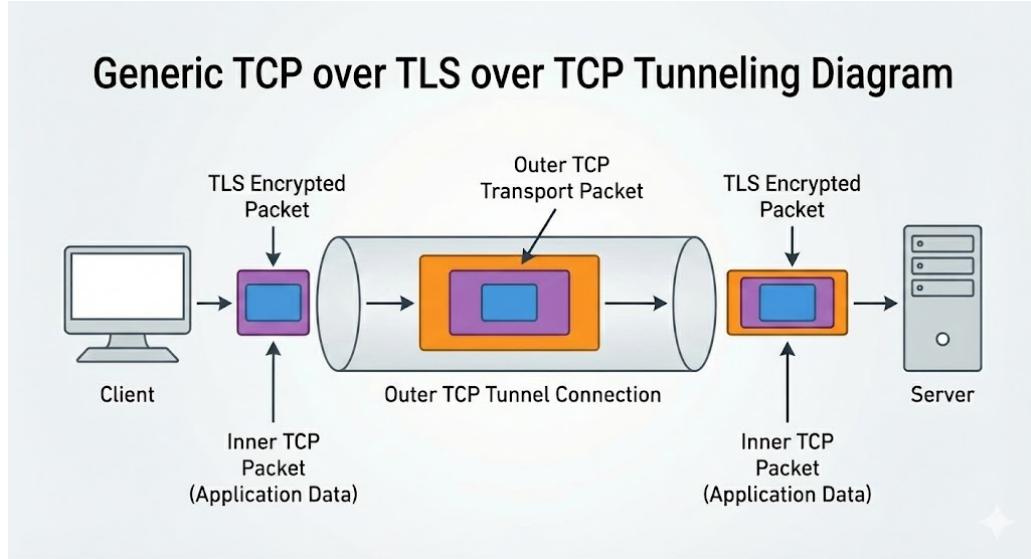


Figura 1: TCP over TLS over TCP

## 2 Truncation Attack

### 2.1 Truncation Attack (Attacco di Troncamento)

Per comprendere la natura di questo attacco, immaginiamo una comune conversazione telefonica:

- **Chiusura normale:** Gli interlocutori segnalano esplicitamente la fine della conversazione (es. “Ciao, alla prossima”) prima di riagganciare.
- **Truncation Attack:** La linea cade improvvisamente o viene tagliata mentre una persona sta ancora parlando. Chi ascolta potrebbe erroneamente dedurre che l’interlocutore avesse terminato il discorso, quando in realtà la frase è rimasta a metà.

Il protocollo TCP gestisce la terminazione della connessione tramite il flag FIN. Tuttavia, esiste una vulnerabilità strutturale nell’interazione tra i livelli ISO/OSI:

- **Mancata protezione del Trasporto:** Poiché TLS opera a un livello superiore (Applicazione/Sessione), non cifra né autentica gli header TCP.
- **L’Attacco:** Un attaccante (*Man-in-the-Middle*) può iniettare un pacchetto FIN falsificato (*spoofing*). Lo stack TCP della vittima accetta questo pacchetto come un comando di chiusura valido e termina la connessione.
- **Il paradosso dell’integrità:** In questo scenario, TLS garantisce perfettamente l’integrità crittografica di ogni *singolo record* (con HMAC) ricevuto fino a quel momento, ma non possiede nativamente l’**integrità della sessione completa**.

**Il Problema Fondamentale:** Senza un segnale di chiusura cifrato ed esplicito all’interno del tunnel TLS, il Client e il Server non hanno modo di distinguere se la connessione è stata chiusa perché la transazione è finita “normalmente” o se è intervenuto un attaccante, causando la perdita della parte finale dello scambio dati.

### 2.2 Soluzione al Truncation Attack: Il "Close Notify"

Per distinguere una chiusura legittima da un attacco, TLS introduce un meccanismo esplicito a livello di protocollo: l’alert **Close Notify**.

- **Natura del messaggio:** È un Alert di livello **warning** (codice descrizione 0). Poiché viaggia all’interno di un Record TLS, è **cifrato e autenticato** (protetto da MAC).
- **Funzione:** Informa esplicitamente la controparte che: *“Non verranno trasmessi ulteriori dati su questa connessione”*.
- **Emesso da chiunque:** Può essere inizializzato da qualsiasi delle due parti (Client o Server).

- **Semantica "Half-close" (Chiusura Parziale):** Come mostrato nella Figura 2, la chiusura non è immediata e brutale. Chi invia il `close_notify` dice "Io ho finito di scrivere", ma la connessione rimane attiva finché anche l'altra parte non risponde con il proprio `close_notify`.
- **Regola di Sicurezza (Truncation Defense):** Una connessione che termina bruscamente (ad esempio ricevendo un TCP FIN o RST) senza aver prima ricevuto un `close_notify` valido, deve essere considerata un potenziale **Truncation Attack**. I dati ricevuti ma non ancora elaborati dovrebbero essere trattati con sospetto o scartati.

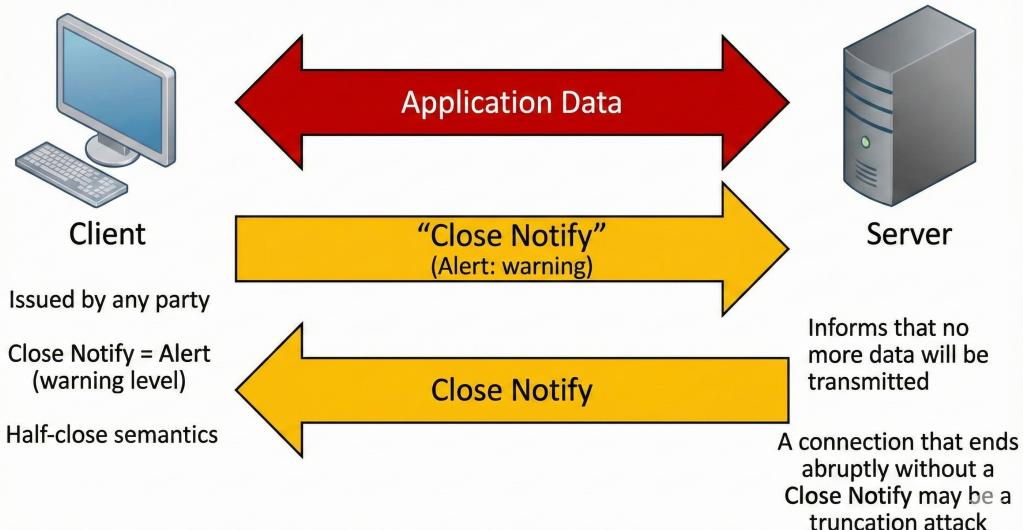


Figura 2: `send_notify`

#### Nota: Debolezza Strutturale

È importante ricordare che, sebbene il `close_notify` protegga dall'integrità della sessione (troncamento dei dati), **TLS non può proteggere il TCP**. L'attaccante può comunque chiudere il canale di trasporto (TCP DoS) inviando un RST spoofato. TLS se ne accorgerà (perché manca il `close_notify`), ma la connessione cadrà comunque.

#### Nota

##### Perché il TCP FIN non basta? (Autenticazione vs Trasporto)

La differenza fondamentale risiede nell'**autenticazione**.

- **TCP FIN:** È un pacchetto del livello di trasporto (Layer 4). Non ha protezione crittografica. Chiunque si trovi sulla rete (Man-in-the-Middle) può falsificarlo ("spoofing") e chiudere la connessione.
- **TLS Close Notify:** È un messaggio del livello applicativo/sessione (sopra TCP). Fa parte del flusso crittografato. Per generare un `close_notify` valido, l'attaccante dovrebbe conoscere le chiavi di sessione per calcolare il MAC corretto. Essendo impossibile falsificarlo, la sua presenza garantisce che la richiesta di chiusura provenga autenticamente dalla controparte legittima.

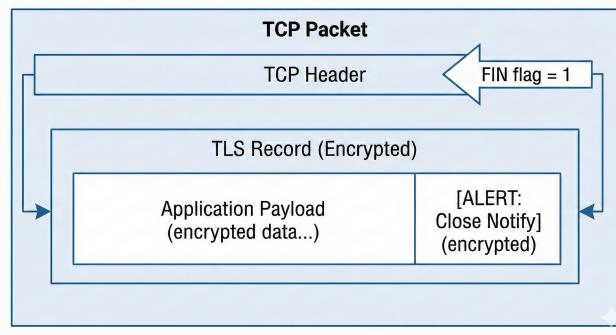


Figura 3: TCP FIN + TLS close\_notify

### 3 Rinegoziazione (Renegotiation)

Ci siamo chiesti: se una Sessione TLS può coprire più connessioni TCP (tramite *Session Resumption*), è possibile il contrario? Ovvero, eseguire nuovi handshake all'interno di una singola connessione TCP persistente?

La risposta è affermativa. Poiché le connessioni TCP moderne sono longeve (*Keep-Alive*), TLS prevede un meccanismo di **Rinegoziazione**.

- **Definizione:** La rinegoziazione è un meccanismo che permette di avviare un nuovo Handshake (e quindi stabilire nuovi parametri di crittografia o autenticazione) all'interno di un tunnel TLS già esistente e cifrato.
- **Come funziona:** Mentre i dati fluiscono, una delle due parti invia un messaggio di `HelloRequest` (in TLS 1.2). Questo innesca un secondo handshake che viene completamente cifrato dalle chiavi del primo.

Supponiamo di trovarci nel seguente scenario:

1. Il Client si connette a un sito web via HTTPS. L'handshake iniziale garantisce la cifratura, ma il Client è anonimo (non invia il proprio certificato).
2. L'utente naviga e decide di accedere a un'area riservata.
3. Il Server, accorgendosi che la risorsa richiede maggiore sicurezza, chiede una **Rinegoziazione** richiedendo stavolta il **Certificato Client**.
4. Avviene un nuovo handshake (invisibile all'utente, perché dentro al tunnel cifrato) dove il Client si autentica.

La rinegoziazione non deve essere confusa con il semplice *rekeying* (la rotazione delle chiavi) né con la *session resumption*; si tratta di un meccanismo molto più flessibile che consente di eseguire un **nuovo handshake completo** all'interno di una connessione già attiva.

La peculiarità di questo processo è che il nuovo handshake avviene all'interno del tunnel sicuro preesistente: di conseguenza, tutti i messaggi scambiati per negoziare i nuovi parametri sono **crittografati** dalla ciphersuite della sessione precedente, rimanendo invisibili a osservatori esterni. Al termine, viene generato un **nuovo ID di sessione**.

**A cosa serve?** Lo scopo principale è elevare il livello di sicurezza in corso d'opera. Lo scenario più comune è l'**autenticazione ritardata del Client**: un utente può iniziare a navigare su un sito pubblico in modo anonimo (solo il server è autenticato), ma quando tenta di accedere a un'area riservata, il server avvia una rinegoziazione per richiedere il certificato dell'utente (passando così alla *Mutual Authentication*) o per negoziare una cifratura più robusta, il tutto senza dover chiudere e riaprire la connessione TCP.

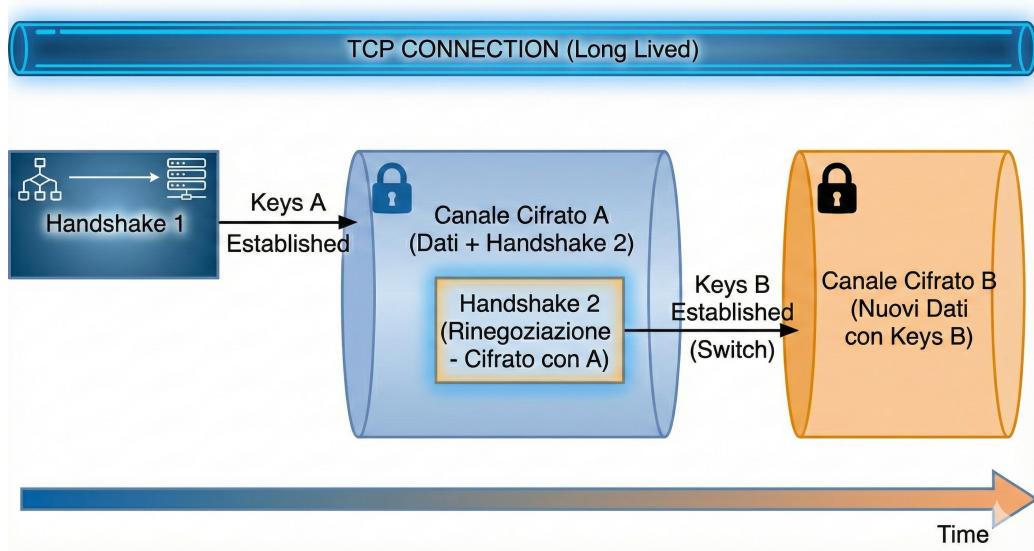


Figura 4: Single TCP connection, multiple TLS Sessions

### Nota

#### Rinegoziazione vs. Ripresa di Sessione (Resumption)

Questi due concetti sono spesso confusi, ma sono opposti:

- **Session Resumption (Ripresa):** Serve a *evitare* un handshake completo. Si usa quando ci si *riconnette* dopo che una sessione precedente è stata chiusa. È un handshake abbreviato per ristabilire rapidamente una sessione (stessi parametri, nuove chiavi) risparmiando risorse.
- **Renegotiation (Rinegoziazione):** È un handshake completo (o quasi) che avviene *durante* una sessione attiva. Serve a *cambiare* i parametri della sessione (es. richiedere un certificato client, cambiare ciphersuite) senza chiudere la connessione.

Sebbene concettualmente si parli di "nuova sessione", è tecnicamente impreciso affermare che la vecchia sessione venga "chiusa" (non viene inviato alcun `close_notify`).

Il processo è più simile a una **sostituzione a caldo (hot swap)**:

- **Continuità del Canale:** Il flusso di dati TCP non subisce alcuna interruzione né chiusura.
- **Lo "Switch":** Al termine del nuovo handshake (che avviene protetto dalle vecchie chiavi), le parti si inviano il messaggio `ChangeCipherSpec`.
- **Passaggio di Consegnate:** Questo messaggio funge da interruttore istantaneo: dal byte successivo in poi, le vecchie chiavi vengono scartate dalla memoria e il traffico viene cifrato immediatamente con le nuove chiavi appena negoziate.

### 3.1 Vulnerabilità e Superficie di Attacco

L'introduzione della rinegoziazione, sebbene utile per la flessibilità (es. autenticazione ritardata), ha ampliato significativamente la superficie di attacco del protocollo. Come spesso accade in sicurezza informatica, l'aggiunta di complessità funzionale introduce nuovi rischi se non accuratamente isolata.

Un flusso tipico di rinegoziazione si svolge come segue:

1. **Handshake Iniziale:** Autenticazione e scambio chiavi standard.
2. **Sessione 1 (Attiva):** Scambio di dati applicativi cifrati con le chiavi iniziali.
3. **Handshake di Rinegoziazione:** Una delle parti richiede nuovi parametri. Questo handshake avviene *all'interno* del tunnel cifrato della Sessione 1.
4. **Sessione 2 (Attiva):** Generazione di un nuovo ID sessione e nuove chiavi per proteggere i dati successivi.

Il design originale della rinegoziazione presentava difetti logici fondamentali che sono stati sfruttati per attacchi di *Plaintext Injection* (iniezione di testo in chiaro):

- **Mancanza di “Binding” Crittografico (Il problema dell’Indistinguibilità):** Sebbene il secondo handshake sia cifrato, il protocollo originale non verificava che fosse crittograficamente legato al primo. Per il Server, il secondo handshake appariva come una *nuova* negoziazione indipendente (spesso con `session_id = 0`), arrivata semplicemente attraverso un canale cifrato.  
*Conseguenza:* Un attaccante (Man-in-the-Middle) poteva stabilire una connessione con il server, inviare dei dati parziali, e poi “infiltrare” l’handshake della vittima come se fosse una rinegoziazione della *sua* connessione.
- **Gestione dei Dati Applicativi e Bufferizzazione:** Durante la rinegoziazione, il flusso dati non si interrompe necessariamente. La rinegoziazione ha la precedenza sui dati applicativi, ma il comportamento del server è critico:
  1. Se il client invia dati applicativi mentre il server sta gestendo la rinegoziazione, il server deve metterli in un **buffer**.
  2. Una volta completata la rinegoziazione e attivate le chiavi della **Sessione 2** (magari autenticata tramite certificato Client), il server svuota il buffer.
  3. **Il Rischio:** Il server elabora i dati bufferizzati (che potrebbero essere stati inviati *prima* dell’autenticazione) nel contesto di sicurezza della **NUOVA** sessione autenticata.
- **Exploitation:** Combinando i due punti precedenti, è possibile un attacco in cui i dati inviati dall’attaccante (es. una richiesta HTTP parziale) vengono preposti (“prefixing”) ai dati legittimi della vittima. Il server, unendo i flussi dopo la rinegoziazione, vedrà un’unica richiesta valida composta da:

[Dati Attaccante] + [Credenziali Vittima]

Eseguendo così la richiesta dell’attaccante con i privilegi della vittima.

## 4 Renegotiation Attack

La scoperta della vulnerabilità di rinegoziazione rappresenta un momento di svolta nella storia di TLS, dimostrando come anche protocolli maturi possano nascondere difetti logici critici.

Questa vulnerabilità fu identificata per la prima volta da **Marsh Ray** (di PhoneFactor Inc.) tra l’Agosto e il Settembre del 2009. Data la gravità del problema — che risiedeva nel design del protocollo stesso e non in una specifica implementazione software — la scoperta fu mantenuta riservata per coordinare una risposta, venendo resa pubblica solo nel Novembre 2009.

Un aspetto notevole di questa vicenda fu la reattività degli organi di standardizzazione (IETF). Di norma, la modifica di uno standard crittografico richiede anni; in questo caso, la criticità era tale che la comunità lavorò febbrilmente, producendo una correzione allo standard (la RFC 5746) in soli **tre mesi**, un tempo record rispetto ai canonici cicli di sviluppo di un anno e mezzo.

Tecnicamente, si tratta di un attacco di **Plaintext Injection** (iniezione di testo in chiaro). L’attaccante, posizionato come Man-in-the-Middle, non decifra il traffico della vittima, ma sfrutta la rinegoziazione per “incollare” il proprio traffico arbitrario all’inizio del flusso legittimo della vittima.

Il server, non distinguendo tra la sessione iniziale dell’attaccante e la rinegoziazione innescata con la vittima, concatena i due flussi di dati. Il risultato è che il comando dell’attaccante viene elaborato dal server **nel contesto di sicurezza e autenticazione della vittima**.

Inizialmente, l’efficacia dell’attacco sembrava limitata e teorica. Tuttavia, a metà Novembre 2009, **Anil Kurmus**, uno studente dell’ETH di Zurigo, dimostrò il contrario trasformando questa vulnerabilità in un exploit pratico contro **Twitter**.

L’attacco ideato da Kurmus era ingegnoso:

1. L’attaccante avviava una connessione TLS con Twitter e inviava una richiesta parziale per postare un tweet (es. `POST /statuses/update...`), lasciandola in sospeso.
2. Quando la vittima tentava di connettersi, l’attaccante sfruttava la vulnerabilità di rinegoziazione per far sì che la richiesta di handshake della vittima venisse interpretata dal server come una continuazione della sessione dell’attaccante.
3. Il server univa i flussi: prendeva la richiesta di “post” dell’attaccante e usava i cookie di autenticazione inviati dalla vittima subito dopo.

- Risultato:** L'attaccante riusciva a postare tweet a nome della vittima, sfruttando le sue credenziali attive, senza mai doverle rubare o decifrare.

Twitter reagì immediatamente con l'unica contromisura disponibile al tempo: **disabilitare completamente la rinegoziazione** sui propri server, accettando il compromesso di rompere la compatibilità con alcuni client pur di chiudere la falla.

L'evento fu uno shock per il mondo della sicurezza e per gli standard body. Ha dimostrato che funzioni accessorie e complesse, se non isolate correttamente, possono compromettere l'intera architettura di sicurezza.

### La Regola d'Oro (KISS Principle)

La lezione fondamentale da portare a casa è: **MANTENERE LE COSE SEMPLICI (Keep It Simple, Stupid).**

Ogni funzionalità aggiuntiva (come la rinegoziazione) aumenta la superficie di attacco. Nella sicurezza informatica, la complessità è il nemico numero uno: un protocollo più semplice è più facile da analizzare, implementare e difendere.

## 4.1 Come funziona l'attacco

Nello schema in Figura 5, non è necessario assumere che l'attaccante debba prevedere il momento esatto della connessione del Client. Operando come Man-in-the-Middle, l'attaccante stabilisce preventivamente il canale con il Server (inviando il payload parziale) e rimane in attesa; non appena intercetta l'handshake della vittima, lo inoltra all'interno del tunnel già attivo, trasformandolo in una richiesta di rinegoziazione.

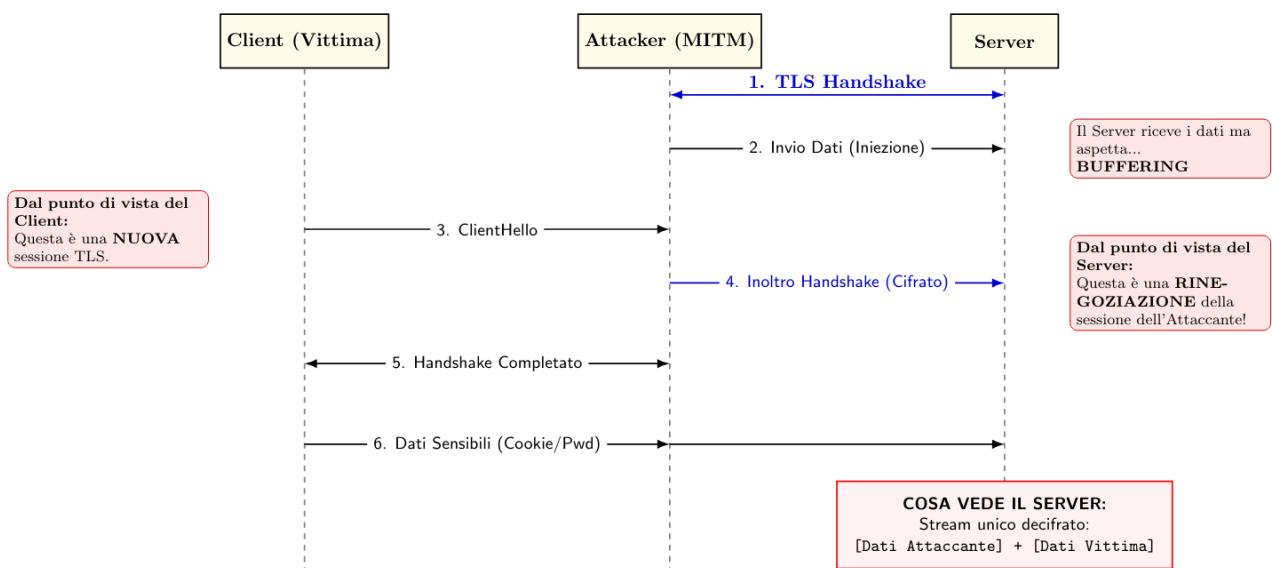


Figura 5: Schema "Renegotiation Attack"

- Setup (Fase 1):** L'Attaccante (MITM) stabilisce una connessione TLS valida con il Server. In questa fase il Client (vittima) **non** è ancora coinvolto.
- Iniezione (Fase 2):** L'Attaccante invia dei dati applicativi incompleti (es. l'inizio di una richiesta POST) attraverso questo canale cifrato. Il Server riceve i dati ma, essendo incompleti, li mette in **buffer** (attende il resto).
- Intercettazione (Fase 3):** Il Client tenta di connettersi inviando un **ClientHello**. Dal punto di vista del Client, questa è l'inizio di una **NUOVA sessione TLS**.
- L'Inganno (Fase 4):** L'Attaccante intercetta il **ClientHello** e, invece di rispondere direttamente, lo **inoltra** al Server all'interno del tunnel cifrato già esistente.
- Disallineamento:** Qui avviene il cuore dell'attacco:
  - Il **Server** interpreta l'handshake in arrivo come una richiesta di **RINEGOZIAZIONE** della sessione dell'Attaccante.

- Il Client crede di negoziare una sessione pulita.
- Esecuzione (Fase 5-6): L'handshake si completa (tramite l'attaccante). Il Client invia ora i suoi dati sensibili (es. Cookie/Password).
  - Risultato Finale: Il Server decifra il flusso e accoda i dati del Client a quelli precedentemente bufferizzati dell'Attaccante. Il Server esegue quindi un'unica richiesta composta da:

[Dati Attaccante] + [Dati Vittima]

**Risultato:** L'attaccante è stato in grado di **iniettare un prefisso** plaintext (scelto da lui) all'inizio di una richiesta autenticata dal client.

## 4.2 Esempio Pratico

La gravità dell'attacco dipende dalla logica dell'applicazione. Di seguito è riportato un esempio classico in cui un attaccante ordina una pizza a spese della vittima sfruttando l'iniezione di dati.

### 1. L'Attaccante invia (Prefix):

L'attaccante invia una richiesta incompleta. Nota fondamentale: l'ultima riga dell'header personalizzato **non** ha il ritorno a capo finale (CRLF).

```
GET /pizza?toppings=pepperoni;address=attacker_address HTTP/1.1
X-Ignore-This: (nessun ritorno a capo!)
```

### 2. La Vittima invia (Normale):

La vittima, ignara, invia la sua richiesta legittima tramite la connessione (che crede sia nuova).

```
GET /pizza?toppings=sausage;address=victim_address HTTP/1.1
Cookie: victim_cookie
```

### 3. RISULTATO: Richieste "Incollate" (Glued Requests):

Il server, avendo bufferizzato i dati dell'attaccante e accodato quelli della vittima, vede un unico flusso continuo. La richiesta della vittima viene "mangiata" dall'header dell'attaccante.

```
GET /pizza?toppings=pepperoni;address=attacker_address HTTP/1.1
X-Ignore-This: GET /pizza?toppings=sausage;address=victim_address HTTP/1.1
Cookie: victim_cookie
```

### Nota

#### Analisi del Risultato:

- La prima riga (l'ordine dell'attaccante) è valida.
- La seconda riga inizia con un header inutile (X-Ignore-This). Poiché l'attaccante non ha messo il ritorno a capo, la prima riga della richiesta della vittima (GET /pizza...) viene interpretata dal server come il **contenuto** di quell'header inutile e quindi ignorata.
- La terza riga (Cookie: victim\_cookie), che apparteneva alla vittima, viene ora interpretata come parte della richiesta dell'attaccante.
- **Conclusione:** Il server usa l'account (Cookie) della vittima per inviare una pizza all'indirizzo dell'attaccante!

## 4.3 Esempio: Attacco Twitter di Kurmus

Questo esempio illustra uno scenario tecnicamente più sofisticato rispetto a quello della "Pizza", molto simile a quello realmente utilizzato da Anil Kurmus per attaccare Twitter. Qui l'obiettivo non è fare un ordine, ma **esfiltrare dati** o far pubblicare al server dati sensibili della vittima.

### 1. L'Attaccante invia (Prefix):

L'attaccante invia una richiesta POST a una pagina che accetta input (es. un forum o una API per postare messaggi).

La richiesta termina con il nome di un parametro (es. Message=) **senza** alcun ritorno a capo o valore.

```

POST /forum/send.php HTTP/1.0
... header vari ...
Message = (nessun ritorno a capo!)

```

## 2. La Vittima invia (Normale):

La vittima invia la sua normale richiesta HTTP, che include i suoi header di autenticazione.

```

GET / HTTP/1.1 [...]
Authorization: Basic dXNlcjpwYXNz...

```

## 3. RISULTATO: Richieste "Incollate" (Glued Requests):

Il server concatena i flussi. L'intera richiesta della vittima (inclusi i suoi header segreti) viene interpretata come il **valore** del parametro **Message** dell'attaccante.

```

POST /forum/send.php HTTP/1.0
...
Message = GET / HTTP/1.1 [...] Authorization: Basic dXNlcjpwYXNz...

```

Il concetto sopra esposto è esattamente quello usato contro Twitter:

- **Meccanismo:** L'attacco sfruttava le API di Twitter. L'attaccante inviava un prefisso per postare un tweet (**status=**).
- **Risultato nel Mondo Reale:** Il server di Twitter riceveva la richiesta incollata e pubblicava letteralmente un Tweet sull'account della vittima.
- **Il Danno:** Il testo del Tweet conteneva l'intera richiesta HTTP della vittima, inclusi gli header di autenticazione contenenti **username** e **password codificati in Base64!** In pratica, la vittima "twittava" le proprie credenziali al mondo intero.

## 4.4 Prevenire l'Attacco di Rinegoziazione TLS

A seguito della scoperta della vulnerabilità nel 2009, la comunità di sicurezza ha risposto in tre fasi distinte per mitigare e risolvere il problema.

### 4.4.1 Soluzione Immediata: Disabilitazione

Appena resa pubblica la vulnerabilità, non esisteva ancora una patch software pronta. L'unica contromisura efficace nell'immediato fu quella di **disabilitare completamente la rinegoziazione** lato server. Sebbene efficace nel bloccare l'attacco, questa azione rompeva la compatibilità con alcune funzionalità legittime (come l'autenticazione client richiesta a metà sessione), ma era necessaria per proteggere i sistemi.

### 4.4.2 La Patch Standardizzata: RFC 5746

Per risolvere il problema alla radice senza eliminare la funzionalità, l'IETF standardizzò urgentemente una correzione nel Febbraio 2010: la **RFC 5746** ("TLS Renegotiation Indication Extension").

Il concetto chiave introdotto è il **Binding Crittografico**: bisogna legare matematicamente la nuova rinegoziazione alla sessione precedente, in modo che un attaccante non possa "innestare" una sua sessione in quella della vittima.

Il meccanismo funziona tramite una nuova **Estensione TLS**:

1. **Primo Messaggio (Handshake Iniziale):** Il Client e il Server si scambiano un'estensione vuota (o uno speciale valore di cipher suite SCSV) per segnalare: "*Io supporto la rinegoziazione sicura (RFC 5746)*".
2. **Secondo Messaggio (Rinegoziazione):** Quando avviene una rinegoziazione, l'estensione non è più vuota. Essa deve contenere i dati di verifica (**Finished Message**) dell'handshake precedente.

**Perché funziona?**

- Un attaccante (MITM) non possiede le chiavi della vittima, quindi non può generare o conoscere il **Finished Message** della sessione crittografata della vittima.
- Se il Server riceve una richiesta di rinegoziazione che **non** contiene i dati corretti della sessione precedente, capisce che c'è un tentativo di attacco (discontinuità della sessione) e chiude la connessione.
- Rilevare l'attacco diventa quindi **banale**.

#### 4.4.3 La Soluzione Definitiva: TLS 1.3

Con l'arrivo di TLS 1.3, si è adottato un approccio filosofico diverso: la semplificazione.

##### TLS 1.3: "Keep it Simple"

In TLS 1.3 la rinegoziazione è stata **completamente proibita**.

Applicando la regola del "*Keep it simple*", si è deciso di rimuovere una funzionalità complessa e prona agli errori. Le necessità che prima venivano coperte dalla rinegoziazione (come l'aggiornamento delle chiavi o la richiesta di certificati client) sono ora gestite da meccanismi specifici e più leggeri (es. `KeyUpdate`, `Post-Handshake Authentication`), eliminando strutturalmente la superficie di attacco.

## 5 DTLS (Datagram TLS)

### 5.1 Panoramica e Obiettivi (RFC 4347)

- **Standard:** Definito nella **RFC 4347** (Aprile 2006).
- **Definizione:** È essenzialmente **TLS adattato per UDP**.
- **Obiettivo di design:** L'obiettivo primario era rendere DTLS il più simile possibile a TLS esistente, minimizzando le nuove invenzioni di sicurezza e massimizzando il riutilizzo del codice e dell'infrastruttura TLS.

#### Nota

##### Perché serve DTLS?

TLS standard si affida a TCP per garantire che i dati arrivino integri e in ordine. Se usassimo TLS su UDP, un singolo pacchetto perso romperebbe l'intera crittografia (poiché il calcolo del MAC dipende dall'ordine esatto dei pacchetti).

Tuttavia, applicazioni "Real-Time" come VoIP (telefonate su IP), streaming video live, VPN moderne (es. OpenVPN UDP) e gaming online non possono usare TCP a causa della latenza (l'Head-of-Line Blocking). DTLS nasce per offrire la sicurezza di TLS a queste applicazioni che tollerano la perdita di dati ma richiedono velocità.

### 5.2 Differenze Chiave: Adattare TLS a un mondo inaffidabile

Poiché UDP non fornisce garanzie di trasporto, DTLS deve implementare a livello applicativo le funzionalità che TCP forniva gratuitamente a TLS. Ecco il confronto punto per punto:

#### 1. Ordine di consegna (Orderly Delivery)

*Problema TLS:* TLS assume che i pacchetti arrivino in ordine. Usa un numero di sequenza *implicito* (non inviato nel pacchetto) per il MAC.

*Soluzione DTLS:* Poiché in UDP i pacchetti possono arrivare disordinati, DTLS aggiunge un **numero di sequenza esplicito** (64-bit) direttamente nell'header di ogni Record. Questo permette al ricevente di processare i pacchetti nell'ordine corretto (o scartare i duplicati/replay) anche se arrivano mescolati.

#### 2. Affidabilità (Reliability)

*Problema TLS:* TLS assume che il trasporto sia affidabile. Se invia un messaggio di Handshake, sa che arriverà.

*Soluzione DTLS:* UDP perde pacchetti. DTLS deve gestire la perdita, specialmente durante l'Handshake (che deve completarsi con successo). Introduce quindi un meccanismo di **Timeout** e **Ritrasmissione** interno: se non riceve risposta entro un certo tempo, ritrasmette l'ultimo volo di messaggi.

#### 3. Frammentazione (Packet Size)

*Problema TLS:* I record TLS possono essere grandi fino a 16KB. TCP si occupa di spezzettarli e ricomporli automaticamente.

*Soluzione DTLS:* Un pacchetto UDP non può superare l'MTU (spesso 1500 byte) senza causare frammentazione IP (che è dannosa). DTLS introduce la capacità di **frammentare i messaggi di handshake** a livello di protocollo (es. un certificato grande viene diviso in più messaggi DTLS) e raccomanda l'uso della *Path MTU Discovery* per evitare pacchetti troppo grandi.

#### 4. Orientamento alla Connessione

*Problema TLS:* Si appoggia allo stato della connessione TCP (SYN/FIN).

**Soluzione DTLS:** Poiché UDP è *connectionless* (senza stato), il concetto di "connessione" in DTLS è puramente crittografico. La durata della sessione è definita esplicitamente:

Connessione DTLS = [TLS Handshake ... TLS Closure Alert]

La connessione esiste fintanto che non viene inviato un alert di chiusura (o scade un timeout di inattività).

## 6 Opossum Attack: Application Layer Desynchronization using Opportunistic TLS

**Opossum** è un attacco di desincronizzazione a livello applicativo "cross-protocol" che colpisce i protocolli applicativi basati su TLS che si affidano sia al TLS opportunistico che a quello implicito. Tra i protocolli colpiti figurano HTTP, FTP, POP3, SMTP, LMTP e NNTP.

Opossum consente agli attaccanti di violare l'integrità dei canali TLS sicuri iniettando messaggi inaspettati, causando una desincronizzazione tra client e server. L'attacco richiede una posizione Man-in-the-Middle (MITM) e prende di mira i servizi che supportano simultaneamente sia il TLS implicito (porta dedicata) che il TLS opportunistico (meccanismo di aggiornamento/upgrade).

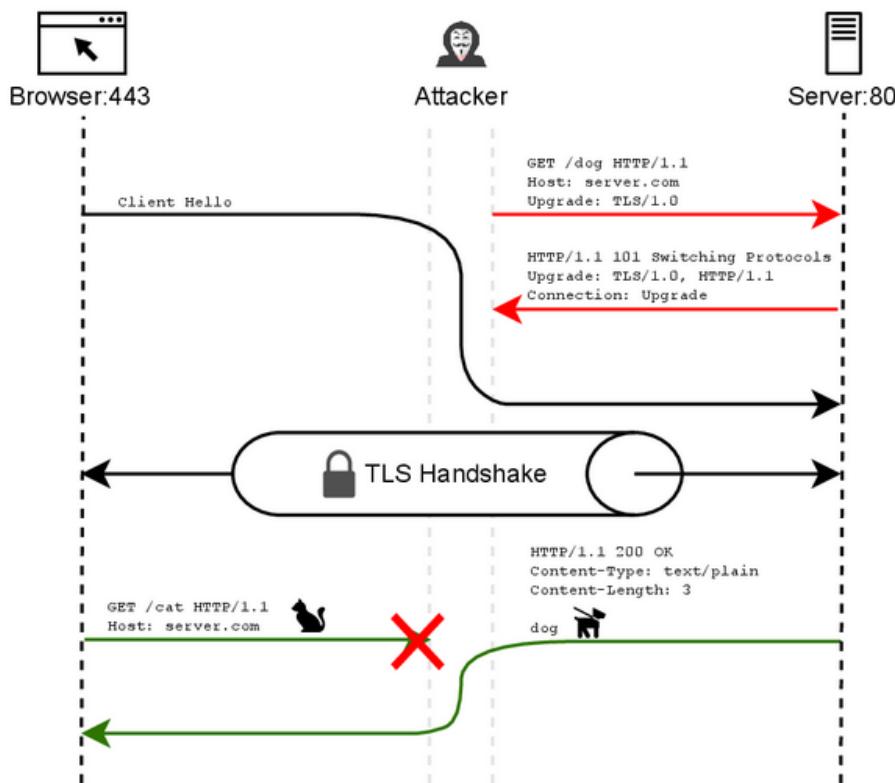


Figura 6: Opossum Attack

Per capire l'attacco Opossum, bisogna prima chiarire la differenza tra i due modi in cui un server può offrire una connessione sicura (TLS):

- **TLS Implicito (Implicit TLS):** Il server ascolta su una porta dedicata (es. 443 per HTTPS). Appena il client si connette, deve iniziare immediatamente l'handshake TLS. Se invia dati in chiaro, la connessione fallisce.
- **TLS Opportunistico (Opportunistic TLS / STARTTLS):** Il server ascolta sulla porta standard "in chiaro" (es. 80 per HTTP). La connessione inizia in chiaro (plaintext); poi il client invia un comando speciale (come STARTTLS) per chiedere di passare alla cifratura.

La vulnerabilità sfruttata consiste nel fatto che molti server moderni supportano entrambi i metodi. L'attacco Opossum sfrutta il fatto che, spesso, il software del server gestisce queste due modalità in modo non perfettamente isolato o con macchine a stati che possono essere confuse.

L'attacco fa leva su una vulnerabilità specifica nelle implementazioni server che supportano il **TLS Implicito**. In teoria, tali server dovrebbero attendersi esclusivamente un handshake TLS all'apertura della connessione; tuttavia, a causa di configurazioni permissive o errori di implementazione, alcuni server accettano e **bufferizzano dati in chiaro** ricevuti prima che l'handshake sia completato, invece di scartarli come traffico non valido.

Sfruttando questa finestra, un attaccante (Man-in-the-Middle) inietta un comando in chiaro (come un comando SMTP RSET o una richiesta HTTP) proprio mentre il Client sta iniziando la negoziazione TLS. Il server, ingannato dalla propria logica di buffering, accoda questi dati malevoli in attesa di processarli.

La **desincronizzazione** si concretizza non appena il tunnel TLS viene stabilito: il server elabora immediatamente i dati presenti nel buffer, eseguendo il comando dell'attaccante *all'interno* del contesto sicuro. Di conseguenza, quando il Client invia la sua prima richiesta legittima, il server si trova in uno stato disallineato (avendo già consumato un'operazione); la risposta inviata potrebbe riferirsi al comando dell'attaccante, oppure la richiesta del Client potrebbe essere erroneamente interpretata come argomento del messaggio precedente, compromettendo l'integrità della sessione.

La Figura 6 illustra un attacco *Cross-Protocol* in cui il Client usa **TLS Implicito** (porta 443) e il Server usa **TLS Opportunistico** (porta 80).

1. **Iniezione:** L'Attaccante intercetta il `ClientHello` del browser e invia al server una richiesta in chiaro (`GET /dog`) con l'header `Upgrade: TLS/1.0`.
2. **Upgrade:** Il Server accetta l'upgrade (101 Switching Protocols) e stabilisce il tunnel TLS usando il `ClientHello` originale della vittima.
3. **Desincronizzazione:**
  - Il **Server** risponde alla richiesta iniettata dall'attaccante (`/dog`) inviando i dati nel tunnel cifrato.
  - Il **Client** invia la sua richiesta legittima (`/cat`).
  - Il **Client riceve "dog"** ma lo associa alla sua richiesta per "cat".

In conclusione, l'integrità della sessione è violata. Il browser visualizzerà il contenuto scelto dall'attaccante (`/dog`) credendo che sia la risposta alla sua richiesta legittima (`/cat`), poiché la risposta è arrivata attraverso un canale TLS valido e autenticato.