Gestione della Connessione TLS e Supporto alle Applicazioni

Trascrizione da Presentazione

1 Protocollo di Alert (Alert Protocol)

TLS (Transport Layer Security) definisce messaggi speciali per veicolare informazioni di "alert" (avviso) tra le parti coinvolte.

- I messaggi dell'Alert Protocol sono incapsulati in Record TLS.
- Di conseguenza, sono crittografati e autenticati (dopo l'handshake iniziale).

1.1 Formato dell'Alert Protocol (2 byte)

- **Primo byte: Livello di Alert (Alert Level)**
  - 'warning(1)' (avviso)
  - 'fatal(02)' (fatale)
- **Secondo Byte: Descrizione dell'Alert (Alert Description)**
  - Ci sono 23 possibili alert definiti.

Spiegazione Didattica

A cosa serve l'Alert Protocol?
L'Alert Protocol è il meccanismo con cui le due parti di una connessione TLS (client e server) si notificano reciprocamente di eventi significativi o errori. È fondamentale per due motivi:

1. **Gestione degli Errori Sicura:** Segnalando un errore (es. un MAC non valido o un certificato scaduto) attraverso un messaggio di alert crittografato, si evita che un attaccante (Man-in-the-Middle) possa intercettare o dedurre informazioni sullo stato della connessione dagli errori.
2. **Chiusura Corretta:** È l'unico modo sicuro per terminare una connessione. Come vedremo, il 'close$_n otify$'$(unalertdilivello'warning')essenzialeperprevenireattacchiditroncamento.$

1.2 Formato del Record TLS per un Alert

Quando un messaggio di alert viene inviato, è contenuto in un Record TLS standard. L'header di questo record avrà i seguenti campi:

- **Contenttype (Tipo di Contenuto):** 0x15 (il codice standard per un Alert)
- **Majorversion (Versione Maggiore):** (es. 0x03 per TLS 1.x)
- **Minorversion (Versione Minore):** (es. 0x03 per TLS 1.2)
- **Length of fragment (Lunghezza):** 0x0002 (poiché il payload dell'alert è sempre di 2 byte)
- **Payload (Payload):** I 2 byte dell'alert (Livello e Descrizione)

1.3 Esempi di Alert

Ecco alcuni degli alert più comuni:

- **unexpected_message:** Messaggio inappropriato ricevuto. È considerato **Fatale**. Non dovrebbe mai essere osservato in una comunicazione tra implementazioni corrette (indica un errore grave di logica o un attacco).

- **bad_record_mac:** Un record è stato ricevuto con un MAC (Message Authentication Code) non corretto. È **Fatale**. Questo indica che il messaggio è stato manomesso o danneggiato, o che le chiavi di sessione sono errate.
- **record_overflow:** La lunghezza del record supera il massimo consentito (2^{14}+2048 byte). È **Fatale**.
- **handshake_failure:** Impossibile negoziare un set accettabile di parametri di sicurezza (ciphersuite) date le opzioni disponibili. È **Fatale**.
- **bad_certificate, unsupported_certificate, certificate_revoked, certificate_expired, certificate_unknown:** Vari problemi con un certificato (corrotto, firme non verificate, non supportato, revocato, scaduto, altri problemi non specificati che lo rendono inaccettabile). Possono essere **Warning o Fatale**, a seconda dell'implementazione e della gravità.

Nota: Se un alert è **fatale**, la connessione deve essere terminata immediatamente e non deve essere permesso il *session resumption* con gli stessi parametri di sicurezza (tutto lo stato di sessione deve essere cancellato).

Spiegazione Didattica

Differenza tra 'Warning' e 'Fatal'

- **Fatal (Fatale):** Questo tipo di alert indica un errore che compromette la sicurezza o la fattibilità della connessione. Quando viene inviato o ricevuto un alert fatale, entrambe le parti devono terminare immediatamente la connessione e invalidare qualsiasi chiave di sessione e ID di sessione associati. Proseguire non è sicuro. Esempi: 'bad_record_mac', '$handshake_failure$'.
- **Warning (Avviso):** Questo tipo di alert notifica un evento che non è necessariamente un errore fatale. La connessione può (ma non deve) continuare. L'esempio più importante è il 'close_notify', $cheunavvisocheinformal'altrapartechenonverrannoinviatipidati, permettendounachiusuraordinat$

1.4 Debolezze e Attacchi DoS (Denial of Service)

Nota importante: TLS NON protegge TCP!

- TLS è esposto ad attacchi DoS a livello TCP.
- Chiunque può terminare una connessione TCP.
- È sufficiente inviare un pacchetto TCP RST (Reset) "spoofato" (con indirizzo IP falsificato).

Spiegazione Didattica

TLS opera sopra TCP (Livello 7 vs Livello 4)
Questo è un concetto fondamentale. TLS opera al livello "Sessione/Applicazione" (Layer 5-7 del modello OSI) e si affida al protocollo TCP (Layer 4) per il trasporto.
TLS protegge i **dati** all'interno del tunnel (confidenzialità, integrità, autenticazione), ma non può proteggere il **tunnel** stesso (la connessione TCP).
Se un attaccante invia un pacchetto TCP RST (Reset) con i giusti numeri di sequenza (che possono essere indovinati o intercettati) al server o al client, il sistema operativo di quella macchina chiuderà la connessione TCP a livello kernel. Il kernel non ha visibilità su TLS; per lui, è solo una connessione TCP come un'altra. TLS si accorgerà solo che la connessione sottostante è "morta" senza un preavviso.

2 Supporto ad Applicazioni non "TLS-aware"

Come proteggere applicazioni che non sono state programmate per usare SSL/TLS?

- **stunnel:** È un programma che crea un tunnel "TCP sopra TLS sopra TCP".
- (Vedi `www.stunnel.org`)
- **Da usare come ULTIMA risorsa...**
- **Problema:** I tunnel TCP-su-TCP causano un grave deterioramento delle prestazioni!
- Il motivo è la presenza di **loop di controllo della congestione contrastanti**.
- È preferibile usare tunnel DTLS, se possibile.

Spiegazione Didattica

Il Problema del "TCP-over-TCP"
Quando si incapsula una connessione TCP (interna) all'interno di un'altra connessione TCP (esterna, ad esempio quella del tunnel TLS), si creano due meccanismi di controllo che si sovrappongono e interferiscono:

1. Il TCP **esterno** (il tunnel) ha il suo controllo di congestione. Se perde un pacchetto, lo ritrasmette.
2. Il TCP **interno** (l'applicazione) ha anch'esso il suo controllo di congestione.

Se il tunnel TCP esterno subisce una perdita di pacchetti, ritrasmetterà. Il TCP interno vedrà questa ritrasmissione come un improvviso aumento della latenza (un RTT, Round-Trip Time, molto alto). In risposta, il TCP interno potrebbe erroneamente concludere che la rete è congestionata e ridurre la sua finestra di invio, o peggio, andare in timeout e ritrasmettere a sua volta pacchetti che erano già in transito. Questo crea un circolo vizioso che porta a prestazioni disastrose.

3 Attacco di Troncamento (Truncation Attack)

Un attaccante (Man-in-the-Middle) può terminare la connessione in qualsiasi momento inviando un pacchetto TCP FIN (Finish) falsificato.

- Parte dello scambio di informazioni inteso viene perso.
- Come possono il Server e il Client distinguere questa terminazione improvvisa da una transazione che si completa "normalmente"?

3.1 Soluzione: Close Notify

- La soluzione TLS è un alert specifico chiamato **Close Notify**.
- Può essere emesso da qualsiasi delle due parti (client o server).
- **Close Notify = Alert (livello 'warning')**.
- Implementa una semantica "Half-close" (chiusura parziale): informa l'altra parte che "Io ho finito di inviare dati, ma sono ancora pronto a ricevere i tuoi".
- **Regola di sicurezza:** Una connessione che termina bruscamente (es. con un TCP FIN o RST) **senza** aver prima ricevuto un Close Notify, deve essere considerata sospetta e potenzialmente vittima di un attacco di troncamento.

Spiegazione Didattica

Perché il TCP FIN non basta?
Un attaccante può intercettare il flusso di dati e, in un momento arbitrario (es. prima che venga inviata un'informazione critica), iniettare un pacchetto TCP FIN per far credere al ricevente che il mittente abbia finito di parlare.
Il '$\text{close}_n otify$'$risolve questo problema per ch un messaggio$**interno**$al flusso TLS, quindi protetto da integrit (autenticato con i$
Se un'applicazione riceve un TCP FIN, deve controllare se ha ricevuto anche un '$\text{close}_n otify$'$. Se non lo ha ricevuto, deve scartare i dati ricevuti in quella sessione, perch potrebbero essere incompleti (troncati)$

4 Rinegoziazione (Renegotiation)

La rinegoziazione (o re-handshake) è un meccanismo che permette di stabilire nuovi parametri di sicurezza *all'interno* di una connessione TLS già attiva.

- Non è "limitata" al *rekeying* (cambio chiavi); è molto di più di una *session resumption* (ripresa di sessione).
- È permesso un **nuovo handshake completo** (opzionalmente).
- Questo nuovo handshake è **crittografato** utilizzando la ciphersuite stabilita nella sessione precedente.
- **Scopo:**
  - Aumentare il livello di sicurezza (es. passare a una cifratura più forte a metà sessione).

- **Autenticare il Client:** Questo è un caso d'uso comune. Un utente naviga su un sito (Sessione 1, solo server autenticato). Quando tenta di accedere a un'area riservata, il server avvia una rinegoziazione chiedendo al client di presentare il suo certificato (Sessione 2, autenticazione mutua).
- Qualsiasi altra cosa necessaria.

- La rinegoziazione **genera un NUOVO ID di sessione**.

Spiegazione Didattica

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.

4.1 Flusso e Problemi della Rinegoziazione

Il flusso tipico è:

1. Handshake Iniziale (Include autenticazione e verifica certificati)
2. Dati Applicativi Crittografati (Sessione TLS 1)
3. **Handshake di Rinegoziazione** (crittografato dalla Sessione 1)
4. Nuova Ciphersuite, Nuova Chiave (NUOVA Sessione TLS 2)

Problemi:

- **Vulnerabilità:** A parte il fatto che è crittografato, l'handshake di rinegoziazione (nella sua forma originale) **NON è distinguibile** da un handshake iniziale per una nuova sessione (es. 'session_id = 0').$Questo fatto puessere (ab) usato?$**Sì.**
- **Gestione Dati Applicativi:** La rinegoziazione, quando richiesta, ha la precedenza sui dati applicativi.
- Se il client invia dati applicativi mentre il server ha iniziato una rinegoziazione, il server **bufferizzerà** quei dati, completerà la rinegoziazione, e **poi** elaborerà i dati bufferizzati nel contesto della NUOVA sessione TLS.
- La rinegoziazione può avvenire nel mezzo di una transazione a livello applicativo... **Possiamo sfruttare questo?**

5 Attacco di Rinegoziazione (Renegotiation Attack)

Questo è un attacco di **iniezione di plaintext**.

- Scoperto da Marsh Ray (PhoneFactor inc.) nell'Agosto-Settembre 2009; reso pubblico a Novembre 2009.
- Efficacia limitata... ma...
- È stato trasformato in un attacco pratico per il **furto di credenziali contro utenti Twitter** da Anil Kurmus (studente ETH) a metà Novembre 2009.
- Twitter risolse immediatamente (disabilitando la rinegoziazione).
- L'applicazione ingegnosa di tale attacco può applicarsi a molti altri casi/scenari.
- Fu uno shock per gli organi di standardizzazione TLS.
- **Regola d'oro da portare a casa: nella sicurezza, MANTENERE LE COSE SEMPLICI!**

5.1 Come funziona l'attacco (Schema MITM)

L'attaccante si posiziona come Man-in-the-Middle (MITM).

1. Il Client stabilisce una connessione TLS con l'Attaccante, e l'Attaccante con il Server. L'attaccante si limita a inoltrare i pacchetti (finora è un MITM passivo).
2. L'Attaccante **invia del plaintext** al Server (es. 'POST /...').
3. Il Server inizia a leggere questa richiesta, ma non è completa.
4. **IMMEDIATAMENTE**, l'Attaccante **avvia una rinegoziazione** con il Server.
5. Il Server **bufferizza** il plaintext parziale dell'attaccante e avvia l'handshake di rinegoziazione.
6. L'Attaccante **inoltra** i messaggi di rinegoziazione al Client.
7. Il Client, ignaro, completa la rinegoziazione, autenticandosi (ad esempio, con il suo cookie).
8. Il Client ora invia la **sua** richiesta (es. 'GET /... Cookie: ...').
9. Il Server riceve la richiesta del Client e la **accoda** ai dati che aveva bufferizzato (quelli dell'attaccante).

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

Spiegazione Didattica

Il Cuore della Vulnerabilità
La vulnerabilità (pre-RFC 5746) risiedeva nel fatto che l'handshake di rinegoziazione non era "legato" crittograficamente alla sessione TLS precedente.
Il server, ricevendo una richiesta di rinegoziazione, non aveva modo di sapere se provenisse legittimamente dal client (che aveva stabilito la Sessione 1) o da un attaccante (che aveva stabilito la Sessione 1 con il server, e un'altra sessione, la Sessione 2, con il client).
Il server "vedeva" solo: 1. Dati parziali (dall'attaccante). 2. Richiesta di rinegoziazione (dall'attaccante). 3. Completamento della rinegoziazione (dal client, ignaro). 4. Resto dei dati (dal client).
Il server, in buona fede, "incollava" i dati 1 e 4, eseguendo la richiesta "mixata" con le credenziali della parte 3 (il client).

5.2 Esempio: Iniezione di prefisso (Attacco "Pizza")

Attaccante invia (plaintext):

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

(Nessun "carriage return" finale, la richiesta è incompleta)

— *Attaccante avvia la rinegoziazione, il Client la completa* —

Vittima invia (dopo rinegoziazione):

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

Risultato (Richieste "incollate" viste dal server):

```
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
```

A seconda di come il server HTTP parsa questa richiesta (spesso ignora tutto dopo il primo header 'Host' o la prima linea di richiesta valida), il server usa il cookie della vittima ('victim$_{c}ookie$')$perprocessarelarichiestadell'attaccante.$**Il ser**

5.3 Esempio: Attacco Twitter di Kurmus

Un esempio più vicino all'attacco reale:

Attaccante invia (plaintext):

```
POST /forum/send.php HTTP/1.0
Message =
```

(Richiesta parziale)

— *Rinegoziazione* —

Vittima invia (dopo rinegoziazione):

```
GET / HTTP/1.1
Host: ...
Authorization: Basic [Base64-coded-user-pass]
...
```

Risultato (Richieste "incollate"):

```
POST /forum/send.php HTTP/1.0
Message = GET / HTTP/1.1
Host: ...
Authorization: Basic [Base64-coded-user-pass]
...
```

Risultato (reale!): Il server interpreta l'intera richiesta 'GET' della vittima (inclusi i suoi header di autenticazione) come il *corpo del messaggio* del 'POST' dell'attaccante. Il server **pubblica un tweet** (o un post sul forum) **che include le credenziali del client** (utente/password codificati in Base64)!

6 Prevenire l'Attacco di Rinegoziazione TLS

- **Soluzione immediata (2009):** Disabilitarla completamente lato server.
- **Soluzione di Standardizzazione:** Creazione di una "patch" standard, l'**estensione di rinegoziazione**.
- Standardizzata (urgentemente) in **RFC 5746** ("TLS Renegotiation Extension", Febbraio 2010).

6.1 Come funziona RFC 5746

- L'estensione "lega" crittograficamente la rinegoziazione alla sessione TLS esistente.
- **Primo handshake:** L'estensione viene inviata vuota (per segnalare il supporto).
- **Handshake di rinegoziazione:** L'estensione trasporta un hash dei messaggi 'finished' dell'handshake **precedente**.
- Entrambe le parti verificano che l'hash corrisponda alla sessione precedente.
- Ora è banale per il server rilevare l'attacco: l'hash inviato dall'attaccante non corrisponderà a quello che il client ha calcolato sulla sua connessione.

6.2 La Soluzione Definitiva: TLSv1.3

- **TLSv1.3: proibisce la rinegoziazione!**
- Segue la regola d'oro: "Keep it simple".
- La complessità della rinegoziazione ha introdotto più problemi di sicurezza di quanti ne abbia risolti. Se è necessaria l'autenticazione del client, TLS 1.3 fornisce meccanismi per richiederla post-handshake in modo più sicuro.

7 E per quanto riguarda DTLS?

7.1 DTLS vs TLS a colpo d'occhio

- **RFC 4347** - Datagram TLS - Aprile 2006.
- È essenzialmente **TLS su UDP**.
- **Obiettivo di design di DTLS:** Essere il più simile possibile a TLS!

Spiegazione Didattica

Perché DTLS?
TLS non può funzionare su UDP. TLS richiede un trasporto affidabile e ordinato (come TCP) per funzionare. Si aspetta che i byte arrivino nello stesso ordine in cui sono stati inviati e che non se ne perda nessuno.
Tuttavia, molte applicazioni (VoIP, gaming online, VPN) usano UDP perché preferiscono la bassa latenza alla garanzia di consegna. Per queste applicazioni, è stato creato DTLS (Datagram TLS).
DTLS fornisce le stesse garanzie di sicurezza di TLS (confidenzialità, integrità, autenticazione) ma adatta il protocollo per funzionare sulla natura inaffidabile e non ordinata dei datagrammi UDP.

7.2 Differenze chiave DTLS vs TLS

DTLS deve risolvere i problemi che TCP risolveva per TLS:

- **TLS assume consegna ordinata:**
- *DTLS:* Aggiunge un **numero di sequenza** esplicito nell'header del record per gestire i pacchetti fuori ordine.
- **TLS assume consegna affidabile:**
- *DTLS:* Aggiunge **timeout e meccanismi di ritrasmissione** a livello di protocollo (per i messaggi di handshake) per gestire la perdita di datagrammi.
- **TLS può generare frammenti grandi (fino a 16384B):**
- *DTLS:* Include capacità di **frammentazione e riassemblaggio** per i messaggi di handshake troppo grandi per un singolo datagramma UDP, e raccomanda l'uso della Path MTU Discovery.
- **TLS assume un protocollo orientato alla connessione:**
- *DTLS:* Una "connessione" DTLS è definita dal TLS handshake (non c'è un 'close$_n$otify'allostessomodo, datocheUDPcon