Istituto di Istruzione Superiore Evangelista Torricelli

Claudio Mola

Anno scolastico 2015 - 2016

|  |
| --- |
| WEB SERVER IN JAVA UTILIZZANDO I SOCKET SSL |
| L'importanza di una comunicazione sicura attraverso internet |



# Sommario

[1. Sommario 1](#_Toc454642990)

[2. Abstract 2](#_Toc454642991)

[3. La crittografia 3](#_Toc454642992)

[3.1. Crittografia Simmetrica 3](#_Toc454642993)

[3.2. Crittografia Asimmetrica 4](#_Toc454642994)

[3.3. L’argoritmo RSA 5](#_Toc454642995)

[3.4. Il protocollo SSL/TLS 6](#_Toc454642996)

[4. Java: Chiavi, Certificati e Socket SSL 7](#_Toc454642997)

[4.1. Gestire chiavi e certificati in Java (Keytool) 7](#_Toc454642998)

[4.2. L’uso dei Socket SSL/TLS in Java 10](#_Toc454642999)

[5. Un Semplice Web Server HTTPS in Java 11](#_Toc454643000)

[5.1. Introduzione 11](#_Toc454643001)

[5.2. Ciclo di vita del Web Server 11](#_Toc454643002)

[5.3. Creazione del Server Socket 12](#_Toc454643003)

[5.4. Attesa del Client 13](#_Toc454643004)

[5.5. Http Request 14](#_Toc454643005)

[5.6. Http Response 16](#_Toc454643006)

[5.7. Il comportamento di un Browser interrogando il Web Server 18](#_Toc454643007)

[6. Analisi dei pacchetti con Wireshark 19](#_Toc454643008)

[6.1. Introduzione 19](#_Toc454643009)

[6.2. Http Sniffing con Wireshark 20](#_Toc454643010)

[6.3. Https Sniffing con Wireshark 21](#_Toc454643011)

[7. Conclusioni 23](#_Toc454643012)

[8. Riferimenti 24](#_Toc454643013)

[8.1. Libri 24](#_Toc454643014)

[8.2. Internet 24](#_Toc454643015)

[8.3. Wikipedia 24](#_Toc454643016)

[8.4. Software 24](#_Toc454643017)

# Abstract

Con questa tesina ho voluto approfondire un argomento che ha saputo incuriosirmi particolarmente: la crittografia e l’importanza che assume ai giorni nostri. Per fare questo ho sviluppato un mio progetto per potermi imbattere nelle problematiche realizzative e verificarne con mano l’effettivo funzionamento.

Partendo dalle conoscenze maturate durante l’anno scolastico sui Socket TCP in Java, sulla crittografia simmetrica e asimmetrica e sul protocollo HTTP (Hypertext Transfer Protocol), **ho voluto sviluppare un semplice Web Server utilizzando i socket**. Esso è in grado di gestire le richieste di trasferimento di pagine web provenienti da un web browser.

**Ho iniziato lo sviluppo implementando le funzioni di base del protocollo HTTP**, interpretando le richieste provenienti dal browser (**HTTP Request**) ed impacchettando le risposte (**HTTP Response**). Successivamente ho approfondito le mie conoscenze sui Socket SSL in Java implementandoli nel mio web server. In questo modo la mia comunicazione tramite il protocollo HTTP avviene all’interno di una connessione criptata TLS (Transport Layer Security) così **diventando** a tutti gli effetti **il protocollo HTTPS** (HTTP over TLS).

Il protocollo HTTPS comporta molti vantaggi dal punto di vista della sicurezza. Assicura che la comunicazione tra l’utente ed il sito web non sia né intercettata né alterata da terzi e dà una garanzia soddisfacente che si stia comunicando esattamente con il sito web voluto. Per fare ciò è necessario, oltre ai due interlocutori, anche di una terza parte fidata, una Certification Authority (CA), per la creazione del certificato digitale. In questo progetto, non avendo a disposizione una CA, ho dovuto provvedere a **creare un mio certificato self-signed**; in questo modo ho potuto anche constatare i comportamenti dei browsers in presenza di un certificato non riconosciuto da una CA.

Per finire **ho messo alla prova il web server** **confrontando la versione HTTP con quella HTTPS, utilizzando il software Open Source Wireshark,** il quale permette di osservare in tempo reale tutto il traffico presente sulla rete. Ho verificato che i pacchetti provenienti dal mio web server HTTP fossero trasmessi in chiaro e visibili da chiunque fosse riuscito ad intercettare la comunicazione, mentre nella versione HTTPS la comunicazione viene cifrata, impedendo a qualsiasi malintenzionato di visionare quanto trasmesso e/o alterarlo.

Da questa esperienza ho potuto constatare con mano quanto sia importante una comunicazione sicura, soprattutto oggigiorno con l’enormità di dati sensibili che viaggiano su internet, come ad esempio quelli scambiati durante gli acquisti online.

# La crittografia

Questo capitolo ha lo scopo di introdurre le basi della crittografia, andando a trattare tutti gli argomenti necessari a comprendere cosa nasconde la dicitura HTTPS.

## Crittografia Simmetrica

La crittografia simmetrica rappresenta un metodo semplice per cifrare testo in chiaro dove la chiave per cifrare è la stessa per decifrare, rendendo l'algoritmo molto performante e semplice da implementare.

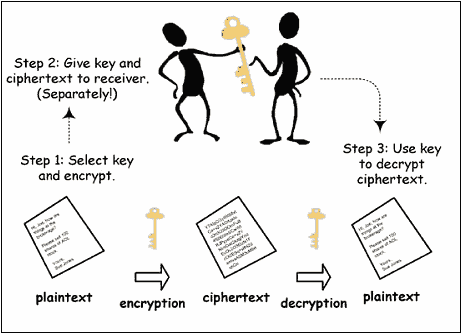
Uno dei primi sistemi crittograﬁci moderni a chiave simmetrica è il DES (Data Encryption Standard), un algoritmo simmetrico con chiave privata da 64 bit, sviluppato per l’IBM nel 1976 e diventato uno standard negli USA per la protezione di dati sensibili.

Il DES inizialmente ha suscitato molte discussioni per via della sua chiave di cifratura corta. Si supponeva che dietro questa scelta vi fosse la National Security Agency (NSA) e l'inserimento di una backdoor. Con la potenza di calcolo disponibile attualmente si può forzare una chiave DES in poche ore, esaminando tutte le possibili combinazioni (Attacco di forza bruta).

Una caratteristica desiderata per ogni algoritmo di criptazione è quello che prende il nome di **effetto valanga**: un cambiamento di pochi bit nel *plaintext* deve provocare un cambiamento di quanti più bit nel *ciphertext*. Il DES possiede un forte effetto valanga.

Attualmente il DES non è più utilizzato come standard negli USA ed è stato rimpiazzato dal **AES** (Advanced Encryption Standard) che utilizza una chiave che può essere di 128, 192 o 256 bit.

Gli algoritmi simmetrici presentano alcuni limiti; quello più evidente è che le persone per comunicare devono essere in possesso della stessa chiave e, di fatto, questo limita la diffusione e il suo utilizzo.



## Crittografia Asimmetrica

L’idea alla base della crittograﬁa asimmetrica è quello di avere due chiavi diverse, una pubblica per cifrare ed una privata per decifrare, che deve essere mantenuta assolutamente segreta.

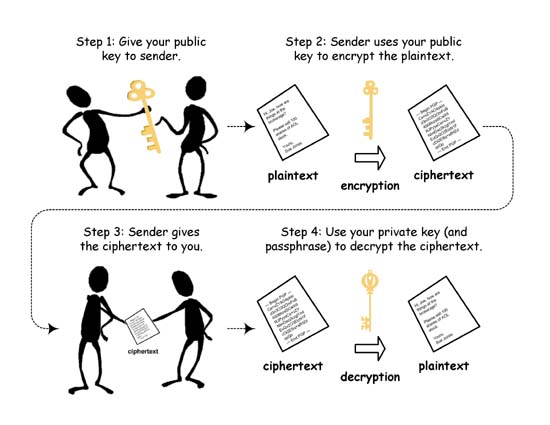
Formalmente è necessario trovare una funzione (“il lucchetto”) la cui trasmissione su canali insicuri non comprometta l’algoritmo, che sia facile da applicare (parte pubblica che chiude il lucchetto) ma difﬁcile da invertire (parte privata che apre il lucchetto).

Questo meccanismo è implementato negli algoritmi di crittograﬁa asimmetrici, come ad esempio nell’algoritmo RSA.

Con la crittografia asimmetrica si risolvono due problemi, quello della riservatezza e quello della autenticità del mittente semplicemente utilizzando le chiavi in modo diverso:

* Per garantire la **riservatezza** si cifra il messaggio con la chiave pubblica e solo il possessore della chiave privata sarà in grado di decifrarlo.
* Per garantire l’**autenticità** del mittente invece il messaggio viene cifrato con la chiave privata e solo con la corrispondente chiave pubblica sarà possibile decifrare il messaggio. La chiave pubblica sarà conservata in registri consultabili ma gestiti in modo sicuro. Questo si chiama firma elettronica. In più, oltre a garantire il mittente, è possibile garantire anche il contenuto del messaggio generando un “hashing” dello stesso, aggiungendolo in fondo al messaggio.

Se si volesse garantire sia la riservatezza che l’autenticità, basterebbe combinare entrambe le tecniche.



Il principale svantaggio degli algoritmi a cifratura asimmetrica sta nella complessità dei calcoli che rendono poco efficiente la loro implementazione soprattutto con l’aumentare della lunghezza della chiave.

In pratica, per motivi prestazionali, il client e il server usano questa tecnica per scambiarsi una chiave simmetrica in modo sicuro e poi passano a un algoritmo di crittografia tradizionale.

Per evitare la necessità di scambiare in anticipo in modo sicuro le chiavi pubbliche, si usano i certificati: **Un certificato** contiene una chiave pubblica autenticata mediante la firma digitale di una **Certification Authority** (CA); chi riceve il certificato può verificare direttamente l’autenticità della chiave pubblica usando la chiave pubblica della CA (che deve essere nota).

Nel corso degli anni le raccomandazioni sulla lunghezza della chiave sono mutate per via della maggior potenza di calcolo degli elaboratori moderni, attualmente si consiglia una chiave a 2048 bit.

## L’argoritmo RSA

L’algoritmo RSA fu descritto nel 1977 da Rivest, Shamir e Adleman al MIT e fu brevettato nel 1983. Il cuore della crittograﬁa asimmetrica è una funzione facile da computare ma difﬁcile da invertire, a meno di non conoscere un particolare dato (la chiave): l’algoritmo RSA “lavora” sfruttando i numeri primi e come chiave utilizza un numero **n** ottenuto proprio dal prodotto di due numeri primi **p** e **q**, cioè **n = p · q**.

Per decrittare un messaggio cifrato con RSA è necessario decomporre la chiave **n** nei due numeri primi **p** e **q**: questo è computazionalmente impegnativo da ottenere, basti pensare che nel 2005 un gruppo di ricerca riuscì a scomporre un numero di 640 bit in due numeri primi da 320 bit impiegando per cinque mesi un cluster con 80 processori da 2,2 GHz.

Un attuale utilizzo è quello di sfruttare RSA per codiﬁcare un unico messaggio contenente una chiave segreta, tale chiave verrà poi utilizzata per scambiarsi messaggi tramite la cifratura simmetrica (ad esempio AES).

Il funzionamento dell’algoritmo RSA è il seguente:

1. Alice deve spedire un messaggio segreto a Bob;
2. Bob sceglie due numeri primi molto grandi e li moltiplica tra loro (generazione delle chiavi);
3. Bob invia ad Alice “in chiaro” il numero che ha ottenuto;
4. Alice usa questo numero per crittografare il messaggio;
5. Alice manda il messaggio cifrato a Bob, che chiunque può vedere ma non decifrare;
6. Bob riceve il messaggio e utilizzando i due fattori primi, che solo lui conosce, decifra il messaggio.

La forza (o la debolezza) dell’algoritmo si basa sull’assunzione mai dimostrata (nota come RSA assumption) che il problema di calcolare un numero composto di cui non si conoscono i fattori sia computazionalmente non trattabile.

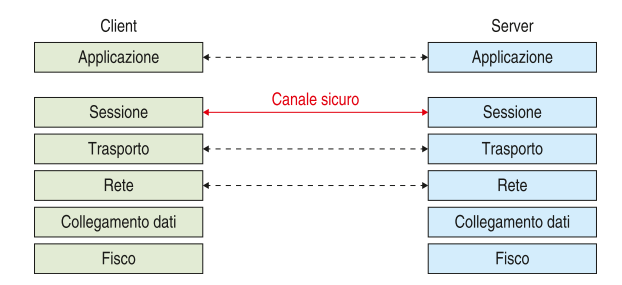
Questo sistema però comporta un problema, cioè che le funzioni matematiche che generano il codice cifrato e quelle inverse impiegano troppo tempo per essere utilizzate per la cifratura di interi documenti, per questo sono nati i **sistemi di crittografia misti** che uniscono la tecnica a cifratura asimmetrica per scambiarsi una chiave segreta che verrà utilizzata per una normale comunicazione basata su crittografia simmetrica. Infatti i vantaggi di un metodo compensano gli svantaggi dell’altro.

## Il protocollo SSL/TLS

Lo standard più diffuso per la protezione dei servizi offerti tramite Internet è Secure Socket Layer (SSL) ed il suo successore Transport Layer Security (TLS): si tratta di un insieme di protocolli crittograﬁci che aggiungono funzionalità di cifratura e autenticazione a protocolli preesistenti al livello di sessione. Questo protocollo è nato al ﬁne di garantire la privacy delle trasmissioni su Internet, permettendo alle applicazioni client/server di comunicare in modo da prevenire le intrusioni, le manomissioni e le falsiﬁcazioni dei messaggi.

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

1. **privatezza del collegamento**: la riservatezza del collegamento viene garantita mediante algoritmi di crittograﬁa a chiave simmetrica (ad esempio **DES** e **AES**);
2. **autenticazione**: l’autenticazione dell’identità viene effettuata con la crittograﬁa a chiave pubblica (per esempio **RSA**): in questo modo si garantisce ai client di comunicare con il server corretto, introducendo a tale scopo anche meccanismi di **certiﬁcazione** sia del server che del client;
3. **affidabilità**: il livello di trasporto include un controllo sull’integrità del messaggio con un sistema detto MAC (Message Authentication Code) che utilizza funzioni hash sicure come SHA e MD5: avviene la veriﬁca di integrità sui dati spediti in modo da avere la certezza che non siano stati alterati durante la trasmissione.



TSL è un protocollo di livello 5 (sessione) che opera quindi al di sopra del livello di trasporto composto da due livelli:

1. **TLS Record Protocol**: opera a livello più basso, direttamente al di sopra di un protocollo di trasporto affidabile come il TCP ed è utilizzato per i protocolli del livello superiore, tra cui l’Handshake Protocol, offrendo in questo modo i servizi di sicurezza;
2. **TLS Handshake Protocol**: si occupa della fase di negoziazione, in cui si autentica l’interlocutore e si stabiliscono le chiavi segrete condivise.

# Java: Chiavi, Certificati e Socket SSL

## Gestire chiavi e certificati in Java (Keytool)

Il Java Development Kit include un tool (da usare da linea di comando) per gestire chiavi e certificati:

keytool

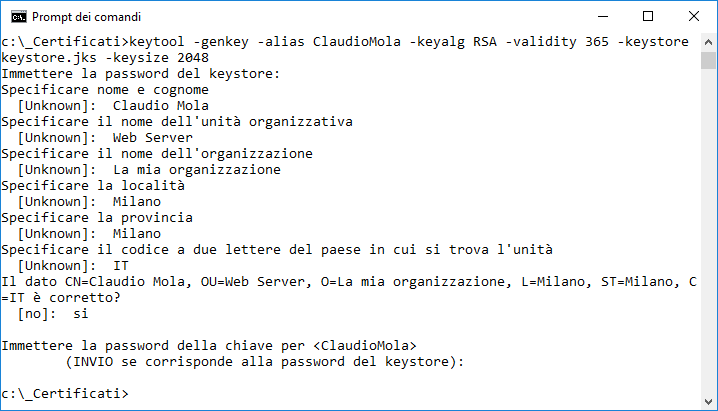
Le chiavi pubbliche e private sono memorizzate in un keystore e i certificati ritenuti “fidati” sono memorizzati in un truststore. Il formato del keystore e del truststore è proprietario, ma keytool offre funzioni per import/export di chiavi e certificati nei formati standard.

Per generare una coppia di chiavi in un keystore il comando è:

keytool -genkey [opzioni] -alias nome -keylag RSA -validity giorni   
-keystore keystore -keysize bits

Il tool richiede alcune informazioni sull’identità della persona che genera le chiavi, che saranno memorizzate all’interno delle chiavi stesse ed è protetto da una password.

Ecco un esempio di che cosa avviene durante la creazione di una coppia di chiavi:



È possibile visualizzare il contenuto di un keystore con il comando:

keytool -list -v -keystore keystore

Per quanto riguarda invece il processo per generare un **certificato,** richiede **tre passi**:

1. **creazione** di una Certificate Request a partire dalla chiave pubblica nel keystore
2. **invio** della Certificate Request alla Certification Authority (CA), che produrrà il certificato
3. **importazione** del certificato della CA nel truststore

Se non si ha a disposizione una CA, si può generare un **self-signed certificate**, quindi senza la garanzia sull’identità data dalla CA, ma è adeguato se i due end-point si fidano reciprocamente e possono scambiarsi i certificati in maniera sicura.

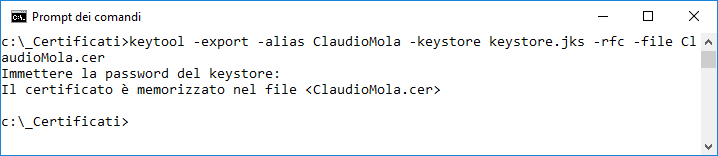
Quindi i passi diventano:

1. **generazione** del certificato dalla chiave pubblica nel keystore
2. **importazione** del certificato nel truststore

Per eseguire il primo passo, cioè generare il certificato la sintassi sarà:

keytool -export -alias nome -keystore keystore -rfc -file fileCertificato

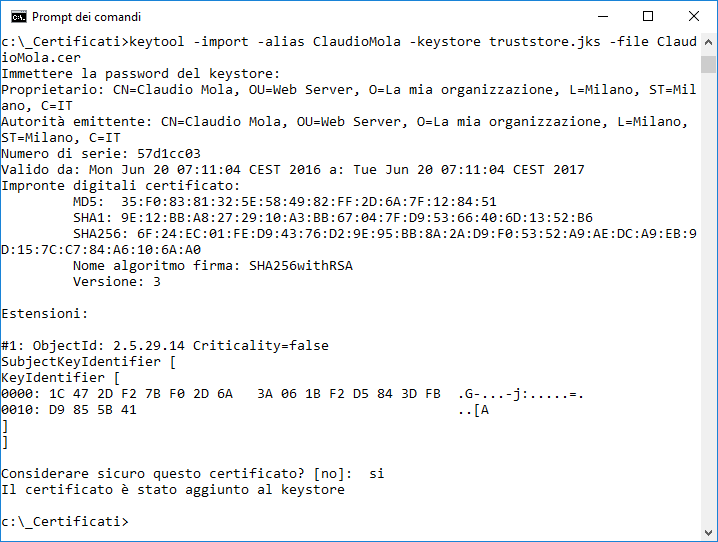
Questo è quello che succede durante la generazione di un certificato:



A questo punto è necessario importare il certificato nel truststore:

keytool -import -alias nome -keystore truststore -file fileCertificato

Questo è quello che succede durante l’importazione del certificato. Verrà mostrato il proprietario, l’ente emittente e le impronte del certificato nei vari algoritmi di hashing:



## L’uso dei Socket SSL/TLS in Java

In Java l’uso di SSL/TLS si basa sulle classi SSLSocket e SSLServerSocket, che estendono rispettivamente Socket e ServerSocket. Una volta effettuata la creazione dei socket, non c’è differenza per l’applicazione rispetto all’uso di socket non crittografati.

Le classi e le interfacce necessarie sono nei package:

Il primo passo è la creazione di una SocketFactory, che è un oggetto che astrae l’operazione di creazione di un socket. Per creare una SocketFactory in grado di creare Socket SSL occorre usare il metodo static getDefault() della classe SSLSocketFactory.



Una volta ottenuta una factory, si può usare il metodo createSocket() per creare il socket vero e proprio:



Una volta creato, il socket si usa come un normale client socket, ma per rendere possibile la creazione del socket SSL, il programma deve conoscere il keystore e il truststore e le relative password. È possibile fornire tali informazioni usando opportune proprietà di sistema che possono essere impostate con il metodo System.setProperty():

La creazione di server socket SSL è analoga alla creazione di socket, occorre usare le classi SSLServerSocketFactory e SSLServerSocket.



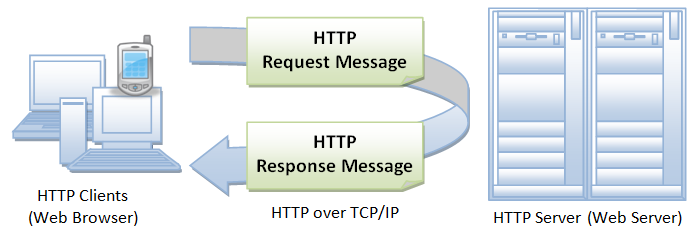
Una volta creato, il server socket si usa esattamente come un ServerSocket non crittografato, il keystore e il truststore devono essere specificati con le stesse proprietà di sistema. Per default i socket creati dalla factory effettuano l’autenticazione del solo server, ma se si desiderasse l’autenticazione anche del client, occorrerebbe richiamare il metodo setNeedClientAuth().



# Un Semplice Web Server HTTPS in Java

## Introduzione

Lo scopo principale di questo progetto è applicare quanto visto in precedenza, sia sulla cifratura, sia sui Socket SSL, per la creazione di un semplice Web Server HTTPS. Le sue funzionalità principali devono comprendere il leggere e l’interpretare una richiesta HTTP (**HTTP Request**), ricavandone quanto necessario per generare una risposta HTTP (**HTTP Response**).



Dovendo prima di tutto garantire la riservatezza della comunicazione, non ho dato priorità all’implementazione totale del protocollo HTTP, ma solo alla parte necessaria per una comunicazione minima, cioè leggere la richiesta ed inoltrare la pagina voluta. Inoltre il Web Server, coinvolgendo una sola coppia di processi alla volta, avrà una comunicazione di tipo unicast. Questo ha un impatto sulle prestazioni generali del web server, ma ne ha reso anche più semplice lo sviluppo.

## Ciclo di vita del Web Server

La struttura di questo Web Server è riassumibile in 4 passi:

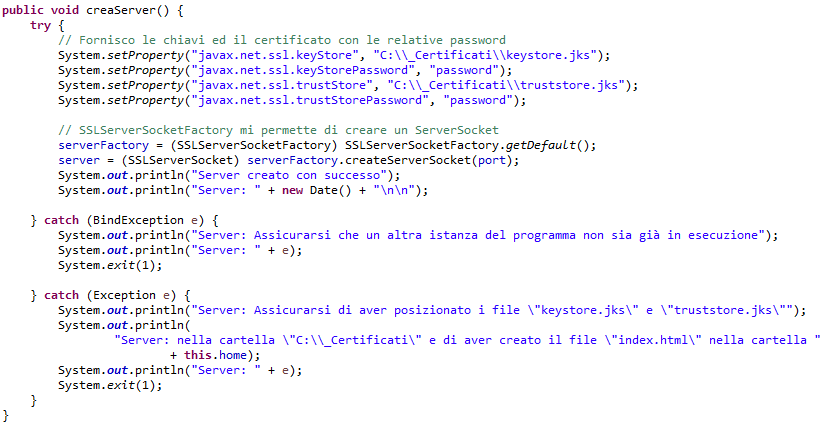
creaServer() => connessione() => clientHeader() => inviaFile()

La creazione del Server Socket (creaServer()) viene eseguita solo all’avvio del Web Server, mentre mettersi in attesa del client (connessione()), leggere la richiesta (clientHeader()) e inviare la risposta (inviaFile()) sono eseguiti in un ciclo senza fine.

## Creazione del Server Socket

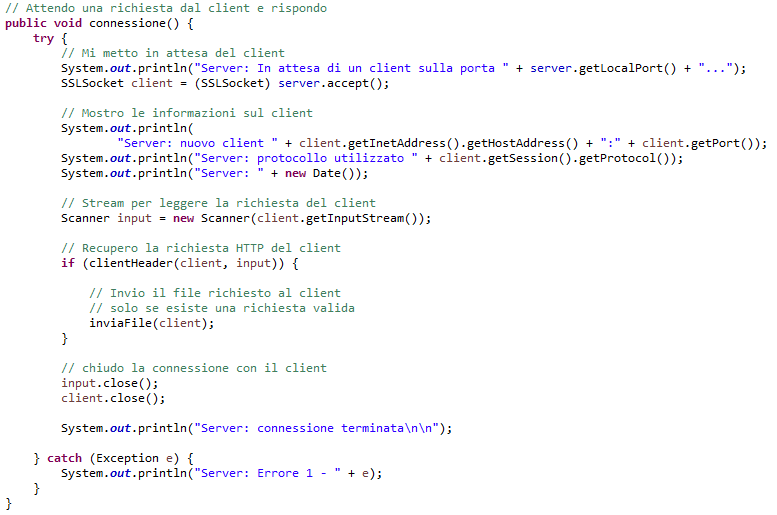
In questo primo passo viene creato il Server Socket attraverso la classe **SSLServerSocketFactory**. Prima di tutto è necessario specificare la posizione delle coppie di chiavi (keystore) e dei certificati digitali (truststore) con le relative password. Per comodità sia i percorsi che le password vengono indicati all’interno del sorgente, ma sarebbe opportuno, ad esempio, passarli come parametri.

Questa è l’unica differenza che si può trovare tra un Socket SSL ed un Socket TCP in chiaro, in quanto sarà SSLSocketFactory ad occuparsi di tutti i dettagli della configurazione di un Secure Socket.

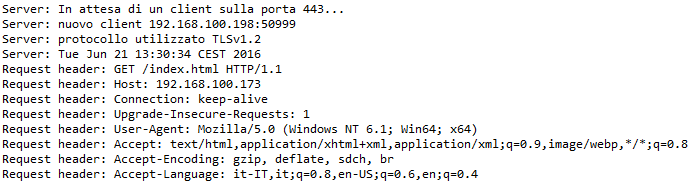


## Attesa del Client

Il secondo passo consiste nel **mettersi in attesa del Client**. Questo metodo (public void connessione()) rappresenta il corpo del programma e da qui verranno gestite tutte le successive fasi.



Qui di seguito è mostrato un printscreen di quello che accade durante la connessione di un client.   
Il server è in attesa sulla porta 443 (porta di default per HTTPS), successivamente un client richiede una connessione ed attraverso il protocollo TLS viene instaurata una comunicazione sicura.



## Http Request

In questo terzo passo viene effettuata la **lettura dell’Header HTML** (HTTP Request).

Con il metodo seguente, leggo riga per riga il **Request message** inviato dal client, dove la prima riga sarà sempre la **Request line** con la seguente sintassi:

request-method-name request-URI HTTP-version

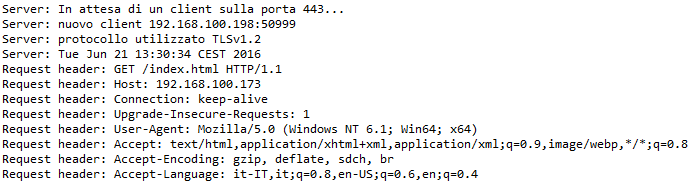
**Request-method-name**: il protocollo HTTP definisce una serie di metodi come GET o POST per mandare una richiesta al server.

**Request-URI:** specifica la risorsa richiesta.

**HTTP-version**: attualmente sono in uso tre versioni: HTTP/1.0, HTTP/1.1 e HTTP/2.0. Quest’ultima è molto recente, infatti la sua specifica è stata pubblicata solo nel 2015.

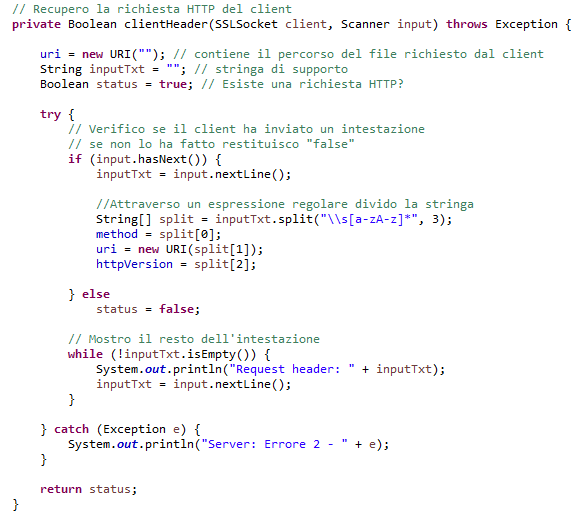
Di seguito viene mostrato quanto ricevuto dal web server.

Attraverso il metodo GET il client fa una richiesta della risorsa “/index.html” utilizzando il protocollo HTTP versione 1.1. Le successive righe della richiesta si chiamano **Request Header** e forniscono informazioni aggiuntive come ad esempio l’IP dell’host e l’User-Agent.   
Una riga vuota segna la fine dell’Header e l’inizio di un eventuale messaggio di richiesta.



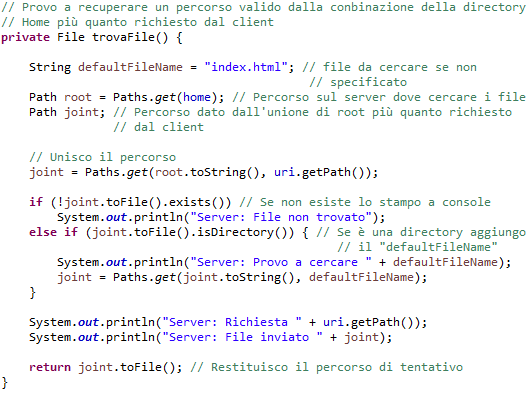
In questo web server, non dovendo fornire funzionalità avanzate, mi interesserà unicamente, come informazione, solo la risorsa richiesta nella Request Line. Il metodo più pratico è stato suddividere la stringa ad ogni spazio attraverso un’espressione regolare.

Nell’eventualità che il client non invii alcun messaggio, la successiva fase non servirebbe, per questo la funzione restituirà vero se la richiesta esiste, altrimenti restituirà falso, terminando di conseguenza la comunicazione con il client.

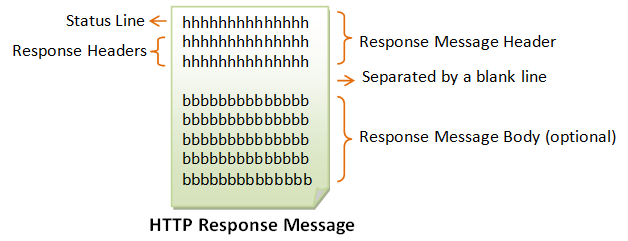


## Http Response

Inquest’ultimo passo viene effettuato l’**invio del file richiesto** (HTTP Response). Dal momento che già conosciamo qual è la risorsa desiderata, bisogna verificare se è effettivamente disponibile. Quindi, si procederà ad unire la directory di root del web server, che si sarà precedentemente impostata (in questo esempio è “C:\\_File\”) alla risorsa richiesta (ad esempio “/index.html”). Come risultato della funzione sarà restituito il file “C:\\_File\index.html”.



A questo punto si può procedere ad inviare la risposta al client, impacchettando il messaggio come richiesto dal protocollo HTTP (HTTP Response message). Esso è composto da una Status Line e da un Response Header. Dopo una riga bianca sarà aggiunto il corpo del messaggio; in questo caso, il file ricavato precedentemente sarà letto dalla classe FileInputStream e scritto sul Buffer di uscita verso il client.



La **Status Line** ha la seguente sintassi:

HTTP-version status-code reason-phrase

**HTTP-version**: viene indicata la versione del protocollo usata per la risposta.

**Status-code**: Indica il risultato della richiesta, è rappresentato da 3 cifre, le più comuni sono 200 e 404.

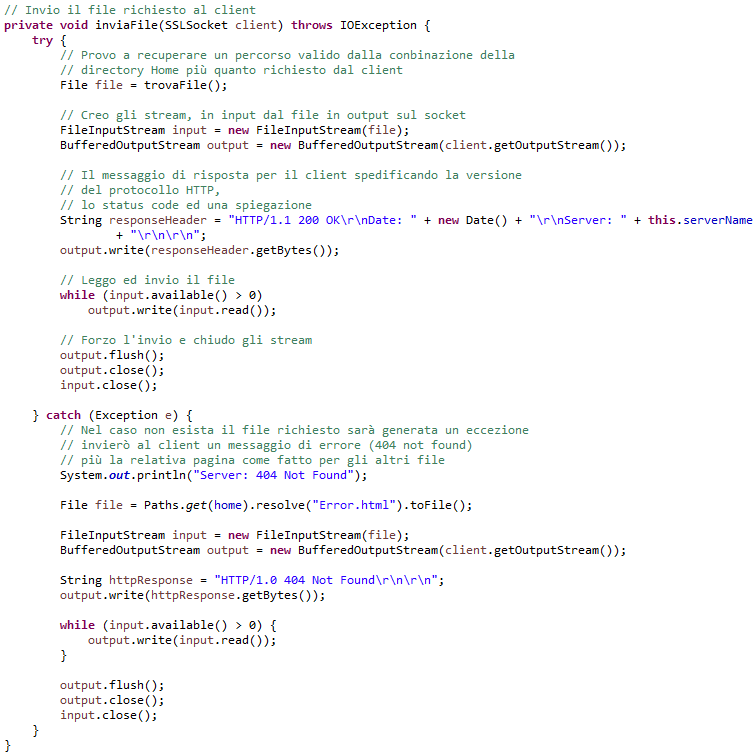
**Reason-phrase**: Indica una spiegazione dello status code, ad esempio “200 OK” o "404 Not Found".

In questo Web Server sono state implementate solo le precedenti due condizioni (“OK” e “Not Found”).

Quindi, se il file richiesto esiste, sarà inviato con la seguente status line, inserendo anche qualche informazione aggiuntiva, come la data corrente ed il nome del Web Server.



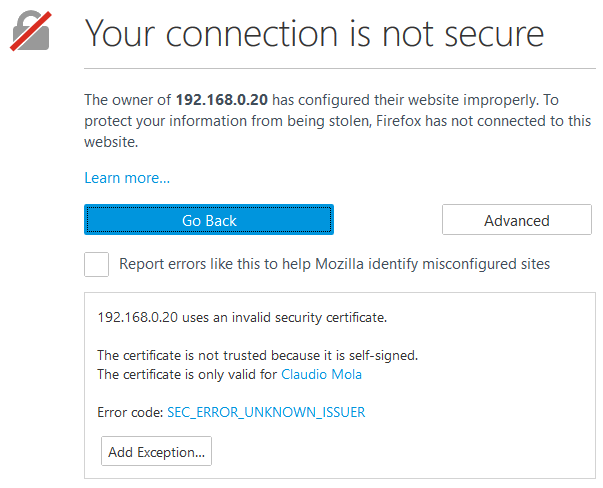
In caso contrario, ad esempio se il file che si è provato ad aprire non esiste, viene inviata la Status Line “HTTP/1.0 404 Not Found” e dopo la riga bianca una pagina html per segnalare l’errore.



## Il comportamento di un Browser interrogando il Web Server

Un browser, al primo tentativo di connessione con il web server, richiederà il suo certificato digitale e proverà a verificarne l’identità attraverso una CA (Certification Authority). In questo caso, non avendo a disposizione una CA, ho dovuto generare un certificato detto **self-signed**, quindi fungendo da autorità di certificazione di me stesso.

Il Browser ci avvertirà di questo mostrando una pagina simile alla seguente, in cui si potrà scegliere di fidarci del certificato o di non visitare il sito.



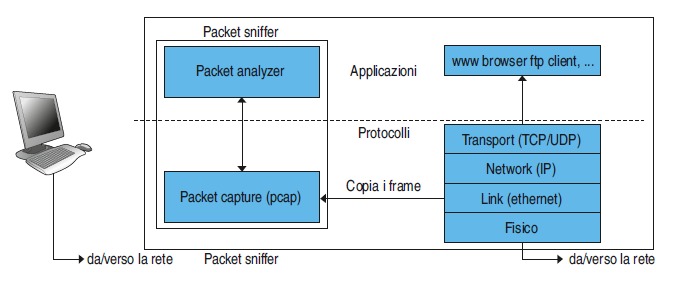
In questo caso scegliamo di fidarci, dal momento che ne conosciamo la fonte. Decidiamo di salvare il certificato localmente e di aggiungerlo come eccezione: così facendo, sarà considerato affidabile e nel caso in cui, in futuro, il web server ci invii un certificato diverso, ci sarà segnalato tempestivamente.

Questa eventualità può essere verificata specificando nel Web Server un certificato differente, così facendo il browser non identificherà più il server come sicuro. Questa operazione è compiuta regolarmente da un normale browser attraverso una CA, verificando l’identità del server e prevenendo dunque attacchi di tipo “**man in the middle**”.

# Analisi dei pacchetti con Wireshark

## Introduzione

Come già visto in precedenza, il protocollo HTTPS garantisce l’autenticazione del sito web, la protezione della privacy e l’integrità dei dati. Attraverso l’utilizzo dei certificati digitali, è possibile garantire l’autenticazione, dimostrata nel capitolo precedente. Ora vogliamo verificare che il Web Server provveda anche a tutelare la privacy. Per fare ciò utilizzeremo un **packet sniffer.** Il suo scopo è osservare i messaggi scambiati tra diversi dispositivi, inviati e ricevuti, copiandoli passivamente.



Il packet sniffer è organizzato in due parti:

1. la **libreria di cattura dei pacchetti** (pcap), riceve una copia di ogni frame che a livello di collegamento viene inviato o ricevuto dal computer. Questo consente di ottenere tutti i messaggi ricevuti o inviati da tutti i protocolli/applicazioni in esecuzione;
2. l’**analizzatore di pacchetti** visualizza il contenuto di tutti i campi all’interno dei messaggi.

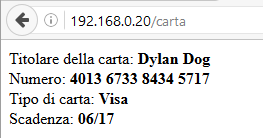
Un potente programma packet sniffer è **Wireshark**, che consente di visualizzare i contenuti di tutti i messaggi inviati/ricevuti dai protocolli a differenti livelli della pila protocollare.

Inizieremo con una copia del Web Server modificata per inviare e ricevere messaggi in chiaro (HTTP), in modo tale da poter dimostrare il funzionamento di Wireshark e di come sia possibile visionare quanto inviato e ricevuto.

Successivamente, proveremo la medesima tecnica sul Web Server sicuro (HTTPS) per poter constatare che effettivamente i messaggi siano cifrati ed in che modo avvenga l’Handshake TLS.

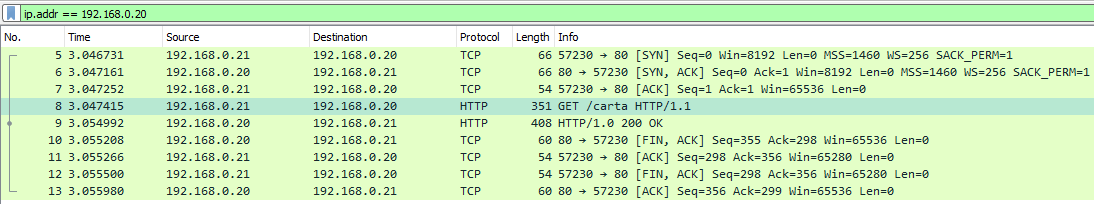
## Http Sniffing con Wireshark

Come primo passo, è necessario creare un caso di studio facile da analizzare. Quindi è stata creata una semplice pagina HTML nella cartella “/carta” del Web Server contenente gli ipotetici dati di una carta di credito del sig. Dylan Dog.

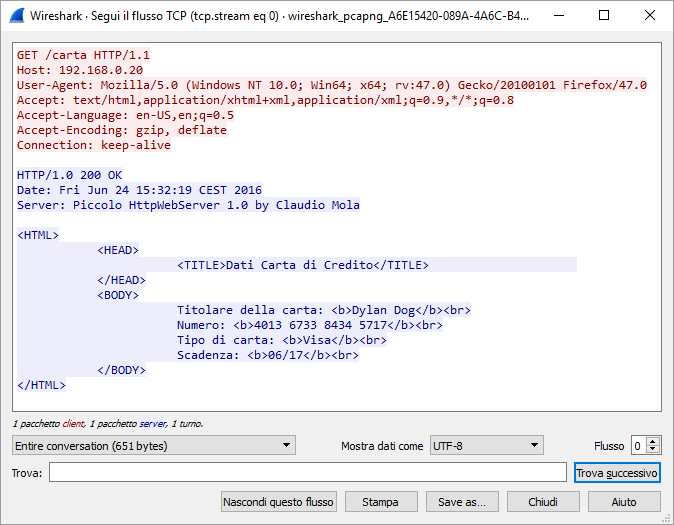


Mandando in esecuzione Wireshark si deve prima di tutto selezionare l’interfaccia di rete che si vuole utilizzare, in questo caso “Connessione alla rete locale (LAN)” ed impostare un filtro sull’indirizzo ip del Web Server **“ip.addr == 192.168.0.20”** in modo tale davisualizzare solo i pacchetti di nostro interesse.

Ora, connettendosi al Web Server all’indirizzo “/carta”, sulla pagina di Wireshark compariranno tutti i pacchetti scambiati tra il client ed il server ed in particolare sono presenti due messaggi HTTP, il primo è la richiesta fatta dal Browser al Web Server (HTTP Request), mentre il secondo è la risposta con la pagina richiesta (HTTP Response).



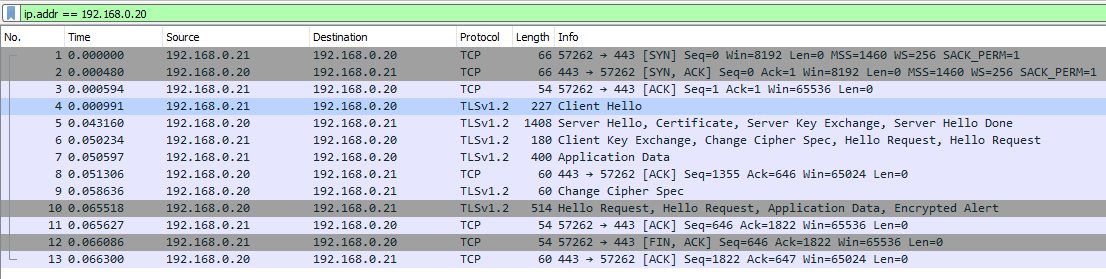
Volendo approfondire l’analisi di questi due messaggi, si può notare che tutta la comunicazione transiti in chiaro sulla rete, rendendone possibile la lettura a chiunque riesca ad intercettarla. Questo renderebbe il sig. Dylan Dog un po’ più povero, ma fortunatamente oggigiorno è difficile trovare negozi online che non utilizzino il protocollo HTTPS.



## Https Sniffing con Wireshark

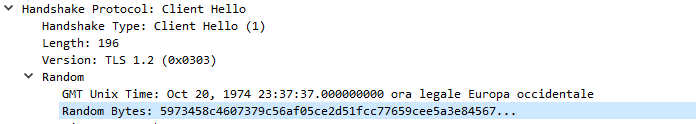
Seguendo la medesima metodologia dello sniffing HTTP, proveremo ora ad analizzare che cosa un eventuale utente esterno vedrebbe con una comunicazione HTTPS.

La prima cosa che si nota è che i due messaggi HTTP (Request e Response) non sono più presenti in maniera esplicita ed al loro posto troviamo l’Handshake del protocollo TLS. Esso provvederà a negoziare la suite di cifratura, ad autenticare il server e a scambiarsi la chiave di sessione.

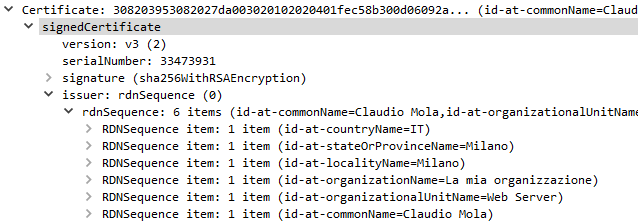


TLS Handshake Protocol si sviluppa nei seguenti passaggi:

1. Il Client invia un messaggio “Client Hello” al server indicando le suite di cifratura supportate insieme ad un numero casuale (No.4);

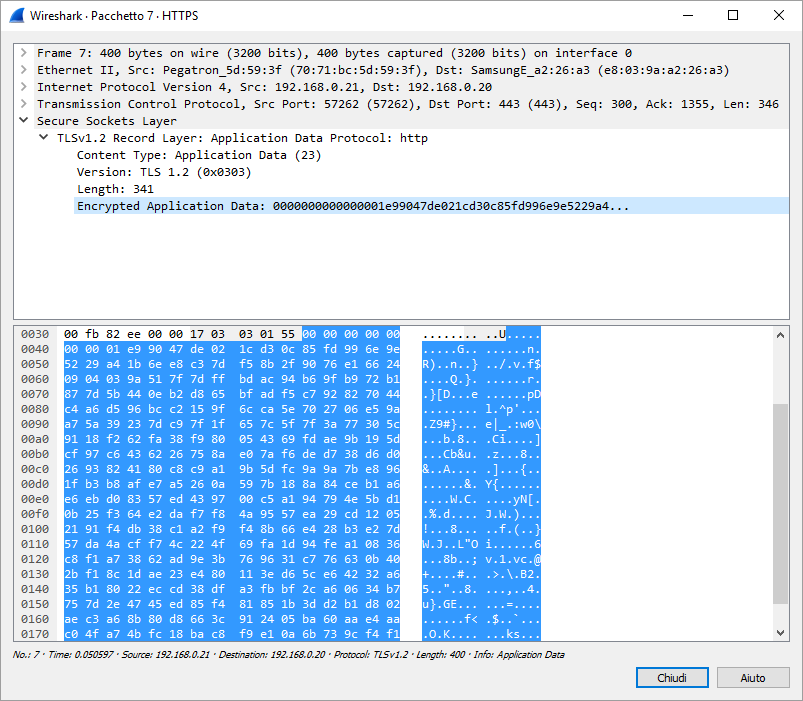


1. Il Server risponde con un messaggio “Server Hello” inviando anche lui un numero casuale (No.5);
2. Il Server invia il suo certificato per autenticarsi ed il messaggio “Server Hello Done” (No.5);



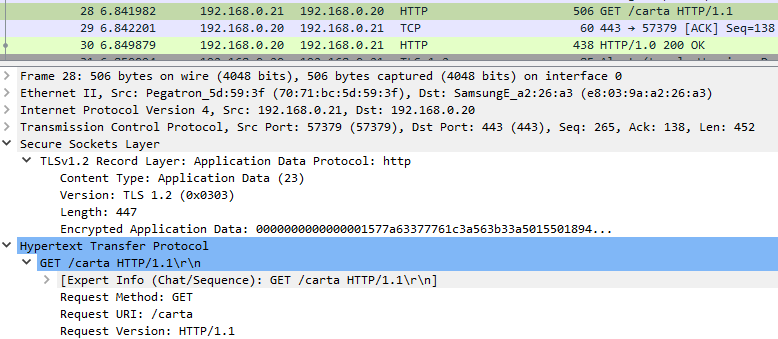
1. Il Client crea con i numeri random precedentemente scambiati un Pre-Master Secret, lo cifra con la chiave pubblica del server e lo invia con il messaggio “Client Key Exchange” (No.6);
2. Il Server ed il Client generano un Master Secret ed una Session Key basati sul Pre-Master Secret;
3. Il Client manda il messaggio “Change Cipher Spec” per indicare che ha iniziato ad usare la nuova Session Key per cifrare/decifrare i messaggi (No.6);
4. Infine il Server manda anche lui il messaggio “Change Cipher Spec” per indicare che ha iniziato ad usare la nuova Session Key per cifrare/decifrare i messaggi (No.9);

Si può notare che in realtà, HTTP Request e Response sono presenti ma cifrati (No. 7 e 10). Infatti i due messaggi **Application Data** nascondono al loro interno i nostri HTTP. Questo è proprio ciò che volevamo vedere, in quanto ci permette di validare il Web Server come sicuro e permette al sig. Dylan di dormire sonni tranquilli.



Come ulteriore conferma si può indicare a Wireshark dove trovare la Session Key generata dal Browser (impostando una variabile d’ambiente) per poter decifrare i messaggi ed avere la conferma che siano proprio i due Application data di nostro interesse.

Come si può notare, questa volta ricatturando i pacchetti viene mostrata sia la versione cifrata che quella decifrata.



# Conclusioni

Sviluppando questo progetto ho avuto modo di ampliare le mie conoscenze sulla crittografia, ma in particolar modo è stato appagante utilizzare quanto appreso durante l’anno scolastico, come punto di partenza per comprendere argomenti come i Socket SSL ed utilizzarli per creare un Web Server, in maniera del tutto autonoma, seguendo semplicemente le regole del protocollo.

Anche l’analisi del traffico di rete mi ha dato modo di verificare le mie conoscenze, rendendo possibile una comprensione quasi immediata di quanto catturato, grazie soprattutto allo studio precedentemente effettuato sul protocollo.

Si può notare come in Java sia del tutto trasparente l’Handshake effettuato dal protocollo TLS, lasciando tutta la gestione alla classe SSLServerSocketFactory. Questo, di fatto, è stato un ottimo punto di partenza, in quando la sua facile implementazione mi ha dato modo di iniziare il progetto agilmente e studiarne solo in seguito i punti più complicati.

Oggi è fondamentale garantire la sicurezza delle informazioni che transitano in rete: il protocollo HTTPS rende possibile questo, ma è necessario implementarlo nella maniera corretta, iniziando dalla scelta della lunghezza della chiave, che attualmente, secondo quanto consigliato dagli stessi creatori dell’algoritmo RSA, dovrebbe essere di ameno 2048 bit. Fortunatamente, i moderni browser segnalano qualsiasi tipo di non conformità con il protocollo SSL/TLS, come ad esempio un certificato non firmato da una CA o anche una chiave troppo corta. Inizialmente, nella fase di creazione delle chiavi, non avevo impostato il campo relativo alla lunghezza, browser come Firefox o Chrome si rifiutavano di continuare la comunicazione, mentre Internet Explorer segnalava solamente l’anomalia.

Attualmente il protocollo TLS risulta inviolato, ma questo non significa che lo sarà per sempre, quindi è molto importante rimanere sempre aggiornati sull’evoluzione della crittografia, per non rischiare che i nostri dati finiscano nelle mani sbagliate.

# Riferimenti

## Libri

1. Sistemi e reti (Luigi Lo Russo, Elena Bianchi) HOEPLI
2. Tecnologie e progettazione di sistemi informatici (Paolo Camagni, Riccardo Nikolassy) HOEPLI
3. Codici & segreti (Simon Singh)

## Internet

1. HTTP (HyperText Transfer Protocol) (<https://www3.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html>)
2. Java Secure Socket Extension Reference Guide

(<https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html>)

1. Programmazione di rete - Socket SSL/TLS

(http://docplayer.it/1153220-Lezione-5-socket-ssl-tls-corso-di-programmazione-in-rete-laurea-magistrale-in-ing-informatica-universita-degli-studi-di-salerno.html)

## **Wikipedia**

1. Data Encryption Standard  
   (<https://it.wikipedia.org/wiki/Data_Encryption_Standard>)
2. Triple DES   
   (<https://it.wikipedia.org/wiki/Triple_DES>)
3. Advanced Encryption Standard   
   (<https://it.wikipedia.org/wiki/Advanced_Encryption_Standard>)
4. Certificato Digitale  
   (<https://it.wikipedia.org/wiki/Certificato_digitale>)
5. HTTPS  
   (<https://it.wikipedia.org/wiki/HTTPS%0Ahttps://it.wikipedia.org/wiki/HTTPS>)

## **Software**

1. Eclipse 4.5.2 (scrittura del Web Server)
2. Java 1.8 (esecuzione del Web Server)
3. Mozilla Firefox 47.0 (prova del Web Server)
4. Wireshark 2.0.4 (analisi dei pacchetti)
5. Microsoft Word (scrittura della tesina)
6. Gimp (editare gli screenshot)
7. Bootstrap (modello utilizzato per la presentazione)