

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



1. SOMMARIO

1. SOMMARIO	1
2. ABSTRACT	2
3. LA CRITTOGRAFIA	3
3.1. CRITTOGRAFIA SIMMETRICA	3
3.2. CRITTOGRAFIA ASIMMETRICA	4
3.3. L'ALGORITMO RSA	5
3.4. IL PROTOCOLLO SSL/TLS	6
4. JAVA: CHIAVI, CERTIFICATI E SOCKET SSL	7
4.1. GESTIRE CHIAVI E CERTIFICATI IN JAVA (KEYTOOL)	7
4.2. L'USO DEI SOCKET SSL/TLS IN JAVA	10
5. UN SEMPLICE WEB SERVER HTTPS IN JAVA	11
5.1. INTRODUZIONE	11
5.2. CICLO DI VITA DEL WEB SERVER	11
5.3. CREAZIONE DEL SERVER SOCKET	12
5.4. ATTESA DEL CLIENT	13
5.5. HTTP REQUEST	14
5.6. HTTP RESPONSE	16
5.7. IL COMPORTAMENTO DI UN BROWSER INTERROGANDO IL WEB SERVER	18
6. ANALISI DEI PACCHETTI CON WIRESHARK	19
6.1. INTRODUZIONE	19
6.2. HTTP SNIFFING CON WIRESHARK	20
6.3. HTTPS SNIFFING CON WIRESHARK	21
7. CONCLUSIONI	23
8. RIFERIMENTI	24
8.1. LIBRI	24
8.2. INTERNET	24
8.3. WIKIPEDIA	24
8.4. SOFTWARE	24

2. 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 oggi con l'enormità di dati sensibili che viaggiano su internet, come ad esempio quelli scambiati durante gli acquisti online.

3.2. 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.

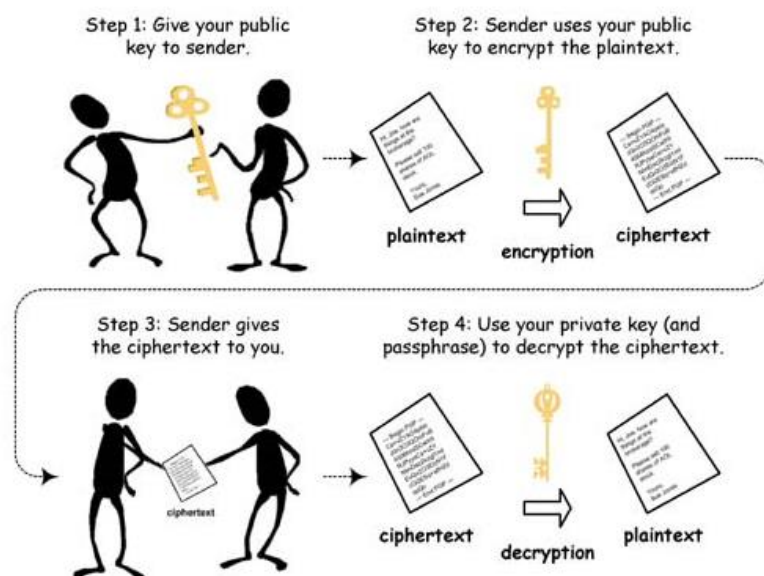
Formalmente è necessario trovare una funzione ("il lucchetto") la cui trasmissione su canali insicuri non comprometta l'algoritmo, che sia facile da applicare (parte pubblica che chiude il lucchetto) ma difficile da invertire (parte privata che apre il lucchetto).

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.

3.3. L'ALGORITMO 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 $n = p \cdot q$.

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.

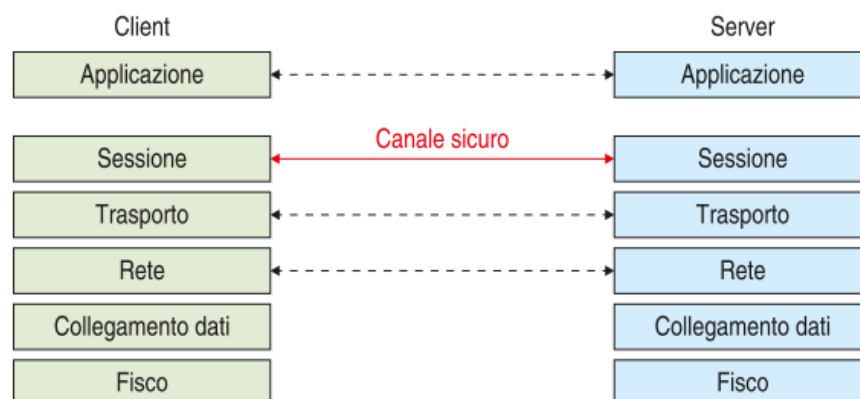
Questo sistema però comporta un problema, cioè che le funzioni matematiche che generano il codice cifrato e quelle inverse impiegano troppo tempo per essere utilizzate per la cifratura di interi documenti, per questo sono nati i **sistemi di crittografia misti** che uniscono la tecnica a cifratura asimmetrica 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.

3.4. 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.

4. JAVA: CHIAVI, CERTIFICATI E SOCKET SSL

4.1. 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

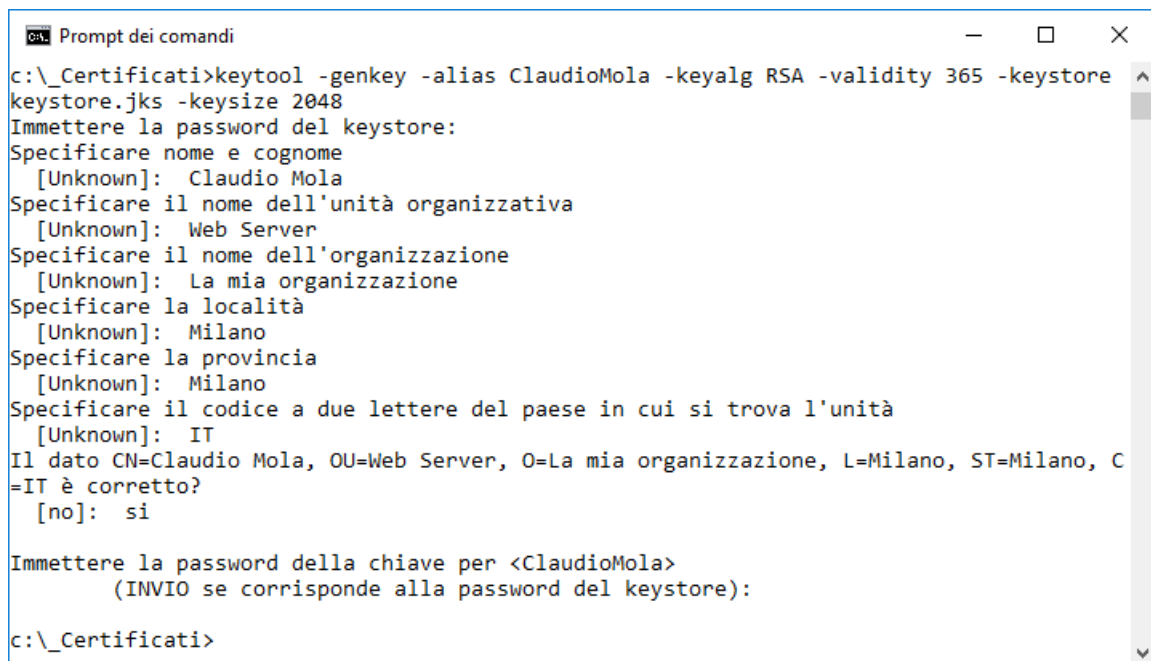
Le chiavi pubbliche e private sono memorizzate in un **keystore** e i certificati ritenuti “fidati” sono memorizzati in un **truststore**. Il formato del keystore e del truststore è proprietario, ma keytool offre funzioni per import/export di chiavi e certificati nei formati standard.

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

```
keytool -genkey [opzioni] -alias nome -keyalg 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:



```
Prompt dei comandi
c:\_Certificati>keytool -genkey -alias ClaudioMola -keyalg RSA -validity 365 -keystore
keystore.jks -keysize 2048
Immettere la password del keystore:
Specificare nome e cognome
[Unknown]: Claudio Mola
Specificare il nome dell'unità organizzativa
[Unknown]: Web Server
Specificare il nome dell'organizzazione
[Unknown]: La mia organizzazione
Specificare la località
[Unknown]: Milano
Specificare la provincia
[Unknown]: Milano
Specificare il codice a due lettere del paese in cui si trova l'unità
[Unknown]: IT
Il dato CN=Claudio Mola, OU=Web Server, O=La mia organizzazione, L=Milano, ST=Milano, C
=IT è corretto?
[no]: si

Immettere la password della chiave per <ClaudioMola>
(INVIO se corrisponde alla password del keystore):

c:\_Certificati>
```

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.

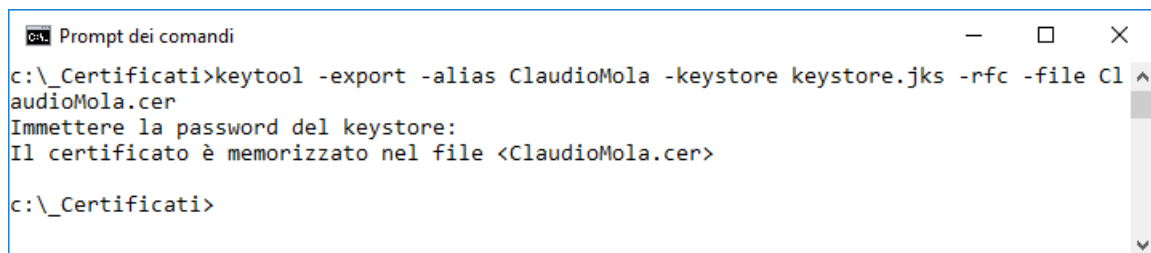
Quindi i passi diventano:

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:



```
Prompt dei comandi
c:\_Certificati>keytool -export -alias ClaudioMola -keystore keystore.jks -rfc -file Cl
audioMola.cer
Immettere la password del keystore:
Il certificato è memorizzato nel file <ClaudioMola.cer>
c:\_Certificati>
```

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:

```
Prompt dei comandi
c:\_Certificati>keytool -import -alias ClaudioMola -keystore truststore.jks -file Claud
ioMola.cer
Immettere la password del keystore:
Proprietario: CN=Claudio Mola, OU=Web Server, O=La mia organizzazione, L=Milano, ST=Mil
ano, C=IT
Autorità emittente: CN=Claudio Mola, OU=Web Server, O=La mia organizzazione, L=Milano,
ST=Milano, C=IT
Numero di serie: 57d1cc03
Valido da: Mon Jun 20 07:11:04 CEST 2016 a: Tue Jun 20 07:11:04 CEST 2017
Impronte digitali certificato:
    MD5: 35:F0:83:81:32:5E:58:49:82:FF:2D:6A:7F:12:84:51
    SHA1: 9E:12:BB:A8:27:29:10:A3:BB:67:04:7F:D9:53:66:40:6D:13:52:B6
    SHA256: 6F:24:EC:01:FE:D9:43:76:D2:9E:95:BB:8A:2A:D9:F0:53:52:A9:AE:DC:A9:EB:9
D:15:7C:C7:84:A6:10:6A:A0
    Nome algoritmo firma: SHA256withRSA
    Versione: 3

Estensioni:
#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 1C 47 2D F2 7B F0 2D 6A 3A 06 1B F2 D5 84 3D FB .G-...-j:.....=.
0010: D9 85 5B 41 ..[A
]
]

Considerare sicuro questo certificato? [no]: si
Il certificato è stato aggiunto al keystore

c:\_Certificati>
```

4.2. 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:

```
javax.net.*
javax.net.ssl.*
```

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**.

```
SocketFactory factory = SocketFactory.getDefault();
```

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

```
Socket client = factory.createSocket(host, porta);
```

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()**:

```
System.setProperty("javax.net.ssl.keyStore",
                   "C:\\Certificati\\keystore.jks");
System.setProperty("javax.net.ssl.keyStorePassword",
                   "unA_passworD_sicurA");
System.setProperty("javax.net.ssl.trustStore",
                   "C:\\Certificati\\truststore.jks");
System.setProperty("javax.net.ssl.trustStorePassword",
                   "unA_passworD_sicurA");
```

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

```
ServerSocketFactory serverFactory = SSLServerSocketFactory.getDefault();
SSLServerSocket server = (SSLServerSocket)
    serverFactory.createServerSocket(porta);
```

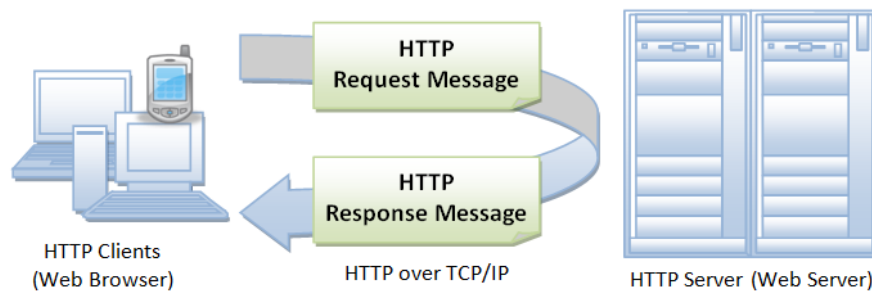
Una volta creato, il server socket si usa esattamente come un ServerSocket non crittografato, il keystore e il truststore devono essere specificati con le stesse proprietà di sistema. Per default i socket creati dalla factory effettuano l'autenticazione del solo server, ma se si desiderasse l'autenticazione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

```
server.setNeedClientAuth(true);
```

5. UN SEMPLICE WEB SERVER HTTPS IN JAVA

5.1. 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.

5.2. 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.

5.3. 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.

```
public void creaServer() {
    try {
        // Fornisco le chiavi ed il certificato con le relative password
        System.setProperty("javax.net.ssl.keyStore", "C:\\_Certificati\\keystore.jks");
        System.setProperty("javax.net.ssl.keyStorePassword", "password");
        System.setProperty("javax.net.ssl.trustStore", "C:\\_Certificati\\truststore.jks");
        System.setProperty("javax.net.ssl.trustStorePassword", "password");

        // SSLServerSocketFactory mi permette di creare un ServerSocket
        serverFactory = (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
        server = (SSLServerSocket) serverFactory.createServerSocket(port);
        System.out.println("Server creato con successo");
        System.out.println("Server: " + new Date() + "\\n\\n");
    } catch (BindException e) {
        System.out.println("Server: Assicurarsi che un'altra istanza del programma non sia già in esecuzione");
        System.out.println("Server: " + e);
        System.exit(1);
    } catch (Exception e) {
        System.out.println("Server: Assicurarsi di aver posizionato i file \\\"keystore.jks\\\" e \\\"truststore.jks\\\"");
        System.out.println("Server: nella cartella \\\"C:\\_Certificati\\\" e di aver creato il file \\\"index.html\\\" nella cartella "
            + this.home);
        System.out.println("Server: " + e);
        System.exit(1);
    }
}
```

5.4. 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.

```
// Attendo una richiesta dal client e rispondo
public void connessione() {
    try {
        // Mi metto in attesa del client
        System.out.println("Server: In attesa di un client sulla porta " + server.getLocalPort() + "...");
        SSLSocket client = (SSLSocket) server.accept();

        // Mostro le informazioni sul client
        System.out.println(
            "Server: nuovo client " + client.getInetAddress().getHostAddress() + ":" + client.getPort());
        System.out.println("Server: protocollo utilizzato " + client.getSession().getProtocol());
        System.out.println("Server: " + new Date());

        // Stream per leggere la richiesta del client
        Scanner input = new Scanner(client.getInputStream());

        // Recupero la richiesta HTTP del client
        if (clientHeader(client, input)) {

            // Invio il file richiesto al client
            // solo se esiste una richiesta valida
            inviaFile(client);
        }

        // chiudo la connessione con il client
        input.close();
        client.close();

        System.out.println("Server: connessione terminata\n\n");
    } catch (Exception e) {
        System.out.println("Server: Errore 1 - " + e);
    }
}
```

Qui di seguito è mostrato un screenshot 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.

```
Server: In attesa di un client sulla porta 443...
Server: nuovo client 192.168.100.198:50999
Server: protocollo utilizzato TLSv1.2
Server: Tue Jun 21 13:30:34 CEST 2016
```

5.5. 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.

```
GET /index.html HTTP/1.1
Host: 192.168.100.173
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 6.1; Win64; x64)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch, br
Accept-Language: it-IT,it;q=0.8,en-US;q=0.6,en;q=0.4
```

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.

```
// Recupero la richiesta HTTP del client
private Boolean clientHeader(SSLSocket client, Scanner input) throws Exception {

    uri = new URI(""); // contiene il percorso del file richiesto dal client
    String inputTxt = ""; // stringa di supporto
    Boolean status = true; // Esiste una richiesta HTTP?

    try {
        // Verifico se il client ha inviato un'intestazione
        // se non lo ha fatto restituisco "false"
        if (input.hasNext()) {
            inputTxt = input.nextLine();

            //Attraverso un'espressione regolare divido la stringa
            String[] split = inputTxt.split("\\s[a-zA-Z]*", 3);
            method = split[0];
            uri = new URI(split[1]);
            httpVersion = split[2];

        } else
            status = false;

        // Mostro il resto dell'intestazione
        while (!inputTxt.isEmpty()) {
            System.out.println("Request header: " + inputTxt);
            inputTxt = input.nextLine();
        }

    } catch (Exception e) {
        System.out.println("Server: Errore 2 - " + e);
    }

    return status;
}
```


5.6. HTTP RESPONSE

In quest'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".

```
// Provo a recuperare un percorso valido dalla combinazione della directory
// Home più quanto richiesto dal client
private File trovaFile() {

    String defaultFileName = "index.html"; // file da cercare se non
                                           // specificato
    Path root = Paths.get(home); // Percorso sul server dove cercare i file
    Path joint; // Percorso dato dall'unione di root più quanto richiesto
               // dal client

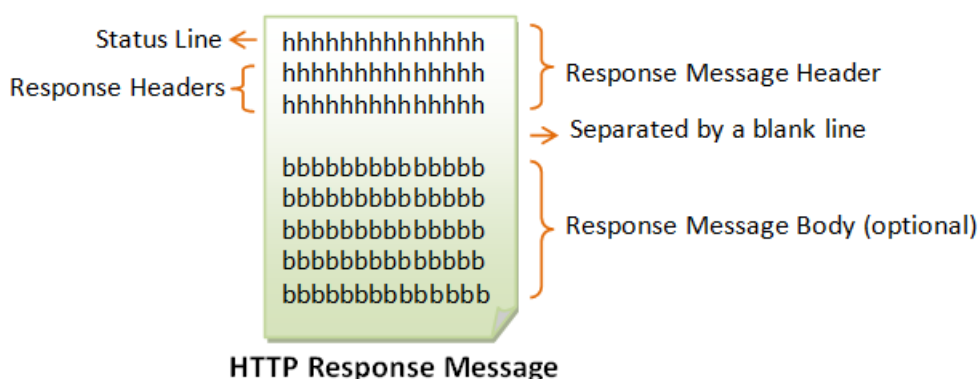
    // Unisco il percorso
    joint = Paths.get(root.toString(), uri.getPath());

    if (!joint.toFile().exists()) // Se non esiste lo stampo a console
        System.out.println("Server: File non trovato");
    else if (joint.toFile().isDirectory()) { // Se è una directory aggiungo
                                           // il "defaultFileName"
        System.out.println("Server: Provo a cercare " + defaultFileName);
        joint = Paths.get(joint.toString(), defaultFileName);
    }

    System.out.println("Server: Richiesta " + uri.getPath());
    System.out.println("Server: File inviato " + joint);

    return joint.toFile(); // Restituisco il percorso di tentativo
}
```

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.



HTTP Response Message

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.

```
HTTP/1.0 200 OK
Date: Tue Jun 21 19:45:46 CEST 2016
Server: Piccolo HttpsWebServer 1.0 by Claudio Mola
```

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.

```
// Invio il file richiesto al client
private void inviaFile(Socket client) throws IOException {
    try {
        // Provo a recuperare un percorso valido dalla combinazione della
        // directory Home più quanto richiesto dal client
        File file = trovaFile();

        // Creo gli stream, in input dal file in output sul socket
        FileInputStream input = new FileInputStream(file);
        BufferedOutputStream output = new BufferedOutputStream(client.getOutputStream());

        // Il messaggio di risposta per il client specificando la versione
        // del protocollo HTTP,
        // lo status code ed una spiegazione
        String responseHeader = "HTTP/1.1 200 OK\r\nDate: " + new Date() + "\r\nServer: " + this.serverName
            + "\r\n\r\n";
        output.write(responseHeader.getBytes());

        // Leggo ed invio il file
        while (input.available() > 0)
            output.write(input.read());

        // Forzo l'invio e chiudo gli stream
        output.flush();
        output.close();
        input.close();

    } catch (Exception e) {
        // Nel caso non esista il file richiesto sarà generata un'eccezione
        // invierò al client un messaggio di errore (404 not found)
        // più la relativa pagina come fatto per gli altri file
        System.out.println("Server: 404 Not Found");

        File file = Paths.get(home).resolve("Error.html").toFile();

        FileInputStream input = new FileInputStream(file);
        BufferedOutputStream output = new BufferedOutputStream(client.getOutputStream());

        String httpResponse = "HTTP/1.0 404 Not Found\r\n\r\n";
        output.write(httpResponse.getBytes());

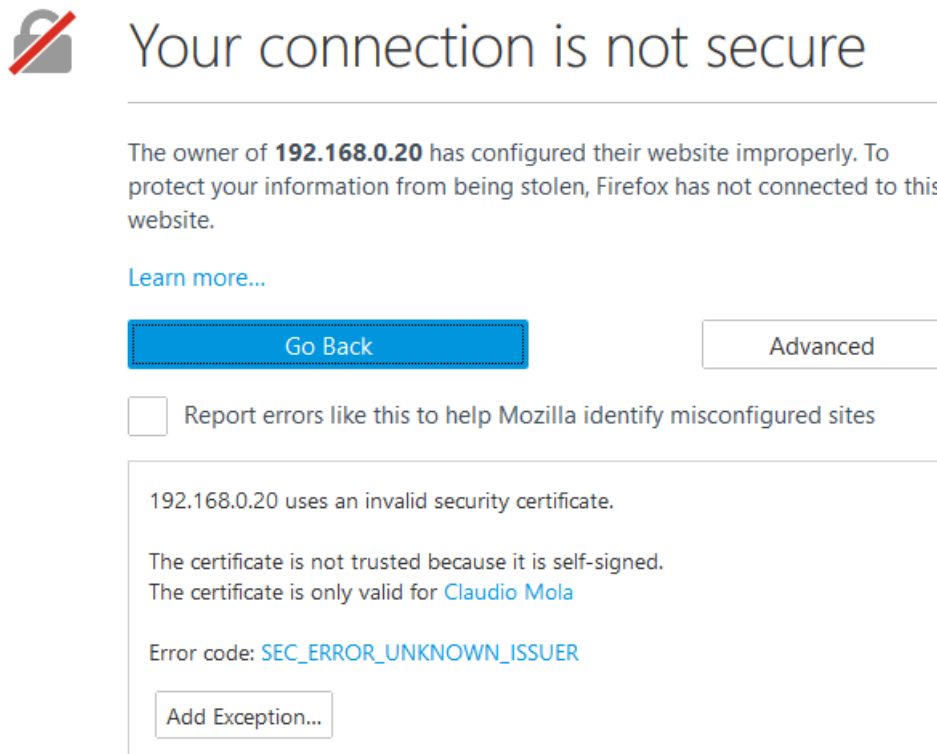
        while (input.available() > 0) {
            output.write(input.read());
        }

        output.flush();
        output.close();
        input.close();
    }
}
```

5.7. 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.



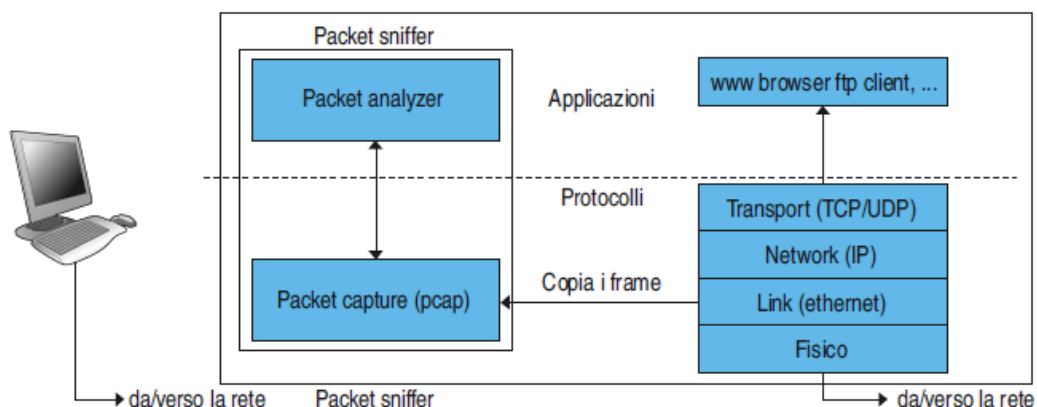
In questo caso scegliamo di fidarci, dal momento che ne conosciamo la fonte. Decidiamo di salvare il certificato localmente e di aggiungerlo come eccezione: così facendo, sarà considerato affidabile e nel caso in cui, in futuro, il web server ci invii un certificato diverso, ci sarà segnalato tempestivamente.

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"**.

6. ANALISI DEI PACCHETTI CON WIRESHARK

6.1. 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.

6.2. 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.

← ⓘ 192.168.0.20/carta

Titolare della carta: **Dylan Dog**
Numero: **4013 6733 8434 5717**
Tipo di carta: **Visa**
Scadenza: **06/17**

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 da visualizzare 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).

ip.addr == 192.168.0.20						
No.	Time	Source	Destination	Protocol	Length	Info
5	3.046731	192.168.0.21	192.168.0.20	TCP	66	57230 → 80 [SYN] Seq=0
6	3.047161	192.168.0.20	192.168.0.21	TCP	66	80 → 57230 [SYN, ACK] S
7	3.047252	192.168.0.21	192.168.0.20	TCP	54	57230 → 80 [ACK] Seq=1
8	3.047415	192.168.0.21	192.168.0.20	HTTP	351	GET /carta HTTP/1.1
9	3.054992	192.168.0.20	192.168.0.21	HTTP	408	HTTP/1.0 200 OK
10	3.055208	192.168.0.20	192.168.0.21	TCP	60	80 → 57230 [FIN, ACK] S
11	3.055266	192.168.0.21	192.168.0.20	TCP	54	57230 → 80 [ACK] Seq=25
12	3.055500	192.168.0.21	192.168.0.20	TCP	54	57230 → 80 [FIN, ACK] S
13	3.055980	192.168.0.20	192.168.0.21	TCP	60	80 → 57230 [ACK] Seq=35

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 oggi giorno è difficile trovare negozi online che non utilizzino il protocollo HTTPS.

Wireshark · Segui il flusso TCP (tcp.stream eq 0) · wireshark_pcapng_A6E15420-089A-4A6C-B4...

```
GET /carta HTTP/1.1
Host: 192.168.0.20
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:47.0) Gecko/20100101 Firefox/47.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive

HTTP/1.0 200 OK
Date: Fri Jun 24 15:32:19 CEST 2016
Server: Piccolo HttpWebServer 1.0 by Claudio Mola

<HTML>
  <HEAD>
    <TITLE>Dati Carta di Credito</TITLE>
  </HEAD>
  <BODY>
    Titolare della carta: <b>Dylan Dog</b><br>
    Numero: <b>4013 6733 8434 5717</b><br>
    Tipo di carta: <b>Visa</b><br>
    Scadenza: <b>06/17</b><br>
  </BODY>
</HTML>
```

6.3. 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.

ip.addr == 192.168.0.20						
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.0.21	192.168.0.20	TCP	66	57262 → 443 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
2	0.000480	192.168.0.20	192.168.0.21	TCP	66	443 → 57262 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
3	0.000594	192.168.0.21	192.168.0.20	TCP	54	57262 → 443 [ACK] Seq=1 Ack=1 Win=65536 Len=0
4	0.000991	192.168.0.21	192.168.0.20	TLSv1.2	227	Client Hello
5	0.043160	192.168.0.20	192.168.0.21	TLSv1.2	1408	Server Hello, Certificate, Server Key Exchange, Server Hello Done
6	0.050234	192.168.0.21	192.168.0.20	TLSv1.2	180	Client Key Exchange, Change Cipher Spec, Hello Request, Hello Request
7	0.050597	192.168.0.21	192.168.0.20	TLSv1.2	400	Application Data
8	0.051306	192.168.0.20	192.168.0.21	TCP	60	443 → 57262 [ACK] Seq=1355 Ack=646 Win=65024 Len=0
9	0.058636	192.168.0.20	192.168.0.21	TLSv1.2	60	Change Cipher Spec
10	0.065518	192.168.0.20	192.168.0.21	TLSv1.2	514	Hello Request, Hello Request, Application Data, Encrypted Alert
11	0.065627	192.168.0.21	192.168.0.20	TCP	54	57262 → 443 [ACK] Seq=646 Ack=1822 Win=65536 Len=0
12	0.066086	192.168.0.21	192.168.0.20	TCP	54	57262 → 443 [FIN, ACK] Seq=646 Ack=1822 Win=65536 Len=0
13	0.066300	192.168.0.20	192.168.0.21	TCP	60	443 → 57262 [ACK] Seq=1822 Ack=647 Win=65024 Len=0

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);
 - ▼ Handshake Protocol: Client Hello
 - Handshake Type: Client Hello (1)
 - Length: 196
 - Version: TLS 1.2 (0x0303)
 - ▼ Random
 - GMT Unix Time: Oct 20, 1974 23:37:37.000000000 ora legale Europa occidentale
 - Random Bytes: 5973458c4607379c56af05ce2d51fcc77659cee5a3e84567...
2. Il Server risponde con un messaggio "Server Hello" inviando anche lui un numero casuale (No.5);
3. Il Server invia il suo certificato per autenticarsi ed il messaggio "Server Hello Done" (No.5);
 - ▼ signedCertificate
 - version: v3 (2)
 - serialNumber: 33473931
 - > signature (sha256WithRSAEncryption)
 - ▼ issuer: rdnSequence (0)
 - ▼ rdnSequence: 6 items (id-at-commonName=Claudio Mola,id-at-organizationalUn
 - > RDNSquence item: 1 item (id-at-countryName=IT)
 - > RDNSquence item: 1 item (id-at-stateOrProvinceName=Milano)
 - > RDNSquence item: 1 item (id-at-localityName=Milano)
 - > RDNSquence item: 1 item (id-at-organizationName=La mia organizzazione)
 - > RDNSquence item: 1 item (id-at-organizationalUnitName=Web Server)
 - > RDNSquence item: 1 item (id-at-commonName=Claudio Mola)
4. 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);
5. Il Server ed il Client generano un Master Secret ed una Session Key basati sul Pre-Master Secret;
6. 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);

7. 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.

```
▼ Secure Sockets Layer
  ▼ TLSv1.2 Record Layer: Application Data Protocol: http
    Content Type: Application Data (23)
    Version: TLS 1.2 (0x0303)
    Length: 341
    Encrypted Application Data: 0000000000000001e99047de021cd30c85fd996e9e5229a4...
```

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.

28	6.841982	192.168.0.21	192.168.0.20	HTTP	506 GET /carta HTTP/1.1
29	6.842201	192.168.0.20	192.168.0.21	TCP	60 443 → 57379 [ACK] Seq=138
30	6.849879	192.168.0.20	192.168.0.21	HTTP	438 HTTP/1.0 200 OK

```
> Frame 28: 506 bytes on wire (4048 bits), 506 bytes captured (4048 bits) on interface 0
> Ethernet II, Src: Pegatron_5d:59:3f (70:71:bc:5d:59:3f), Dst: SamsungE_a2:26:a3 (e8:03:9a:a2:26:a3)
> Internet Protocol Version 4, Src: 192.168.0.21, Dst: 192.168.0.20
> Transmission Control Protocol, Src Port: 57379 (57379), Dst Port: 443 (443), Seq: 265, Ack: 138, Len: 452
▼ Secure Sockets Layer
  ▼ TLSv1.2 Record Layer: Application Data Protocol: http
    Content Type: Application Data (23)
    Version: TLS 1.2 (0x0303)
    Length: 447
    Encrypted Application Data: 0000000000000001577a63377761c3a563b33a5015501894...
▼ Hypertext Transfer Protocol
  ▼ GET /carta HTTP/1.1\r\n
    > [Expert Info (Chat/Sequence): GET /carta HTTP/1.1\r\n]
    Request Method: GET
    Request URI: /carta
    Request Version: HTTP/1.1
```

7. 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 quanto 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.

8. RIFERIMENTI

8.1. 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)

8.2. INTERNET

4. **HTTP (HyperText Transfer Protocol)**
(https://www3.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html)
5. **Java Secure Socket Extension Reference Guide**
(<https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html>)
6. **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>)

8.3. WIKIPEDIA

7. **Data Encryption Standard**
(https://it.wikipedia.org/wiki/Data_Encryption_Standard)
8. **Triple DES**
(https://it.wikipedia.org/wiki/Triple_DES)
9. **Advanced Encryption Standard**
(https://it.wikipedia.org/wiki/Advanced_Encryption_Standard)
10. **Certificato Digitale**
(https://it.wikipedia.org/wiki/Certificato_digitale)
11. **HTTPS**
(<https://it.wikipedia.org/wiki/HTTPS%0Ahttps://it.wikipedia.org/wiki/HTTPS>)

8.4. SOFTWARE

12. **Eclipse 4.5.2** (scrittura del Web Server)
13. **Java 1.8** (esecuzione del Web Server)
14. **Mozilla Firefox 47.0** (prova del Web Server)
15. **Wireshark 2.0.4** (analisi dei pacchetti)
16. **Microsoft Word** (scrittura della tesina)
17. **Gimp** (editare gli screenshot)
18. **Bootstrap** (modello utilizzato per la presentazione)