



UNIVERSITÀ DEGLI STUDI DI SIENA
Facoltà di Ingegneria

Corso di Laurea Specialistica in
INGEGNERIA INFORMATICA

Progettazione di un authenticator per JBoss compatibile con l'infrastruttura Open Portal Guard del Comune di Grosseto

Tesi di Laurea di

Carlo Politi

Relatore:
Prof. Marco Maggini

Correlatore:
Dott. Ludwig Bargagli

Anno Accademico 2007/2008

Quest'opera è stata rilasciata sotto la licenza Creative Commons Attribuzione-Non commerciale-Condividi allo stesso modo 2.5 Italia. Per leggere una copia della licenza visita il sito web <http://creativecommons.org/licenses/by-nc-sa/2.5/it/> o spedisci una lettera a Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

*Dedicato ai miei genitori
perché senza di loro non sarei mai
arrivato a questo punto*



Indice

1	Introduzione.....	9
1.1	Single Sign-On.....	9
1.2	Autenticazione ed autorizzazione.....	10
1.3	LDAP.....	14
1.4	Il progetto Open Portal Guard.....	17
1.5	Organizzazione della Tesi.....	20
2	OPGAuthenticator.....	22
2.1	Studio iniziale.....	22
2.2	Memorizzazione delle credenziali degli utenti in Tomcat.....	30
	2.2.1 UserDatabaseRealm.....	32
	2.2.2 JNDIRealm.....	35
2.3	Memorizzazione delle credenziali degli utenti in JBoss.....	39
	2.3.1 UsersRolesLoginModule.....	42
	2.3.2 LdapLoginModule.....	44
2.4	OPG: primo approccio.....	46
	2.4.1 Configurazione di Apache HTTP Server.....	47
	2.4.1.1 mod_proxy.....	48
	2.4.1.2 mod_jk.....	52
	2.4.1.2.1 Configurazione di Apache.....	52
	2.4.1.2.2 Configurazione di Tomcat.....	54
	2.4.1.2.3 Step finale per l'attivazione.....	54
	2.4.2 BasicAuthenticator.....	54
2.5	OPGAuthenticator: il nuovo autenticatore.....	62
	2.5.1 Compatibilità dell'autenticatore.....	67

2.5.1.1	Collocazione della classe OPGAuthenticator.class.....	69
2.5.1.1.1	Tomcat: catalina.jar.....	69
2.5.1.1.2	JBoss: jbossweb.jar.....	70
2.5.2	Attivazione di OPGAuthenticator.....	73
2.6	OPG e la sicurezza.....	74
2.6.1	Mutua autenticazione tra Apache e Tomcat.....	77
2.6.2	Mutua autenticazione tra Apache e JBoss.....	90
3	LoginModuleOPG.....	93
3.1	JBoss, JBossSX ed i Login Module.....	93
3.2	Moduli astratti di login.....	101
3.2.1	AbstractServerLoginModule.....	101
3.2.2	UsernamePasswordLoginModule.....	106
3.2.3	Scelta delle classi astratte per il login.....	112
3.3	LoginModuleOPG come alternativa ad OPGAuthenticator.....	114
4	Conclusioni.....	126
	Bibliografia.....	129

Indice delle figure

1.1	Schema interazione sistemi autenticazione/autorizzazione.....	12
1.2	Schema tradizionale di autenticazione.....	13
1.3	Schema di comunicazione LDAP.....	16
1.4	Esempio albero directory LDAP.....	17
2.1	Schema DTD di jboss-web.xml.....	41
2.2	Schema Proxy Server.....	51
2.3	Pop-up richiesta dati generato dal BasicAuthenticator.....	58
2.4	Schema dell'attacco "Man In The Middle".....	74
2.5	Vari tipi di modalità di sicurezza nello stack TCP/IP.....	75
2.6	Stack di protocolli SSL.....	76
3.1	Errore HTTP 403 durante l'accesso ad una risorsa protetta.....	117
3.2	Inserimento dati non corretti nel pop-up per l'autenticazione.....	122
3.3	Risultato esecuzione servlet protetta con LoginModuleOPG.....	122
4.1	Schema riassuntivo del sistema Open Portal Guard.....	128

Indice delle tabelle

2.1	Possibili valori per il cipher-suite.....	86
2.2	Lista tag speciali per RSA e DH.....	88

Capitolo 1

Introduzione

1.1 Single Sign-On

La sicurezza informatica e la protezione di risorse contenenti dati sensibili sono argomenti sempre più attuali e di notevole interesse: è infatti importante evitare che venga garantito l'accesso a determinate risorse a coloro che non possiedono l'autorizzazione. Nel corso degli anni sono stati studiati sistemi, o protocolli, di autenticazione ed autorizzazione sempre più sofisticati che vanno sotto il nome di **Single Sign-On (SSO)**, traducibile con *autenticazione unica* od *identificazione unica*; sono infatti sistemi specializzati che permettono ad un utente di autenticarsi una sola volta e di accedere a tutte le risorse informatiche alle quali è abilitato. Gli obiettivi del SSO sono molteplici:

- semplificare la gestione delle password: maggiore è il numero delle password da gestire, maggiore è la possibilità che saranno utilizzate password simili le une alle altre e facili da memorizzare, abbassando il livello di sicurezza;
- semplificare la gestione degli accessi ai vari servizi;
- semplificare la definizione e la gestione delle politiche di sicurezza.

Vi sono anche varie tipologie di approcci per la creazione di un sistema di SSO:

- **centralizzato**: questa soluzione prevede la presenza di un database globale e centralizzato di utenti ed il principio è quello di centralizzare anche la politica di sicurezza. Questo approccio è destinato principalmente a servizi dipendenti tutti dalla stessa entità, ad esempio all'interno di un'azienda;
- **federativo**: con questo approccio differenti gestori, "federati" tra di loro, gestiscono i dati di uno stesso utente. L'accesso ad uno dei sistemi federati permette automaticamente l'accesso a tutti gli altri sistemi. Questa soluzione è stata sviluppata per rispondere ad un bisogno di gestione decentralizzata degli utenti: ogni gestore federato mantiene il controllo della propria politica di sicurezza;
- **cooperativo**: quest'ultima soluzione parte dal principio che ciascun utente dipenda, per ciascun servizio, da uno solo dei gestori cooperanti. In questo modo se si tenta di accedere, ad esempio, alla rete locale, l'autorizzazione viene effettuata dal gestore che ha in carico l'utente per l'accesso alla rete. Come per l'approccio federativo, in questa maniera ciascun gestore gestisce in un modo indipendente la propria politica di sicurezza; permette, poi, di rispondere ai bisogni di strutture istituzionali nelle quali gli utenti sono dipendenti da una entità, ad esempio in università, laboratori di ricerca, amministrazioni, ecc.

1.2 Autenticazione ed autorizzazione

E' importante distinguere queste due fasi dal momento che possono (e dovrebbero) essere realizzate da sistemi separati.

Autenticazione (authentication): meccanismo mediante il quale un sistema può identificare con sicurezza i suoi utenti. I sistemi di autenticazione forniscono una risposta alle domande [1]:

- chi è l'utente?
- l'utente è realmente chi dice di essere?

Un sistema di autenticazione può essere semplice (ma insicuro) come un sistema di sfida basato su password in chiaro oppure complesso come il protocollo di autenticazione Kerberos del Massachusetts Institute of Technology (MIT); in ogni caso, i sistemi di autenticazione utilizzano un'**informazione segreta condivisa** che è conosciuta (o disponibile) solo alla persona che si sta autenticando e al sistema di autenticazione. Tale informazione può essere la classica password, qualche proprietà fisica dell'individuo (es. impronte digitali, caratteristiche della retina, ecc.) o altre informazioni derivate quali possono essere i sistemi basati su **smart card**. Per verificare l'identità di un utente, il sistema di autenticazione esegue una serie di sfide in cui l'utente fornisce i suoi dati segreti (password, impronte digitali, ecc.): se il sistema di autenticazione riesce a verificare che l'informazione segreta condivisa è stata fornita correttamente, l'utente può considerarsi autenticato.

Autorizzazione (authorization): è il processo mediante il quale il sistema determina quale sia il livello di accesso a cui l'utente autenticato può accedere. Per esempio, un DBMS potrebbe essere stato progettato per fornire solo diritti di lettura a certi utenti mentre ad altri anche quelli di scrittura. I sistemi di autorizzazione forniscono risposta alle domande [2]:

- l'utente X è autorizzato ad accedere alla risorsa R?
- l'utente X è autorizzato ad eseguire l'operazione P?
- l'utente X è autorizzato ad eseguire l'operazione P sulla risorsa R?

Il sistema di autenticazione e quello di autorizzazione sono strettamente legati: un sistema di autenticazione dipende dal sistema di autenticazione sicuro per garantire che gli utenti siano quelli che dicono di essere e, di conseguenza, ostacolare gli utenti non autorizzati dall'ottenere l'accesso alle risorse protette. Nel diagramma sottostante (**figura 1.1**) un utente lavora su un client, interagendo con il sistema di autenticazione che ne verifica la sua identità: terminata questa fase, la comunicazione avviene con il server che a sua volta interagisce con il sistema di autorizzazione che determina i diritti ed i privilegi che dovrebbero essere assegnati all'utente del client.

Authentication vs. Authorization

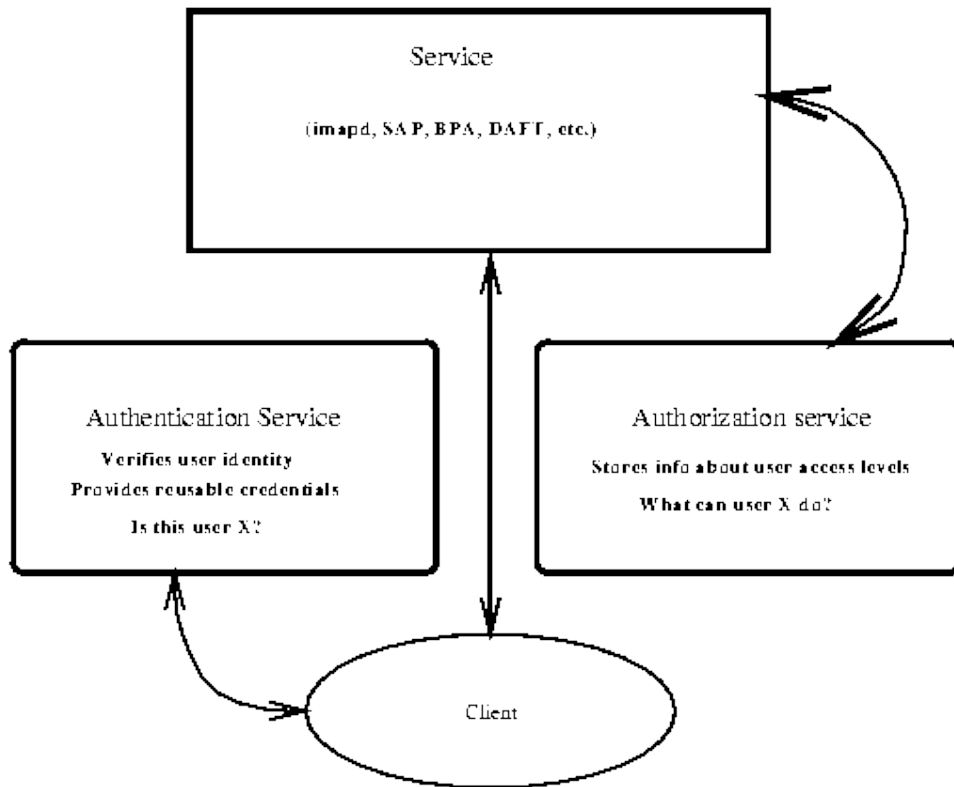


Figura 1.1: Tipica interazione tra sistemi di autenticazione/autorizzazione ed una tipica applicazione client/server

Il sistema di autenticazione tradizionale, altamente insicuro e purtroppo ancora molto utilizzato, è rappresentato dal metodo di autenticazione locale (**figura 1.2**). In questo modello, la coppia [username, password] di ciascun utente autenticabile, è memorizzata localmente su di un server. Gli utenti mandano il loro username e password in chiaro (nel caso di autenticazione di tipo Digest la password è inviata sotto forma di stringa codificata come un digest) al server che confronta tali dati con quelli memorizzati in locale. Se il confronto dà esito positivo, l'utente è considerato autenticato: questo è il principale modello utilizzato per l'autenticazione della login su tradizionali sistemi multi utenti.

Traditional (host/passwd based) Authentication Method

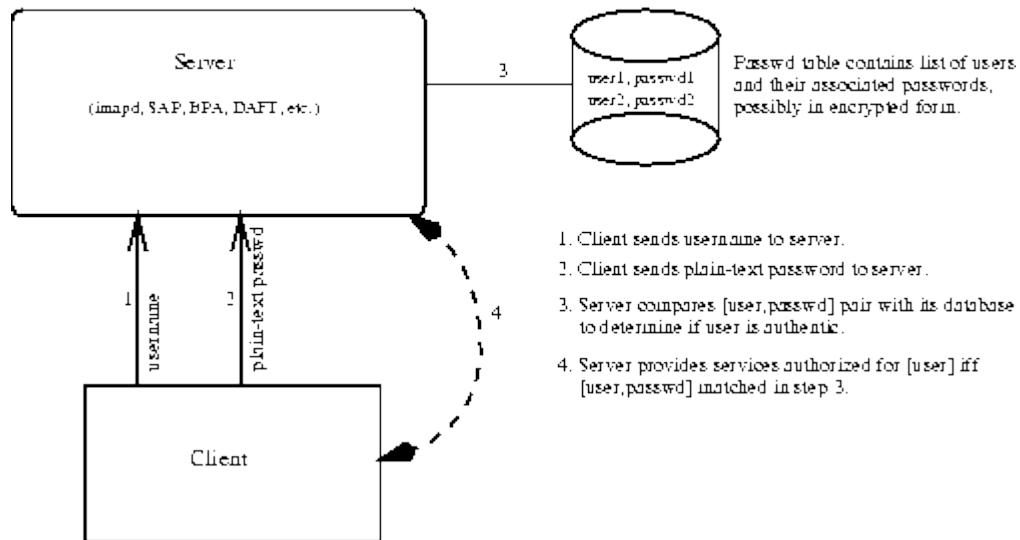


Figura 1.2: Metodo di autenticazione tradizionale

Questo modello ha non poche debolezze [2]:

1. in alcuni casi la password degli utenti è memorizzata in chiaro sul server: chiunque possa accedere al database delle password ottiene sufficienti informazioni per attuare un attacco di tipo impersonificazione, utilizzando i dati di un qualunque utente del sistema;
2. nei casi in cui la password degli utenti siano memorizzate crittate sul server, la password è inviata in chiaro sul canale, probabilmente non sicuro, tra client al server; chiunque potrebbe effettuare un attacco di tipo snoop (ossia potrebbe spiare) la coppia [username, password] dalla comunicazione tra client e server ed utilizzarla per effettuare l'accesso;
3. ciascun sistema separato, ossia ogni server di autenticazione, deve avere la propria copia delle informazioni di autenticazione di ciascun utente. Questo comporta che gli utenti devono conservare le proprie password su ciascun server a cui intendono autenticarsi: questo comporta che essi andranno a scegliere, per convenienza e praticità, password meno sicure o, caso non raro, sempre la stessa password in modo da non dover ricordarne una per ogni sistema;
4. l'autenticazione non è riutilizzabile ossia gli utenti devono autenticarsi

separatamente su ciascun sistema o applicazione a cui vogliono accedere: questo comporta che gli utenti debbano costantemente digitare la propria password; così facendo essi tenderanno a scegliere, per convenienza e praticità, password meno sicure;

5. non c'è alcun modo all'interno di questo modello, di fare la mutua autenticazione del client e del server. Un sistema che impersona il server (ad esempio mediante spoofing dell'indirizzo IP) non può essere distinto dal client dal server reale; in questo modo, è possibile che un server trojan horse raccolga le coppie [username, password] e che le usi per autenticarsi al vero server.

Con la nascita ed evoluzione di Internet e con il notevole incremento dei suoi utenti, sono sorti problemi di sicurezza in quanto la suite di protocolli **TCP/IP** non prevede nessun meccanismo atto a garantire confidenzialità e privacy tra gli utenti; gli scenari in cui questi due aspetti sono richiesti sono molteplici, ad esempio accedere in maniera privata ad una directory remota dislocata su di un server: ancora oggi per fare l'accesso remoto ad una directory si utilizza il protocollo FTP che, purtroppo, richiede ancora una semplice autenticazione mediante password trasmessa in chiaro, benché esista SFTP che permette di effettuare connessioni FTP in modo sicuro.

1.3 LDAP

Una risposta a questo bisogno di confidenzialità e privacy è stata data con la nascita di **LDAP (Lightweight Directory Access Protocol)** che, come suggerisce il nome, è un protocollo leggero per accedere ai servizi di directory, basati sul protocollo **X.500**, e può essere utilizzato direttamente sopra lo stack TCP/IP. Con il termine **Directory**, intendiamo un particolare database specializzato per la lettura e la ricerca: ha la possibilità di contenere informazioni basate su attributi o descrizioni e di supportare sofisticate capacità di ricerca attraverso dei filtri; la caratteristica principale di una directory è quella di fornire risposte veloci ad operazioni di consultazione o di ricerca su enormi volumi di dati. Un **Servizio di Directory** è un programma, o un insieme di programmi, che provvedono ad

organizzare e memorizzare informazioni su reti di computer e su risorse condivise disponibili tramite rete. Il servizio di directory fornisce anche un controllo degli accessi sull'utilizzo delle risorse condivise, in modo da favorire il lavoro dell'amministratore del sistema. Possiamo dire che i servizi di directory forniscono uno strato di astrazione tra le risorse e gli utenti. Ci sono differenti vie per implementare un servizio di directory e i vari metodi supportano differenti tipi di informazioni che possono essere memorizzate in una directory: alcuni servizi di directory sono locali, fornendo servizi ad un ristretto contesto, quale ad esempio una singola macchina; altri, invece, sono globali ed offrono i servizi ad un contesto più ampio quale può essere l'intera Internet. Tipicamente i servizi globali sono distribuiti, ciò vuol dire che i dati contenuti sono memorizzati in diverse macchine, ognuna delle quali provvede al servizio di directory: un esempio di questa struttura è rappresentata dall'Internet **Domain Name Service (DNS)**. Prima di LDAP, per accedere ai dati memorizzati in una directory X.500, un client doveva supportare il **Directory Access Protocol (DAP)** che però imponeva una notevole penalizzazione delle risorse in gioco, in quanto basato sulla specifica **Open System Interconnection (OSI)**, rimpiazzata successivamente dalla suite TCP/IP e da altri protocolli. LDAP, dunque, nasce proprio come sostituto del DAP che risultava molto più oneroso dal punto di vista dell'impiego di risorse. La struttura di un LDAP è client/server: un client LDAP invia una richiesta ad un server LDAP che elabora la richiesta ricevuta, accede eventualmente ad un directory database e ritorna dei risultati al client (**figura 1.3**).

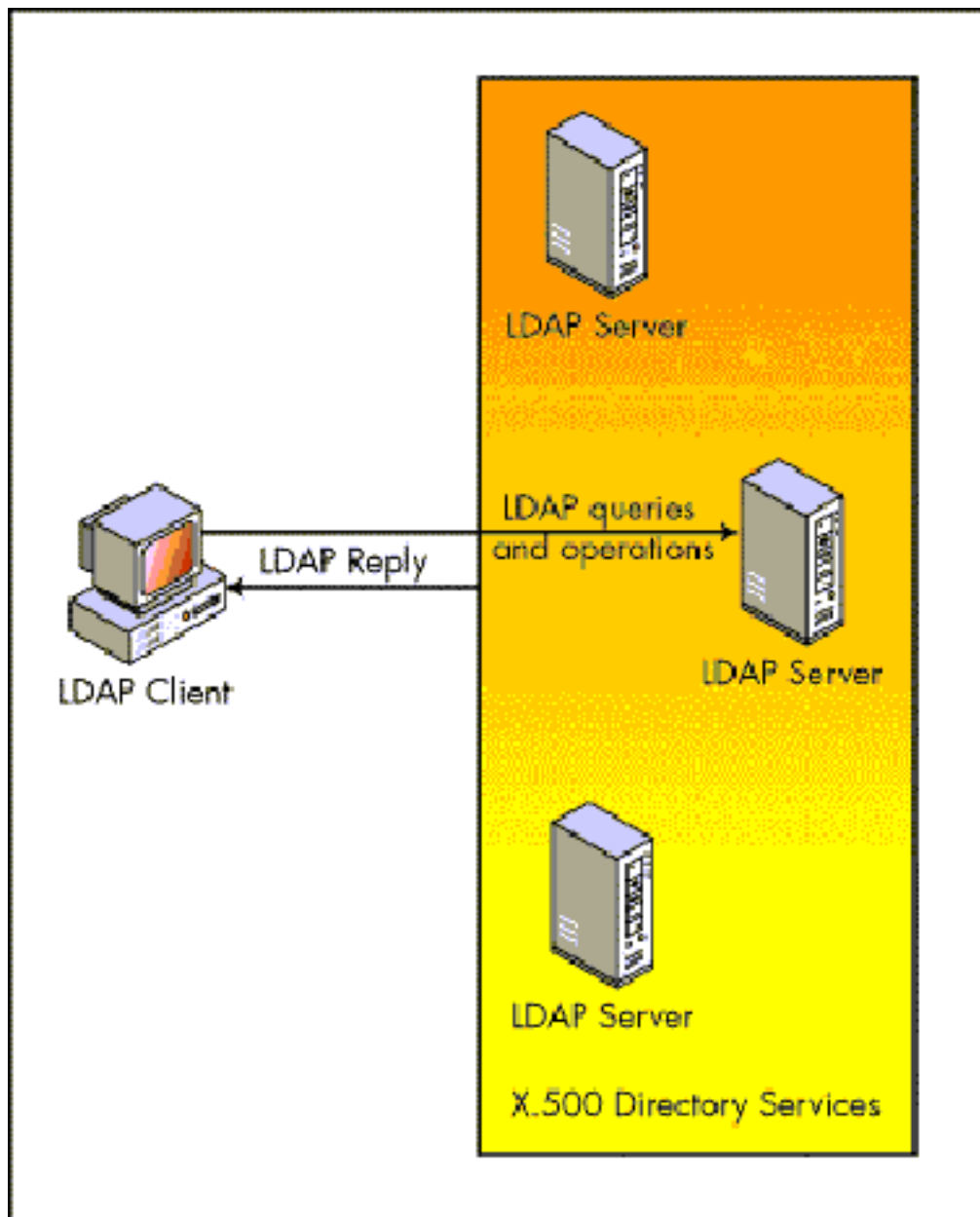


Figura 1.3: Schema di comunicazione con un server LDAP

Le informazioni, all'interno di un database LDAP, sono strutturate in **entry** ossia una collezione di attributi aventi un unico nome globale, il **Distinguished Name (DN)** che è usato per riferirsi ad una particolare entry, senza avere ambiguità. Ogni attributo delle entry ha un tipo ed uno o più valori; i tipi sono di solito stringhe mnemoniche come **cn** utilizzato per i **common name** (i nomi comuni) oppure *mail* per gli indirizzi di posta. In LDAP le entry di una directory sono strutturate secondo un'architettura gerarchica ad albero: le entry che rappresentano il paese si trovano alla radice dell'albero, al disotto troviamo quelle che rappresentano stati ed organizzazioni nazionali. Seguono poi altri tipi di entry

che possono rappresentare organizzazioni, persone, stampanti, documenti, ecc. (figura 1.4).

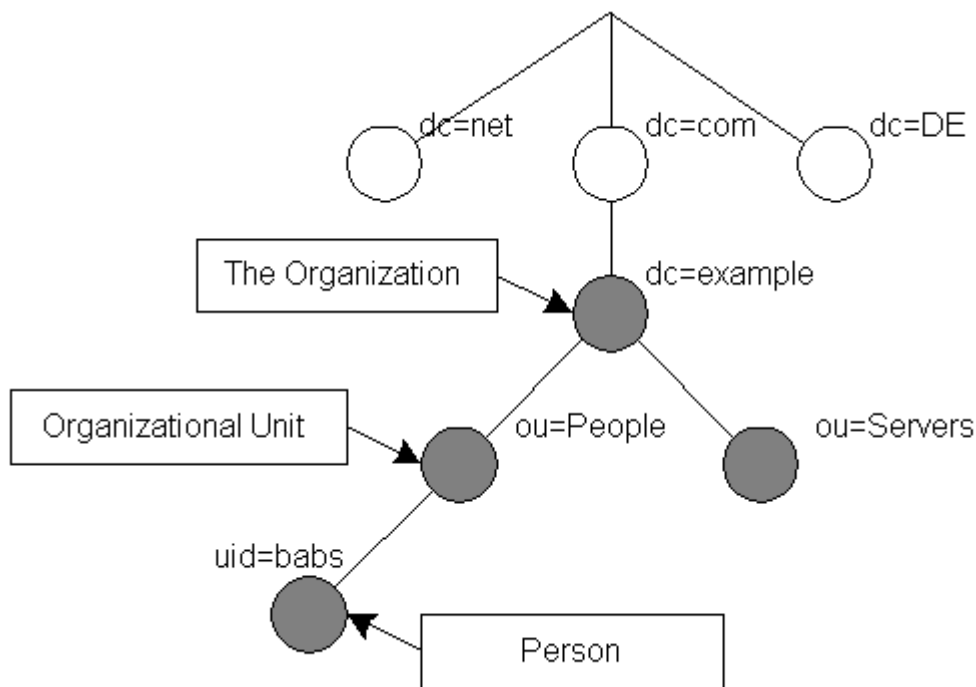


Figura 1.4: Albero directory LDAP (nomi utilizzati in Internet)

Una entry è indicata, come abbiamo detto, dal suo DN che è costruito prendendo il nome stesso dell'entry (chiamato **Relative Distinguished Name, RDN**) e concatenandolo ai nomi delle entry dei suoi predecessori nell'albero. Nella figura precedente, il DN è così formato, partendo dal suo RDN: *uid=babs, ou=People, dc=example, dc=com*.

Molti servizi di directory non prevedono protezione per i dati e le informazioni, permettendo a chiunque di accedere a tutti i dati; LDAP, invece, include un meccanismo nel quale un client si può autenticare o provare la sua identità ad una directory server: in questo modo si ha la possibilità di gestire servizi di privacy e di integrità delle informazioni.

1.4 Il progetto Open Portal Guard

Abbiamo accennato in apertura di questo capitolo al fatto che l'autenticazione avviene mediante un'informazione segreta condivisa e che, oltre ad avvenire

tramite l'utilizzo di password, può avvenire mediante un'informazione derivata come può essere una smart card. E' verso questa direzione che il Comune di Grosseto ha ideato il progetto chiamato **Open Portal Guard (OPG) [3]** atto a rendere sicuri i servizi elettronici della Pubblica Amministrazione offerti ai cittadini, attraverso l'utilizzo di una smart card per l'autenticazione che può essere una delle seguenti:

- la **Carta d'Identità Elettronica (CIE)** realizzata dal Ministero degli Interni e rilasciata dai Comuni;
- la **Carta Nazionale dei Servizi (CNS)** realizzata dal Ministero dell'Innovazione e della Tecnologia.

OPG permette di proteggere i servizi sensibili presenti sul proprio portale web mediante:

- un sistema di autenticazione Single Sign-On che utilizza:
 - la coppia [username, password] per ragione di legacy, ossia per motivo di retro compatibilità;
 - autenticazione client SSL mediante certificati e chiavi X.509 memorizzati su file o su vari tipi di smart card, ossia i due modelli indicati sopra.
- un sistema di accesso dichiarativo centralizzato.

Nel Codice dell'Amministrazione Digitale [4], la Carta Nazionale dei Servizi è definita come *il documento rilasciato su supporto informatico per consentire l'accesso per via telematica ai servizi erogati dalla pubblica amministrazione* (articolo 1 lettera bb). Tale definizione è rimasta inalterata (articolo 1, comma 1, lettera d)) nel decreto legislativo 4 aprile 2006, n. 159 recante "Disposizioni integrative e correttive al decreto legislativo 7 marzo 2005, n. 82, recante codice dell'amministrazione digitale". Il medesimo decreto legislativo aggiorna anche le regole per l'accesso ai servizi erogati in rete dalle pubbliche amministrazioni. L'articolo 64 contiene queste regole e di seguito lo riportiamo per comodità in modo integrale:

“64. Modalità di accesso ai servizi erogati in rete dalle pubbliche amministrazioni.

1. La carta d'identità elettronica e la carta nazionale dei servizi costituiscono strumenti per l'accesso ai servizi erogati in rete dalle pubbliche amministrazioni per i quali sia necessaria l'autenticazione informatica.

2. Le pubbliche amministrazioni possono consentire l'accesso ai servizi in rete da esse erogati che richiedono l'autenticazione informatica anche con strumenti diversi dalla carta d'identità elettronica e dalla carta nazionale dei servizi, purché tali strumenti consentano di accertare l'identità del soggetto che richiede l'accesso. L'accesso con carta d'identità elettronica e carta nazionale dei servizi è comunque consentito indipendentemente dalle modalità di accesso predisposte dalle singole amministrazioni.

3. Ferma restando la disciplina riguardante le trasmissioni telematiche gestite dal Ministero dell'economia e delle finanze e dalle agenzie fiscali, con decreto del Presidente del Consiglio dei Ministri o del Ministro delegato per l'innovazione e le tecnologie, di concerto con il Ministro per la funzione pubblica e d'intesa con la Conferenza unificata di cui all'articolo 8 del decreto legislativo 28 agosto 1997, n. 281, è fissata la data, comunque non successiva al 31 dicembre 2007, a decorrere dalla quale non è più consentito l'accesso ai servizi erogati in rete dalle pubbliche amministrazioni, con strumenti diversi dalla carta d'identità elettronica e dalla carta nazionale dei servizi. E' prorogato alla medesima data il termine relativo alla procedura di accertamento preventivo del possesso della Carta di identità elettronica (CIE), di cui all'articolo 8, comma 5, del decreto del Presidente della Repubblica 2 marzo 2004, n. 117, limitatamente alle richieste di emissione di Carte nazionali dei servizi (CNS) da parte dei cittadini non residenti nei comuni in cui è diffusa la CIE.”

Un ruolo centrale, in questa tesi, è ricoperto dal concetto di **Authenticator**. Con tale termine intendiamo un oggetto, che nello specifico del caso è una classe Java, la cui funzione è quella di ottenere l'autenticazione per una connessione di rete; usualmente tale processo avviene chiedendo informazioni particolari all'utente

che richiede una determinata risorsa protetta. Tali risorse protette sono, in particolare, applicazioni web installate su application server quale **Apache Tomcat** a cui possono accedere solo determinati utenti. Come abbiamo visto in precedenza, lo schema classico di autenticazione è quello mediante inserimento di [username, password] e la quasi totalità degli authenticator presenti in un application server seguono questo schema. L'entrata in uso delle smart card a cui abbiamo accennato prima, ha portato a pensare all'ipotesi di utilizzo anche all'interno di application server ma si è posto il problema che nessun autenticatore presente era già compatibile a supportare questo metodo di autenticazione. All'inizio del lavoro di tesi, tutti gli sforzi sono stati concentrati verso la creazione di un authenticator per Tomcat, che è un application server largamente utilizzato, essendo open source ma la vera sfida era rappresentata dal portare questi sforzi verso un altro tipo di application server, ossia, **JBoss**; quest'ultimo, a differenza di Tomcat, è un application server **Java 2 Enterprise Edition (J2EE) 1.4** compliant e non un semplice servlet container. Come vedremo in seguito, la creazione del nuovo autenticatore è stata completata, risultando compatibile con entrambi gli application server su cui l'abbiamo sviluppato. Qui di seguito elenchiamo l'ambiente con cui abbiamo operato:

1. Linux Ubuntu 7.10
2. Java SE Development Kit (JDK) 1.6
3. Eclipse WTP 3.2, come ambiente di sviluppo
4. Apache HTTP Server, come web server
5. Apache Tomcat 5.5.25
6. JBoss AS 4.2.1

1.5 Organizzazione della Tesi

Di seguito viene riportato il contenuto dei capitoli che compongono questo lavoro di tesi:

- nel primo capitolo abbiamo dato una descrizione del contesto e le motivazioni che hanno spinto a sviluppare un nuovo authenticator;

- nel secondo capitolo verrà prima presentato OPG, poi analizzato nel dettaglio quanto abbiamo fatto per arrivare alla creazione di un authenticator utilizzabile sia con Tomcat sia con JBoss. Presenteremo anche una soluzione atta a rendere la comunicazione più sicura tra Apache HTTP Server e l'application server di interesse, mediante l'utilizzo della mutua autenticazione in SSL;
- nel capitolo terzo invece discuteremo dell'architettura JBossSX di JBoss e di come abbiamo pensato alla creazione di un login module come alternativa all'authenticator;
- nel quarto ed ultimo capitolo tratteremo le conclusioni del lavoro di tesi.

Capitolo 2

OPGAuthenticator

2.1 Studio iniziale

Abbiamo svolto il lavoro suddividendolo in tappe: la prima era quella di leggere la documentazione disponibile relativa a Tomcat, studiandone in particolare le classi atte all'autenticazione/autorizzazione. Il motivo per cui siamo partiti da questo application server è, come abbiamo accennato, la sua grande diffusione che ha portato alla creazione di molta documentazione, presente sia nel sito ufficiale di questo prodotto software sia realizzata da terze parti e consultabile presso appositi forum di discussione. In questa prima fase, abbiamo visto il comportamento dei vari authenticator, in particolare di quello **BASIC**, anche grazie all'utilizzo di due servlet di prova, una protetta mediante questo tipo di autenticazione ed una priva. Ricordiamo che con il termine servlet intendiamo una componente applicativa server-side sviluppata in Java che risponde direttamente alle request http ed è in grado di restituire in streaming un flusso di response http [5]. Scopo di questa fase preliminare era vedere il flusso dati e verificare il comportamento dell'autenticator che è stato specificato, per le servlet, all'interno del rispettivo file di configurazione rappresentato dal file **web.xml**. Questo file viene chiamato **deployment descriptor** ed è un file XML che contiene tutte le definizioni necessarie al servlet container (quindi Tomcat e JBoss) per poter avviare ed eseguire correttamente l'applicazione web installata nell'application server. Il file web.xml segue le regole dettate dal suo **XML Schema** (che serve a descrivere il contenuto di un file XML) e contiene numerose informazioni, alcune

obbligatorie alcune opzionali, circa l'applicazione a cui fa riferimento. Vediamo come sono costituite queste due servlet, prendendone in considerazione solo una, dal momento che sono identiche, questo per comprendere cosa succede. Premettiamo che le servlet che abbiamo creato, generano come output una pagina HTML e quindi le possiamo richiamare e visualizzare all'interno di un qualunque browser web quale ad esempio Internet Explorer o Firefox. Vediamo quali sono i casi possibili di esito quando viene richiamata una delle due servlet: un primo caso è che venga scatenata un'eccezione, e questo succede se chi ha richiamato la servlet non è stato autenticato; il verificarsi di questa casistica comporta che nell'output della servlet viene visualizzato un messaggio di errore sulla mancata autenticazione. Questo avviene soprattutto se viene eseguita la servlet non protetta in quanto, nel codice, viene fatta esplicita richiesta di accesso alle informazioni di autenticazione dell'utente, non presenti ovviamente se viene eseguita la servlet (o in generale una risorsa) non protetta. Va precisato che se viene eseguita la servlet protetta ma chi la richiede non viene autenticato correttamente, in quel caso è l'application server ad intervenire e a generare una schermata di errore **HTTP 403** che informa che non si può accedere a tale risorsa. L'altra casistica che si può avere eseguendo la servlet è quella in cui l'utente si è autenticato correttamente: in questo caso, come accennato, la servlet esegue alcune istruzioni che vanno a richiedere i dati di autenticazione dell'utente, in particolare va a leggere e visualizzare nell'output quale sia il nome dell'utente (il suo userid) che ha richiesto la risorsa e tale valore di userid è stato associato, all'interno delle politiche di sicurezza dell'application server, ai ruoli **prova1** e **proveVarie**; specifichiamo che *proveVarie* è il ruolo che abbiamo associato alla servlet protetta mentre *prova1* è fittizio e non è associato effettivamente ad alcuna risorsa. Riportiamo ora il contenuto di una delle due servlet, in particolare quella che si chiama **Prova.java** per poi commentare alcune parti di interesse del codice:

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import javax.servlet.ServletException;
```

```

public class Prova extends javax.servlet.http.HttpServlet implements
javax.servlet.Servlet {

private static final String CONTENT_TYPE = "text/html; charset=ISO-8859-15";

public Prova() {
    super();
}

protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    response.setContentType(CONTENT_TYPE);
    PrintWriter out = response.getWriter();
    out.println("<html><body>");
    try {
        out.println("remoteUser: " +request.getRemoteUser());
        out.println("<br>ha prova1 come ruolo?:" +
request.isUserInRole("prova1"));
        out.println("<br>ha proveVarie come ruolo?: " +
request.isUserInRole("proveVarie"));
    } catch (Exception e) {
        out.println("<br>errore!");
    }
    out.println("</body></html>");
    out.close();
}

protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
}
}

```

Ora che abbiamo dato il codice della servlet, possiamo notare per prima cosa che, come avevamo detto, l'output generato è un documento HTML. Questo viene abilitato mediante le seguenti istruzioni:

```

private static final String CONTENT_TYPE = "text/html; charset=ISO-8859-15";
...
response.setContentType(CONTENT_TYPE);

```

La prima istruzione, infatti, specifica, all'interno di una costante, che il Content Type del documento deve essere di tipo HTML utilizzando il set di caratteri ISO 8859-15 mentre la seconda istruzione imposta l'output della servlet mediante il valore della costante del Content Type.

L'istruzione seguente, **String getRemoteUser()**, dell'interfaccia **javax.servlet.http.HttpServletRequest** serve a leggere, dalla request http, l'username e restituisce *null* solo nel caso in cui chi ha richiesto la risorsa non risulti essere autenticato correttamente.

request.getRemoteUser()

Ovviamente nel caso venga eseguita la servlet non protetta, il valore in uscita di tale metodo sarà sempre *null* mentre nel caso di quella protetta che viene richiamata dall'utente con i giusti diritti, verrà letto e stampato il suo username. Ricordiamo che se l'utente non possiede i diritti e tenta di accedere ad una risorsa protetta nessuna istruzione della servlet verrà eseguita e verrà mostrato, come già accennato, la pagina di errore HTTP 403.

Discorso analogo per la lettura dei ruoli dell'utente, unica cosa particolare è che, per motivi di sicurezza e privacy, non esiste un metodo Java che può ritornare tutti i ruoli di un utente e quindi è necessario andare a richiedere, di volta in volta, mediante il metodo **boolean isUserInRole(String)** appartenente sempre all'interfaccia **javax.servlet.http.HttpServletRequest**, se nella politica di sicurezza dell'application server è stato definito un particolare ruolo per l'utente in questione. Nello specifico andiamo a richiedere, come abbiamo accennato, se l'utente possiede i ruoli *prova1* e *proveVarie*; tale metodo, nel caso di corretta associazione userid/ruolo restituisce il valore boolean *true*, *false* in caso contrario:

request.isUserInRole("prova1")
request.isUserInRole("proveVarie")

Consideriamo il file web.xml che descrive le caratteristiche delle servlet utilizzate: in tale descrittore vengono definite le risorse che si vogliono proteggere, il tipo di autenticazione richiesta (Basic, Form, Client-Cert, ecc.), il realm name ossia il dominio applicativo associato al security constraint; segue poi

l'elenco dei ruoli ammessi per questa applicazione. Se configuriamo l'applicazione con autenticazione BASIC, all'atto della sua esecuzione viene presentata, da parte del browser web, una finestra in cui ci viene chiesto di immettere i dati di autenticazione, ossia la coppia [username, password]. Qui è riportato il descrittore web.xml delle servlet di test in cui possiamo riconoscere i nomi delle servlet, il realm, il role name ed anche il tipo di autenticazione utilizzata.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <display-name>Servlet di testing</display-name>
  <servlet>
    <description></description>
    <display-name>Prova</display-name>
    <servlet-name>Prova</servlet-name>
    <servlet-class>Prova</servlet-class>
  </servlet>
  <servlet>
    <description></description>
    <display-name>Prova2</display-name>
    <servlet-name>Prova2</servlet-name>
    <servlet-class>Prova2</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>
      Prova
    </servlet-name>
    <url-pattern>
      /Prova
    </url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>
      Prova2
    </servlet-name>
    <url-pattern>
```

```

        /Prova2
    </url-pattern>
</servlet-mapping>

<security-constraint>
    <web-resource-collection>
        <web-resource-name>
            proveVarie
        </web-resource-name>
        <url-pattern>
            /Prova
        </url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>
            proveVarie
        </role-name>
    </auth-constraint>
</security-constraint>

<login-config>
    <auth-method>
        BASIC
    </auth-method>
    <realm-name>
        Prove
    </realm-name>
</login-config>

<security-role>
    <description>
        esempio di autenticazione
    </description>
    <role-name>
        proveVarie
    </role-name>
</security-role>
</web-app>

```

Nel codice riportato abbiamo utilizzato l'autenticazione di tipo Basic (specificata mediante il tag **<auth-method>**) ed abbiamo anche definito il nome del Realm (tag **<realm-name>**) che è referenziato per autenticare le credenziali dell'utente per questa particolare applicazione web. Queste due cose sono definite all'interno dell'elemento opzionale **<login-config>** che serve a configurare la modalità con cui l'utente si autentica:

```
<login-config>  
  <auth-method>  
    BASIC  
  </auth-method>  
  <realm-name>  
    Prove  
  </realm-name>  
</login-config>
```

Altri tag importanti per la definizione della sicurezza sono quelli appartenenti all'elemento **<security-role>** in cui viene specificato il nome dei ruoli (nello specifico è *proveVarie*), ossia identifica il gruppo degli utenti che possono accedere all'applicazione (tag **<role-name>**):

```
<security-role>  
  <description>  
    esempio di autenticazione  
  </description>  
  <role-name>  
    proveVarie  
  </role-name>  
</security-role>
```

Aggiungiamo poi una piccola nota, relativa a **<role-name>**: all'interno del web.xml possiamo andare a specificare più di uno di questi elementi, definendo così tutti i possibili gruppi di utenti e per ciascun gruppo definire a quali parti dell'applicazione possono accedere. Se ad esempio abbiamo il gruppo studenti e professori, potremmo definire questi gruppi nel seguente modo:

```

<security-role>
  <description>
    Gruppo studenti
  </description>
  <role-name>
    studente
  </role-name>
</security-role>
<security-role>
  <description>
    Gruppo professori
  </description>
  <role-name>
    professore
  </role-name>
</security-role>

```

La configurazione dei vincoli, dell'accessibilità dei singoli gruppi di utenti, viene definita all'interno dell'elemento **<security-constraint>**: al suo interno troviamo i tag **<web-resource-name>** in cui andiamo a specificare i componenti dell'applicazione web a cui i vincoli di sicurezza devono essere applicati; nel nostro caso definiamo la sicurezza per l'URL **/Prova** (tag **<url-pattern>**). Per quanto invece riguarda l'associazione al gruppo, questo avviene mediante il tag **<role-name>** dell'elemento **<auth-constraint>**.

```

<security-constraint>
  <web-resource-collection>
    <web-resource-name>
      proveVarie
    </web-resource-name>
    <url-pattern>
      /Prova
    </url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>
      proveVarie
    </role-name>
  </auth-constraint>

```

```
</security-constraint>
```

Ricordiamo che possiamo andare a definire più elementi `<security-constraint>`, uno per ogni collezione di indirizzi che vogliamo proteggere. Tornando al caso degli studenti e professori, potremmo scrivere:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Professori</web-resource-name>
    <url-pattern>/professori/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>professore</role-name>
  </auth-constraint>
</security-constraint>
```

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Studenti</web-resource-name>
    <url-pattern>/studenti/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>studente</role-name>
    <role-name>professore</role-name>
  </auth-constraint>
</security-constraint>
```

2.2 Memorizzazioni delle credenziali degli utenti in Tomcat

Prima di iniziare la trattazione a discutere di come memorizzare le credenziali degli utenti in Tomcat, introduciamo la notazione `$CATALINA_HOME` con la quale intendiamo la cartella dove risiede Tomcat, ad esempio `/opt/jakarta/tomcat` (nel caso di installazione in ambiente Linux). Questa notazione ci è comoda dal momento che faremo molte volte riferimento alle sottocartelle di Tomcat, ad esempio `$CATALINA_HOME/conf` che è una cartella molto importante perché contiene i file di configurazione di questo application server.

Abbiamo detto che la fase di autenticazione è quella in cui un utente (o un agente web) si identifica sul server e solo dopo essere stato riconosciuto correttamente, potrà accedere alle risorse che ha richiesto. Detto questo dobbiamo però ancora rispondere ad una domanda importante ossia dove vengano memorizzate le credenziali dei singoli utenti. Tomcat mette a disposizione vari tipi di Realm, ad esempio:

- UserDatabaseRealm
- MemoryRealm
- JDBCRealm
- JNDIRealm

Di questi possibili realm nelle nostre prove ne abbiamo utilizzati solamente due che andiamo a presentare. Per specificare quale realm deve essere usato, dobbiamo inserire l'elemento **Realm** all'interno del file di un particolare file XML chiamato **server.xml**, presente in \$CATALINA_HOME/conf, specificando il realm mediante l'attributo *className* e fornendo al realm i parametri di configurazione. Un esempio di configurazione generica di un realm è rappresentato dal seguente frammento:

```
<Realm className="some.realm.implementation.className"  
    customAttribute1="some custom value"  
    customAttribute2="some other custom value"  
    <!-- etc... -->  
>
```

Il file server.xml è un file molto importante perché è il principale file di configurazione di Tomcat; qui di seguito riportiamo un tipico schema di questo file.

```
<Server port="8005" shutdown="SHUTDOWN" debug="0">  
    <Service name="Catalina">  
        <Connector port="8080"  
            maxThreads="150" minSpareThreads="25" maxSpareThreads="75"  
            enableLookups="false" redirectPort="8443" acceptCount="100"
```

```

debug="0" connectionTimeout="20000"
disableUploadTimeout="true" />

<Connector port="8009"
enableLookups="false" redirectPort="8443" debug="0"
protocol="AJP/1.3" />

<Engine name="Catalina" defaultHost="localhost" debug="0">
  <Realm
    className="org.apache.catalina.realm.UserDatabaseRealm"
    debug="0" resourceName="UserDatabase"/>

    <Host name="localhost" debug="0" appBase="webapps"
      unpackWARs="true" autoDeploy="true"
      xmlValidation="false" xmlNamespaceAware="false">
        </Host>
      </Engine>
    </Service>
  </Server>

```

Nel listato possiamo distinguere varie sezioni:

- l'elemento radice di tutto il documento è rappresentato dal tag **<Server>**;
- l'elemento **<Service>** rappresenta un gruppo di **Connector** associati ad un **Engine**;
- gli elementi **Connector** rappresentano l'interfaccia tra i client esterni che mandano richieste a (e ricevono da) un particolare **Service**;
- gli elementi **Container** rappresentano i componenti le cui funzioni sono quelle di processare richieste in ingresso e di creare le corrispondenti risposte. Un **Engine** gestisce tutte le richieste per un **Service**, un **Host** gestisce tutte le richieste per un particolare virtual host mentre un **Context** gestisce tutte le richieste per una specifica applicazione web;
- possiamo trovare anche elementi annidati all'interno di qualsiasi **Container** mentre ve ne sono altri che possono essere annidati all'interno del **Context**.

2.2.1 UserDatabaseRealm

Il primo realm che andiamo a presentare è l'**UserDatabaseRealm** che carica in memoria, da un file statico, le credenziali degli utenti: tali dati vivono finché Tomcat non è disattivato e vengono caricati in memoria unicamente all'avvio di questo application server. Il file di default nel quale vengono memorizzati i permessi in questo realm è **tomcat-users.xml** situato in `$CATALINA_HOME/conf`. In questo file andiamo ad impostare gli utenti a cui è consentito l'accesso alle risorse web ed è un semplice file XML la cui radice è rappresentata dal tag **tomcat-users**, i cui unici elementi sono **role** ed **user**. Ciascuno elemento *role* ha un singolo attributo **rolename** mentre ciascun elemento *user* ha tre elementi che sono **username**, **password** e **roles**. Il file `tomcat-user.xml` che viene fornito di default con Tomcat ha la seguente forma:

```
<tomcat-users>
  <role rolename="tomcat"/>
  <role rolename="role1"/>
  <user username="tomcat" password="tomcat" roles="tomcat" />
  <user username="role1" password="tomcat" roles="role1" />
  <user username="both" password="tomcat" roles="tomcat,role1" />
</tomcat-users>
```

Il significato di *username* e *password* è piuttosto ovvio mentre merita prendere in considerazione l'elemento *roles*. Un *role* è un gruppo di utenti per i quali le applicazioni web possono definire un determinato insieme di caratteristiche.

Nel nostro caso abbiamo creato un file `tomcat-user.xml` così fatto, in cui abbiamo definito un solo utente (*utente1*) associandolo al ruolo *proveVarie*.

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
  <role rolename="proveVarie"/>
  <user username="utente1" password="12345" roles="proveVarie"/>
</tomcat-users>
```

Risulta evidente che scrivere in chiaro i dati di accesso in un file di testo è quanto di più lontano dall'avere un sistema sicuro: chiunque abbia i diritti per accedere (e

modificare) alla cartella \$CATALINA_HOME/conf, potrebbe leggere le password contenute in questo file, modificarle o crearsi un proprio account per utilizzare risorse che non gli spettano. E' dunque chiaro che affidare la sicurezza di dati sensibili ad un sistema che legge i dati di accesso in questo modo è da scartare, benché può andare bene per progetti in cui il grado di segretezza può ritenersi trascurabile o irrilevante per l'applicazione stessa. Affinché Tomcat possa leggere i dati scritti all'interno di questo file, è necessario che sia presente, e che sia attiva, questa dichiarazione all'interno del file server.xml:

```
<Resource name="UserDatabase" auth="Container"
    type="org.apache.catalina.UserDatabase"
    description="User database that can be updated and saved"
    factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
    pathname="conf/tomcat-users.xml" />
```

Per completezza riportiamo il file server.xml completo in modo da mostrare dove inserire questo elemento di configurazione:

```
<?xml version="1.0" encoding="UTF-8"?>
<Server port="8005" shutdown="SHUTDOWN">
<!-- Comment these entries out to disable JMX MBeans support used for the
administration web application -->
    <Listener className="org.apache.catalina.core.AprLifecycleListener" />
    <Listener className="org.apache.catalina.mbeans.ServerLifecycleListener" />
    <Listener
className="org.apache.catalina.mbeans.GlobalResourcesLifecycleListener" />
    <Listener
className="org.apache.catalina.storeconfig.StoreConfigLifecycleListener"/>

<!-- Global JNDI resources -->
<GlobalNamingResources>
<!-- Test entry for demonstration purposes -->
    <Environment name="simpleValue" type="java.lang.Integer" value="30"/>
    <Resource name="UserDatabase" auth="Container"
        type="org.apache.catalina.UserDatabase"
        description="User database that can be updated and saved"
        factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
        pathname="conf/tomcat-users.xml" />
```

```

</GlobalNamingResources>

<!-- Define the Tomcat Stand-Alone Service -->
<Service name="Catalina">

<!-- Define a non-SSL HTTP/1.1 Connector on port 8080 -->
<Connector port="8080" maxHttpHeaderSize="8192"
    maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
    enableLookups="false" redirectPort="8443" acceptCount="100"
    connectionTimeout="20000" disableUploadTimeout="true"
proxyPort="80"/>

<!-- Define an AJP 1.3 Connector on port 8009 -->
<Connector port="8009" enableLookups="false" redirectPort="443"
protocol="AJP/1.3" debug="0"/>

<!-- Define an AJP 1.3 Connector on port 8449 -->
<Connector port="8449" scheme="https" enableLookups="false" protocol="AJP/
1.3" debug="0" secure="true"/>

<!-- Define the top level container in our container hierarchy -->
<Engine name="Catalina" defaultHost="localhost">

<!-- Because this Realm is here, an instance will be shared globally -->
<Realm className="org.apache.catalina.realm.UserDatabaseRealm"
    resourceName="UserDatabase"/>
<Host name="localhost" appBase="webapps"
    unpackWARs="true" autoDeploy="true"
    xmlValidation="false" xmlNamespaceAware="false">
</Host>
</Engine>
</Service>
</Server>

```

2.2.2 JNDIRealm

Il realm **JNDIRealm** fornisce un meccanismo mediante il quale Tomcat può accedere alle informazioni richieste per autenticare gli utenti, definire i loro ruoli di sicurezza attraverso un directory server, o altro servizio, attraverso API JNDI.

Le **Java Naming and Directory Interface (JNDI)** è un API per servizi di directory che consente ai client di effettuare il discover ed il lookup di dati e di oggetti attraverso nome. Come tutte le API Java che si interfacciano con sistemi host, anche JNDI è indipendente dall'implementazione sottostante. In aggiunta specifica un componente software chiamato **Service Provider Interface (SPI)** per supportare componenti rimpiazzabili (o riutilizzabili) che sono utilizzate, ad esempio, anche in **JDBC (Java Database Connectivity)**, **JAXP (Java API for XML Processing)** o **JBIC (Java Business Integration)**.

L'API JNDI è utilizzata da Java RMI e dalle API di J2EE per fare il look up di oggetti all'interno di una rete. Le API forniscono:

- un meccanismo per collegare un oggetto ad un nome;
- un'interfaccia di directory look-up che permette di fare query;
- un'interfaccia ad eventi che permette ai client di determinare quando una entry di una directory è stata modificata;
- estensioni LDAP per supportare le capacità aggiuntive del servizio LDAP.

Le SPI permettono di supportare praticamente ogni tipo di servizio di naming e di directory includendo:

- LDAP
- DNS
- NIS
- RMI
- servizi di naming di CORBA
- File System

JNDI organizza i propri nomi in gerarchia. Un nome può essere una qualunque stringa tipo *com.mydomain.ejb.MyBean* e può essere anche un oggetto che supporta l'interfaccia *Name*: la stringa è il più comune metodo per nominare un oggetto. Un nome è unito ad un oggetto nella directory memorizzando o l'oggetto od un riferimento all'oggetto nella directory dei servizi identificato mediante un nome.

Abbiamo detto che il JNDIRealm consente di connettere a vari servizi di directory, quale ad esempio LDAP, ed è proprio quest'ultimo che abbiamo connesso a Tomcat. In un server LDAP le informazioni non sono modificabili direttamente da tutti a meno, ovviamente, di essere a conoscenza della login (in particolare della password) dell'amministratore. Le password memorizzate all'interno di un database LDAP possono essere in forma cifrata o meno. Per creare i dati da memorizzare all'interno del server LDAP possiamo creare un file di testo, che noi abbiamo chiamato **datiLDAP**, mediante un qualsiasi editor di testi, ad esempio **gedit** oppure **vi**. Come server LDAP utilizziamo **OpenLDAP** che possiamo installare in questo modo:

- Per prima cosa dobbiamo installare il demone ldap (slapd), installando i pacchetti **slapd** e **ldap-utils**;
- Durante l'installazione, dobbiamo inserire sia il proprio dominio sia la password scelta per la directory dell'amministratore.

Questo conclude la fase di installazione, per avviare il demone LDAP dobbiamo eseguire il comando:

```
sudo /etc/init.d/slapd start
```

Per arrestare il demone LDAP:

```
sudo /etc/init.d/slapd stop
```

Ora che abbiamo illustrato brevemente come installare ed avviare OpenLDAP, possiamo riportare il contenuto del file dati usato per LDAP.

```
dn: ou=groups, dc=comune, dc=grosseto, dc=it  
objectClass: organizationalUnit  
ou: groups
```

```
dn: cn=sed, ou=groups, dc=comune, dc=grosseto, dc=it  
objectClass: groupOfUniqueNames  
cn: sed
```

uniqueMember: uid=utente1

dn: cn=proveVarie, ou=groups, dc=comune, dc=grosseto, dc=it

objectClass: groupOfUniqueNames

cn: proveVarie

uniqueMember: uid=utente1, ou=people,dc=comune, dc=grosseto, dc=it

dn: ou=people, dc=comune, dc=grosseto, dc=it

objectClass: organizationalUnit

ou: people

dn: uid=utente1, ou=people, dc=comune, dc=grosseto, dc=it

objectClass: inetOrgPerson

cn: utente1

sn: utente

uid: utente1

givenName: nome

displayName: nome cognome

userPassword: 12345

Tutte le informazioni contenute in un database LDAP vengono archiviate, come abbiamo già accennato, in una struttura ad albero. E' necessario determinare la struttura della directory (il **DIT** o **Directory Information Tree**). Per iniziare consideriamo un albero base con due nodi sopra la radice:

- nodo **People**: è dove i propri utenti vengono salvati;
- nodo **Groups**: è dove i propri gruppi vengono salvati.

E' necessario determinare quale deve essere la radice di LDAP. Di base l'albero viene determinato dal proprio dominio internet: se il dominio, ad esempio, fosse *example.com*, la radice sarebbe identificata come *dc=example, dc=com*; guardando il file dati sopra riportato, notiamo che la radice è identificata da: *dc=comune, dc=grosseto, dc=it*. Esiste un solo gruppo di utenti, identificato da *cn=sed, ou=groups, dc=comune, dc=grosseto, dc=it* a cui appartiene un unico utente *utente1* (identificato da *uid=utente1, ou=people, dc=comune, dc=grosseto, dc=it*) a cui è associato il ruolo *proveVarie* (*cn=proveVarie, ou=groups, dc=comune, dc=grosseto, dc=it*).

Per popolare il database LDAP possiamo utilizzare il seguente comando da console:

```
sudo ldapadd -x -D "cn=admin, dc=comune, dc=grosseto, dc=it" -w 1234 -v -f /home/sed/datiLDAP
```

dove **cn=admin** è l'username dell'utente dell'amministratore e **-w 1234** è la sua password; **-f** specifica, invece, il file da cui leggere i dati. Una volta creata la struttura possiamo riavviare il demone LDAP mediante il comando:

```
sudo /etc/init.d/slaped restart
```

Per attivare la connessione ad una database LDAP all'interno di Tomcat è necessario che sia attiva questa direttiva, da inserire nel server.xml, come abbiamo visto per l'UserDatabaseRealm.

```
<Realm className="org.apache.catalina.realm.JNDIRealm" debug="99"  
    connectionURL="ldap://localhost:389"  
    userPattern="uid={0}, ou=people, dc=comune, dc=grosseto, dc=it"  
    roleBase="ou=groups, dc=comune, dc=grosseto, dc=it"  
    roleName="cn"  
    roleSearch="(uniqueMember={0})" />
```

Nella precedente dichiarazione, il server LDAP si trova sulla stessa macchina di Tomcat (**connectionURL="ldap://localhost:389"**, dove **389** è la porta d'ascolto di LDAP). L'attributo **userPattern** descrive dove gli utenti sono memorizzati all'interno di LDAP, **roleBase** descrive dove sono memorizzati i ruoli, **roleName** è il nome dell'attributo che contiene il nome del ruolo (ossia il gruppo) mentre **roleSearch** definisce il filtro LDAP da soddisfare durante la ricerca dei ruoli.

2.3 Memorizzazioni delle credenziali degli utenti in JBoss

Le cose che abbiamo detto circa la memorizzazione delle credenziali in Tomcat, le possiamo riportare anche in JBoss. Dobbiamo dire che, in quest'ultimo, vanno configurate un po' di cose in più, ossia bisogna modificare il file XML principale di configurazione, chiamato **login-config.xml**, in cui specificare il modulo login

incaricato di gestire i dati provenienti dall'autenticatore, impostato nel descrittore web.xml. Introduciamo, prima di continuare, la notazione **\$JBOSS_HOME** intendendo, come abbiamo fatto con Tomcat, la directory dove è installato JBoss. A differenza di Tomcat, JBoss prevede più modalità di esecuzione, possiamo cioè decidere di avviare l'application server con una configurazione minimale, una di default oppure full, quindi più pesante. All'interno della cartella **\$JBOSS_HOME/server** troviamo tre sottocartelle chiamate, rispettivamente, **minimal**, **default** e **all**; all'interno di ciascuna delle tre sottocartelle c'è un'ulteriore cartella chiamata **conf** in cui troviamo il file di configurazione login-config.xml.

Per attivare una delle tre possibili modalità di esecuzione di JBoss, da terminale va lanciato il comando, intendendo con **nomeserver** uno dei tre valori possibili dei nomi delle sotto cartelle viste in precedenza (*minimal*, *default* e *all*). Se il parametro **-c** viene omissso, la configurazione predefinita è quella *default*:

```
$JBOSS_HOME/bin/run.sh -c nomeserver
```

Possiamo spendere qualche discorso parlando dell'architettura di JBoss: questo application server è, infatti, costituito da un server **JMX MBean** che monitora e gestisce le risorse, da un micro kernel e da un set di componenti di servizio pluggabili chiamati, appunto MBean. Questo set di componenti, permette di assemblare facilmente differenti configurazioni che si adattano in modo molto flessibile ai requisiti di utilizzo; non è necessario eseguire un server pesante, monolitico, tutte le volte, è possibile rimuovere componenti non necessari (riducendo così il tempo di avvio del server) o integrarne con altri nuovi mediante la scrittura di nuovi MBeans. Prima però di procedere oltre dobbiamo premettere una cosa molto importante e cioè, affinché si possa effettuare una qualsiasi autenticazione di applicazioni installate all'interno di JBoss, è necessario creare un file XML di configurazione, chiamato **jboss-web.xml**, da aggiungere nella directory **WEB-INF** del file WAR (o EAR) che contiene l'applicazione Web che andiamo ad installare all'interno di JBoss e che vogliamo proteggere mediante un meccanismo di autenticazione. Sebbene sia possibile specificare tutto (o quasi) all'interno del file web.xml, ci sono casi in cui risulta necessario aggiungere

settaggi specifici dell'application server JBoss. Di seguito riportiamo lo schema DTD (Document Type Definition) di jboss-web.xml (figura 2.1).

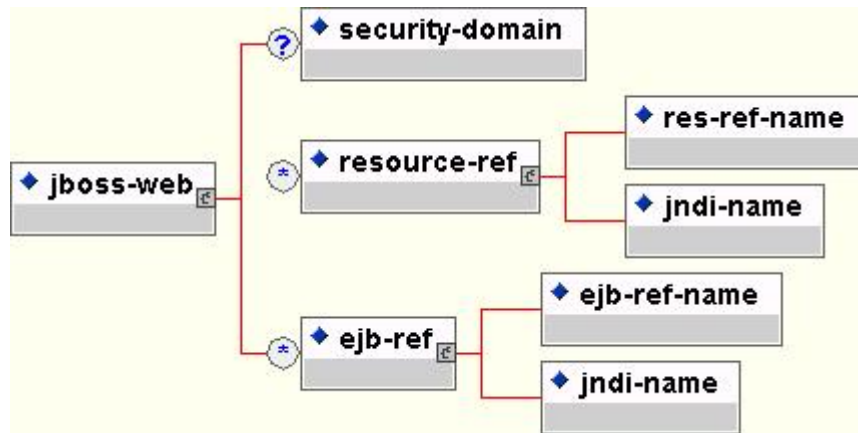


Figura 2.1: Schema DTD jboss-web.xml

Un tag importante, senza il quale non si realizza correttamente l'autenticazione dell'applicazione, è `<security-domain>`, mediante il quale definiamo il nome JNDI dell'implementazione del manager della sicurezza che dovrebbe essere usato per effettuare l'autenticazione e l'autorizzazione dei client web. Nel nostro caso, questo ulteriore file è così formato, in cui indichiamo il role name, che nel nostro caso ricordiamo è *proveVarie*, dell'applicazione Web da proteggere:

```

<?xml version="1.0" encoding="UTF-8"?>
<jboss-web>
  <security-domain>
    java:/jaas/proveVarie
  </security-domain>
</jboss-web>

```

Fatta questa modifica possiamo procedere ad illustrare due moduli login presenti all'interno di JBoss che permettono di fare l'analogo di quelli visti con Tomcat e che sono:

- UsersRolesLoginModule
- LdapLoginModule

2.3.1 UsersRolesLoginModule

Questo modulo login è l'analogo del UserDatabaseRealm di Tomcat ma per configurarlo abbiamo bisogno di definire, all'interno di due file di proprietà, ossia file di testo, la mappatura username/password e quella username/ruoli. Nel nostro caso abbiamo creato, mediante un editor di testi, due file chiamati, rispettivamente, **proveVarie-users.properties** e **proveVarie-roles.properties** all'interno della sottocartella conf della configurazione server di JBoss che vogliamo avviare. Nel nostro caso, noi lavoriamo sempre con la configurazione di default.

Un generico file di proprietà che definisce la mappatura tra username e password ha la seguente struttura:

```
username1=password1
username2=password2
...
```

Quello relativo alla mappatura tra username e ruoli, invece, ha la seguente struttura:

```
username1=role1,role2,...
username2=role1,role3,...
...
```

Prima di mostrare la configurazione che abbiamo utilizzato con il modulo login UsersRolesLoginModule, conviene mostrare la struttura generica del file di configurazione login-config.xml:

```
<?xml version='1.0'?>
<!DOCTYPE policy PUBLIC
  "-//JBoss//DTD JBOSS Security Config 3.0//EN"
  "http://www.jboss.org/j2ee/dtd/security_config.dtd">

<policy>
<application-policy name="security-domain-name">
```

```

<authentication>
  <login-module code="login.module1.class.name" flag="control_flag">
    <module-option name = "option1-name">option1-value</module-option>
    <module-option name = "option2-name">option2-value</module-option>
    ...
  </login-module>

  <login-module code="login.module2.class.name" flag="control_flag">
    ...
  </login-module>
  ...
</authentication>
</application-policy>
</policy>

```

Per ogni valore dei ruoli che vogliamo gestire all'interno di JBoss, è necessario definire una nuova sezione identificata dall'elemento **<application-policy>**, al cui interno andiamo a definire tutti i login module che vogliamo che si occupino del trattamento dei dati per quel ruolo. Nel nostro caso abbiamo configurato così l'elemento application-policy:

```

<application-policy name = "proveVarie">
  <authentication>
    <login-module
      code="org.jboss.security.auth.spi.UsersRolesLoginModule" flag =
      "required">
      <module-option name="usersProperties">
        proveVarie-users.properties
      </module-option>
      <module-option name="rolesProperties">
        proveVarie-roles.properties
      </module-option>
    </login-module>
  </authentication>
</application-policy>

```

Possiamo notare che i login module vengono definiti mediante il tag `<login-module>` in cui dobbiamo dichiarare il nome della classe da utilizzare mediante

l'attributo **code** (identificato da **<login-module code="org.jboss.security.auth.spi.UsersRolesLoginModule" flag = "required">**). All'interno di questo elemento dobbiamo elencare le opzioni relative al login module, nello specifico, quale file deve essere usato per la mappatura username/password (valore del tag **<module-option name="usersProperties">**) e quale per quella relativa ad username/ruoli (valore del tag **<module-option name="rolesProperties">**).

Riportiamo qui di seguito il contenuto file proprietà proveVarie-users.properties, in cui, ricordiamo, troviamo l'associazione dell'utente (nel nostro caso del solo utente chiamato *utente1*) con la sua password:

utente1=12345

L'altro file di interesse è proveVarie-roles.properties, in cui troviamo invece l'associazione tra l'utente *utente1* ed il ruolo *proveVarie*:

utente1=proveVarie

2.3.2 LdapLoginModule

Anche JBoss permette la connessione ad un database LDAP e questo avviene utilizzando il modulo login LdapLoginModule. Come abbiamo visto nel paragrafo precedente, per utilizzare questo modulo dobbiamo attivarlo mediante apposita dichiarazione nel file login-config.xml. La seguente configurazione, rispetto a quella vista per Tomcat, è molto più articolata perché prevede molti più parametri da gestire. Il server LDAP, facciamo notare, si trova sulla stessa macchina dove è installato JBoss.

```
<application-policy name = "proveVarie">
  <authentication>
    <login-module code =
      "org.jboss.security.auth.spi.LdapLoginModule" flag = "required">
      <module-option name = "java.naming.factory.initial">
        com.sun.jndi.ldap.LdapCtxFactory
    </module-option>
```

```

        <module-option name = "java.naming.provider.url">
            ldap://localhost:389
        </module-option>
        <module-option name =
"java.naming.security.authentication">
            simple
        </module-option>
        <module-option name = "principalDNPrefix">
            uid=
        </module-option>
        <module-option name = "principalDNSuffix">
            ,ou=people, dc=comune, dc=grosseto, dc=it
        </module-option>
        <module-option name = "rolesCtxDN">
            ou=groups, dc=comune, dc=grosseto, dc=it
        </module-option>
        <module-option name = "uidAttributeID">
            uniqueMember
        </module-option>
        <module-option name = "matchOnUserDN">
            true
        </module-option>
        <module-option name = "roleAttributeID">
            cn
        </module-option>
        <module-option name = "roleAttributesDN">
            false
        </module-option>
    </login-module>
</authentication>
</application-policy>

```

E' bene spiegare il significato delle proprietà definite all'interno del listato precedente:

- **principalDNPrefix** e **principalDNSuffix** sono il prefisso ed il suffisso da aggiungere all'username per formare il DN dell'utente;
- **rolesCtxDN** è un DN fisso utilizzato per la ricerca dei ruoli dell'utente. Non va inteso come il DN di dove sono i ruoli attuali, bensì come il DN di

dove sono gli oggetti che contengono i ruoli degli utenti (ad esempio, nel caso di Active Directory è il DN di dove è l'account dell'utente);

- **uidAttributeID** è il tipo di attributo nell'oggetto ruolo che rappresenta l'identificativo dell'utente. E' generalmente utilizzato per localizzare i ruoli degli utenti mediante una query LDAP;
- **matchOnUserDN** è un flag che specifica se la ricerca deve essere esatta su tutto il DN dell'utente. Se assume valore *false*, l'username è utilizzato come valore per il matching; in caso di *true*, invece, **userDN** è utilizzato come valore del matching;
- **roleAttributeIsDN** è un flag che indica se il ruolo dell'utente contiene il DN completo dell'oggetto ruolo oppure se contiene il nome del ruolo. Se fissato a *false*, il nome del ruolo è preso dal valore dell'attributo ruolo dell'utente, altrimenti l'attributo ruolo rappresenta il DN di un oggetto ruolo;
- **roleAttributeID** il nome dell'attributo ruolo del context che corrisponde al nome del ruolo. Se **roleAttributeIsDN** è settato a *true*, questa proprietà è il DN del context da cercare per l'attributo **roleNameAttributeID**. Se l'attributo **roleAttributeIsDN** assume valore *false*, questa proprietà è il nome dell'attributo per il nome del ruolo.

2.4 OPG: primo approccio

Nei precedenti paragrafi abbiamo visto i modi con cui abbiamo memorizzato le credenziali degli utenti sia in Tomcat sia in JBoss ma ora è venuto il momento di vedere più da vicino la struttura di Open Portal Guard per poi, nei prossimi paragrafi, affrontare la trattazione della realizzazione del nuovo autenticatore per entrambi gli application server. La struttura dell'architettura dell'OPG è piuttosto complessa perché entrano in gioco diverse componenti quali:

- un application server che può essere Tomcat o JBoss;
- un server LDAP;
- un Apache HTTP Server su cui abbiamo installato vari moduli, tra i quali citiamo **mod_proxy** (per effettuare il reverse proxy), **mod_python** (per

integrare il linguaggio Python all'interno del server Apache) e **mod_ssl** (per il supporto crittografico mediante protocollo SSL/TLS);

- un lettore di smart card compatibile con le schede CNS e CIE.

Il punto centrale attorno a cui ruota un po' tutto il sistema è Apache HTTP Server che sia riceve le richieste di accesso alle risorse protette sia provvede, mediante i vari moduli installati, un programma scritto dal SED del Comune di Grosseto e la lettura della smart card inserita in un apposito lettore di smart card, a generare due variabili nella request http che vengono inviate all'application server che, a sua volta, provvederà a convalidare o meno i diritti dell'utente. Dal momento che OPG è una struttura modulare, per semplicità, omettiamo l'estrazione dei dati da smart card, il look-up all'interno del server LDAP, così come lo scambio dei certificati SSL durante la connessione; prendiamo in considerazione unicamente il caso in cui un utente richiede una certa risorsa protetta ed Apache ha già generato due variabili relative a quanto è stato letto dalla smart card e dal server LDAP. Le variabili che vengono generate nell'elaborazione di Apache sono chiamate **OPG_USERNAME** ed **OPG_USERROLES** e sono quelle che, nei prossimi paragrafi, andremo a gestire all'interno dell'autenticatore: la prima variabile contiene l'username (o il codice fiscale) dell'utente mentre la seconda specifica la lista dei ruoli che sono associati all'utente che si vuole autenticare.

2.4.1 Configurazione di Apache HTTP Server

In questo paragrafo andiamo ad illustrare come configurare Apache affinché trasmetta le due variabili, introdotte prima, all'application server. Un punto molto importante è quello relativo a come far comunicare Apache con Tomcat (e poi JBoss). Durante il nostro studio ci siamo serviti di due moduli che abbiamo installato in Apache e sono:

- mod_proxy
- mod_jk

Il primo modulo, **mod_proxy** [6], implementa un proxy/gateway per Apache e può essere connesso ad altri moduli proxy per protocolli quali FTP, CONNECT (per SSL), HTTP/0.9, HTTP/1.0 e HTTP/1.1;

L'altro modulo, **mod_jk** [7], è un connettore molto utilizzato per connettere container JSP, quali Tomcat con Apache, Netscape, iPlanet, SunOne ed anche IIS, e che fa uso del protocollo **Apache JServe Protocol (AJP)**. Questo protocollo consente, infatti, di trasferire richieste provenienti da un web server ad un application server. AJP è tipicamente utilizzato per bilanciare il carico quando uno o più web server front-end devono interfacciarsi con uno o più application server; le sessioni, inoltre, sono indirizzate verso il corretto application server utilizzando un meccanismo di routing.

2.4.1.1 mod_proxy

Il primo modulo che andiamo ad utilizzare per trasferire richieste da Apache ad un application server è mod_proxy: vogliamo che le richieste provenienti dalla porta 80 utilizzata per http (o la 443 nel caso di https) vengano reindirizzate all'application server, che risiede dietro il web server, verso la porta 8080 (o 8443 nel caso di https). Iniziamo col modificare un file di configurazione di Apache chiamato **default** che si trova, nel nostro caso, all'interno della cartella **/etc/apache2/sites-available**. Elenchiamo qui di seguito il contenuto base di tale file per poi mostrare quanto abbiamo aggiunto:

```
#NameVirtualHost
```

```
<VirtualHost *:80>
```

```
    ServerAdmin webmaster@localhost
```

```
    DocumentRoot /var/www/
```

```
    <Directory />
```

```
        Options FollowSymLinks
```

```
        AllowOverride None
```

```
    </Directory>
```

```
    <Directory /var/www/>
```

```
        Options Indexes FollowSymLinks MultiViews
```

```
        AllowOverride None
```

```
        Order allow,deny
```

```

        allow from all
        # This directive allows us to have apache2's default start page
        # in /apache2-default/, but still have / go to the right place
        #RedirectMatch ^/$ /apache2-default/
</Directory>
ScriptAlias /cgi-bin/ /usr/lib/cgi-bin/
<Directory "/usr/lib/cgi-bin">
    AllowOverride None
    Options +ExecCGI -MultiViews +SymLinksIfOwnerMatch
    Order allow,deny
    Allow from all
</Directory>

ErrorLog /var/log/apache2/error.log
# Possible values include: debug, info, notice, warn, error, crit,
# alert, emerg.
LogLevel warn
CustomLog /var/log/apache2/access.log combined
ServerSignature On

Alias /doc/ "/usr/share/doc/"
<Directory "/usr/share/doc/">
    Options Indexes MultiViews FollowSymLinks
    AllowOverride None
    Order deny,allow
    Deny from all
    Allow from 127.0.0.0/255.0.0.0 ::1/128
</Directory>
<Location ...>
    ...
</Location>
</VirtualHost>

```

Mostriamo anche il pezzo relativo che abbiamo utilizzato per effettuare la comunicazione mediante `mod_proxy`, inserendola al posto della direttiva `<Location>` che permette di limitare la dichiarazione di quanto è rinchiuso e specificato al suo interno unicamente all'URL specificato. Il loro utilizzo è simile a quello della direttiva `<Directory>` e non dovrebbe essere utilizzata per controllare l'accesso a locazioni fisiche nel file system (cosa di cui si occupa la direttiva `<Directory>`). Nel caso illustrato, andiamo unicamente a considerare una

connessione http non protetta, rimandando ad altri paragrafi la trattazione della connessione https:

```
<Location /testApp2/>  
    Order allow,deny  
    Allow from all  
    ProxyPass http://localhost:8080/testApp/  
    ProxyPassReverse http://localhost:8080/testApp/  
    RequestHeader set OPG_USERNAME utente1  
    RequestHeader set OPG_USERROLES [prova1,prova2,prova3,proveVarie]  
</Location>
```

Vediamo che abbiamo creato una sezione <Location> relativa all'URL **/testApp2/** (<Location /testApp2/>) ma dobbiamo subito dire che tale URL non esiste fisicamente né all'interno di Apache né nell'application server, è un indirizzo virtuale che possiamo spiegare in questo modo: quando andremo a richiamare, mediante un browser web, l'URL **http://localhost/testApp2/**, la nostra richiesta verrà elaborata da Apache che, trovando il match sull'URL indicato in <Location>, la elaborerà e la inoltrerà verso **http://localhost:8080/testApp/** ossia verso un'applicazione che giace all'interno dell'application server e quindi verso un indirizzo reale, non fittizio come l'altro. All'interno della direttiva <Location> che abbiamo poco fa indicato, troviamo diverse impostazioni:

- **Allow**: specifica gli host abilitati a vedere l'URL mappato, nel nostro caso, tutti sono abilitati;
- **Order**: indica l'ordine con il quale vengono elaborati gli host specificati dalle direttive Allow e Deny;
- **ProxyPass**: serve a mappare l'URL remoto in richieste locali;
- **ProxyPassReverse**: permette di aggiustare l'URL in una response header inviata dal server su cui agisce il **reverse proxy**. Il reverse proxy è un **proxy server (figura 2.2)** installato in prossimità di uno o più server ed è tipicamente utilizzato davanti ad un web server. Tutte le connessioni che arrivano ad uno dei web server, sono indirizzate attraverso il server proxy che può o gestire lui stesso le richieste oppure inoltrarle interamente o parzialmente verso i web server principali;

- **RequestHeader**: consente di modificare l'intestazione della richiesta HTTP prima che questa venga recuperata dal content handler dell'applicazione che ha richiesto tale risorsa.

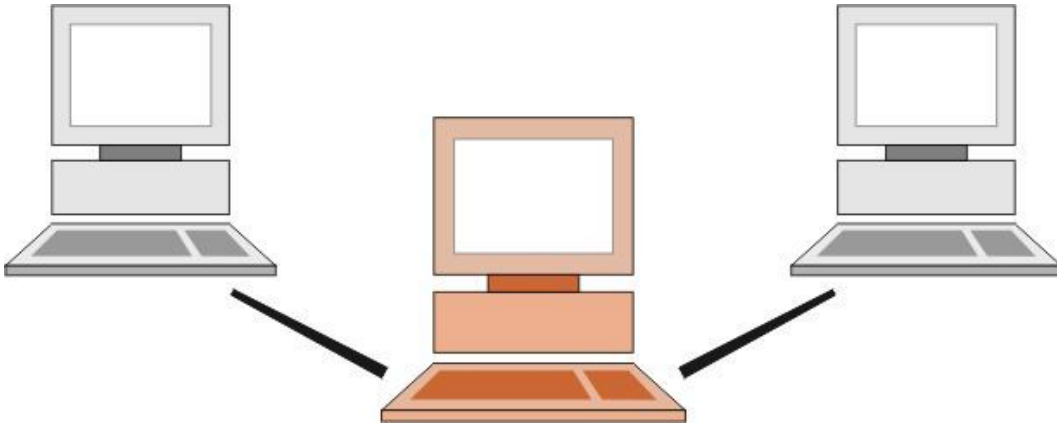


Figura 2.2: Rappresentazione di Proxy Server in cui il computer nel mezzo agisce come proxy server tra gli altri due.

Mediante la direttiva RequestHeader andiamo ad impostare, in modo statico ma solo ai fini delle prove, il valore delle variabili OPG_USERNAME ed OPG_USERROLES, rispettivamente ad *utente1* ed *[prova1,prova2,prova3,proveVarie]*. A regime, i valori di queste variabili vengono generati in automatico ma, per motivi di praticità, li abbiamo fissati noi a valori noti. E' importante notare il pattern della stringa dei ruoli dal momento che deve avere i nomi dei ruoli separati da virgola: la stringa così formata, deve iniziare con una parentesi quadra aperta e terminare con una chiusa (esempio **[ruolo1,ruolo2,...,ruoloN]**).

Ogni volta che modifichiamo la configurazione di Apache, come in questo caso, dobbiamo provvedere a riavviare il web server mediante il comando, da terminale:

```
sudo /etc/init.d/apache2 restart
```

2.4.1.2 mod_jk

Leggermente più complessa è la configurazione del modulo **mod_jk** in quanto dobbiamo interagire con ben quattro file di configurazione (tre per Apache ed uno per Tomcat). Vediamo innanzi tutto la parte relativa ad Apache per poi affrontare la configurazione di Tomcat.

2.4.1.2.1 Configurazione di Apache

Come nel caso di `mod_proxy`, dobbiamo definire una sezione `<Location>` nel file `default`, in cui però scriviamo unicamente i comandi per la creazione delle due variabili passate alla request http, delegando ad altri file la gestione della connessione tra Apache e Tomcat:

```
<Location /testApp2/>  
    # variabili nome e ruoli  
    RequestHeader set OPG_USERNAME utente1  
    RequestHeader set OPG_USERROLES [prova1,prova2,prova3,proveVarie]  
</Location>
```

La configurazione del connettore `mod_jk` avviene mediante il file di testo **workers.properties** che abbiamo creato in `/etc/apache2` che andiamo ora a riportare; con il termine **worker** intendiamo gli enti che in `mod_jk` si occupano effettivamente della connessione tra Apache e Tomcat:

```
workers.tomcat_home=$CATALINA_HOME  
workers.java_home=$JAVA_HOME  
ps=  
worker.ajp13.port=8009  
worker.ajp13.host=localhost  
worker.ajp13.type=ajp13
```

In questo file appena mostrato, abbiamo definito quale siano le directory dove risiede Tomcat (direttiva **workers.tomcat_home**) e la Java Virtual Machine da utilizzare (**workers.java_home**).

Con la direttiva **ps**, indichiamo quale è il separatore di ambiente, con **ps=/** facciamo riferimento a sistemi Unix mentre con **ps=** a sistemi Windows.

Con **worker.ajp13.port** indichiamo il valore della porta da usare per comunicare con l'application server. Se viene specificato il valore *8009* stiamo utilizzando Tomcat, *8019* invece è usata per Jetty.

Mediante **worker.ajp13.host** indichiamo l'host dove il worker Tomcat è in ascolto.

L'ultima direttiva, **worker.ajp13.type**, specifica il tipo di worker da utilizzare. Valori possibili sono *ajp12*, *ajp13*, *jni* e *lb*.

Affinché `mod_jk` venga utilizzato da Apache, dobbiamo dire a quest'ultimo dove si trovano i file di configurazione del connettore e per far questo, utilizzando un editor di testo, andiamo a scrivere all'interno del file di configurazione **httpd.conf** in `/etc/apache2/`, le seguenti direttive:

```
JkWorkersFile /etc/apache2/workers.properties  
JkLogFile /etc/apache2/mod_jk.log  
JkLogLevel info  
JkLogStampFormat "[%a %b %d %H:%M:%S %Y] "  
JkAutoAlias /etc/apache2/config-store  
JkMount /testApp/* ajp13
```

Mediante **JkWorkersFile** indichiamo ad Apache dove trovare il file di configurazione, `workers.properties`, di `mod_jk`.

Con la direttiva **JkLogFile** specifichiamo quale deve essere il file da usare come file di log.

Il livello di log viene settato mediante la direttiva **JkLogLevel**: nello specifico viene detto di visualizzare le attività standard (default) di `mod_jk`.

Il formato della data ed ora riportata nel log viene configurata mediante **JkLogStampFormat**.

Con **JkAutoAlias** viene settato un alias per la context della directory **webapps** di Tomcat all'interno dello spazio dei documenti di Apache.

L'ultima direttiva, **JkMount**, specifica un punto di mount tra il context e il worker di Tomcat.

2.4.1.2.2 Configurazione di Tomcat

In questo nuovo paragrafo andiamo a mostrare cosa c'è da modificare in Tomcat, precisamente andiamo a modificare nuovamente il file di configurazione server.xml, inserendo dopo il tag di chiusura `</Host>`, la seguente direttiva:

```
<Listener className="org.apache.jk.config.apacheConfig"
  modJk="/usr/lib/apache2/modules/mod_jk.so"
  workersConfig="/etc/apache2/workers.properties" />
```

Nella precedente direttiva andiamo a dire a Tomcat di caricare la classe relativa al modulo mod_jk (attributo **className="org.apache.jk.config.apacheConfig"**) passandole sia il path dove risiede fisicamente la plug-in mod_jk (attributo **modJk**) sia quale è il file di configurazione da utilizzare (attributo **workersConfig**).

2.4.1.2.3 Step finale per l'attivazione

Ora che abbiamo visto come configurare sia Apache che Tomcat per funzionare con mod_jk, il prossimo ed ultimo passo è quello di eseguire il seguente comando per attivare il modulo mod_jk:

```
sudo a2enmod jk
```

Per rendere effettive tutte le modifiche è necessario, come sempre, riavviare sia Apache che Tomcat.

2.4.2 BasicAuthenticator

Fino ad ora abbiamo introdotto le modalità di interfacciamento tra Apache e Tomcat e quali sono le variabili che dobbiamo leggere da dentro l'application server. Il primo passo, per arrivare alla costruzione di un nuovo authenticator è rappresentato dallo studiare quelli che sono già presenti, tenendoli come base di

partenza. Ricordiamo che Tomcat, così come JBoss, ne mette a disposizione diversi che qua elenchiamo:

- BasicAuthenticator
- FormAuthenticator
- SSLAuthenticator
- DigestAuthenticator
- NonLoginAuthenticator (non è un vero e proprio autenticatore, non viene mai utilizzato in pratica e la sua funzione non è quella di autenticare)

Di tutti questi forniti dall'application server, noi andiamo ora a presentare quello relativo alla Basic Authentication, il cui sorgente può essere scaricato da <http://tomcat.apache.org/download-55.cgi> insieme a tutti i sorgenti di Tomcat. Questo tipo di autenticazione rappresenta un processo nel quale il server di autenticazione richiede, al client, la coppia [username, password]; tale coppia di valori viene controllata dal server nel suo database locale e se tale coppia è presente, l'utente viene autenticato ed autorizzato ad accedere alle risorse da lui richieste. Questo processo di autenticazione viene ripetuto ogni volta che un utente richiede di utilizzare una diversa risorsa. Ci possiamo rendere conto che questo procedimento risulta essere noioso per un utente che si trova costretto a ricordare (ed ad inserire) la propria coppia di [username, password] ogni volta voglia accedere a risorse differenti.

Terminata questa breve introduzione sull'autenticazione Basic, torniamo al file di interesse ossia al sorgente del Basic Authenticator che si chiama **BasicAuthenticator.java** situato all'interno del percorso **\$CATALINA_SOURCE/container/catalina/src/share/org/apache/catalina/authenticator**, dove con **\$CATALINA_SOURCE** intendiamo la cartella in cui sono stati estratti i sorgenti di Tomcat. Qui di seguito riportiamo il contenuto di questo sorgente Java per poi illustrarne le caratteristiche e le problematiche che abbiamo incontrato durante il suo studio.

```
package org.apache.catalina.authenticator;  
import java.io.IOException;  
import java.security.Principal;
```

```

import javax.servlet.http.HttpServletResponse;
import org.apache.catalina.connector.Request;
import org.apache.catalina.connector.Response;
import org.apache.catalina.deploy.LoginConfig;
import org.apache.catalina.util.Base64;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.tomcat.util.buf.ByteChunk;
import org.apache.tomcat.util.buf.CharChunk;
import org.apache.tomcat.util.buf.MessageBytes;

public class BasicAuthenticator
    extends AuthenticatorBase {
    private static Log log = LogFactory.getLog(BasicAuthenticator.class);

    public static final byte[] AUTHENTICATE_BYTES = {
        (byte) 'W', (byte) 'W', (byte) 'W', (byte) '-',
        (byte) 'A', (byte) 'u', (byte) 't', (byte) 'h', (byte) 'e', (byte) 'n',
        (byte) 't', (byte) 'i', (byte) 'c', (byte) 'a', (byte) 't', (byte) 'e'
    };

    protected static final String info =
        "org.apache.catalina.authenticator.BasicAuthenticator/1.0";

    public String getInfo() {
        return (info);
    }

    public boolean authenticate(Request request, Response response, LoginConfig
    config)
        throws IOException {
        Principal principal = request.getUserPrincipal();
        String ssold = (String) request.getNote(Constants.REQ_SSOLD_NOTE);
        if (principal != null) {
            if (log.isDebugEnabled())
                log.debug("Already authenticated '" + principal.getName() +
                "'");
            if (ssold != null)
                associate(ssold, request.getSessionInternal(true));
        }
        return (true);
    }
}

```

```

if (ssold != null) {
    if (log.isDebugEnabled())
        log.debug("SSO Id " + ssold + " set; attempting " +
            "reauthentication");
    if (reauthenticateFromSSO(ssold, request))
        return true;
}
String username = null;
String password = null;

MessageBytes authorization =
    request.getCoyoteRequest().getMimeHeaders().getValue("authorizat
ion");
if (authorization != null) {
    authorization.toBytes();
    ByteChunk authorizationBC = authorization.getByteChunk();
    if (authorizationBC.startsWithIgnoreCase("basic ", 0)) {
        authorizationBC.setOffset(authorizationBC.getOffset() + 6);
        CharChunk authorizationCC = authorization.getCharChunk();
        Base64.decode(authorizationBC, authorizationCC);
        int colon = authorizationCC.indexOf(':');
        if (colon < 0) {
            username = authorizationCC.toString();
        } else {
            char[] buf = authorizationCC.getBuffer();
            username = new String(buf, 0, colon);
            password = new String(buf, colon + 1,
                authorizationCC.getEnd() - colon - 1);
        }
        authorizationBC.setOffset(authorizationBC.getOffset() - 6);
    }

    principal = context.getRealm().authenticate(username, password);
    if (principal != null) {
        register(request, response, principal,
            Constants.BASIC_METHOD,
            username, password);
        return (true);
    }
}
}

```

```

MessageBytes authenticate =
    response.getCoyoteResponse().getMimeHeaders()
        .addValue(AUTHENTICATE_BYTES,
0,
AUTHENTICATE_BYTES.length);
CharChunk authenticateCC = authenticate.getCharChunk();
authenticateCC.append("Basic realm=\"");
if (config.getRealmName() == null) {
    authenticateCC.append(request.getServerName());
    authenticateCC.append(':');
    authenticateCC.append(Integer.toString(request.getServerPort()));
} else {
    authenticateCC.append(config.getRealmName());
}
authenticateCC.append("\");
authenticate.toChars();
response.sendError(HttpServletResponse.SC_UNAUTHORIZED);
return (false);
}
}

```

La prima problematica che abbiamo incontrato, di grande importanza, è stata l'individuazione dell'istruzione responsabile della visualizzazione, all'interno del browser web, del pop-up utilizzato per richiedere all'utente la sua coppia di dati [username, password] (figura 2.3).

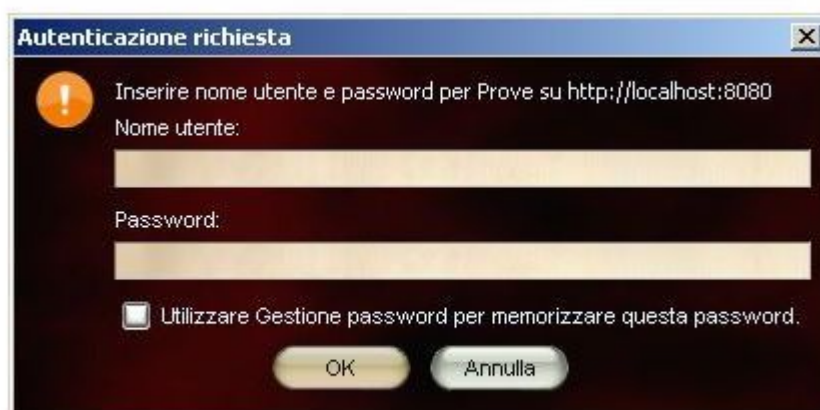


Figura 2.3: Pop-up di richiesta immissione [username, password] generato utilizzando l'autenticazione di tipo BASIC

Ad una prima osservazione del codice ci rendiamo conto che non è così facile individuare quale sia l'istruzione responsabile della visualizzazione del pop-up della figura precedente; solo un'analisi step-by-step del sorgente ci ha portato all'individuazione di una serie di istruzioni che concorrono alla creazione della finestra di input, benché quella principale è rintracciabile nella seguente:

```
response.sendError(HttpServletResponse.SC_UNAUTHORIZED);
```

Tale istruzione svolge una duplice funzione perché, oltre a richiedere i dati all'utente, genera la visualizzazione di una pagina di errore: se, infatti, l'utente dovesse scegliere di fare click sul bottone con la scritta "Annulla" perché o non conosce i dati d'accesso o per qualsiasi altro motivo, viene generata una pagina HTML che avvisa che si è verificato un errore di accesso non autorizzato, contraddistinto da un errore **HTTP 401** (la costante **HttpServletResponse.SC_UNAUTHORIZED** ha, appunto, valore *401*). Una volta che l'utente immette i dati, viene letto il valore assunto dal campo **authorization** della request http e il risultato viene memorizzato come un array di byte chiamato **MessageBytes** appartenente alla classe **org.apache.tomcat.util.buf.MessageBytes** che ha il compito di rappresentare tutti gli elementi dei messaggi della request e della response http:

```
MessageBytes authorization =
```

```
request.getCoyoteRequest().getMimeHeaders().getValue("authorization");
```

Una volta in possesso del valore di authorization, l'autenticatore inizia il vero processo di elaborazione che porta alla ricostruzione della coppia [username, password] in quanto i dati immessi vengono codificati internamente dal pop-up. Vediamo di rendere chiaro il processo con un esempio, provando ad inserire, nel pop-up, questi dati: *[username, password] = [utente1, changeit]*. La prima volta che l'autenticatore viene invocato, il valore assunto dalla variabile authorization risulta essere nullo e questo porta all'esecuzione del ramo in cui viene generata una sfida, o challenge, rivolta all'utente mediante la quale gli vengono chiesti, appunto, i propri dati segreti. Per prima cosa viene inserito un nuovo campo, etichettato come **WWW-Authenticate**, all'interno della response http ([8], [9]);

nel nostro caso, il valore dell'etichetta è scritto all'interno del byte[] **AUTHENTICATE_BYTES**.

MessageBytes authenticate =

```
response.getCoyoteResponse().getMimeHeaders()  
.addValue(AUTHENTICATE_BYTES, 0, AUTHENTICATE_BYTES.length);
```

Successivamente viene recuperato il nome del Realm dell'applicazione Web a cui si cerca di accedere che, nel nostro caso, ricordiamo assume il valore *Prove*; tale valore viene concatenato al valore della variabile **authenticate** andando a formare la variabile **authenticateCC**. Alla fine quest'ultima variabile risulta assumere il seguente valore:

Basic realm="Prove"

A questo punto si hanno tutti i pezzi per poter creare il pop-up eseguendo il metodo `sendError(HttpServletResponse.SC_UNAUTHORIZED)` a cui abbiamo accennato sopra: se leggiamo l'**RFC-2616 [8]** e prestiamo attenzione sia ai codici di stato sia ai campi relativi, in particolare al caso dell'errore 401, ci accorgiamo che è espressamente richiesto, affinché si realizzi la fase di challenge, che vada inserito, all'interno dell'header della response http, il campo **WWW-Authenticate** contenente il valore del Realm dell'applicazione, scritto nel formato presentato dalla variabile **authenticateCC**. Sempre secondo le specifiche RFC-2616, il client può replicare con un appropriato campo **Authorization** nell'header della request http. Un utente (o agente) che desideri autenticarsi su un server, generalmente ma non necessariamente, dopo aver ricevuto una **Response 401**, replica con una request che contiene un campo **Authentication** contenente le informazioni di autenticazione dell'utente per il Realm della risorsa che ha richiesto, esempio:

Authorization = "Authorization" ":" credentials

Se una richiesta è autenticata ed è specificato il Realm, le stesse credenziali di un utente dovrebbero essere valide per tutte le sue richieste all'interno di quel Realm,

restando nell'ipotesi che lo schema di autenticazione non agisca in modo diverso, ad esempio, mediante richiesta di credenziali che possano variare a seconda dei valori della fase di challenge o sincronizzati col clock. I dettagli di accesso del protocollo HTTP possono essere reperiti all'interno di un altro documento RFC, il numero **2617** [9].

Una volta che l'utente ha fornito la coppia [username, password], a questo punto viene recuperato dall'header della request, proprio il valore del campo Authorization a cui abbiamo accennato; nel nostro caso di studio, la variabile authorization assume il seguente valore:

Basic dXRlbnRIMTjjaGFuZ2VpdA==

A questo punto, viene creata una nuova variabile **authorizationBC** di tipo ByteChunk che contiene il valore assunto della variabile authorization; su questa nuova variabile si va a verificare se il suo valore inizia con la parola basic seguita da uno spazio ("**basic** "), non tenendo conto se la parola è scritta in minuscolo o maiuscolo: nel nostro caso, il contenuto di authorizationBC inizia con la parola "basic " e questo comporta che viene creata una nuova variabile, chiamata **authorizationCC** di tipo CharChunk. Viene applicato il metodo **decode(ByteChunk, CharChunk)**, appartenente alla classe **org.apache.catalina.util.Base64**, che prende in input la variabile authorizationBC, a cui però sono stati tolti i sei byte iniziali che compongono proprio la parola "basic ", e questa nuova variabile authorizationCC allo scopo di effettuare la conversione da stringhe in Base64 in stringhe scritte in esadecimale.

Base64.decode(authorizationBC, authorizationCC);

Tutta questa serie di passi porta a far assumere alla variabile authorizationCC il seguente valore ossia l'username e la password, precedentemente inseriti alla richiesta di input, separati dal carattere ":"; questi due dati segreti vengono poi memorizzati separatamente all'interno di altrettante variabili, dai nomi esplicativi, **username** e **password**. Nel nostro caso authorizationCC assume il valore:

utente1:changeit

Una volta che sono state popolate queste due nuove variabili, non resta che passarle in input ad un ulteriore metodo, appartenente alla classe **org.apache.catalina.Realm**, allo scopo di ottenere un valore del **Principal** (ossia un'entità, tipicamente un utente, che è identificata dal sistema di sicurezza) associato alla coppia [username, password]: con tale istruzione si va ad invocare il modulo di login che è stato configurato per quella particolare applicazione (come avviene nel caso di JBoss in cui si possono definire singolarmente i moduli di login in relazione al dominio di sicurezza) oppure globale (come avviene per Tomcat). Torneremo a parlare di moduli di login per JBoss più avanti nel dettaglio.

```
principal = context.getRealm().authenticate(username, password);
```

Ora che siamo in possesso di un oggetto **principal** autenticato da un qualche modulo di login, possiamo procedere alla sua registrazione mediante il seguente metodo della classe **org.apache.catalina.authenticator.AuthenticatorBase**:

```
register(request, response, principal, Constants.BASIC_METHOD, username,  
password);
```

2.5 OPGAuthenticator: il nuovo autenticatore

Nei precedenti paragrafi abbiamo illustrato un po' i punti chiave, le difficoltà e le intenzioni che ci muovevano a creare un authenticator compatibile con Open Portal Guard: siamo quindi pronti ad illustrare quanto abbiamo realizzato, ossia l'oggetto di questo lavoro di tesi. Qui di seguito riportiamo il contenuto del nuovo autenticatore che abbiamo chiamato **OPGAuthenticator** (file **OPGAuthenticator.java**), ricordando che la base di partenza da cui siamo partiti per la sua realizzazione, è quella dell'autenticator di tipo BASIC. Al listato facciamo poi seguire la spiegazione del codice.

```
package org.apache.catalina.authenticator;
```

```

import java.io.IOException;
import java.security.Principal;
import java.util.Vector;
import org.apache.catalina.connector.Request;
import org.apache.catalina.connector.Response;
import org.apache.catalina.deploy.LoginConfig;
import org.apache.catalina.realm.GenericPrincipal;

public class OPGAuthenticator extends AuthenticatorBase {
    private static final String OPG_METHOD = "OPG";
    protected static final String info =
"org.apache.catalina.authenticator.OPGAuthenticator/1.1b";

    public String getInfo() {
        return(info);
    }

    public boolean authenticate(Request request, Response response, LoginConfig
config) throws IOException {
        String role = null;
        Vector roles = new Vector();
        String username = null;
        String password = null;
        Principal principal = null;

        try {
            role = request.getHeader("OPG_USERROLES");
            roles = getRoles(role);
            if(roles != null) {
                username = request.getHeader("OPG_USERNAME");
                principal = new GenericPrincipal(context.getRealm(), username,
password, roles);
            }
            else {
                return(mostraErrore("[OPG] Abort: lista dei ruoli nulla",
username));
            }
        } catch (Exception e) {
            principal = null;
        }
        if (principal != null) {

```

```

        register(request, response, principal, OPG_METHOD, username,
password);
        return(true);
    } else {
        return(mostraErrore("[OPG] Abort: principal non creato
correttamente", username));
    }
}
private boolean mostraErrore(String errore, String utente) {
    System.out.println(errore);
    if(utente != null)
        System.out.println(" per l'utente " + utente);
    return(false);
}

private Vector getRoles(String userRoles) {
    Vector ruoli = new Vector();
    String res[] = null;

    if ((userRoles.charAt(0) == '[') && (userRoles.charAt(userRoles.length()-1)
== ']')) {
        res = (userRoles.substring(1, userRoles.length()-1)).split(",");
        for(int i = 0; i < res.length; i++) {
            ruoli.add(res[i]);
        }
    } else {
        ruoli = null;
    }
    return ruoli;
}
}

```

Ora che abbiamo riportato il sorgente, possiamo spiegarne il funzionamento. Per prima cosa ricordiamo che affinché un qualsiasi authenticator venga eseguito, è necessario che sia specificato all'interno del file descrittore web.xml dell'applicazione che vogliamo proteggere: sarà poi compito dell'application server richiamare, a quel punto, la giusta classe associata al nome simbolico specificato all'interno del tag <auth-method> del deployment descriptor web.xml. Tutte le volte che un utente fa richiesta di accesso ad una risorsa protetta,

l'application server invoca l'autenticator specificato nel file web.xml, passandogli, come parametri di ingresso, la request servlet http, la response servlet http e la rappresentazione degli elementi di configurazione dell'applicazione web che è racchiusa, nel file web.xml, all'interno del tag <login-config>. L'application server invoca l'autenticatore mediante un metodo che hanno tutti gli authenticator: **boolean authenticate(Request, Response, LoginConfig)**. La request servlet http è un'interfaccia tra il client e la servlet e consente l'accesso alle informazioni contenute nell'header del protocollo http; la response servlet http permette invece di manipolare le informazioni dell'header http da restituire al client. Il metodo `authenticate` inizia facendo una serie di assegnazioni preparatorie alle variabili che conterranno l'identificativo dell'utente e la lista dei suoi ruoli per poi andare a leggere, dalla request http, il valore dei ruoli e memorizzando tale stringa all'interno di una lista. La lista dei ruoli viene letta mediante il metodo **String getHeader(String)** memorizzandola in una variabile **role** di tipo String:

```
role = request.getHeader("OPG_USERROLES");
```

Una volta in possesso della stringa dei ruoli, andiamo a creare una nuova lista dei ruoli, scandendo la variabile **role** mediante il metodo **Vector getRoles(String)**, ottenendo la variabile **roles** di tipo Vector che andremo ad utilizzare per creare il Principal:

```
roles = getRoles(role);
```

Il metodo `getRoles` ha il compito di creare una lista, di tipo Vector, a partire dalla stringa in ingresso: la stringa che viene letta deve avere, come abbiamo accennato in precedenza, un certo pattern, ossia deve essere scritta come [ruolo1,ruolo2, ...,ruoloN] (dove ruolo1, ruolo2 ed ruoloN sono i ruoli associati all'utente). Qualora la stringa dei ruoli non rispecchi tale forma, abbiamo programmato l'autenticator in modo tale che fallisca il processo di autenticazione restituendo l'errore booleano *false*; se, invece, la lista dei ruoli ha il giusto pattern, il processo di autenticazione procede andando a leggere l'identificativo dell'utente mediante il seguente metodo e salvando questo nuovo valore nella variabile **username** di tipo String:

```
username = request.getHeader("OPG_USERNAME");
```

Grazie al metodo `getHeader` siamo in grado di accedere alla request http e di leggere i dati che ci interessano; con la coppia [username, ruoli] possiamo procedere a popolare la variabile **principal** mediante il costruttore **GenericPrincipal(Realm, String, String, java.util.List)** della classe **org.apache.catalina.realm.GenericPrincipal**:

```
principal = new GenericPrincipal(context.getRealm(), username, password, roles);
```

Come possiamo vedere, il costruttore della classe `GenericPrincipal` prende in ingresso la coppia [username, ruoli] caratterizzata dalle variabili `username` (di tipo `String`) e `roles` (di tipo `Vector`) viste in precedenza, il **realm** preso dal `context` ed anche una variabile chiamata **password** di tipo `String`, in cui viene specificata la password dell'utente. Tale variabile è fissata al valore costante *null* dal momento che non effettuiamo autenticazione mediante la coppia [username, password].

Avevamo visto, analizzando il caso dell'autenticazione BASIC, che la variabile `principal` veniva popolata mediante l'invocazione del modulo `login` configurato per la particolare applicazione: nel caso di questo nuovo autenticatore, invece, non siamo interessati a demandare l'autorizzazione delle credenziali ad ulteriori moduli di `login`. E' grazie al costruttore della classe `GenericPrincipal` che abbiamo potuto costruirci un `Principal` autenticato già a livello di `authenticator` e non più a livello di `login module`, come avviene nell'autenticatore `Basic` che popola la variabile `principal` nel seguente modo:

```
principal = context.getRealm().authenticate(username, password);
```

Abbiamo accennato che il nostro nuovo autenticatore controlla il pattern della stringa dei ruoli: in caso di stringa mal formata, o eventualmente nulla, l'autenticatore è programmato in modo tale da scrivere, nel file di log dell'application server, un messaggio d'errore in cui segnaliamo che la lista dei ruoli non è stata popolata correttamente. Verificandosi questo evento sfortunato, oltre alla scrittura nel log, viene assegnato valore *null* alla variabile `principal`. Nel caso la lista dei ruoli sia non vuota, andiamo a fare un controllo sul valore assunto

dalla variabile `principal`, verificando se assume valore non nullo: in tal caso l'autenticatore procede alla vera registrazione del `principal` nella sessione mediante il metodo **`register(Request, Response, Principal, String, String, String, String)`** della classe `org.apache.catalina.authenticator.AuthenticatorBase`. Facciamo notare che tutti gli autenticatori estendono questa classe. Il metodo `register` riceve in ingresso la request http, la response http, la variabile `principal` appena creata, un'ulteriore variabile `String`, chiamata **`OPG_METHOD`** di valore costante **`OPG`** che serve ad indicare il metodo di autenticazione e due altre variabili `String` che rappresentano l'username e la password (ricordiamo che la variabile che contiene la password ha costantemente valore nullo perché non siamo interessati all'autenticazione mediante la coppia [username, password]).

```
register(request, response, principal, OPG_METHOD, username, password);
```

Con l'esecuzione della precedente istruzione, si conclude il processo di autenticazione e l'autenticator termina restituendo il valore booleano *true* per il metodo `authenticate`, per poi ripassare il controllo all'application server; in caso d'errore, andiamo a notificare l'errore nel file di log dell'application server per poi terminare l'esecuzione restituendo *false* al metodo `authenticate`.

2.5.1 Compatibilità dell'autenticatore

Quando abbiamo iniziato lo studio degli autenticator presenti all'interno di Tomcat e di JBoss, ci siamo posti come priorità che il nuovo autenticator fosse totalmente compatibile con questi application server ma anche, se era possibile, con **IBM WebSphere** ed **Apache Geronimo2**; sorprendentemente, durante la fase di ricerca, ci siamo accorti che tutti questi application server hanno una base comune, ossia adottano Tomcat (o parte del suo core) come JSP Servlet Container; questo vuol dire che qualsiasi autenticator creato e che risulti funzionare in Tomcat funzionerà, senza apportare modifiche al codice, anche su altri application server. Questo risultato è piuttosto importante perché, a priori, non era nota tale compatibilità che è stata accertata solamente mediante analisi approfondita dei sorgenti, studiando il contenuto delle classi dei vari application

server (in particolare JBoss): tale risultato ci ha così permesso di velocizzare il completamento della creazione dell'authenticator. Aggiungiamo che la compatibilità di questo nuovo autenticatore è pressoché totale e non necessita di modifiche, finché andiamo a considerare application server che si basano su Tomcat 5.xx; se andiamo a considerare versioni inferiori abbiamo riscontrato problemi perché siamo costretti ad utilizzare classi e costrutti differenti da quelli che utilizziamo con OPGAuthenticator per Tomcat 5.xx (o superiore). Un'altra questione molto importante che abbiamo analizzato è stata quella di capire il modo migliore di recuperare l'informazione relativa alla coppia *[OPG_USERNAME, OPG_USERROLES]* e se c'era un modo che fosse il più compatibile con i vari application server considerati. Se ci ricordiamo quanto abbiamo accennato nel precedente paragrafo, ogni autenticatore inizia la propria esecuzione dal metodo boolean `authenticate(Request, Response, LoginConfig)`: questo ci ha avvantaggiati perché, una volta avviata l'esecuzione dell'authenticator, siamo già in possesso dell'interfaccia con la quale interagire con i dati della request http, in quanto è lo stesso application server a passarla all'autenticatore quando invoca il metodo `authenticate`. Nel precedente paragrafo abbiamo visto che abbiamo fatto uso del metodo `getHeader` nel seguente modo:

```
role = request.getHeader("OPG_USERROLES");  
username = request.getHeader("OPG_USERNAME");
```

L'interfaccia `request` mette a disposizione, però, anche un altro metodo chiamato `getCoyoteRequest()` che consente di accedere alla request http di **Coyote**, ossia un Connector HTTP/1.1 che permette a Catalina, che è il nome in codice di Tomcat, di funzionare come web server stand alone oltre che come JSP Servlet Container. Volendo far uso della request di Coyote, è possibile leggere i campi di interesse nel seguente modo:

```
role = request.getCoyoteRequest().getHeader("OPG_USERROLES");  
username = request.getCoyoteRequest().getHeader("OPG_USERNAME");
```

Abbiamo dovuto fare delle scelte che rendessero l'OPGAuthenticator compatibile con il maggior numero di versioni di application server, basati appunto su Tomcat,

ed abbiamo preferito utilizzare il primo sistema di accesso ai campi dell'header dal momento che il metodo `getCoyoteRequest()` è disponibile unicamente a partire dalla versione 5.5 di Tomcat.

2.5.1.1 Collocazione della classe OPGAuthenticator.class

Tralasciando IBM WebSphere e Apache Geronimo che non erano oggetto diretto di studio, continuiamo la trattazione relativa al modo di utilizzare l'`OPGAuthenticator`: affinché quest'ultimo sia operativo dobbiamo collocare la classe del nuovo autenticatore, chiamata **`OPGAuthenticator.class`**, all'interno del giusto file JAR cosicché l'application server la possa riconoscere e caricare al momento del suo start-up. Nei prossimi due paragrafi andiamo a mostrare dove collocare la classe del nostro autenticatore, diversificando la spiegazione per Tomcat e per JBoss.

2.5.1.1.1 Tomcat: catalina.jar

Durante lo studio di Tomcat ci siamo accorti che tutti i suoi autenticatori sono situati all'interno di un archivio JAR chiamato **`catalina.jar`**, situato in **`$CATALINA_HOME/server/lib`**. All'interno di questo file JAR troviamo il package **`org.apache.catalina.authenticator`** che contiene le implementazioni dei vari tipi di metodi di autenticazione (BASIC, FORM, CLIENT-CERT, DIGEST e NONE): è all'interno di questo package che andiamo a collocare la classe `OPGAuthenticator.class`. In questo package troviamo anche un'importante classe chiamata `AuthenticatorBase` che viene utilizzata per implementazioni custom degli autenticatori: abbiamo detto che tutti gli autenticatori estendono ed implementano questa classe e così avviene anche per il nostro autenticatore. Non è sufficiente copiare la classe di un autenticatore all'interno del package accennato poco fa, affinché questi risulti utilizzabile: dobbiamo, infatti, creare l'associazione tra il nome che si vuole dare al metodo di autenticazione e la classe che lo implementa. Questa mappatura avviene andando a modificare, con un editor di testi, il file di proprietà chiamato **`Authenticators.properties`** (che

troviamo in **catalina.jar\org\apache\catalina\startup**). Questo file di configurazione viene caricato in memoria ogni volta che Tomcat viene avviato. Riportiamo il contenuto di questo file di proprietà, in cui possiamo notare l'associazione tra nome simbolico ed il nome della classe. Facciamo anche notare che OPGAuthenticator appartiene al package org.apache.catalina.authenticator come tutti gli altri autenticatori.

```
BASIC=org.apache.catalina.authenticator.BasicAuthenticator  
CLIENT-CERT=org.apache.catalina.authenticator.SSLAuthenticator  
DIGEST=org.apache.catalina.authenticator.DigestAuthenticator  
FORM=org.apache.catalina.authenticator.FormAuthenticator  
NONE=org.apache.catalina.authenticator.NonLoginAuthenticator  
OPG=org.apache.catalina.authenticator.OPGAuthenticator
```

2.5.1.1.2 JBoss: jbossweb.jar

In questo nuovo paragrafo andiamo a mostrare come abilitare l'autenticatore per OPG all'interno di JBoss: i passi da seguire sono simili a quelli che abbiamo fatto per Tomcat. Durante lo studio abbiamo scoperto che in JBoss tutti gli authenticator sono situati all'interno dell'archivio JAR **jbossweb.jar** (situato in **\$JBOSS_HOME/server/[server_scelto]/deploy/jboss-web.deployer/**), precisamente all'interno del percorso **jbossweb.jar\org\apache\catalina\authenticator**. Una cosa va subito notata ossia che il package è lo stesso che è utilizzato da Tomcat, ossia org.apache.catalina.authenticator. Abbiamo detto nei precedenti paragrafi che JBoss (ed altri application server) si basa su Tomcat e già qui possiamo renderci conto di tale affermazione. Non abbiamo dovuto compilare nuovamente il sorgente dell'OPGAuthenticator per farlo funzionare con JBoss, abbiamo semplicemente preso la classe che avevamo ottenuto per Tomcat, inserendola in questo package ottenendone la piena compatibilità a livello di librerie. Come in Tomcat abbiamo dovuto modificare alcuni file di configurazione. Il primo che andiamo a prendere in considerazione è il file **jboss-service.xml** (che si trova in **\$JBOSS_HOME/server/[server_scelto]/deploy/jboss-web.deployer/META-INF/**) che qui riportiamo integralmente.

```

<?xml version="1.0" encoding="UTF-8"?>
<!-- $Id: jboss-service.xml 60489 2007-02-12 08:22:28Z scott.stark@jboss.org $ -->
<server>
<mbean code="org.jboss.web.tomcat.service.JBossWeb"
      name="jboss.web:service=WebServer"
      xmbean-dd="META-INF/webserver-xmbean.xml">
<attribute name="Authenticators" serialDataType="jbx">
  <java:properties xmlns:java="urn:jboss:java-properties"
    xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
    xs:schemaLocation="urn:jboss:java-properties
      resource:java-properties_1_0.xsd">
    <java:property>
      <java:key>BASIC</java:key>
      <java:value>org.apache.catalina.authenticator.BasicAuthenti
cator
      </java:value>
    </java:property>
    <java:property>
      <java:key>CLIENT-CERT</java:key>
      <java:value>org.apache.catalina.authenticator.SSLAuthentic
ator
      </java:value>
    </java:property>
    <java:property>
      <java:key>DIGEST</java:key>
      <java:value>org.apache.catalina.authenticator.DigestAuthen
ticator</java:value>
    </java:property>
    <java:property>
      <java:key>FORM</java:key>
      <java:value>org.apache.catalina.authenticator.FormAuthenti
cator</java:value>
    </java:property>
    <java:property>
      <java:key>OPG</java:key>
      <java:value>org.apache.catalina.authenticator.OPGAuthenti
cator</java:value>
    </java:property>
    <java:property>
      <java:key>NONE</java:key>

```

```

        <java:value>org.apache.catalina.authenticator.NonLoginAuth
        henticator</java:value>
    </java:property>
</java:properties>
</attribute>
<attribute name="DefaultSecurityDomain">java:/jaas/other</attribute>
<attribute name="Java2ClassLoadingCompliance">>false</attribute>
<attribute name="UseJBossWebLoader">>false</attribute>
<attribute name="FilteredPackages">javax.servlet</attribute>
<attribute name="LenientEjbLink">>true</attribute>
<attribute name="DeleteWorkDirOnContextDestroy">>false</attribute>
<attribute
                        name
                        =
"ManagerClass">org.jboss.web.tomcat.service.session.JBossCacheManager
</attribute>
<attribute name="SubjectAttributeName">j_subject</attribute>
<attribute
                        name
                        =
"SessionIdAlphabet">ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+.*
</attribute>
<attribute name="SnapshotMode">instant</attribute>
<attribute name="SnapshotInterval">2000</attribute>
<attribute name="UseJK">>false</attribute>
<attribute name="Domain">jboss.web</attribute>
<depends optional-attribute-name="SecurityManagerService"
        proxy-type="attribute">jboss.security:service=JaasSecurityManager
</depends>
<depends>jboss:service=TransactionManager</depends>
<depends>jboss.jca:service=CachedConnectionManager</depends>
</mbean>
</server>

```

Il file mostrato è lungo ed articolato ma la sezione di interesse è quella interna al tag `<attribute name="Authenticators">`. Al suo interno abbiamo definito l'associazione tra il nome simbolico (OPG) e la classe associata (`org.apache.catalina.authenticator.OPGAuthenticator`) mediante appositi tag che riportiamo:

```

<java:property>
    <java:key>OPG</java:key>

```

```
<java:value>
    org.apache.catalina.authenticator.OPGAuthenticator
</java:value>
</java:property>
```

Anche in JBoss è presente il file di proprietà `Authenticators.properties` (posizionato in `jbossweb.jar\org\apache\catalina\startup`) che possiamo modificare come nel paragrafo precedente. Abbiamo notato come non sia necessario modificare sia il file `Authenticators.properties` che `jboss-service.xml` dal momento che le mappature che vengono definite in questo file di configurazione XML hanno precedenza su quelle fatte nell'altro e quindi è del tutto inutile andare a modificare il file `Authenticators.properties`.

2.5.2 Attivazione di OPGAuthenticator

Nei due precedenti paragrafi ci siamo preoccupati della collocazione fisica dell'autenticatore all'interno dell'application server; l'ultimo passaggio necessario che andiamo a realizzare è quello di specificare il valore **OPG**, all'interno del tag `<auth-method>` nel deployment descriptor `web.xml` dell'applicazione Web che vogliamo proteggere con Open Portal Guard. Qui riportiamo solo la sezione `<login-config>` relativa al file `web.xml` della nostra applicazione di test.

```
<login-config>
    <auth-method>
        OPG
    </auth-method>
    <realm-name>
        Prove con OPG
    </realm-name>
</login-config>
```

2.6 OPG e la sicurezza

Un autenticatore, qualunque esso sia, da solo non fornisce alcuna garanzia di essere immune da attacchi quali **Man in the middle** (figura 2.4); è necessario mettersi nella situazione di adottare una qualche difesa ed un modo è quello di far ricorso a protocolli specifici quale il **Security Socket Layer (SSL)** [10].

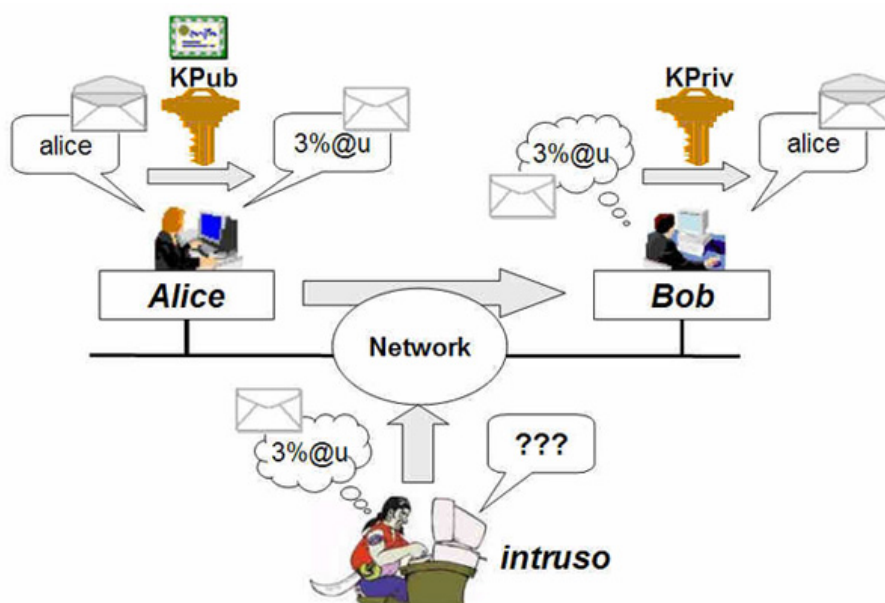


Figura 2.4: Schema di attacco chiamato “Man In The Middle”

Oggi giorno praticamente tutte le società, enti o individui possiedono un proprio sito Web; aumenta sempre di più anche il numero di utenti che possiedono un collegamento Internet ed è questo aumento di utenza che spinge molte aziende ad offrire sempre più servizi Web, quali ad esempio il commercio elettronico: in questo clima di libera creazione di servizi, non possiamo dimenticare che Internet ed il Web sono estremamente insicuri e facilmente oggetto di attacchi. Cresce sempre più l'esigenza di offrire servizi Web che siano sicuri o, almeno, che offrano qualche garanzia, visto che la totale sicurezza è impossibile da ottenere. Il protocollo SSL e la sua successiva evoluzione, chiamata **Transport Layer Security (TLS)**, stanno assumendo sempre più importanza proprio per il bisogno di ottenere servizi Web sicuri.

Quando affrontiamo problematiche di sicurezza nel Web, è bene capire a che livello esse possano essere messe in pratica nell'ambito dello stack dei protocolli TCP/IP (figura 2.5).

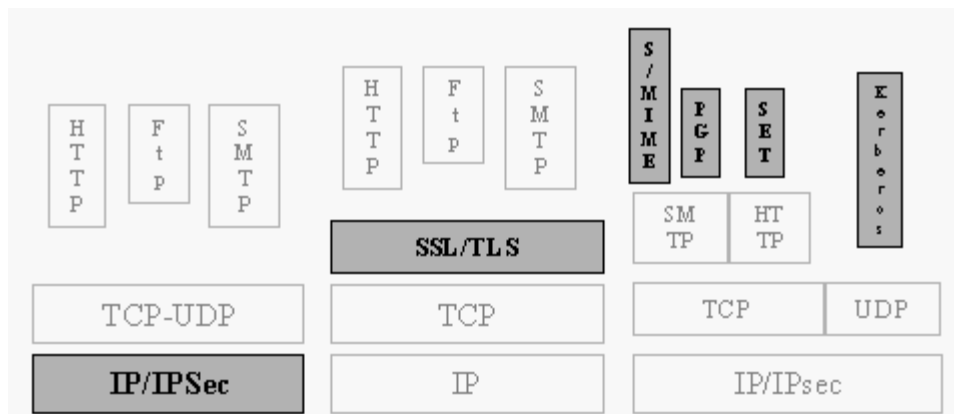


Figura 2.5: Tipi di modalità di sicurezza

Un primo approccio di implementazione della sicurezza è quello di far uso di **IPSec**, con il quale è possibile fornire una funzione di filtraggio, ossia permettere l'elaborazione dell'header solo al traffico selezionato, il tutto in modo trasparente per l'utente e per le applicazioni. Fornire una sicurezza a livello IP è vantaggioso per un'azienda perché può assicurare un networking sicuro sia alle applicazioni che sono dotate di propri meccanismi di sicurezza, sia a tutte le altre applicazioni che non ne sono provviste. A livello di trasporto, invece, sono possibili due soluzioni ossia l'SSL ed il TLS, totalmente trasparenti alle applicazioni. Per il livello di applicazione, la sicurezza potrebbe essere dettagliata per le specifiche necessità di una data applicazione; nel contesto della Web Security un importante esempio di questo è approccio è rappresentato dal **Security Electronic Transaction (SET)**. In Internet sono stati sviluppati meccanismi di sicurezza specifici quali **PGP** o **S/MIME** per l'e-mail, **Kerberos** per l'autenticazione in ambienti distribuiti, ecc. Il Secure Socket Layer è stato sviluppato da Netscape: la versione 3 di tale protocollo è stata progettata con un processo di revisione pubblica e sulla base di input dal mondo delle aziende, ed è stata pubblicata come bozza Internet; successivamente raggiunto un certo consenso per produrre uno standard Internet, venne costituito il gruppo di lavoro TLS nell'ambito della **Internet Engineering Task Force (IETF)** affinché si realizzasse uno standard comune. Questa prima versione del TLS può essere considerata fondamentalmente come la versione 3.1 di SSL ed ha il vantaggio di essere molto simile e retro compatibile con SSL versione 3. Il protocollo SSL combina le capacità dei tre principali ma separati protocolli dell'IPSec e li applica per ottenere una protezione a livello di trasporto; questi protocolli inclusi nell'SSL

comprendono autenticazione, crittaggio e scambio chiavi mentre nell'IPSec questi sono forniti da protocolli separati: **Authentication Header (AH)**, **Encapsulating Security Payload (ESP)** e protocolli per scambio chiavi quali ad esempio **ISAKMP/Oakley**. I dati dell'SSL sono sempre trasmessi in un formato speciale che incorpora un checksum simile all'AH ed un identificatore per la security association simile all'ESP. Quando due host iniziano a comunicare utilizzando SSL, il messaggio iniziale utilizza uno speciale protocollo di handshake per stabilire gli algoritmi di crittaggio e le chiavi da utilizzare. L'architettura SSL è stata progettata per impiegare TCP con un servizio affidabile end-to-end; SSL però non è un unico protocollo bensì è formato da due livelli (**figura 2.6**).

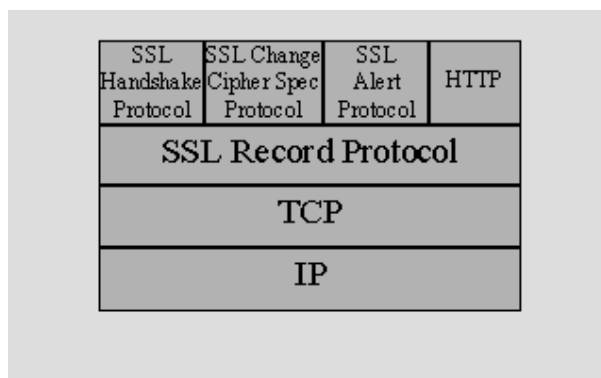


Figura 2.6: Stack di protocolli SSL

Il **SSL Record Protocol** fornisce servizi di sicurezza di base per i vari protocolli di livello superiore, in particolare, il protocollo HTTP che fornisce il servizio di trasferimento per le interazioni Web Client/Server, può operare su SSL. Nell'ambito dell'SSL sono definiti tre protocolli di più alto livello quali: **Handshake Protocol**, **Change Cipher Spec Protocol** ed **Alert Protocol**. È importante sottolineare due concetti importanti quali sessione SSL e connessione SSL che sono così definiti:

- **connessione**: una connessione è un trasporto, nella definizione del modello a livelli OSI, che fornisce un adatto tipo di servizio. Per SSL, questa connessione è una relazione tra nodi paritari (peer-to-peer), transitoria ed è associata ad una sola sessione;

- **sessione**: una sessione SSL è un'associazione tra un client ed un server; le sessioni vengono create dal protocollo Handshake e definiscono un set di parametri di sicurezza crittografica che possono essere condivisi su più connessioni. Le sessioni vengono utilizzate per evitare di svolgere la costosa operazione di negoziazione di nuovi parametri di sicurezza per ciascuna connessione.

Ogni coppia di parti, ad esempio applicazioni come HTTP tra client e server, può avere più connessioni sicure ed, in teoria, sarebbe possibile anche avere più sessioni simultanee fra le parti ma, in realtà, questa funzionalità non viene utilizzata. Ad ogni sessione vengono associati più stati ed appena una sessione è stabilita, vi è uno stato operativo corrente, per la lettura e la scrittura, ovvero per la ricezione e l'invio; durante il protocollo di Handshake, inoltre, vengono creati degli stati provvisori di lettura e scrittura che diventano stati correnti al suo completamento. Il protocollo SSL Record fornisce due servizi per le connessioni SSL:

- **segretezza**: il protocollo Handshake definisce una chiave segreta condivisa utilizzata per la crittografia convenzionale del payload SSL;
- **integrità del messaggio**: il protocollo Handshake definisce anche una chiave segreta condivisa che viene utilizzata per creare un codice MAC (Message Authentication Code).

2.6.1 Mutua autenticazione tra Apache e Tomcat

Ora che abbiamo introdotto brevemente SSL ci possiamo chiedere come poterlo applicare per instaurare una connessione sicura tra Apache HTTP Server e Tomcat (nel prossimo paragrafo affronteremo lo stesso problema per JBoss) [11]. Premettiamo che è raro che la macchina su cui gira l'application server sia la stessa che ospita il web server: nelle nostre prime fasi di testing abbiamo lavorato proprio in questa situazione, per poi spostarci nella direzione di avere macchine dedicate a compiti diversi. In questo e nel prossimo paragrafo andremo ad utilizzare due tool per la gestione e generazione di chiavi pubbliche/private e

certificati **X.509**: il primo è il framework **OpenSSL** [12] e l'altro è il **keytool** appartenente a Java [13].

Prima di procedere è bene introdurre alcuni termini. In crittografia, una **Certificate Authority o Certification Authority (CA)** è un ente di terza parte, pubblico o privato, abilitato a rilasciare un certificato digitale tramite procedura di certificazione che segue standard internazionali e conforme alla normativa europea e nazionale in materia. Viene fatto utilizzo della crittografia a doppia chiave, o asimmetrica, in cui una delle due chiavi viene resa pubblica all'interno del certificato (**chiave pubblica**), mentre la seconda, univocamente correlata alla prima, rimane segreta ed associata al titolare (**chiave privata**). Una coppia di chiavi può essere attribuita ad un solo titolare. La CA è in possesso di un certificato con il quale sono firmati tutti i certificati emessi agli utenti. Ora che abbiamo introdotto cosa intendiamo con CA, possiamo procedere con la spiegazione dei passi atti a creare la mutua autenticazione SSL.

Il primo passo che andiamo ad eseguire è quello relativo alla creazione di un certificato digitale per il **Root Certification Authority (RootCA)** che, nella situazione reale, potrebbe essere rilasciato da un'organizzazione globalmente fidata quale **Verisign, Thawte e CACert**. Una RootCA è una CA il cui compito è quello di firmare i certificati delle CA ad essa subordinate. Per sua natura la RootCA, essendo l'inizio della catena di certificazione, ha il proprio certificato di tipo self-signed che contiene la sua chiave pubblica. Anticipiamo che tutti i certificati che andremo a generare, saranno poi verificati dalla RootCA: il primo che andiamo a considerare è, quindi, proprio il Root Certificate e la sua chiave privata. Questo certificato è di tipo self-signed e reso attendibile sia dalla **Public Key Infrastructure (PKI)** che dalla **Privilege Management Infrastructure (PMI)** e generato dal seguente comando, eseguito all'interno di una shell o terminale, appartenente al toolkit OpenSSL che permette di implementare SSL/TLS in ambiente Linux. Una nota circa la scelta della password: per tutti i certificati, ed anche per i keystore, abbiamo utilizzato la parola chiave **changeit** ma è importante, oltre che buona cosa, utilizzare differenti password, oltre che più complesse di questa, per crittare differenti chiavi.

```
openssl req -out rootca.pem -keyout rootcakey.pem -new -x509 -days 1000
```

Il parametro **req** è il comando principale per creare ed elaborare le richieste di certificato in formato **PKCS #10**, **-out** serve a specificare il file di output, con **-keyout** specifichiamo il file in cui memorizzare una nuova coppia di chiavi private. L'opzione **-new** serve a creare un nuovo certificato che viene generato andando a richiedere all'utente alcuni dati significativi. **-x509** specifica che i certificati generati in output devono essere di tipo self-signed. Questo parametro è tipicamente utilizzato per generare certificati di test o il certificato della RootCA. Con **-days**, invece, si va a specificare il numero di giorni per cui il certificato è valido. Il comando appena digitato permette di ottenere sia una chiave privata sia un certificato digitale, caratterizzati dall'essere a *1024* bit. Qualora volessimo aumentare il numero di bit, perché interessati ad un grado di sicurezza più alto, possiamo andare a creare un file di configurazione, con un editor di testi, in cui andiamo a specificare il numero di bit ed altri parametri. Ad esempio proviamo a creare il file chiamato **config_ssl.conf** al cui interno scriviamo queste due righe:

```
[ req ]
default_bits = 2048
```

Ogni volta che vogliamo usare questo file di configurazione, è sufficiente aggiungere il parametro **-config config_ssl.conf** ad ogni invocazione del comando **openssl**. La direttiva **default_bits** serve a specificare il numero di bit con il quale vogliamo andare a lavorare: di default tale valore è fissato a *1024*. Nel proseguo della nostra trattazione, ed anche per scopo dimostrativo, continueremo ad utilizzare i valori di default e *1024* bit.

La maggior parte delle versioni di OpenSSL richiedono un file contenente un numero che viene utilizzato per tenere traccia dell'attuale numero di serie dei seguenti certificati che stanno per essere rilasciati. Possiamo semplificare il tutto mediante il seguente comando che crea un file di testo, chiamato **rootca.srl**, con dentro un numero (**000001**):

```
echo 000001 > rootca.srl
```

Quest'ultimo file, in generale, è formato da una sola riga e contiene un numero pari di cifre esadecimali che sta ad indicare il numero di serie da utilizzare.

Il certificato RootCA che abbiamo creato in precedenza, lo andiamo ad utilizzare per scopi di autenticazione e per assicurare che i certificati nella nostra configurazione di sicurezza non siano falsificati. Durante il processo di creazione di certificati è richiesta anche la presenza di un altro utente PMI che agisca come **root of trust**, chiamato **Source of Authority (SoA)**; tale utente necessita di un certificato e di una chiave che devono essere firmati mediante il certificato del RootCA in modo da garantirne l'autenticità attraverso i seguenti comandi. Il certificato che andiamo così a creare ha una validità di *1000* giorni (parametro **-days 1000**)

```
openssl req -out soareq.pem -keyout soakey.pem -new
```

```
openssl x509 -in soareq.pem -out soa.pem -CA rootca.pem -CAkey  
rootcakey.pem -days 1000 -req
```

Il parametro **x509** abilita l'utility per la creazione e firma di certificati. Con **-in** specifichiamo il file certificato da leggere, **-CA** specifica la CA da usare per firmare i certificati, con **-CAkey** indichiamo la chiave privata con cui firmare un certificato. Il parametro **-req** serve a specificare la richiesta di un certificato e con **-days** andiamo ad impostare i giorni di validità del certificato. Se volessimo utilizzare il file `rootca.srl` creato in precedenza, a quest'ultimo comando possiamo aggiungere il parametro **-CAserial rootca.srl**: il numero di serie che viene usato viene incrementato di volta in volta ad ogni suo utilizzo; nel proseguo non utilizzeremo tale parametro ma è sempre possibile usarlo. Durante il processo di installazione capiterà molte volte di convertire certificati in differenti formati: il formato di default di OpenSSL è il **PEM** ma qualche volta dovremo utilizzare anche il formato binario **DER**, utilizzato principalmente quando vogliamo caricare certificati all'interno di una struttura OpenLDAP. Con la seguente istruzione effettuiamo la conversione di un certificato dal formato PEM a quello DER:

```
openssl x509 -in soa.pem -out soa.der -outform DER
```

Ora che siamo in possesso di questi primi certificati digitali, vediamo come utilizzarli, insieme ad altri che andremo a creare fra non molto, in modo da realizzare la mutua autenticazione tra Apache e Tomcat [14]; nel prossimo paragrafo andremo a fare la stessa cosa con JBoss.

Vediamo le modifiche da fare all'interno di Tomcat per abilitare una connessione sicura: per prima cosa dobbiamo abilitare la direttiva che si occupa di gestire le connessioni sicure sulla porta 8443, modificando il file \$CATALINA_HOME/conf/server.xml e configurando al suo interno lo specifico **Connector**, ossia un endpoint mediante il quale le richieste sono ricevute e le risposte sono ritornate. Facciamo presente che ciascun connector, inoltre, passa le proprie richieste allo specifico Container, generalmente all'Engine, che provvede ad elaborarle.

<!--Define a SSL HTTP/1.1 Connector on port 8443 -->

<Connector

```
    port="8443"
    maxThreads="150"
    minSpareThreads="25"
    maxSpareThreads="75"
    enableLookups="false"
    disableUploadTimeout="true"
    acceptCount="100"
    scheme="https"
    secure="true"
    clientAuth="true"
    sslProtocol="TLS"
    keystoreFile="/home/sed/tomcat.keystore"
    keystorePass="changeit"/>
```

L'aver definito questi parametri per il Connector non è sufficiente da solo a rendere operativo SSL dal momento che dobbiamo creare una coppia di chiavi per instaurare la comunicazione tra Tomcat ed il client, rappresentato in questo caso dal web server ossia Apache. Una delle chiavi che andremo a generare deve essere memorizzata nel **keystore** che è un ulteriore formato di memorizzazione per i certificati e le chiavi. Analizzando il precedente frammento di codice notiamo che, mediante l'attributo **keystoreFile**, andiamo a definire il percorso dove è situato

proprio il keystore, mentre con **keystorePass** andiamo a specificare quale sia la password necessaria ad aprire tale file. Non meno importante è la presenza dell'attributo **clientAuth** che abbiamo settato al valore boolean *true* il cui significato è quello di abilitare l'autenticazione del client da parte del server, ossia dell'application server in questione; qualora fosse stato settato il valore *false*, solamente il client avrebbe autenticato il server, situazione tipica in una configurazione SSL.

Ora che abbiamo illustrato gli attributi necessari senza i quali non si può attuare la mutua autenticazione, possiamo procedere con la creazione del keystore e per farlo utilizziamo il comando **keytool**, fornito insieme alla distribuzione Java. Nella spiegazione che andiamo ad illustrare, facciamo principalmente uso dei valori di default del **keytool** ma ciò non toglie che possiamo raffinare il suo utilizzo. Qualora volessimo forzare, ad esempio, l'utilizzo dell'algoritmo crittografico **RSA** con una chiave lunga 2048 bit, possiamo aggiungere il parametro **-keyalg RSA -keysize 2048** al seguente comando, oppure, se volessimo cambiare la validità dei certificati e delle chiavi generate possiamo aggiungere il parametro **-validity numerogiorni**, dove *numerogiorni* rappresenta il numero di giorni per cui si vuole rendere valido quanto si sta creando. Osserviamo che, tipicamente, un certificato generato senza il parametro che ne specifica la durata della sua validità, ha un tempo di vita di 90 giorni mentre per l'algoritmo di generazioni delle chiavi, il valore di default è il **Digital Signature Algorithm (DSA)** a 1024 bit. Col seguente comando andiamo a generare, con **-genkey**, una coppia di chiavi pubblica/privata, assegnandole mediante il parametro **-alias** l'alias *tomcat*, all'interno del file keystore chiamato **tomcat.key** specificato dal parametro **-keystore**, specificando con **-storepass** anche quale sia la password (**changeit**) per accedervi.

```
keytool -genkey -keystore tomcat.keystore -storepass changeit -alias tomcat
```

Completato questo passaggio, procediamo col creare un certificato usando la chiave pubblica appena creata per poi firmarlo con il certificato del RootCA per poi importarlo nel keystore: non dobbiamo dimenticarci di importare, tra i certificati fidati, anche il certificato RootCA altrimenti la catena di autenticazioni non potrà completarsi correttamente. In questi passaggi andiamo ad utilizzare sia

keytool che OpenSSL: notiamo che la validità del certificato **tomcat.pem** che andiamo a definire è di 365 giorni.

```
keytool -certreq -keystore tomcat.keystore -storepass changeit -alias tomcat -file tomcatreq.pem
```

Nel comando precedente, **-certreq** serve a generare un **Certificate Signing Request (CSR)** utilizzando il formato PKCS #10. Il CSR va poi mandato ad una CA affinché lo certifichi, ottenendo anche un certificato o una catena di certificati che devono essere utilizzati per rimpiazzare la catena di certificati (che di solito sono di tipo self-signed) già presenti nel keystore. Il comando che segue invece serve a certificare il CSR col certificato della nostra RootCA.

```
openssl x509 -in tomcatreq.pem -out tomcat.pem -CA rootca.pem -CAkey rootcakey.pem -days 365 -req
```

```
keytool -import -keystore tomcat.keystore -storepass changeit -alias rootca -file rootca.pem -trustcacerts
```

Il comando precedente ha la funzione di importare il certificato RootCA nel keystore. Il parametro **-trustcacerts** specifica che il certificato deve essere considerato parte della catena di certificazione e quindi da ritenersi fidato. Col comando seguente andiamo ad importare il certificato, ottenuto dalla certificazione del CSR mediante RootCA, all'interno del keystore.

```
keytool -import -keystore tomcat.keystore -storepass changeit -alias tomcat -file tomcat.pem
```

Il certificato che abbiamo così creato sarà utilizzato per la cifratura dei messaggi tra client, rappresentato dal web server, e l'application server Tomcat; abbiamo però bisogno di un ulteriore certificato che andiamo ad usare per il client e che firmeremo col certificato RootCA e questo perché, ricordiamo, vogliamo implementare la mutua autenticazione tra Apache e Tomcat: purtroppo il primo, non supportando chiavi crittate, ci obbliga a creare un certificato che non sia cifrato e questo lo realizziamo con le seguenti istruzioni utilizzando nuovamente

OpenSSL. Notiamo che abbiamo settato la validità di questo certificato, anche in questo caso, a 365 giorni. La non cifratura del certificato avviene mediante il parametro **-nodes**.

```
openssl req -out httpdreq.pem -keyout httpdkey.pem -new -nodes
```

Col comando seguente, come abbiamo accennato, andiamo a certificare, mediante RootCA, il file creato con la precedente istruzione.

```
openssl x509 -in httpdreq.pem -out httpd.pem -CA rootca.pem -CAkey rootcakey.pem -req -days 365
```

Dobbiamo ora convertire il certificato e le chiavi, generate al passo precedente, in un nuovo formato chiamato **PKCS #12**, appartenente alla famiglia dei **Public-Key Cryptography Standards (PKCS)**, utilizzato per memorizzare chiavi private e certificati a chiave pubblica; il file **httpd.p12** che andiamo a generare, lo possiamo importare all'interno dei certificati ritenuti validi ed accettati dal proprio browser web, ad esempio Mozilla Firefox. Nel comando che segue, il parametro **pkcs12** abilita l'utility per creare e elaborare file in formato PKCS #12 mentre con **-export** specifichiamo che vogliamo creare un file invece di processarlo. Il parametro **-inkey** serve a specificare il file contenente la chiave privata mentre con **-in** indichiamo il file da cui leggere i certificati e le chiavi private in formato PEM.

```
openssl pkcs12 -in httpd.pem -inkey httpdkey.pem -out httpd.p12 -export
```

Torniamo ad occuparci nuovamente di Tomcat dal momento che dobbiamo importare, all'interno del security keystore della Java Virtual Machine che è stata configurata per eseguire l'application server in questione, tutti i certificati che vogliamo che lui consideri affidabili. Il keystore a cui stiamo facendo riferimento è rintracciabile all'interno della cartella **\$JAVA_HOME/jre/lib/security/cacerts** e vi è associata, di default, la password *changeit*; grazie al comando keytool possiamo completare l'operazione nel seguente modo, andando ad importare il solo certificato del RootCA.

```
keytool -import -keystore $JAVA_HOME/jre/lib/security/cacerts -storepass  
changeit -alias rootca -file rootca.pem -trustcacerts
```

Quest'ultima parte di istruzioni che segue, riguarda la configurazione del proxy di Apache così da poter supportare comunicazioni SSL verso Tomcat; dobbiamo configurare le porte sulle quali Apache deve mettersi in ascolto ossia la porta 80 per l'HTTP e la 443 per l'HTTPS. Questo avviene andando a modificare con un editor di testi, il file `/etc/apache2/ports.conf`, in modo che contenga le due seguenti dichiarazioni:

```
Listen 80
```

```
Listen 443
```

Dobbiamo ora procedere col definire il comportamento di Apache nel caso di connessione sicura mediante HTTPS, andando a configurare il modulo `mod_ssl` (che è un'interfaccia tra OpenSSL ed Apache) definendo i certificati e gli algoritmi crittografici da usare durante la comunicazione sicura. Tutto questo, insieme alle direttive di reverse proxy e a quelle che avevamo visto in precedenza per scrivere le variabili `OPG_USERNAME` ed `OPG_USERROLES`, le abbiamo inserite nella sezione `<VirtualHost *:443>`, relativa alle connessioni HTTPS, all'interno del file di configurazione `/etc/apache2/sites-available/default`:

```
<VirtualHost *:443>
```

```
SSLEngine On
```

```
SSLProxyEngine On
```

```
SSLProtocol All
```

```
SSLCipherSuite ALL:!ADH:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP
```

```
SSLCertificateFile /etc/apache2/httpd.pem
```

```
SSLCertificateKeyFile /etc/apache2/httpdkey.pem
```

```
SSLCACertificateFile /etc/apache2/rootca.pem
```

```
SSLProxyCACertificateFile /etc/apache2/rootca.pem
```

```
SSLProxyMachineCertificateFile /etc/apache2/httpdcertkey.pem
```

```
SSLProxyCipherSuite ALL:!ADH:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP
```

```
<Location /testApp/>
```

```
Order allow,deny
```

```
Allow from all
```

```

RequestHeader set OPG_USERNAME utente1SSL
RequestHeader          set          OPG_USERROLES
[prova1,prova2,prova3,proveVarie]
ProxyPass https://localhost:8443/testApp/
ProxyPassReverse https://localhost:8443/testApp/
</Location>
</VirtualHost>

```

Illustriamo ora le varie direttive utilizzate. Mediante le direttive del modulo mod_ssl **SSLCipherSuite** e **SSLProxyCipherSuite**, utilizzata nelle connessione mediante proxy, andiamo a specificare quali sono i cifratori, o cipher, da usare nella fase di Handshaking di SSL. I possibili valori utilizzabili sono divisi in quattro gruppi:

1. **Key Exchange Algorithm:** RSA o varianti di Diffie-Hellman;
2. **Authentication Algorithm:** RSA, Diffie-Hellman, DSS o niente;
3. **Cipher/Encryption Algorithm:** DES, 3DES, RC4, RC2, IDEA o niente;
4. **MAC Digest Algorithm:** MD5, SHA o SHA1.

Qui di seguito riportiamo i possibili valori che possono essere selezionati per i cipher (**tabella 2.1**) [15]:

Tag	Descrizione
Key Exchange Algorithm:	
kRSA	RSA key exchange
kDhr	Diffie-Hellman key exchange with RSA key
kDHd	Diffie-Hellman key exchange with DSA key
kEDH	Ephemeral (temp.key) Diffie-Hellman key exchange (no cert)
Authentication Algorithm:	
aNULL	No authentication
aRSA	RSA authentication
aDSS	DSS authentication
aDH	Diffie-Hellman authentication
Cipher Encoding Algorithm:	
eNULL	No encoding
DES	DES encoding
3DES	Triple-DES encoding
RC4	RC4 encoding
RC2	RC2 encoding
IDEA	IDEA encoding
MAC Digest Algorithm:	
MD5	MD5 hash function
SHA1	SHA1 hash function

SHA	SHA hash function
Aliases:	
SSLv2	all SSL version 2.0 ciphers
SSLv3	all SSL version 3.0 ciphers
TLSv1	all TLS version 1.0 ciphers
EXP	all export ciphers
EXPORT40	all 40-bit export ciphers only
EXPORT56	all 56-bit export ciphers only
LOW	all low strength ciphers (no export, single DES)
MEDIUM	all ciphers with 128 bit encryption
HIGH	all ciphers using Triple-DES
RSA	all ciphers using RSA key exchange
DH	all ciphers using Diffie-Hellman key exchange
EDH	all ciphers using Ephemeral Diffie-Hellman key exchange
ADH	all ciphers using Anonymous Diffie-Hellman key exchange
DSS	all ciphers using DSS authentication
NULL	all ciphers using no encryption

Tabella 2.1: Valori possibili per i cipher

Interessante è la possibilità sia di specificare quali cifratori usare sia l'ordine di preferenza: per velocizzare tale scelta, sono state introdotte delle alias per certi gruppi di cifratori (SSLv2, SSLv3, TLSv1, EXP, LOW, MEDIUM e HIGH). I tag della tabella 2.1 possono essere uniti per formare quello che è il **cipher-spec**, ossia la lista dei cifratori che possono essere usati in una connessione sicura, mentre i possibili prefissi che possiamo andare ad utilizzare sono i seguenti:

- **nessuno**: aggiunge un cifratore alla lista;
- **+**: aggiunge il cifratore alla lista, posizionandolo nell'attuale posizione della lista;
- **-**: toglie il cifratore dalla lista (ma può essere aggiunto in seguito se necessario);
- **!**: distrugge dalla lista il cifratore (e non può essere ulteriormente aggiunto).

La stringa cipher-spec **ALL:!ADH:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP** che abbiamo specificato, va così intesa alla luce di quanto detto fino ad ora: vengono aggiunti tutti i cifratori per poi eliminare tutti quelli basati su Anonymous Diffie-Hellman, cioè quelli senza autenticazione, poi vengono aggiunti quelli basati su RC4 e RSA, seguiti nell'ordine da quelli con grado di robustezza alta (ossia quelli basati su 3DES), media (cifratori a 128 bit) e bassa (quelli non adatti all'esportazione e che sono basati su DES); alla fine della lista

vengono aggiunti tutti i cifratori della versione 2.0 di SSL e quelli adatti all'esportazione. E' possibile far uso anche di particolari tag, o alias, qualora fossimo interessati a creare connessione SSL utilizzando cifratori basati su RSA o DH (**tabella 2.2**):

Cipher-Tag	Protocol	Key Ex.	Auth.	Enc.	MAC	Type
RSA Ciphers:						
DES-CBC3-SHA	SSLv3	RSA	RSA	3DES(168)	SHA1	
DES-CBC3-MD5	SSLv2	RSA	RSA	3DES(168)	MD5	
IDEA-CBC-SHA	SSLv3	RSA	RSA	IDEA(128)	SHA1	
RC4-SHA	SSLv3	RSA	RSA	RC4(128)	SHA1	
RC4-MD5	SSLv3	RSA	RSA	RC4(128)	MD5	
IDEA-CBC-MD5	SSLv2	RSA	RSA	IDEA(128)	MD5	
RC2-CBC-MD5	SSLv2	RSA	RSA	RC2(128)	MD5	
RC4-MD5	SSLv2	RSA	RSA	RC4(128)	MD5	
DES-CBC-SHA	SSLv3	RSA	RSA	DES(56)	SHA1	
RC4-64-MD5	SSLv2	RSA	RSA	RC4(64)	MD5	
DES-CBC-MD5	SSLv2	RSA	RSA	DES(56)	MD5	
EXP-DES-CBC-SHA	SSLv3	RSA(512)	RSA	DES(40)	SHA1	export
EXP-RC2-CBC-MD5	SSLv3	RSA(512)	RSA	RC2(40)	MD5	export
EXP-RC4-MD5	SSLv3	RSA(512)	RSA	RC4(40)	MD5	export
EXP-RC2-CBC-MD5	SSLv2	RSA(512)	RSA	RC2(40)	MD5	export
EXP-RC4-MD5	SSLv2	RSA(512)	RSA	RC4(40)	MD5	export
NULL-SHA	SSLv3	RSA	RSA	None	SHA1	
NULL-MD5	SSLv3	RSA	RSA	None	MD5	
Diffie-Hellman Ciphers:						
ADH-DES-CBC3-SHA	SSLv3	DH	None	3DES(168)	SHA1	
ADH-DES-CBC-SHA	SSLv3	DH	None	DES(56)	SHA1	
ADH-RC4-MD5	SSLv3	DH	None	RC4(128)	MD5	
EDH-RSA-DES-CBC3-SHA	SSLv3	DH	RSA	3DES(168)	SHA1	
EDH-DSS-DES-CBC3-SHA	SSLv3	DH	DSS	3DES(168)	SHA1	
EDH-RSA-DES-CBC-SHA	SSLv3	DH	RSA	DES(56)	SHA1	
EDH-DSS-DES-CBC-SHA	SSLv3	DH	DSS	DES(56)	SHA1	
EXP-EDH-RSA-DES-CBC-SHA	SSLv3	DH(512)	RSA	DES(40)	SHA1	export
EXP-EDH-DSS-DES-CBC-SHA	SSLv3	DH(512)	DSS	DES(40)	SHA1	export
EXP-ADH-DES-CBC-SHA	SSLv3	DH(512)	None	DES(40)	SHA1	export
EXP-ADH-RC4-MD5	SSLv3	DH(512)	None	RC4(40)	MD5	export

Tabella 2.2: Lista tag speciali per RSA e DH

Le direttive **SSL**Engine e **SSL**ProxyEngine, per connessioni proxy, attivano o meno l'uso dell'engine per il protocollo SSL/TLS. Con la direttiva **SSL**Protocol abilitiamo, invece, i tipi di protocolli SSL che il client deve avere per poter effettuare la connessione con il server. I suoi possibili valori sono:

- SSLv2;
- SSLv3;
- TLSv1;
- All (abbreviazione per “+SSLv2 +SSLv3 +TLSv1”).

Con la direttiva **SSLCertificateFile** specifichiamo un certificato crittato di tipo PEM ed, opzionalmente, anche il corrispondente file contenente la chiave privata RSA o DSA; **SSLCertificateKeyFile** ci permette di specificare quale sia il file contenente la chiave privata crittata in formato PEM per il server. **SSLCACertificateFile** e **SSLProxyCACertificateFile** sono le direttive utilizzate per selezionare il file di certificati della Certification Authority rispettivamente dei client e dei server con cui Apache può instaurare una connessione sicura e fidata. Questi file (che nel nostro caso è rappresentato unicamente dal `/etc/apache2/rootca.pem`) specificati da queste ultime due direttive, sono semplicemente la concatenazione, in ordine di preferenza, dei vari certificati codificati in formato PEM. L’ultima direttiva che consideriamo è **SSLProxyMachineCertificateFile** il cui compito è quello di specificare il file che contiene i certificati e le chiavi usate per l’autenticazione del proxy: anche in questo caso, il file che andiamo a specificare è realizzato come una semplice concatenazione dei certificati codificati in formato PEM che vengono inseriti in ordine di preferenza. Attualmente però va detto che non vi è ancora il supporto, in Apache, per le chiavi private codificate. Veniamo finalmente a vedere come realizzare il file **httpdcertkey.pem**, specificato dall’ultima direttiva presa in considerazione a partire dal file **httpd.pem**:

```
cp httpd.pem httpdcertkey.pem
```

```
cat httpdkey.pem >> httpdcertkey.pem
```

Con queste ultime due istruzioni terminiamo i passaggi per la realizzazione della mutua autenticazione tra il web server e l’application server: se tutti i passi sono stati eseguiti correttamente, una volta che abbiamo provveduto a riavviare sia Apache che Tomcat, digitando dal browser web l’indirizzo **https://localhost/testApp/Prova**, verrà visualizzato il risultato della chiamata alla

Servlet che genererà, come suo output, una pagina HTML con il nome del principal dell'utente che l'ha richiamata (nel caso in questione e per come è stato configurato l'ambiente, il nome del utente è **utente1SSL**) e se appartiene o meno ai ruoli *prova1* e *proveVarie*. Specifichiamo che, ovviamente, quando sia l'autenticatore per OPG sia la mutua autenticazione tra Apache e l'application server saranno in produzione, il nome dell'utente ed i suoi ruoli verranno generati in modo dinamico e non statico come riportato qui a titolo di esempio.

2.6.2 Mutua autenticazione tra Apache e JBoss

Il procedimento per abilitare la mutua autenticazione tra Apache e JBoss non è tanto diverso da quanto abbiamo visto nel paragrafo precedente, anche se dobbiamo apportare delle modifiche al file di configurazione di questo nuovo application server, fermo restando quella relativa ad Apache. Per prima cosa dobbiamo abilitare, come nel caso di Tomcat, il Connector relativo alla porta 8443 all'interno del file di configurazione server.xml situato in \$JBOSS_HOME/server/[server_scelto]/deploy/jboss-web.deployer/:

```
<!-- Define a SSL HTTP/1.1 Connector on port 8443 -->
```

```
<Connector
```

```
    port="8443"
    address="{jboss.bind.address}"
    maxThreads="100"
    strategy="ms"
    maxHttpHeaderSize="8192"
    protocol="HTTP/1.1"
    SSLEnabled="true"
    emptySessionPath="true"
    scheme="https"
    secure="true"
    clientAuth="true"
    sslProtocol = "TLS"
    keystoreFile="/home/sed/tomcat.keystore"
    keystorePass="changeit"
    truststoreFile="/home/sed/server.truststore"
    truststorePass="changeit" />
```

A partire della versione **4.2.1** di JBoss è necessario che siano presenti, come nel precedente listato, gli attributi **protocol="HTTP/1.1"** e **SSLEnabled="true"** [16]: la spiegazione del loro utilizzo è che tale versione di JBoss utilizza, come JSP Servlet Container, la versione 6.0 di Tomcat. Trova quindi fondamento l'affermazione, che avevamo fatto in precedenza, relativa al fatto che dietro a molti application service ci sia la presenza di Apache Tomcat come JSP Servlet Container: volendo, quindi, abilitare la comunicazione in SSL all'interno di Tomcat 6.0 e, per quanto detto, le versioni di JBoss a partire dalla 4.2.1, è fondamentale ricordare di aggiungere questi due nuovi attributi all'interno del connector per l'HTTPS [17].

Abbiamo, infine, inserito all'interno del keystore e del truststore, specificati rispettivamente mediante **keystoreFile** e **truststoreFile**, i file contenenti i certificati digitali validi: nel caso di Tomcat 5.5, abbiamo usato il truststore interno alla distribuzione Java mentre, in questa nuova configurazione, abbiamo preferito utilizzarne uno diverso, chiamato **server.truststore**, in cui importare il certificato del RootCA, come nel caso precedente, nel seguente modo:

```
keytool -import -v -keystore server.truststore -storepass changeit -file rootca.pem
```

Il parametro **-v** sta ad indicare di visualizzare le informazioni del certificato in modalità più dettagliata (modalità "verbose"). Il file keystore **tomcat.keystore** è, invece, lo stesso file di cui abbiamo illustrato la generazione nel precedente sottoparagrafo. Qui di seguito, a titolo riassuntivo, elenchiamo tutti i files che abbiamo generato, specificando se sono per Apache o per Tomcat/JBoss.

Usati per la configurazione di Apache

- httpdcertkey.pem
- httpdkey.pem
- httpd.p12
- httpd.pem
- rootca.pem

- rootcakey.pem
- rootca.srl
- soareq.pem
- soakey.pem
- soa.pem

Usati per la configurazione di Apache Tomcat e JBoss

- tomcat.keystore
- tomcat.pem
- tomcatreq.pem
- server.truststore (solo a partire da Tomcat 6 o JBoss 4.2.1)

Capitolo 3

LoginModuleOPG

3.1 JBoss, JBossSX ed i Login Module

In questo capitolo presenteremo un aspetto, non secondario, affrontato nello studio di JBoss ossia quello riguardante i **Login Module**. La motivazione che ci ha spinti ad interessarci a questa tematica è subentrata analizzando la struttura di questo application server che è di tipo a livelli sovrapposti, fatta appunto di moduli interconnessi. La struttura altamente complessa di JBoss ci ha portati, quasi involontariamente, verso l'idea di provare a realizzare un Login Module che avesse le caratteristiche funzionali dello stesso autenticatore che avevamo realizzato per Tomcat (e per JBoss). In Tomcat possiamo trovare qualcosa che si avvicini ai login module ma lì, questi oggetti, vengono chiamati **Realm** ([18], [19]) e sono dei “database” di username/password che hanno il compito di identificare gli utenti validi di una generica applicazione (o insieme di applicazioni), oltre ad una enumerazione della lista dei ruoli associati a ciascun utente valido. Sebbene la specifica delle Servlet descriva un meccanismo portabile per le applicazioni mediante il quale è possibile dichiarare i requisiti di sicurezza (questo avviene mediante il file di deployment descriptor web.xml visto in precedenza), non esiste un'API portabile che definisca l'interfaccia tra un servlet container e le informazioni associate riguardanti l'utente e i suoi ruoli. In molti casi ci si accontenta di collegare il servlet container ad un esistente database di autenticazione o ad un meccanismo già presente all'interno del sistema in produzione. Tomcat definisce, quindi, un'interfaccia Java, chiamata

org.apache.catalina.Realm che può essere implementata mediante componenti plug-in per stabilire questo tipo di connessione. Esistono ben cinque plug-in standard atte a supportare connessioni con varie sorgenti di autenticazione:

1. **JDBCRealm**: permette di accedere alle informazioni di autenticazione, memorizzate in un database relazionale, mediante driver JDBC;
2. **DataSourceRealm**: permette di accedere alle informazioni di autenticazione, memorizzate in un database relazionale, mediante un datasource JNDI JDBC;
3. **JNDIRealm**: permette di accedere alle informazioni di autenticazione, memorizzate in un directory LDAP, mediante JNDI;
4. **MemoryRealm**: permette di accedere alle informazioni di autenticazione memorizzate in un oggetto collezione che viene letto ed inizializzato da un documento XML, il file tomcat-users.xml che abbiamo già incontrato in precedenza;
5. **JAASRealm**: permette di accedere alle informazioni di autenticazione mediante il framework **Java Authentication & Authorization Service (JAAS)**.

Possiamo, dunque, scrivere una nostra implementazione del Realm da integrare all'interno di Tomcat purché siano rispettati i seguenti passi:

1. implementare org.apache.catalina.Realm;
2. collocare la classe del nuovo realm all'interno della cartella \$CATALINA_HOME/server/lib;
3. dichiarare il realm all'interno del file di configurazione server.xml nel seguente modo:

```
<Realm className="... nome della classe per questa implementazione" ... altri attributi per questa implementazione... />
```

L'elemento <Realm> può trovarsi annidato all'interno di uno qualunque dei seguenti elementi che influisce in modo differente sulla visibilità dell'elemento Realm:

- **all'interno dell'elemento <Engine>**: il Realm è condiviso da tutte le applicazioni web su tutti gli host virtuali, a meno che non venga fatto un override da un oggetto Realm annidato all'interno di un elemento subordinato quale <Host> o <Context>;
- **all'interno di un elemento <Host>**: il Realm è condiviso da tutte le applicazioni web solo su questo preciso host virtuale a meno che non venga fatto un override da un oggetto Realm annidato all'interno di un elemento subordinato quale <Context>;
- **all'interno di un elemento <Context>**: il Realm è usato unicamente per quell'applicazione Web.

4. dichiarare il realm all'interno di un descrittore MBean.

Mettiamo da parte questa introduzione sui Realm per discutere della sicurezza all'interno JBoss. Abbiamo detto fin dal primo capitolo che per autorizzare l'accesso ad una qualunque risorsa è necessario, innanzi tutto, autenticare chi la richiede; la sicurezza è, dunque, una parte fondamentale di qualunque applicazione enterprise. Lo standard Java 2 Enterprise Edition definisce un semplice modello di sicurezza basato su ruoli per gli **Enterprise Java Bean (EJB)** e per i componenti Web. JBoss, invece, lo estende facendo uso del framework **JBossSX** che fornisce supporto sia per il modello di sicurezza di J2EE sia per l'integrazione di una security di tipo custom mediante il security proxy layer. L'architettura di JBossSX si basa sul JAAS che implementa il framework **Pluggable Authentication Module (PAM)** ed estende l'architettura del controllo degli accessi di Java 2 per supportare l'autorizzazione user-based. Le API di JAAS vennero rilasciate inizialmente come prodotto aggiuntivo per il JDK 1.3 ma dalla versione 1.4 ne è parte integrante; JBossSX fa uso soltanto della parte relativa all'autenticazione per implementare il modello dichiarativo di sicurezza di J2EE. Le classi che formano il core di JAAS possono essere divise in tre sotto categorie denominate **common**, **authentication** e **authorization**; riportiamo qui di seguito la lista delle classi relative alle prime due categorie perché sono quelle utilizzate da JBossSX [20].

Classi relative alla categoria **common**:

1. **javax.security.auth.Subject**
2. **java.security.Principal**

Classi relative alla categoria **authentication**:

1. **javax.security.auth.callback.Callback**
2. **javax.security.auth.callback.CallbackHandler**
3. **javax.security.auth.login.Configuration**
4. **javax.security.auth.login.LoginContext**
5. **javax.security.auth.spi.LoginModule**

Durante il processo di autenticazione, un **Subject** viene popolato mediante l'identità ad esso associato o quelli che vengono chiamati **Principal**; va notato che ci possono essere più principal associati ad un subject: un utente può avere il principal relativo al suo nome (es. *Mario Rossi*), il principal relativo al suo numero di matricola (*105*) e quello relativo al suo username (*mariorossi*), ciascuno dei quali permettono di distinguerlo da altri subject. Per recuperare i principal associati ad un subject possiamo utilizzare due metodi, il primo dei quali restituisce tutti i principal contenuti in un subject, mentre il secondo restituisce quei principal che sono istanze della classe *nomeClasse* o di una delle sue sotto classi. Questi metodi restituiscono un set vuoto se il subject non possiede almeno un principal:

- **public Set getPrincipals() {...}**
- **public Set getPrincipals(Class nomeClasse) {...}**

L'autenticazione di un subject richiede un login JAAS che possiamo riassumere nelle seguenti fasi:

1. un'applicazione istanzia un **LoginContext** e passa il nome della configurazione di login ed un **CallbackHandler** per popolare gli oggetti di

- tipo Callback che sono richiesti dalla configurazione dei Login Module;
2. il LoginContext consulta la configurazione per caricare tutti i Login Module specificati nella configurazione;
 3. l'applicazione invoca il metodo di login **LoginContext.login**;
 4. il metodo di login invoca tutti i Login Module caricati nei passi precedenti e ciascun modulo di login tenta di autenticare il subject invocando il metodo **handle** sul relativo CallbackHandler per ottenere informazioni necessarie al processo di autenticazione. L'informazione richiesta viene poi passata al metodo handle sotto forma di array di oggetti Callback. Se il processo va a buon fine, i Login Module associano i principal rilevanti e le credenziali al subject;
 5. il LoginContext restituisce all'applicazione lo status dell'autenticazione: positivo se si rientra dal metodo login ma in caso negativo è il metodo login che lancia un'eccezione di tipo **LoginException**;
 6. se l'autenticazione ha successo, l'applicazione recupera il subject autenticato mediante il metodo **LoginContext.getSubject**;
 7. alla fine del processo di autenticazione, tutti i principal e le relative informazioni associate al subject, mediante il metodo login, possono essere rimosse attraverso il metodo di logout **LoginContext.logout**.

Tutto il ciclo di vita di un Login Module viene pilotato dall'oggetto LoginContext e si compone di due fasi:

- il LoginContext crea ciascun Login Module che sia stato configurato utilizzando il proprio costruttore pubblico senza argomenti;
- ciascun Login Module è inizializzato mediante una chiamata al proprio metodo **initialize (public void initialize(Subject subject, CallbackHandler callbackHandler, Map sharedState, Map options))** ed è garantito il fatto che il subject sia non nullo;
- il metodo **login (boolean login() throws LoginException)** viene invocato per iniziare il processo di autenticazione. Per esempio, potrebbe essere richiesto all'utente di inserire la coppia [username, password] che verrebbe verificata confrontandola con i dati memorizzati all'interno di un database

LDAP; altre implementazioni potrebbero interfacciarsi a smart card o dispositivi biometrici. La validazione dell'identità dell'utente da parte di ciascuno Login Module viene definita la **fase 1** dell'autenticazione di JAAS. Se questo metodo lancia un'eccezione di tipo LoginException possiamo concludere che siamo davanti ad un fallimento del processo; se viene ritornato un valore booleano *true* il processo è andato a buon fine mentre, se ritorna *false*, il modulo di login deve essere ignorato;

- Se l'autenticazione del LoginContext ha complessivamente esito positivo, viene invocato il metodo **commit (boolean commit() throws LoginException)** di ciascun Login Module configurato. Se la fase 1 ha successo per un Login Module, allora il metodo commit continua con la **fase 2** associando i principal rilevanti, le credenziali pubbliche e/o quelle private al subject. Se invece la fase 1 non ha avuto successo, il commit rimuove qualunque stato di autenticazione memorizzato precedentemente, ad esempio gli username o le password. Se il metodo commit genera un'eccezione LoginException siamo nel caso in cui non è possibile portare a buon fine il processo; se il valore di ritorno è *true*, il processo ha avuto successo mentre se viene ritornato *false*, anche in questo caso il modulo di login deve essere ignorato;
- Se l'autenticazione del LoginContext ha complessivamente esito negativo, viene richiamato il metodo **abort (boolean abort() throws LoginException)** di ciascun Login Module che rimuove o distrugge qualunque stato di autenticazione che sia stato creato dai metodi login o initialize. L'impossibilità a completare la fase di abort è rappresentata dalla generazione dell'eccezione LoginException; se il valore di ritorno è *true*, il processo ha avuto successo mentre se viene ritornato *false*, anche in questo caso, il modulo di login deve essere ignorato;
- Per rimuovere lo stato di autenticazione dopo un login che ha avuto successo, l'applicazione invoca il metodo **logout (boolean logout() throws LoginException)** sul LoginContext e, di conseguenza, su ciascun Login Module. Tale metodo rimuove i principal e le credenziali originariamente associate con il subject durante l'operazione di commit. Facciamo presente che è buona norma che le credenziali vengano distrutte una volta rimosse, questo per motivi di sicurezza. Come gli altri metodi,

un errore nell'impossibilità di completamento del processo è dato dalla generazione dell'eccezione `LoginException`; se il valore di ritorno è `true`, il processo ha avuto successo mentre se viene ritornato `false`, anche in questo caso il modulo di login deve essere ignorato.

Quando un Login Module deve comunicare con l'utente per ottenere informazioni necessarie all'autenticazione, viene utilizzato un oggetto di tipo `CallbackHandler`; le applicazioni implementano l'interfaccia `CallbackHandler` e la passano al `LoginContext` che la reindirizza direttamente ai moduli login sottostanti. I moduli login utilizzano il `CallbackHandler` sia per raccogliere informazioni, o input, dagli utenti, quali password o PIN della smart card, sia per fornire informazioni all'utente circa lo status; permettendo all'applicazione di specificare il `CallbackHandler`, i sottostanti `LoginModule` restano indipendenti dai differenti modi in cui le applicazioni interagiscono con gli utenti: una possibile implementazione del `CallbackHandler` per un'applicazione che fa uso di GUI potrebbe essere quella di visualizzare una finestra di sollecito di input da parte dell'utente; per un'applicazione o ambienti non-GUI, quali potrebbero essere le applicazioni server, la `CallbackHandler` potrebbe semplicemente ottenere informazioni attraverso le API dell'application server. L'interfaccia `CallbackHandler` ha un metodo da implementare che è:

```
void handle(Callback[] callbacks)  
throws java.io.IOException,  
UnsupportedCallbackException;
```

`Callback` è un'interfaccia per la quale vengono fornite diverse implementazioni di default e viene utilizzata dai Login Module per richiedere informazioni necessarie al meccanismo di autenticazione: il Login Module passa un array di `Callback` direttamente al metodo `CallbackHandler.handle` durante la fase di login dell'autenticatore e se un `CallbackHandler` non sa come utilizzare l'oggetto `Callback` che gli è stato passato dal metodo `handle`, genera un'eccezione di tipo **`UnsupportedCallbackException`** in modo da abortire il processo di login. L'architettura di JBoss permette all'utente di costruirsi il proprio Login Module nel caso in cui, quelli presenti all'interno del framework di JBossSX, non siano

sufficienti o non adatti per particolari esigenze. Dobbiamo dire che JBossSX mette a disposizione dell'utente ben sei metodi per ottenere le informazioni associate ad un Subject:

1. **java.util.Set getPrincipals()**
2. **java.util.Set getPrincipals(java.lang.Class nomeClasse)**
3. **java.util.Set getPrivateCredentials()**
4. **java.util.Set getPrivateCredentials(java.lang.Class nomeClasse)**
5. **java.util.Set getPublicCredentials()**
6. **java.util.Set getPublicCredentials(java.lang.Class nomeClasse)**

Il framework JBossSX permette di accedere alle identità ed ai ruoli di un Subject mediante i metodi `getPrincipals()` e `getPrincipals(java.lang.Class)`, che abbiamo elencato poco prima, ed il cui modello di utilizzo è il seguente:

- le identità dell'utente (es. username, numero di matricola, ecc.) sono memorizzate come oggetti di tipo `java.security.Principal` in un insieme di Subject `Principal`. L'implementazione del `Principal` deve basare i confronti e l'uguaglianza sul nome del `principal`. Un'adatta implementazione è disponibile come classe **`org.jboss.security.SimplePrincipal`** mentre altre istanze di `Principal` possono essere aggiunte all'insieme di Subject `Principal` se richieste;
- i ruoli associati all'utente sono memorizzati nell'insieme `Principal` ma sono raggruppati in un insieme di ruoli etichettati mediante istanze di **`java.security.acl.Group`**. L'interfaccia `Group` definisce una collezione di `Principal` e/o `Group` ed è una sotto interfaccia di `java.security.Principal`. Un numero qualsiasi di insiemi di ruoli può essere assegnato al Subject benché, attualmente, il framework JBossSX faccia uso di due ben noti set di ruoli chiamati **`Roles`** e **`CallerPrincipal`**. Il `Group` chiamato `Roles` è una collezione di `Principal` per i ruoli etichettati come noti nel dominio dell'applicazione sotto il quale il Subject è stato autenticato; il `Group` `CallerPrincipal`, invece, consiste della singola identità `Principal` assegnata all'utente nel dominio dell'applicazione: se però un Subject non ha un `Group` `CallerPrincipal`, l'identità assunta all'interno dell'applicazione è la

stessa di quella dell'ambiente operativo o environment.

3.2 Moduli astratti di login

Per semplificare la corretta implementazione del modello d'uso del Subject, JBossSX include due moduli astratti di login il cui compito è quello di gestire il popolamento del Subject autenticato con un modello che fa rispettare il corretto uso del Subject:

- **org.jboss.security.auth.spi.AbstractServerLoginModule**
- **org.jboss.security.auth.spi.UsernamePasswordLoginModule**

3.2.1 AbstractServerLoginModule

Il più generico dei moduli è rappresentato dalla classe **AbstractServerLoginModule** che fornisce una concreta implementazione dell'interfaccia **javax.security.auth.spi.LoginModule**, offrendo metodi astratti per i task chiave specifici all'infrastruttura di sicurezza dell'ambiente di funzionamento. Qua di seguito riportiamo il sorgente del modulo (file **AbstractServerLoginModule.java**).

```
package org.jboss.security.auth.spi;
import java.security.Principal;
import java.security.acl.Group;
import java.util.Enumeration;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.lang.reflect.Constructor;
import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.login.LoginException;
import javax.security.auth.spi.LoginModule;
import org.jboss.logging.Logger;
import org.jboss.security.NestableGroup;
import org.jboss.security.SimpleGroup;
```

```

import org.jboss.security.SimplePrincipal;

public abstract class AbstractServerLoginModule implements LoginModule
{
    protected Subject subject;
    protected CallbackHandler callbackHandler;
    protected Map sharedState;
    protected Map options;
    protected Logger log;
    protected boolean useFirstPass;
    protected boolean loginOk;
    protected String principalClassName;
    protected Principal unauthenticatedIdentity;

    public void initialize(Subject subject, CallbackHandler callbackHandler,
        Map sharedState, Map options)
    {
        this.subject = subject;
        this.callbackHandler = callbackHandler;
        this.sharedState = sharedState;
        this.options = options;
        log = Logger.getLogger(getClass());
        log.trace("initialize");
        String passwordStacking = (String) options.get("password-stacking");
        if( passwordStacking != null &&
passwordStacking.equalsIgnoreCase("useFirstPass") )
            useFirstPass = true;

        principalClassName = (String) options.get("principalClass");
        String name = (String) options.get("unauthenticatedIdentity");
        if( name != null )
        {
            try
            {
                unauthenticatedIdentity = createIdentity(name);
                log.trace("Saw unauthenticatedIdentity="+name);
            }
            catch(Exception e)
            {
                log.warn("Failed to create custom unauthenticatedIdentity", e);
            }
        }
    }
}

```

```

    }
  }
}
public boolean login() throws LoginException
{
  log.trace("login");
  loginOk = false;
  if( useFirstPass == true )
  {
    try
    {
      Object identity = sharedState.get("javax.security.auth.login.name");
      Object credential = sharedState.get("javax.security.auth.login.password");
      if( identity != null && credential != null )
      {
        loginOk = true;
        return true;
      }
    }
    catch(Exception e)
    {
      log.error("login failed", e);
    }
  }
  return false;
}

```

```

public boolean commit() throws LoginException
{
  log.trace("commit, loginOk="+loginOk);
  if( loginOk == false )
    return false;

  Set principals = subject.getPrincipals();
  Principal identity = getIdentity();
  principals.add(identity);
  Group[] roleSets = getRoleSets();
  for(int g = 0; g < roleSets.length; g ++)
  {
    Group group = roleSets[g];

```

```

String name = group.getName();
Group subjectGroup = createGroup(name, principals);
if( subjectGroup instanceof NestableGroup )
{
    SimpleGroup tmp = new SimpleGroup("Roles");
    subjectGroup.addMember(tmp);
    subjectGroup = tmp;
}
Enumeration members = group.members();
while( members.hasMoreElements() )
{
    Principal role = (Principal) members.nextElement();
    subjectGroup.addMember(role);
}
}
return true;
}

public boolean abort() throws LoginException
{
    log.trace("abort");
    return true;
}

public boolean logout() throws LoginException
{
    log.trace("logout");
    Principal identity = getIdentity();
    Set principals = subject.getPrincipals();
    principals.remove(identity);
    return true;
}

abstract protected Principal getIdentity();
abstract protected Group[] getRoleSets() throws LoginException;

protected boolean getUseFirstPass()
{
    return useFirstPass;
}

```

```

protected Principal getUnauthenticatedIdentity()
{
    return unauthenticatedIdentity;
}

protected Group createGroup(String name, Set principals)
{
    Group roles = null;
    Iterator iter = principals.iterator();
    while( iter.hasNext() )
    {
        Object next = iter.next();
        if( (next instanceof Group) == false )
            continue;
        Group grp = (Group) next;
        if( grp.getName().equals(name) )
        {
            roles = grp;
            break;
        }
    }
    if( roles == null )
    {
        roles = new SimpleGroup(name);
        principals.add(roles);
    }
    return roles;
}

protected Principal createIdentity(String username)
    throws Exception
{
    Principal p = null;
    if( principalClassName == null )
    {
        p = new SimplePrincipal(username);
    }
    else
    {

```

```

        ClassLoader loader = Thread.currentThread().getContextClassLoader();
        Class clazz = loader.loadClass(principalClassName);
        Class[] ctorSig = {String.class};
        Constructor ctor = clazz.getConstructor(ctorSig);
        Object[] ctorArgs = {username};
        p = (Principal) ctor.newInstance(ctorArgs);
    }
    return p;
}
}

```

3.2.2 UsernamePasswordLoginModule

La seconda classe astratta per il login che possiamo utilizzare all'interno di un Login Module di tipo custom è rappresentata dal **UsernamePasswordLoginModule** che semplifica ulteriormente l'implementazione dei Login Module personalizzati, imponendo che l'username per l'identità dell'utente sia di tipo string-based e che la password per le credenziali dell'autenticazione sia basata su char[]. Tale modulo supporta anche la mappatura di utenti anonimi, caratterizzati dall'aver valore nullo sia all'username che alla password, verso Principal senza ruoli. Riportiamo anche in questo caso, il sorgente di questa ulteriore classe (file **UsernamePasswordLoginModule.java**).

```

package org.jboss.security.auth.spi;
import java.security.Principal;
import java.util.Map;
import java.util.HashMap;
import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.FailedLoginException;
import javax.security.auth.login.LoginException;
import org.jboss.security.Util;
import org.jboss.crypto.digest.DigestCallback;

```

```

public abstract class UsernamePasswordLoginModule extends
AbstractServerLoginModule
{
    private Principal identity;
    private char[] credential;
    private String hashAlgorithm = null;
    private String hashCharset = null;
    private String hashEncoding = null;
    private boolean ignorePasswordCase;

    public void initialize(Subject subject, CallbackHandler callbackHandler,
        Map sharedState, Map options)
    {
        super.initialize(subject, callbackHandler, sharedState, options);
        hashAlgorithm = (String) options.get("hashAlgorithm");
        if( hashAlgorithm != null )
        {
            hashEncoding = (String) options.get("hashEncoding");
            if( hashEncoding == null )
                hashEncoding = Util.BASE64_ENCODING;
            hashCharset = (String) options.get("hashCharset");
            if( log.isTraceEnabled() )
            {
                log.trace("Password hashing activated: algorithm = " + hashAlgorithm
                    + ", encoding = " + hashEncoding
                    + ", charset = " + (hashCharset == null ? "{default}" : hashCharset)
                    + ", callback = " + options.get("digestCallback")
                );
            }
        }
        String flag = (String) options.get("ignorePasswordCase");
        ignorePasswordCase = Boolean.valueOf(flag).booleanValue();
    }

    public boolean login() throws LoginException
    {
        if( super.login() == true )
        {
            Object username = sharedState.get("javax.security.auth.login.name");

```

```

if( username instanceof Principal )
    identity = (Principal) username;
else
{
    String name = username.toString();
    try
    {
        identity = createIdentity(name);
    }
    catch(Exception e)
    {
        log.debug("Failed to create principal", e);
        throw new LoginException("Failed to create principal: "+ e.getMessage());
    }
}
Object password = sharedState.get("javax.security.auth.login.password");
if( password instanceof char[] )
    credential = (char[]) password;
else if( password != null )
{
    String tmp = password.toString();
    credential = tmp.toCharArray();
}
return true;
}

super.loginOk = false;
String[] info = getUsernameAndPassword();
String username = info[0];
String password = info[1];
if( username == null && password == null )
{
    identity = unauthenticatedIdentity;
    super.log.trace("Authenticating as unauthenticatedIdentity="+identity);
}
if( identity == null )
{
    try
    {
        identity = createIdentity(username);
    }
}

```

```

    }
    catch(Exception e)
    {
        log.debug("Failed to create principal", e);
        throw new LoginException("Failed to create principal: "+ e.getMessage());
    }

    if( hashAlgorithm != null )
        password = createPasswordHash(username, password);
    String expectedPassword = getUsersPassword();
    if( validatePassword(password, expectedPassword) == false )
    {
        super.log.debug("Bad password for username="+username);
        throw new FailedLoginException("Password Incorrect/Password
Required");
    }
}
if( getUseFirstPass() == true )
{
    sharedState.put("javax.security.auth.login.name", username);
    sharedState.put("javax.security.auth.login.password", credential);
}
super.loginOk = true;
super.log.trace("User "" + identity + "" authenticated, loginOk="+loginOk);
return true;
}

protected Principal getIdentity()
{
    return identity;
}

protected Principal getUnauthenticatedIdentity()
{
    return unauthenticatedIdentity;
}

protected Object getCredentials()
{
    return credential;
}

```

```

}
protected String getUsername()
{
    String username = null;
    if( getIdentity() != null )
        username = getIdentity().getName();
    return username;
}

protected String[] getUsernameAndPassword() throws LoginException
{
    String[] info = {null, null};
    if( callbackHandler == null )
    {
        throw new LoginException("Error: no CallbackHandler available " +
            "to collect authentication information");
    }
    NameCallback nc = new NameCallback("User name: ", "guest");
    PasswordCallback pc = new PasswordCallback("Password: ", false);
    Callback[] callbacks = {nc, pc};
    String username = null;
    String password = null;
    try
    {
        callbackHandler.handle(callbacks);
        username = nc.getName();
        char[] tmpPassword = pc.getPassword();
        if( tmpPassword != null )
        {
            credential = new char[tmpPassword.length];
            System.arraycopy(tmpPassword, 0, credential, 0, tmpPassword.length);
            pc.clearPassword();
            password = new String(credential);
        }
    }
    catch(java.io.IOException ioe)
    {
        throw new LoginException(ioe.toString());
    }
    catch(UnsupportedCallbackException uce)

```

```

    {
        throw new LoginException("CallbackHandler does not support: " +
uce.getCallback());
    }
    info[0] = username;
    info[1] = password;
    return info;
}

protected String createPasswordHash(String username, String password)
{
    DigestCallback callback = null;
    String callbackClassName = (String) options.get("digestCallback");
    if( callbackClassName != null )
    {
        try
        {
            ClassLoader loader = Thread.currentThread().getContextClassLoader();
            Class callbackClass = loader.loadClass(callbackClassName);
            callback = (DigestCallback) callbackClass.newInstance();
            if( log.isTraceEnabled() )
                log.trace("Created DigestCallback: "+callback);
        }
        catch (Exception e)
        {
            if( log.isTraceEnabled() )
                log.trace("Failed to load DigestCallback", e);
            SecurityException ex = new SecurityException("Failed to load
DigestCallback");
            ex.initCause(e);
            throw ex;
        }
        HashMap tmp = new HashMap(options);
        tmp.put("javax.security.auth.login.name", username);
        tmp.put("javax.security.auth.login.password", password);
        callback.init(tmp);
    }

    String passwordHash = Util.createPasswordHash(hashAlgorithm,
hashEncoding,
hashCharset, username, password, callback);

```

```

        return passwordHash;
    }

    protected boolean validatePassword(String inputPassword, String
expectedPassword)
    {
        if( inputPassword == null || expectedPassword == null )
            return false;
        boolean valid = false;
        if( ignorePasswordCase == true )
            valid = inputPassword.equalsIgnoreCase(expectedPassword);
        else
            valid = inputPassword.equals(expectedPassword);
        return valid;
    }
    abstract protected String getUsersPassword() throws LoginException;
}

```

3.2.3 Scelta delle classi astratte per il login

In questo paragrafo cerchiamo di delineare quale sia la scelta migliore tra le due classi presentate nei due precedenti sottoparagrafi. Dobbiamo per prima cosa porre l'attenzione sulla variabile boolean **loginOk**: tale variabile deve essere settata al valore *true*, se la fase di login ha avuto successo, o a *false* in caso negativo da qualsiasi sottoclasse che effettua l'override del metodo **login()**. L'omissione nel settare correttamente questa variabile, provoca l'esecuzione del metodo **commit()** quando non dovrebbe essere aggiornato il Subject o il suo non aggiornamento quando, invece, dovrebbe essere fatto. Dopo questa precisazione molto importante, senza della quale chi cerca di creare un modulo custom di login può incorrere in errore, cerchiamo di affrontare il confronto delle due classi. Possiamo subito dire che la scelta se implementare `UsernamePasswordLoginModule` o `AbstractServerLoginModule` si può risolvere semplicemente analizzando sia la semantica dell'username, che è di tipo string-based, sia le credenziali dell'utente in questione che si vuole autenticare: se, infatti, la semantica string-based è corretta e valida, possiamo decidere di implementare la classe `UsernamePasswordLoginModule` mentre, in caso

contrario, la scelta migliore è `AbstractServerLoginModule`. Se siamo ancora nell'incertezza su quale classe estendere, possiamo risolvere la situazione andando sempre ad implementare quest'ultima classe. Quando vogliamo scrivere un modulo login custom bisogna prestare attenzione ad alcune precauzioni o regole, tenendo in considerazione quale, delle due classi, vogliamo implementare. Una prima precauzione è quella di implementare una delle due classi di cui abbiamo parlato, in modo da garantire che il proprio modulo di login fornisca la giusta informazione circa il Principal autenticato nella forma attesa dal security manager di JBossSX. Se abbiamo deciso di implementare la classe `AbstractServerLoginModule` dobbiamo provvedere a fare l'override dei seguenti metodi:

- **`void initialize(Subject, CallbackHandler, Map, Map)`**: se vogliamo personalizzare le opzioni da elaborare;
- **`boolean login()`**: per eseguire l'attività di autenticazione, stando attenti a settare correttamente la variabile `loginOk` al valore *true* se il login ha avuto successo, *false* in caso contrario;
- **`Principal getIdentity()`**: per restituire l'oggetto Principal per l'utente autenticato mediante il metodo `login()`;
- **`Group[] getRoleSets()`**: per restituire almeno un Group etichettato come Roles contenente i ruoli assegnati al Principal che è stato autenticato dal metodo `login()`. Un secondo Group di nome `CallerPrincipal` ha la funzione di fornire l'identità dell'applicazione dell'utente piuttosto che quella del dominio di sicurezza.

Se, invece, vogliamo implementare la classe `UsernamePasswordLoginModule`, dobbiamo considerare i seguenti metodi:

- **`void initialize(Subject, CallbackHandler, Map, Map)`**: se vogliamo personalizzare, anche in questo caso, le opzioni da elaborare;
- **`Group[] getRoleSets()`**: per restituire almeno un Group etichettato come Roles contenente i ruoli assegnati al Principal che è stato autenticato dal metodo `login()`. Un secondo Group di nome `CallerPrincipal` ha la funzione

di fornire l'identità dell'applicazione dell'utente piuttosto che quella del dominio di sicurezza;

- **String getUsersPassword()**: per restituire la password attesa per l'attuale username mediante il metodo getUsername(). Il metodo getUsersPassword() viene invocato all'interno del metodo login() dopo che il CallbackHandler restituisce la coppia [username, password].

3.3 LoginModuleOPG come alternativa ad OPGAuthenticator

Nei precedenti paragrafi abbiamo illustrato la tecnologia che sta alla base di un login module e delle regole per la realizzazione di uno di tipo custom. Ora possiamo finalmente entrare nel discorso dello nostro modulo login personalizzato. Durante questo nuovo studio, ci siamo chiesti se era possibile far funzionare un login module con un autenticatore un po' particolare ossia il **NonLoginAuthenticator** che possiamo attivare mediante la dichiarazione `<auth-method>NONE</auth-method>` nel file descrittore web.xml, in modo da spostare tutta la logica di controllo degli accessi sul modulo di login invece di farla gestire, come succede normalmente, all'autenticator così poco conosciuto e mai utilizzato in pratica. Una domanda che ci può venire posta è il perché della scelta di questo autenticatore e la risposta è presto data dal momento che è l'autenticator più semplice e che si limita a restituire sempre, come suo valore di uscita, il valore boolean *true*. Per completezza riportiamo il sorgente di questo autenticatore (**NonLoginAuthenticator.java**).

```
package org.apache.catalina.authenticator;
import java.io.IOException;
import org.apache.catalina.connector.Request;
import org.apache.catalina.connector.Response;
import org.apache.catalina.deploy.LoginConfig;

public final class NonLoginAuthenticator
    extends AuthenticatorBase {
    private static final String info =
        "org.apache.catalina.authenticator.NonLoginAuthenticator/1.0";
```

```

public String getInfo() {
    return (info);
}

public boolean authenticate(Request request,
                            Response response,
                            LoginConfig config)
    throws IOException {
    if (containerLog.isDebugEnabled())
        containerLog.debug("User authentication is not required");
    return (true);
}
}

```

Inizialmente avevamo pensato che bastasse semplicemente la restituzione del valore *true*, da parte dell'authenticator, per garantire l'esecuzione di un qualche modulo di login per l'autenticazione ma a seguito delle prove fatte mediante applicazioni di test, ci siamo resi conto che tale ipotesi era del tutto inesatta, portandoci a riconsiderare quanto avevamo fatto fino a quel momento per poi arrivare alla seguente conclusione: affinché un login module possa essere effettivamente richiamato, è necessario che l'authenticator esegua il metodo **authenticate** dell'interfaccia **Realm**. Nell'esempio qui riportato, utilizziamo il metodo **java.security.Principal authenticate(String username, String credentials)** poiché ci autenticiamo mediante [username, password]:

```

principal = context.getRealm().authenticate(username, password);

```

Se riguardiamo l'authenticator per OPG che abbiamo presentato nel capitolo precedente, in particolare a come avevamo creato l'oggetto principal che, ricordiamo, appartiene all'interfaccia **java.security.Principal**, noteremo che avevamo fatto uso del costruttore della classe **GenericPrincipal** e non del metodo **authenticate** come fanno tutti gli authenticatori standard. Mediante l'uso di quel costruttore di classe possiamo bypassare l'intervento di un qualsiasi modulo di login anche se configurato nei file di configurazione dell'application server in questione, andando così a svolgere il compito di un generico Realm o un Login Module, ossia quello dell'autorizzazione. Riportiamo il codice relativo all'utilizzo

del costruttore della classe `GenericPrincipal` che abbiamo utilizzato con `OPGAuthenticator`.

```
principal = new GenericPrincipal(context.getRealm(), username, password, roles);
```

Una volta che il `Principal` viene creato mediante o un qualche modulo di login esterno o il costruttore illustrato poco prima, un passo fondamentale che deve essere fatto da un `authenticator` è l'esecuzione del metodo **`register(Request request, Response response, java.security.Principal principal, String authType, String username, String password)`**, in modo da consentire la registrazione del `Principal` autenticato precedentemente: senza questo fondamentale passaggio, non viene permesso in alcun modo l'accesso alla risorsa protetta richiesta dall'utente. Riassumendo possiamo elencare cosa succede dal momento in cui una risorsa viene richiesta fino a quando ne viene consentito l'accesso:

1. l'utente fa richiesta di accesso ad una risorsa protetta, ad esempio tramite browser web;
2. l'application server in base all'autenticatore configurato nel descrittore `web.xml`, legge la coppia `[username, password]` fornita dall'utente (nel caso di `Basic` o `Form authentication`) o recupera il certificato digitale (nel caso di `Client-Cert`);
3. l'autenticatore richiama a sua volta, il `Realm` o il `Login Module` configurato per gestire i dati di accesso forniti dall'utente;
4. il `Realm` o il `LoginModule` verifica se effettivamente i dati forniti dall'utente sono validi. In caso di validità, viene restituito un oggetto `Principal` altrimenti un oggetto nullo. Il controllo ritorna poi all'`authenticator` al termine della validazione;
5. qualora sia stato restituito un oggetto `Principal` non nullo, questo viene registrato insieme al tipo di autenticazione nella `request` della sessione corrente e l'utente, a questo punto, è riconosciuto come avente i diritti per accedere alla risorsa da lui richiesta; l'`authenticator` termina restituendo valore booleano `true`. Qualora non avvenga la registrazione del `Principal`, all'utente non viene dato accesso a quanto da lui richiesto e l'autenticatore

termina con valore *false*;

6. il controllo ritorna, in ogni caso, all'applicazione che provvede a visualizzare o meno l'output della risorsa richiesta.

Alla luce di quanto detto fino ad ora, possiamo concludere che il NonLoginAuthenticator non è un autenticatore da usare insieme ad alcun Realm o Login Module ed è, quindi, necessario utilizzarne altri, a seconda delle esigenze del caso. In **figura 3.1** mostriamo la pagina di errore che viene mostrata da JBoss quando tentiamo di accedere ad una risorsa protetta abbinando il NonLoginAuthenticator ed un qualunque Login Module o Realm.



Figura 3.1: Errore HTTP 403 nel caso in cui venga fatto uso di NonLoginAuthenticator unito ad un qualunque modulo login.

Qui di seguito andiamo a riportare la bozza del modulo login che abbiamo ideato, il cui nome è **LoginModuleOPG (LoginModuleOPG.java)**, e che abbiamo provato in unione ad autenticator di tipo Basic.

```
package org.jboss.security.auth.spi;
import java.security.Principal;
import java.security.acl.Group;
import java.util.Map;
import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.login.LoginException;
import javax.security.jacc.PolicyContext;
import javax.servlet.http.HttpServletRequest;
import java.util.StringTokenizer;
import org.jboss.security.SimpleGroup;
import org.jboss.security.SimplePrincipal;
import org.jboss.security.auth.spi.AbstractServerLoginModule;
```

```

public class LoginModuleOPG extends AbstractServerLoginModule {
String nome = null;
String role = null;

    public void initialize(Subject arg0, CallbackHandler arg1, Map arg2, Map
arg3) {
        this.subject = arg0;
        this.callbackHandler = arg1;
        this.sharedState = arg2;
        this.options = arg3;
    }
    public boolean login() throws LoginException {
        this.loginOk = true;
        return true;
    }
    public boolean abort() throws LoginException {
        return true;
    }
}

public boolean commit() throws LoginException {
String userRole = null;
    try {
        HttpServletRequest request = (HttpServletRequest)
PolicyContext.getContext("javax.servlet.http.HttpServletRequest");
        if( request != null) {
            nome = request.getHeader("OPG_USERNAME");
            role = request.getHeader("OPG_USERROLES");
            if(role != null) userRole = (String) role;
        }
    }catch( Exception e) {
        return false;
    }
    if(userRole == null) return(false); // esce
    SimpleGroup gruppoUtente = new SimpleGroup("CallerPrincipal");
    SimpleGroup gruppoRuoli = new SimpleGroup("Roles");
    SimpleGroup roles = new SimpleGroup("Roles");
    if(role != null) {
        if(role.charAt(0) == '[' && role.charAt(role.length()-1) == '']){
            StringTokenizer tokenizer = new
StringTokenizer(role, ",");

```

```

        while(tokenizer.hasMoreTokens()) {
            String roleName = tokenizer.nextToken();
            if(roleName.startsWith("[") roleName =
            roleName.substring(1);
            if(roleName.endsWith("]")) roleName =
            roleName.substring(0, roleName.length() - 1);
            roles.addMember(new
            SimplePrincipal(roleName));
        }
    }
    gruppoRuoli=roles;
    gruppoUtente.addMember(new PrincipalOPG(nome));
    this.subject.getPrincipals().add(gruppoUtente);
    this.subject.getPrincipals().add(gruppoRuoli);
    return true;
}
else {
    return false;
}
}
public boolean logout() throws LoginException {
    this.subject.getPrincipals().clear();
    this.subject.getPublicCredentials().clear();
    this.subject.getPrivateCredentials().clear();
    return true;
}
protected Principal getIdentity() {
    return null;
}
protected Group[] getRoleSets() throws LoginException {
    return null;
}
}
}

```

Notiamo subito che questo modulo di login implementa la classe AbstractServerLoginModule a cui abbiamo accennato in precedenza: potevamo utilizzare anche l'altra classe astratta ma abbiamo notato che a sua volta la classe UsernamePasswordLoginModule implementa AbstractServerLoginModule. Abbiamo quindi ritenuto più vantaggioso e più semplice estendere la classe

AbstractServerLoginModule. Notiamo sia la presenza dei quattro metodi di cui abbiamo parlato e di cui dobbiamo fare l'override quando vogliamo implementare questa classe astratta sia il settaggio dell'importante variabile *loginOk*, impostata a *true* nel metodo login().

All'interno del LoginModuleOPG abbiamo utilizzato anche una classe Bean chiamata **PrincipalOPG (PrincipaOPG.java)** che abbiamo utilizzato per memorizzare il nome dell'utente e di cui riportiamo il sorgente.

```
package org.jboss.security.auth.spi;
import java.security.Principal;
public class PrincipalOPG implements Principal {
    private String name = null;
        public PrincipalOPG(String nameM) {
            this.name = nameM;
        }

        public String getName() {
            return this.name;
        }
    }
}
```

Abbiamo accennato che abbiamo provato il LoginModuleOPG in unione con l'autenticazione di tipo Basic: qui di seguito andiamo ad esporre tale prova di funzionamento. Per prima cosa dobbiamo abilitare l'autenticazione di tipo Basic all'interno del deployment descriptor web.xml:

```
<auth-method>BASIC</auth-method>
```

Dobbiamo provvedere ad inserire le classi **LoginModuleOPG.class** e **PrincipalOPG.class** all'interno dell'archivio JAR **jbosssx.jar**, che si trova in \$JBOSS_HOME/server/[server_scelto]/lib, precisamente nel percorso **jbosssx.jar\org\jboss\security\auth\spi**. Una volta che abbiamo terminato questo passaggio, da farsi ovviamente una volta sola e non tutte le volte che vogliamo usare questo modulo login, dobbiamo modificare il file di configurazione di JBoss login-config.xml, andando a definire il comportamento nel caso in cui il dominio di sicurezza sia, come nella nostra applicazione di test, *proveVarie* (tag

`<application-policy name="proveVarie">`) specificando che si vuole utilizzare la classe `LoginModuleOPG` (tag `<login-module code="org.jboss.security.auth.spi.LoginModuleOPG" flag = "required">`) come nel frammento di istruzioni qui riportate. Come possiamo vedere, il nostro modulo di login non viene richiamato mediante parametri aggiuntivi come invece succede per gli altri Login Module di cui abbiamo parlato nel precedente capitolo:

```
<application-policy name="proveVarie">
  <authentication>
    <login-module code="org.jboss.security.auth.spi.LoginModuleOPG"
      flag = "required">
    </login-module>
  </authentication>
</application-policy>
```

Terminata anche questa fase preparatoria, possiamo avviare sia Apache in modo che provveda a popolare la coppia di variabili `[OPG_USERNAME, OPG_USERROLES]` sia l'application server JBoss, andando successivamente a richiamare tramite browser web, ad esempio Mozilla Firefox, l'URL della nostra applicazione di test **`http://localhost/testApp/Prova`** andando così ad ottenere, in risposta, la visualizzazione del pop-up che ci invita ad immettere la coppia dati `[username, password]` (**figura 3.2**). Facciamo presente che la comparsa di tale finestra di richiesta dati è dovuta all'autenticatore di tipo Basic e che fino a questo punto non è ancora entrato in funzione il modulo di login. Possiamo eliminare il pop-up della figura precedente unicamente andando ad agire sul codice dell'autenticatore, come abbiamo fatto con `OPGAuthenticator`.

Per verificare l'effettivo funzionamento del `LoginModuleOPG`, inseriamo dei dati palesemente fasulli, non presenti all'interno di alcun database di accessi dell'application server all'interno del pop-up della figura 3.2. Come risultato di questo inserimento dati intenzionalmente errato, otteniamo l'accesso alla risorsa: questo lo si giustifica dicendo che i dati di accesso che vengono validati non sono quelli letti attraverso il pop-up ma sono quelli recuperati dall'header HTTP che è stato generato proprio da Apache. JBoss, come possiamo vedere in **figura 3.3**, autentica l'utente permettendogli di accedere alla risorsa protetta, che nel caso specifico è una servlet, che visualizza in output sia il nome dell'utente letto

dall'header (voce **remoteUser**), ossia *utente1* e non quello inserito nel pop-up visto in figura 3.2 cioè *utentesconosciuto*, sia dandoci conferma che i ruoli dell'utente autorizzato sono effettivamente *prova1* e *proveVarie*. Questa prova di funzionamento ci ha permesso sia di dimostrare l'effettivo funzionamento del LoginModuleOPG sia di confermare il fatto che il NonLoginAuthenticator non può essere utilizzato insieme ad alcun modulo di login, essendo privo di quei due metodi, a cui abbiamo accennato in questo paragrafo, necessari sia ad invocare un qualunque Login Module per creare un Principal sia per registrarlo all'interno della sessione corrente.

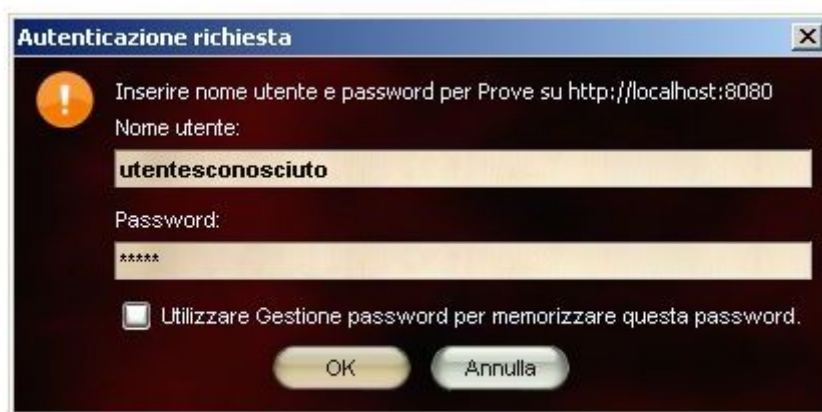


Figura 3.2: Richiesta di dati riempita volontariamente con una coppia di dati volontariamente non validi.

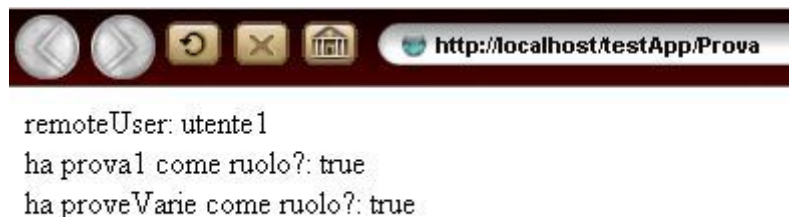


Figura 3.3: Risultato dell'esecuzione della Servlet di prova

Sebbene quanto illustrato fino ad ora circa la creazione del Login Module in questione possa risultare facile, quasi banale, non sono mancati problemi prima di arrivare ad una soluzione funzionante. Una prima difficoltà con cui abbiamo avuto modo di scontrarsi è riuscire ad accedere alla request http: nel caso dell'authenticator, l'oggetto Request era già disponibile in quanto la struttura

stessa degli autenticatori lo mette a disposizione. Ricordiamo che quando l'application server invoca un autenticatore, lo invoca passandogli la request http, la response http ed un oggetto LoginConfig: in quel caso è immediato poter utilizzare i metodi appositi per leggere i valori dalla request http ma nel caso di un Login Module, la situazione è differente, non viene passata dall'esterno alcuna variabile request. Per risolvere questo inconveniente, abbiamo adoperato **Object getContext(String)** della classe **javax.security.jacc.PolicyContext** che appartiene al **Java Authorization Contract for Containers (JACC)**: JACC è parte della specifica Java 1.4 e definisce un modo per le servlet e per i Container EJB per rendere sicuri con un modello di sicurezza basato su ruoli, sia le risorse web che gli EJB.

```
HttpServletRequest request = (HttpServletRequest)  
PolicyContext.getContext("javax.servlet.http.HttpServletRequest");
```

Una volta in possesso della request http, abbiamo potuto accedere regolarmente alla lettura delle informazioni delle due variabili di interesse mediante il metodo `getHeader` già visto in precedenza:

```
nome = request.getHeader("OPG_USERNAME");  
role = request.getHeader("OPG_USERROLES");
```

Una cosa molto interessante che abbiamo notato nello studio di JBoss, è che tutti i login module, forniti a corredo di questo application server, che utilizzano la coppia [username, password] per l'autenticazione, quando effettuano la fase di commit, ossia quando viene eseguito il metodo `commit()`, passano alla struttura del Principal soltanto la lista dei ruoli associati e non anche l'username di chi ha richiesto l'accesso; per l'esattezza ci siamo accorti che il nome dell'utente viene passato ma viene preso solamente quello letto dal pop-up di richiesta di immissione dati, nel caso ad esempio di utilizzo insieme ad autenticatore Basic. Non c'è nulla di strano nel recuperare l'username dal pop-up in una situazione classica ma questa situazione si presta male nel nostro caso, dal momento che il nostro intento è riuscire ad ottenere l'autenticazione/autorizzazione mediante i dati che provengono da Apache e non attraverso quelli forniti in input dall'utente.

Ci siamo posti il problema di scoprire il motivo di questo non passaggio del valore dell'username, benché venisse letto dalla request http, e se era possibile intervenire, in qualche modo, per risolverlo ed ottenere l'effetto voluto. La soluzione a questi interrogativi non si è fatta attendere perché siamo arrivati a capire che il problema risiede nel fatto che, di base, non viene creata una particolare lista, benché sarebbe meglio parlare di oggetto **SimpleGroup (org.jboss.security.SimpleGroup)**. Per bypassare l'username che viene letto dal pop-up dobbiamo creare un SimpleGroup **CallerPrincipal** in modo da inserire, come valore, proprio l'username, cioè l'identità, dell'utente che cerca di accedere alla risorsa protetta. Con i ruoli invece non abbiamo problemi, vengono letti senza difficoltà ed inseriti, a loro volta, in un ulteriore SimpleGroup chiamato **Roles**. Riassumendo, abbiamo risolto il nostro problema creando un SimpleGroup per l'identità dell'utente così da passarlo, insieme a quello generato per i ruoli, al Principal. Possiamo anche rispondere al perché i Login Module, che fanno uso di [username, password], non utilizzano il SimpleGroup CallerPrincipal e la spiegazione è che il framework JBossSX fa l'assunzione che l'identità assunta all'interno dell'applicazione sia la stessa di quella dell'ambiente operativo. Qui di seguito riportiamo i comandi necessari alla creazione del SimpleGroup CallerPrincipal e al suo popolamento mediante l'utilizzo del Bean PrincipalOPG: senza queste istruzioni, non possiamo dunque forzare l'username ad assumere valori diversi da quelli che riceve in fase di input dall'authenticator. Vediamo, inoltre, che il SimpleGroup CallerPrincipal è aggiunto al Principal insieme a SimpleGroup Roles mentre nome è la variabile che contiene l'username letto dal request http:

```
SimpleGroup gruppoUtente = new SimpleGroup("CallerPrincipal");  
SimpleGroup gruppoRuoli = new SimpleGroup("Roles");  
...  
// qua in mezzo c'è il codice per popolare il SimpleGroup gruppoRuoli con i  
ruoli letti dalla Request HTTP  
...  
gruppoUtente.addMember(new PrincipalOPG(nome));  
  
this.subject.getPrincipals().add(gruppoUtente);  
this.subject.getPrincipals().add(gruppoRuoli);
```

Con queste istruzioni chiudiamo la trattazione del LoginModuleOPG: abbiamo visto che può essere un'alternativa possibile all'utilizzo di OPGAuthenticator benché, come abbiamo detto, possiamo utilizzare questo modulo di login unicamente insieme ad un autenticatore purché sia in grado sia di demandare la fase di autorizzazione ad un Login Module sia di effettuare la registrazione del principal nella sessione. Questi due requisiti scartano dalla lista degli autenticatori sia OPGAuthenticator, dal momento che è lui stesso ad autenticare le credenziali di un utente e non fa uso di alcun Login Module, sia NonLoginAuthenticator che, come abbiamo fatto vedere, non è in grado di passare il controllo a nessun modulo di login e tanto meno di registrare un principal.

Capitolo 4

Conclusioni

In questo lavoro di tesi abbiamo affrontato il problema della sicurezza, nello specifico lo studio e la realizzazione di un nuovo autenticatore, che abbiamo chiamato OPGAuthenticator, sia per Tomcat sia per JBoss in modo da supportare il sistema Open Portal Guard. Abbiamo studiato anche la possibilità di realizzazione di un login module specifico per l'application server JBoss, come alternativa possibile all'autenticator OPGAuthenticator. Per entrambe le soluzioni il vincolo che dovevamo rispettare era quello di poter leggere i dati provenienti da un sistema di Single Sign On che permette l'autenticazione tramite due tipi di smart card di interesse per le Pubbliche Amministrazioni Italiane, quali:

- Carta d'Identità Elettronica (CIE);
- Carta Nazionale dei Servizi (CNS).

Abbiamo inoltre studiato come aumentare la sicurezza di Open Portal Guard introducendo la mutua autenticazione SSL tra Apache HTTP Server e l'application server Tomcat/JBoss. A conclusione del lavoro svolto abbiamo provato l'autenticatore in una situazione reale, pronto per essere messo in produzione. Per l'occasione abbiamo quindi abbandonato la configurazione utilizzata durante la fase di sviluppo, ossia quella con una sola macchina, in favore di una a tre macchine, ciascuna con funzioni differenti:

- un client su cui era installato il software di riconoscimento della smart card, in ambito sia Linux che Windows;
- un server Linux su cui girava Apache HTTP Server che gestiva la lettura dei dati provenienti dal certificato digitale della smart card;
- un ultimo server, sia Linux che Windows 2000/XP, su cui era installato l'application server.

Abbiamo prestato particolare attenzione al pieno funzionamento con JBoss. Su quest'ultima macchina abbiamo installato un'applicazione di prova protetta mediante Open Portal Guard: siamo andati a verificare il comportamento dell'applicazione sia con la smart card inserita nell'apposito lettore ottenendo, come risultato, che l'applicazione veniva eseguita correttamente sia in sua assenza avendo come risultato che l'application server non ci consentiva l'accesso alla risorsa protetta, segno del corretto funzionamento dell'autenticatore. Le stesse prove sono poi state fatte con il modulo di login, ottenendo gli stessi risultati anche se possiamo dire che quest'ultima soluzione potrebbe essere più onerosa dal punto di vista computazionale perché viene introdotto uno strato software in più rispetto al gestire il processo di autenticazione ed autorizzazione a livello di autenticatore.

Qui di seguito riportiamo lo schema complessivo del sistema (figura 4.1) in cui sono evidenziate le tre zone (client, web server ed application server). L'utente interagisce col sistema mediante la smart card e direttamente col browser web. Nel livello intermedio troviamo il web server, ossia Apache HTTP Server, al cui interno sono installati vari moduli e la particolare applicazione sviluppata dal SED del Comune di Grosseto. A questo livello avviene il look-up all'interno del server LDAP dei dati letti dalla smart card e la conseguente generazione, nella request http, delle due importanti variabili OPG_USERNAME ed OPG_USERROLES che vengono successivamente elaborate dall'ultimo livello del sistema, quello occupato dall'application server. Qui troviamo l'OPGAuthenticator oppure la soluzione composta dal LoginModuleOPG unito ad un authenticator, ad esempio, quello di tipo Basic. In entrambi i casi è a questo punto del sistema che vengono validati i dati letti dalla smart card e garantito (o meno) l'accesso alle risorse protette mediante Open Portal Guard.

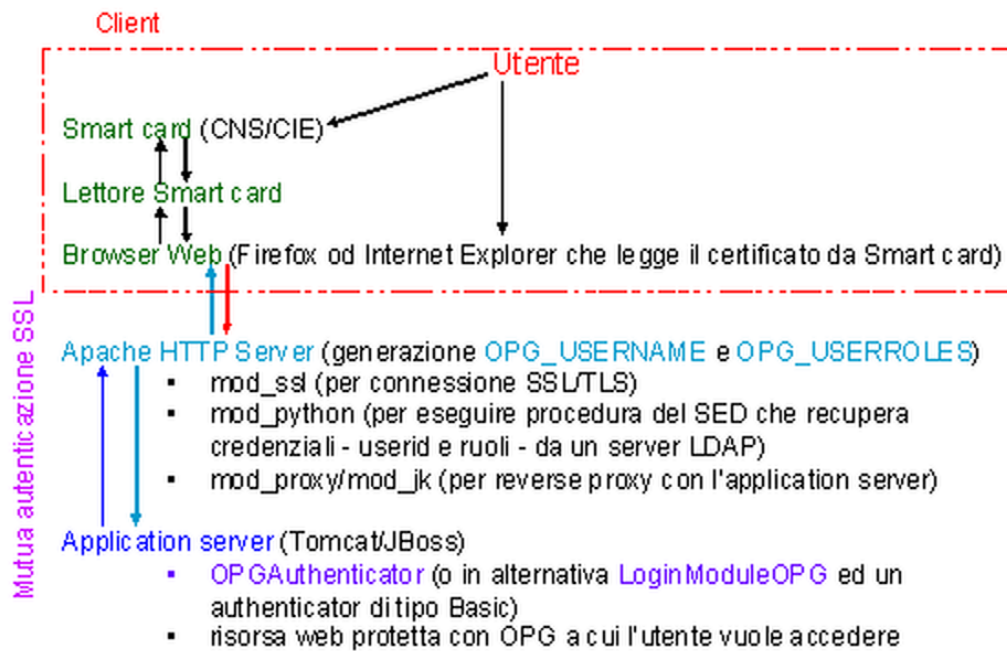


Figura 4.1: Schema riassuntivo del sistema Open Portal Guard

Bibliografia

- [1] “Authentication vs. Authorization” in <http://www.duke.edu/~rob/kerberos/authvauth.html>
- [2] “Traditional Authentication Method” in <http://www.duke.edu/~rob/kerberos/dumbauth.html>
- [3] Bud P. Bruegger, “Open Portal Guard Home Page” in: <http://openportalguard.sourceforge.net/>
- [4] “Codice Amministrazione Digitale” in: http://www.cnipa.gov.it/site/_files/Codice%20Amministrazione%20Digitale_02.pdf
- [5] Massimo Canducci, “Apogeeonline – Servlet” in: <http://www.apogeeonline.com/webzine/2002/06/27/14/200206271401>
- [6] “Apache Module mod_proxy” in: http://httpd.apache.org/docs/2.0/mod/mod_proxy.html
- [7] “The Apache Tomcat Connector – Webserver HowTo – Apache HowTo” in: http://tomcat.apache.org/connectors-doc/webserver_howto/apache.html
- [8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, “Hypertext Transfer Protocol – HTTP/1.1” in: <http://www.ietf.org/rfc/rfc2616.txt> (ultimo aggiornamento Giugno 1999)
- [9] J. Franks, P. Hallam-Baker, J. Lawrence, P. Leach, A. Luotonen, L. Stewart, “HTTP Authentication: Basic and Digest Access Authentication” in: <http://www.ietf.org/rfc/rfc2617.txt> (ultimo aggiornamento Giugno 1999)
- [10] W. Stallings, “Crittografia e sicurezza delle reti”, McGrawHill, 2 edizione, Gennaio 2007
- [11] Jos Koetsier, “Installation of the Delegation Issuing Service in a Virtual Organisation” in:

http://labserv.nesc.gla.ac.uk/projects/dyvose/dis_install/dis_install.html (ultimo aggiornamento 19 Giugno 2006)

[12] “*OpenSSL: Documents*” in ***<http://www.openssl.org/docs/>***

[13] “*keytool – Key and Certificate Management Tool*” in: ***<http://java.sun.com/javase/6/docs/technotes/tools/windows/keytool.html>***

[14] “*The Apache Tomcat 5.5 Servlet/JSP Container: SSL Configuration HOW-TO*” in: ***<http://tomcat.apache.org/tomcat-5.5-doc/ssl-howto.html>***

[15] “*mod_ssl – Apache HTTP Server*” in: ***http://httpd.apache.org/docs/2.2/mod/mod_ssl.html***

[16] “*JBossWiki: SSLSetup*”, in ***<http://wiki.jboss.org/wiki/SSLSetup>*** (ultimo aggiornamento 14 Febbraio 2008)

[17] “*Apache Tomcat 6.0 – SSL Configuration HOW-TO*” in: ***<http://tomcat.apache.org/tomcat-6.0-doc/ssl-howto.html>***

[18] “*The Apache Tomcat 5.5 Servlet/JSP Container: Realm Configuration HOW-TO*” in ***<http://tomcat.apache.org/tomcat-5.5-doc/realm-howto.html>***

[19] “*Apache Tomcat 6.0 – Realm Configuration HOW-TO*” in: ***<http://tomcat.apache.org/tomcat-6.0-doc/realm-howto.html>***

[20] “*Chapter 8. Security on JBoss*” in: ***<http://docs.jboss.org/jbossas/jboss4guide/r3/html/ch8.chapter.html>***

FORTEZZA





Per le figure di pagina 3, 131 e 132 ringrazio, in prima persona, il **Dott. Fabio Torchio** e la **Dott.ssa Anna Maria Emanuele** della Soprintendenza Beni Storici Artistici ed Etnoantropologici per le provincie di Siena e Grosseto. Le immagini sono tratte da Cesare Ripa, *Nova iconologia*, Padova 1618, ad vocem.