UNVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
CURSO DE ESPECIALIZAÇÃO EM TECNOLOGIA JAVA

CARLOS SAMUEL FRANCO CABRAL

TUTORIAL DA API JAVA AUTHENTICATION AND AUTHORIZATION SERVICE (JAAS)

MONOGRAFIA DE ESPECIALIZAÇÃO

CURITIBA - PR
JUNHO 2011

CARLOS SAMUEL FRANCO CABRAL

TUTORIAL DA API JAVA AUTHENTICATION AND AUTHORIZATION SERVICE (JAAS):

Monografia de Especialização apresentada ao Departamento Acadêmico de Informática, da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do título de “Especialista em Tecnologia Java” -
Orientador: Prof. Maurício Correia Lemes Neto

CURITIBA - PR

2011

DEDICATÓRIA

A todos que tem curiosidade e ousadia para desbravar os caminhos ainda desconhecidos.

AGRADECIMENTOS

Agradeço a todos aqueles que direta ou indiretamente permitiram que eu fizesse e concluísse essa especialização.

RESUMO

CABRAL, Carlos. Tutorial da API Java Authentication and Authorization Service (JAAS). 2011. 87 f. Monografia (Especialização em Tecnologia Java) – Programa de Pós-Graduação em Tecnologia, Universidade Tecnológica Federal do Paraná.
Curitiba, 2011.

Será apresentado um breve resumo sobre Java Security para depois apresentar um estudo mais aprofundado sobre a API JAAS da plataforma Java. Também será feito uma breve apresentação de como o Servidor TOMCAT implementa esta solução. Utilizando a implementação do JAAS no Tomcat será feito um exemplo recuperando as informações do usuário desde um banco de dados relacional.

Palavras-chave: (JAAS. Java Authentication. Java Authorization. Java Security. JAASRealm.)

ABSTRACT

CABRAL, Carlos. Java Authentication and Authorization Service (JAAS) Tutorial. 2011. 87 f. Monografia (Especialização em Tecnologia Java) – Programa de Pós-Graduação em Tecnologia, Universidade Tecnológica Federal do Paraná.
Curitiba, 2011.

It will produce a brief overview regarding the Java Security and after it will make a more detailed study of the API JAAS. Also it will present how the TOMCAT implements the JAAS (JAASRealm) and get the user data from a database.

Keywords: (JAAS. Java Authentication. Java Authorization. Java Security. JAASRealm.)

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AES	Advanced Encryption Standard
API	Interface de Programação de Aplicações (Application Programming Interfaces)
BBC	British Broadcasting Corporation
CERT	Centro de Estudos, Resposta e Tratamentos de Incidentes de Segurança no Brasil
CPF	Cadastro de Pessoa Física
FMI	Fundo Monetário Internacional
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
JAAS	Java Authentication and Authorization service
JAR	Java Archive
JCA	Java Cryptography Architecture
JDBC	Java Database Connectivity
JNDI	Java Naming Directory Interface
JRE	Java Runtime Environment
JVM	Máquina Virtual do Java (Java Virtual Machine)
LDAP	Lightweight Directory Access Protocol
MD5	Message-Digest algorithm 5
OWASP	The Open Web Application Security Project
PKI	Infra-estrutura de Chaves Públicas (Public Key Infrastructure)
RC4	Rivest Cipher 4
RSA	Rivest, Shamir e Adleman
SDK	Standard Edition Development Kit
SSL	Secure Sockets Layer
SQL	Structure Query Language
XML	Extensible Markup Language
URL	Uniform Resource Locator

SUMÁRIO

1. INTRODUÇÃO

O que significa a palavra segurança? Segundo o dicionário Aurélio, segurança é "estado, qualidade ou condição de seguro", ele também a define como "condição daquele ou daquilo que se pode confiar" (Ferreira, 2004). E uma aplicação ou programa seguro, seria aquele se pode confiar.

O quanto é possível confiar nos programas e sites utilizados diariamente por milhares de pessoas. Em 2011, uma das redes de vídeos game mais famosas do mundo, a Playstation Network (PSN), da Sony, sucumbiu ao ataque de hackers deixando vulneráveis informações de milhares de usuários. Hackers são pessoas que tem a capacidade de burlar os mecanismos de segurança de um sistema e conseguir acesso não autorizado a certos recursos (Ferreira, 2004).

> "Segurança: Medida da habilidade de um sistema de resistir a um uso não autorizado enquanto continua provendo seus serviços para usuários legítimos. Uma tentativa de quebrar a segurança é chamada de ataque e pode ser realizada de diferentes maneiras. Podendo ser uma tentativa não autorizada de acessar/modificar dados ou serviços, ou recusar serviços para usuários legítimos". (CAMPOS, 2011)

Nos últimos meses alguns jornais do mundo têm publicado invasões que vem ocorrendo a sistemas de grandes empresas, como SONY e FMI. Em um dos ataques realizados à Sony foi utilizada uma simples SQL *Injection* (BBC, 2011). A *Structured Query Language*, SQL, é uma "linguagem padrão para programação, operação interativa e comunicação de sistemas de bancos de dados relacionais" (Ferreira, 2004). O ataque por SQL *Injection,* ou Injeção de SQL, consiste na entrada de códigos SQL pelo usuário, através de um campo de um formulário do sistema que ele está utilizando, com o intuito de acessar informações, ou alterar, ou apagar, do banco de dados (OWASP, 2011). Com este ataque também é possível acessar algumas funcionalidades administrativas do banco de dados, como derrubá-lo.

Não é possível afirmar que os sistemas da SONY ou do FMI são mais ou menos seguros que outros sistemas, mas sim que tinham uma falha segurança e que esta foi descoberta pelos Hackers.

Este trabalho tem como objetivo apresentar um guia para utilização do *Java Authentication and Authorization Service* (JAAS).

O JAAS faz parte de um conjunto de APIs de segurança disponibilizadas pela plataforma Java. Este conjunto de APIs é o Java SE *Security*.

A Interface de Programação de Aplicações (API – *Application Programming Interfaces*) é um conjunto de padrões e rotinas definidos para um software para que suas funcionalidades sejam utilizadas por outros aplicativos.

O JAAS foi adicionado na versão 1.3 do Java *Standard Edition Development Kit* (SDK) como uma API opcional. Só na versão 1.4 o JAAS deixou de ser opcional e passou a fazer parte do SDK. O JAAS possui dois propósitos, autenticação e autorização (Oracle, 2011I).

A autenticação determina quem está executando o código, independente se está executando uma aplicação, um *bean* ou um *servlet.* Por outro lado, autorização é garantir que o usuário tem as permissões necessárias para executar uma determinada ação (Oracle, 2011F).

Além do JAAS, Java SE *Security* tem as APIs para criptografia, chaves públicas, comunicação segura, etc.

Criptografia é "a ciência e arte de escrever mensagens em forma cifrada ou em código" (CERT, 2011). A criptografia no JAAS pode ser utilizada para a transmissão de informações entre o servidor e a aplicação do usuário, para armazenar a senha do usuário no banco de dados, etc. A API de criptografia do Java SE *Security* é conhecida como Java *Cryptography Architecture* (JCA). A JCA oferece uma vasta gama de serviços de criptografia, como por exemplo, Message Digest Algorithm 5 (MD5), que será visto na Seção 2.3.1.

O Java SE *Security* oferece algumas ferramentas para o gerenciamento de chaves públicas. As chaves públicas também são utilizadas na criptografia. Além disso, oferece APIs para alguns protocolos de comunicação segura.

Além de estudar o JAAS, este trabalho explica como implementar o JAAS no servidor Tomcat, utilizando o recurso disponibilizado por eles que é o `JAASRealm`.

O Tomcat é um servidor web para a plataforma Java. Ele pertence à Apache *Software Foundation*, ele é distribuído como *software* livre e o código dele é aberto. O `JAASRealm` é uma implementação do JAAS oferecido pelo Tomcat para a autenticação de usuários.

Todo este trabalho foi baseado na versão 6 da plataforma Java SE.

2. JAVA SE SECURITY

O Java *Security* é uma API que possui um amplo conjunto de ferramentas, implementações dos algoritmos, mecanismos e protocolos de segurança mais utilizados. As áreas abrangidas por estas APIs são criptografia, infra-estrutura de chaves públicas (PKI – *Public Key Infrastructure*), autenticação, comunicação segura e controle de acesso.

As APIs de criptografia e infra-estrutura de chaves públicas são a base para desenvolver uma aplicação segura. Também há interfaces de autenticação e autorização para controle de acesso aos sistemas ou recursos.

Essas APIs permitem que hajam diversas implementações interoperáveis de algoritmos e serviços de seguranças. Esses serviços são fornecidos através de *Providers* que foram feitos utilizando interfaces padrões, como é o caso de uma conexão ao banco de dados ou *Lightweight Directory Access Protocol* (LDAP) para recuperar usuário e senha. Isso permite que o desenvolvedor só se preocupe em como integrar a aplicação dele com essas APIs de segurança fornecidas. Apesar de a plataforma fornecer um grande conjunto de *Providers* que fornecem algum tipo serviço de segurança, a plataforma também permite que o desenvolvedor crie novos ou personalize.

Provider é um nome dado a uma classe que implementa algum serviço de segurança. Como é caso das classes para a criptografia de mensagens, se há uma criptografia do tipo A, por exemplo, então há o ProviderDaCriptografiaA que tem toda a lógica desta criptografia. Os *Providers* não são apenas limitados para criptografias, eles também são utilizados em outras áreas, como conexão com o banco de dados, com um servidor LDAP, etc.

O foco deste trabalho é apresentar o JAAS. Por este motivo, as demais APIs do Java SE Security serão apresentadas com poucos detalhes.

2.1 A segurança da linguagem Java e a validação do *Byte Code*

Uma das primeiras lições que é ensinado quando se aprende a linguagem Java é que ela é uma linguagem que foi desenvolvida para ser segura e de fácil uso. Ela possui gerenciamento automático da memória, possui o conhecido *garbage collection* que limpa da memória as variáveis que não estão sendo utilizadas, verifica o tamanho dos *arrays* e

fornece modificadores de acesso para classes e atributos (*public*, *private*, *protected* e *package*).

Quando um código Java é compilado o resultado é um arquivo `.class`, conhecido como *byte code*. Antes deste *byte code* ser executado pelo Java *runtime*, esse *byte code* é analisado para verificar se cumpre com as especificações da linguagem Java, i.e., se não viola nenhuma das regras da linguagem e se não possui acessos a memória incorretos ou *typecasts* ilegais. *Typecast* é a conversão forçada do tipo de um atributo em outro, por exemplo, uma variável do tipo inteira (int) pode ser convertida em uma do tipo real (double). Um exemplo de *typecast* é paresentado no Código 1.

```
//Exemplo de Typecast
int varInt = 100; //Criando uma variável do tipo inteiro
System.out.println((double)varInt);//Imprime na console o valor dela como
real(double), ou seja, o valor mostrado é 100.0
```

Código 1- Exemplo de typecast

2.2 Sobre a arquitetura

As APIs foram desenhadas seguindo três princípios, são eles Independência de Implementação (*Implementation Independence*), Interoperabilidade de Implementação (*Implementation Interoperability*) e Extensibilidade de Algoritmo (*Algorithm Extensibility*).

- *Implementation Independence:* As aplicações podem invocar a múltiplos *Providers* independentes.
- *Implementation Interoperability:* Uma aplicação não é construída para um *Provider* específico e um *Provider* não é construído para uma aplicação específica.
- *Algorithm Extensibility:* Apesar dos *Providers* fornecidos pela plataforma, haverá aplicações que exijam diferentes padrões.

Um serviço disponível pode ter uma lista de *Providers* ou um específico. No exemplo apresentado no Código 2, o serviço requerido é o de criptografia, para isso é utilizada a classe `MessageDigest`. O texto deve ser criptografado utilizando o algoritmo *Message-Digest algorithm* 5 (MD5). O algoritmo MD5 é explicado na Seção 2.2. Para procurar o *Provider* (ver Figura 1) que tem este algoritmo dentro de uma lista, é utilizado o método

`getInstance("MD5")`, este método retorna um objeto do primeiro *Provider* que tenha a lógica do algoritmo de criptografia MD5. E para criptografar o texto neste algoritmo é chamado o método `digest(byte [] texto)` do objeto retornado.

```
MessageDigest md = MessageDigest.getInstance("MD5");
md.digest( textoEmBytes );
```

Código 2 – Requisição de criptografia usando MD5.

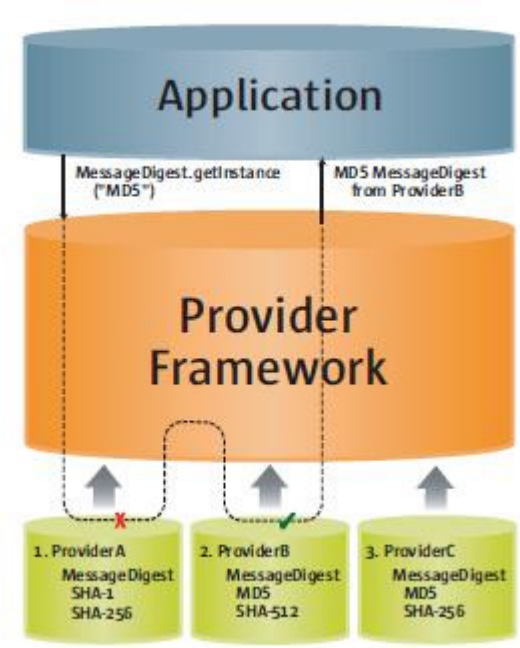

Figura 1 – Etapas para Procurar um Provider

Fonte: Java SE Documentation (Oracle, 2011E)

Para pegar um *Provider* pelo nome (Figura 2) também é utilizado o método `getInstance`, mas passando o nome dele.

```
MessageDigest md = MessageDigest.getInstance("MD5", “ProviderC”);
```

Código 3 - Busca de um provider pelo nome.

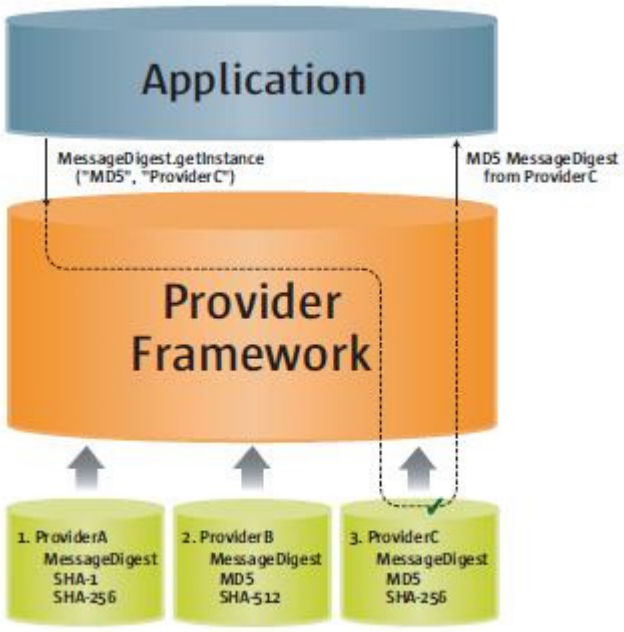

Figura 2 – Pegar provider pelo nome

Fonte: Java SE Documentation (Oracle, 2011E)

Os *Providers* padrões estão definidos no arquivo `java.security` que fica pasta lib/security dentro da pasta de instalação do *Java™ Runtime Environment* (JRE). Mas também é possível mudá-las, chamando certos métodos da classe `java.security.Security`.

2.3 Java Cryptography Architecture (JCA)

A JCA é um conjunto de APIs que disponibilizam serviços de criptografia para facilitar o desenvolvimento, entre outros serviços. Um dos algoritmos que pode ser utilizado para criptografar senhas armazenadas em banco de dados é o MD5. A JCA possui vários outros serviços, mas eles não serão abordados neste trabalho. O MD5 é abordado neste trabalho pois é mencionado em um exemplo mais adiante.

2.3.1 MD5 – *Message-Digest Algorithm* 5

O MD5 (Message-Digest algorithm 5), descrito na RFC 1321, é um algoritmo de *hash* de 128 bits unidirecional desenvolvido pela *RSA Data Security, Inc*.

O *hash* é o resultado da aplicação de algum algoritmo de criptografia. Se aplicado o MD5, o *hash* será composto somente por caracteres hexadecimais, para representar um

caractere hexadecimal são necessários 4 *bits*, ou seja meio *byte*, como ele tem 128 bits (16 *byte*) significa que o resultado desta criptografia terá 32 caracteres, independente da tamanho do texto utilizado.

Por ser unidirecional o texto criptografado não pode ser revertido ao seu valor original. Para comparar a senha fornecida pelo usuário com a senha criptografada armazenada no banco de dados, a senha fornecida também é criptografada e o resultado é comparado com o valor armazenado.

Aplicando o algoritmo MD5 ao texto “senha1234”, sem as aspas duplas, o resultado ou o *hash* é 732002CEC7AEB7987BDE842B9E00EE3B.

Para saber mais detalhes deste algoritmo consulte a RFC 1231 (RIVEST, 1992).

2.4 Infra-estrutura de chaves públicas (PKI)

PKI é um conjunto de APIs que permitem a troca segura de informações através de criptografia de chaves públicas. Isso consiste em cada entidade (pessoa, empresa, etc.) possuir duas chaves, uma pública e uma privada. A chave pública é disponibilizada livremente, a privada não. Para enviar um e-mail criptografado, por exemplo, a uma empresa, é necessário saber a chave pública dela, pois esta será utilizada para criptografar o e-mail. A empresa quando recebe o e-mail utiliza a chave privada para decodificar (CERT, 2011).

Nestas APIs também há as classes necessárias para trabalhar com *key-stores*, além das classes a plataforma oferecer algumas ferramentas para o gerenciamento e criação. As *key-stores* armazenam informações dos certificados digitais. Ao definir uma política de segurança é possível referenciar uma *key-store* , isso é mostrado com mais detalhes na Seção 3.3.

2.5 Autenticação e Controle de Acesso

Este tema, mais conhecido como Java *Authentication and Authorization Service* (JAAS), será discutido na Seção 3.

2.6 Comunicação Segura

A plataforma Java fornece APIs que possibilitam a transmissão segura de informações através da rede. Nesta transmissão segura é utilizado criptografia, como pode ser visto na Seção 2.3.

No Tomcat quando o `JAASRealm` é configurado, é possível exigir que a comunicação entre o navegador do usuário e o servidor seja segura. Esta comunicação utiliza o protocolo *Hyper Text Transfer Protocol Secure* (HTTPS). Ele é a implementação do protocolo *Hyper Text Transfer Protocol* (HTTP) em conjunto com o protocolo *Secure Sockets Layer* (SSL). O SSL está entre a camada de aplicação e de transporte, ele é responsável por manter um canal seguro de comunicação entre o servidor e o usuário. As informações nesta comunicação estão criptografadas.

3. JAAS (Java *Authentication and Authorization Service*)

Como mencionado na introdução, o JAAS é uma API para autenticação e autorização. Nesta seção é mostrado como fazer uma configuração genérica do JAAS. Esta configuração pode ser utilizada para um aplicativo desktop. O nome dado a esta configuração é `JAASGenerico`.

> "Autorização: Dar poder para, permitir". (LUFT, 1990, p. 68)

As classes e as interfaces oferecidas por esta API estão divididas em três grupos: (1) *Common*, (2) *Authentication* e (3) *Authorization*. As classes e Interfaces dos grupos *Authentication* e *Authorization* serão vistas nas subseções JAAS *Authentication* e JAAS *Authorization,* respectivamente.

3.1 *Common*

As classes neste grupo são utilizadas tanto pelas do grupo *Authentication* como pelas do grupo *Authorization*. Este grupo possui três classes: (1) `Subject`, (2) `Principal` e (3) *Credential*.

3.1.1 Classe *Subject*

A classe `Subject` representa quem originou a requisição, e pode representar uma pessoa ou um serviço. Para falar da classe `Subject` será utilizado como exemplo a pessoa Fulano e o cargo dele, na empresa que ele trabalha, é administrador. O nome Fulano representa uma das identificações de uma pessoa, cada pessoa pode ter mais de uma identificação, além do nome uma pessoa pode ser identificada pelo Cadastro de Pessoa Física (CPF), pelo título de Eleitor e outros. O cargo dele, administrador, também pode representar uma identificação, pois em uma empresa há vários cargos e cada um deles tem suas permissões de acesso. Cada identificação é representada pela interface Principal, a classe `Subject` tem o atributo `principals` do tipo `Set`, ou seja, uma coleção de objetos do tipo Principal.

Um `Subject` pode ter seus próprios atributos de segurança, o qual será chamado de *Credential* e no plural *Credentials*. Esta classe possui dois atributos que representam seus *Credentials*, o `publicCredentials` e `privateCredentials` que são do tipo `Set`. Um exemplo de `publicCredentials` é uma chave pública ou um certificado digital e para `privateCredentials` um exemplo seria a chave privada. No `publicCredentials` são armazenados as *credentials* que podem ser vistas por qualquer um. A senha, por exemplo, deveria ser armazenada no `privateCredentials`, pois somente o usuário deve saber sua senha, mas isso não é obrigatório.

Ademais dos atributos já mencionados, esta classe possui o atributo `readOnly`. Este atributo indica se o objeto da classe `Subject` pode ser alterado ou não. Se ele for verdadeiro (*true*) os atributos da classe (`principals`, `publicCredentials` e `privateCredentials`) não podem ser alterados e caso sejam, será disparado uma exceção do tipo `IllegalStateException`.

Esta classe possui outros dois métodos o `doAs` e o `doAsPrivileged`. Eles são utilizados para executar uma ação de um determinado `Subject`. A diferença entre eles é que o `doAsPrivileged` no lugar de utilizar o `Subject` do `AccessControlContext`(Contexto de Controle de acesso) da *thread* atual, ele pode utilizar outro `AccessControlContext`. Assim as permissões são definidas por `Subject` e também por `AccessControlContext`. Lembrando que um objeto da classe `Subject` pode estar relacionado a um Contexto de Controle de Acesso (classe `AccessControlContext`).

3.1.2 Interface `Principal`

`Principal` é uma interface, todo projeto deve criar uma nova classe que a implemente e também a interface `java.io.Serializable`. A interface `Principal` tem quatro métodos que devem ser implementados pela classe nova. O método `getName`, que é utilizado para recuperar a identidade do usuário, neste exemplo o `getName` poderia retornar Fulano, ou administrador, ou o CPF. Os outros métodos são `equals`, `toString` e `hashCode`, o objetivo de cada um destes três métodos não será explicado neste trabalho.

A Oracle não explica porque a nova classe deve implementar a `java.io.Serializable`. Ao implementar esta interface, um objeto da nova classe pode ser serializado e deserializado. O objeto ao ser serializado ele vira uma seqüência de *bytes*, essa

seqüência pode ser salva em um arquivo texto ou enviada pela rede a outro sistema. Desde este arquivo texto a seqüência de *bytes* pode ser deserializada e volta a virar um objeto. Em ambos os processos deve ser utilizado a mesma classe, inclusive a versão. Se alguém tenta deserializar uma seqüência de *byte* que não é da classe que ele está utilizando ou de versão diferente, a máquina virtual do Java dispara uma `java.io.InvalidClassException`. A versão da classe é representa pelo atributo `serialVersionUID`, toda classe que implementa a `Serializable` é aconselhada a ter-lo.

Para representar o nome Fulano, o cargo administrador e o CPF do usuário, foram criadas três classes, a `LoginPrincipal`, `CargoPrincipal` e `CPFPrincipal` respectivamente. A estrutura delas está definida no diagrama de classes abaixo (Figura 3). O `getName` da `LoginPrincipal` retorna o *login* (Fulano), o da `CargoPrincipal` retorna o cargo (administrador) e o da `CPFPrincipal` retorna o CPF.

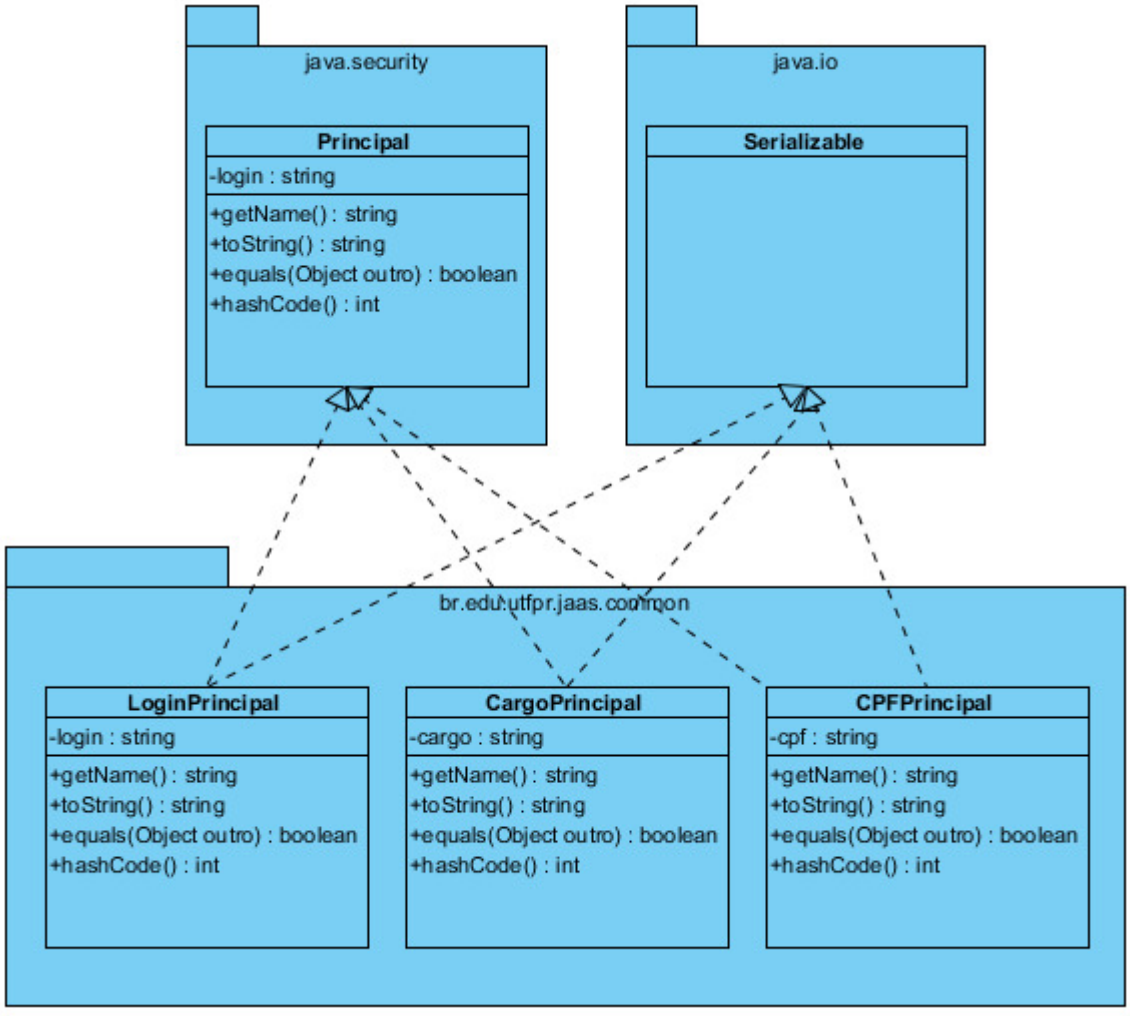

Figura 3 – Diagrama de Classes das classes LoginPrincipal, CargoPrincipal e CPFPrincipal

3.1.3 *Credential*

Credential não é uma classe ou uma interface, é apenas um termo utilizado para definir os objetos que irão popular os atributos `publicCredentials` e `privateCredentials` da classe Subject. Esses objetos podem pertencer a qualquer classe. A Oracle, no JAAS *Reference Guide*, recomenda que o desenvolvedor escolha uma classe que implemente as interfaces `javax.security.auth.Refreshable` e `javax.security.auth.Destroyable` (Oracle, 2011G).

A interface `Refreshable` oferece aos `Credentials` um método para verificar que ele está ativo (`isCurrent()`) e um para atualizá-lo (`refresh()`). E a interface `Destroyable` oferece também dois métodos, o `isDestroyed()` que retorna *true* se o *credential* já foi destruído *false* caso contrário, e o `destroy()` que destrói e limpa as informações relacionadas com o *credential*.

3.2 JAAS *Authentication*

Autenticar é o processo que confirma a identidade da pessoa ou serviço que fez a requisição. Para realizar este processo, no caso mais simples, devem ser utilizadas três classes e um arquivo de configuração: (1) `LoginContext`, (2) `CallbackHandler`, (3) `LoginModule` e o (4) arquivo de configuração que foi chamado de `configuracao_generica.config`.

3.2.1 Arquivo de configuração do *Login* (`configuracao_generica.config`)

Neste arquivo são definidas as tecnologias ou os `LoginModules` que serão utilizados no sistema. Para cada configuração é definida um nome (*name*), no exemplo é utilizado o nome `JAASGenerico`. É possível ter mais de uma entrada (*name*) definida neste arquivo, todas elas são representadas pela classe `javax.security.auth.login.Configuration`, cada uma está indexada pelo *name*.

Segue a estrutura de uma configuração (ver Código 4) extraída da referência *Class Configuration – JavaDoc* fornecida pela Oracle (Oracle, 2011A):

```
Name {
     ModuleClass  Flag    ModuleOptions;
     ModuleClass  Flag    ModuleOptions;
     ModuleClass  Flag    ModuleOptions;
};
```

Código 4 - Modelo para criar um configuração de login.

O parâmetro `Name` deve ser substituído pelo nome da sua aplicação e `ModuleClass` pelo nome da classe e do pacote do `LoginModule` que a sua aplicação utilizará, pode ser definido mais de um. Caso haja mais de um `LoginModule` definido, o resultado da autenticação só será dado após todos serem executados.

O `ModuleOptions` é um espaço onde pode ser definido uma lista de valores que serão utilizados pelo LoginModule. Um exemplo é o *debug*. Se o `debug` é igual a *true* (`debug=true`), significa que o sistema exibirá algumas informações extras enquanto um usuário tenta se autenticar. Outro exemplo é o `digest`, se ele for igual a MD5 (`digest=MD5`), significa que a senha salva no banco foi criptografada utilizando o MD5.

O parâmetro `Flag` define o quão importante é este `LoginModule` no processo de autenticação, há quatro valores possíveis para ele (*required*, *requisite*, *sufficient* e *Optional*):

Required: O usuário deve ser autenticado com sucesso por ela. Caso não tenha sucesso, independente se há outros `LoginModules` definidos na configuração, a autenticação será dada como incorreta ou falha. Mesmo não tendo sucesso os outros `LoginModules` serão executados;

Requisite: Se a autenticação falha, nenhum outro `LoginModule` abaixo dele é executado;

Sufficient: Se autenticação tem êxito, nenhum outro `LoginModule` abaixo dele é executado;

Optional: Independente se falhou ou não, os outro `LoginModule` abaixo dele serão executados.

Para melhor exemplificar, será utilizado um exemplo disponibilizado pela Oracle no JavaTM *Authentication and Authorization Service* (JAAS) – *Reference Guide* (Oracle, 2011G), a Tabela 1.

```
Login2 {
       sample.SampleLoginModule required;
       com.sun.security.auth.module.NTLoginModule sufficient;
       com.foo.SmartCard requisite debug=true;
       com.foo.Kerberos optional debug=true;
    };
```

Código 5 - Exemplo 1 para o arquivo de configuração do login.

A Tabela 1 mostra os possíveis resultados para a configuração dada acima, a do Login2. O texto 'Passou' significa que naquela etapa o LoginModule correspondente autenticou o usuário, o 'Falhou' significa o contrário e o '*' que o `LoginModule` correspondente não foi chamado. Lembrando que o resultado final do processo de autenticação só é dado após passar por todos os `LoginModules`.

Tabela 1 – Login2 o Estado da Autenticação (traduzida)

Login2 Authentication Status									
SampleLoginModule	required	Passou	Passou	Passou	Passou	Falhou	Falhou	Falhou	Falhou
NTLoginModule	sufficient	Passou	Falhou	Falhou	Falhou	Passou	Falhou	Falhou	Falhou
SmartCard	requisite	*	Passou	Passou	Falhou	*	Passou	Passou	Falhou
Kerberos	optional	*	Passou	Falhou	*	*	Passou	Falhou	*
Resultado final da autenticação		Passou	Passou	Passou	Falhou	Falhou	Falhou	Falhou	Falhou

Fonte: JavaTM Authentication and Authorization Service (JAAS) (Oracle,2011G)

Para este exemplo genérico foi criado o arquivo de configuração `configuracao_generica.config`, o conteúdo dele está abaixo:

```
JAASGenerico {
     br.edu.utfpr.jaas.module.GenericoLoginModule required debug=true
digest=MD5;
};
```

O `name` foi substituído pelo `JAASGenerico`, que é o nome dado para esta configuração genérica que está sendo demonstrada nesta seção. Neste exemplo é utilizado um `LoginModule` com o nome `GenericoLoginModule`, esta classe está descrita na subseção relacionada a Interface `LoginModule`. A Flag foi definida como *required*, ou seja, caso a autenticação nela falhe, todo o processo de autenticação será dado como falho. No espaço

destinado a uma lista de valores, foram definidos dois valores, *debug* igual a *true* e *digest* igual a MD5.

Antes de executar o `JAASGenerico` na sua aplicação é necessário informar à maquina virtual onde encontrar o arquivo de configuração, uma das maneiras é executando o comando `System.setProperty("java.security.auth.login.config", "/configuracao_generica.config")` dentro do seu programa.

3.2.2 Classe `LoginContext`

Esta classe contém o mínimo necessário para construir um módulo de autenticação independente da tecnologia escolhida. Todos os construtores desta classe têm o parâmetro `name` do tipo `String`. Este `name` representa o nome da configuração no arquivo. Para o arquivo citado acima, o `configuração_generica.config`, o valor do parâmetro `name` é "`JAASGenerico`". Quando é criada uma instância desta classe, internamente dela é criado uma instância vazia de um `Subject`.

O método `login` executa os `LoginModules` definidos no arquivo de configuração para um determinado *name*. Neste exemplo o `LoginModule` definido é o `GenericoLoginModule`.

Dentro do método LoginContext.login é chamado método `login` da classe `GenericoLoginModule`, se a autenticação nele for bem sucedida, após ele é executado o método `commit` da classe `GenericoLoginModule`. Dentro do método `commit` a instância da classe `Subject` é populada com os `principals` e `credentials` correspondentes, caso contrário o processo é abortado, chamando o método `abort` da classe `GenericoLoginModule`, e os `principals` e `credentials` são destruídos. Caso tenha mais de um `LoginModule`, somente após a execução de todos e em caso de sucesso que a instância do `Subject` é populada.

A Figura 4 mostra o diagrama de seqüência para a configuração JAASGenerico.

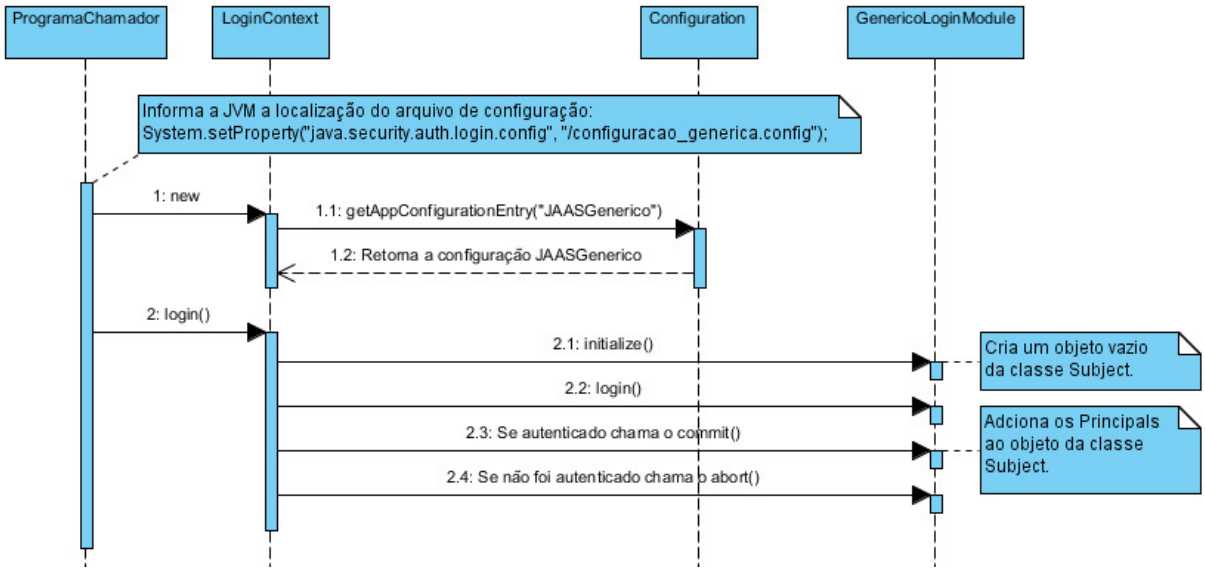

Figura 4 - Diagrama de Seqüência incluindo a classe GenericoLoginModule

3.2.3 Interface `LoginModule`

O `LoginModule` é apenas uma interface, cada desenvolvedor deve criar uma classe ou mais classes que a implemente, isso depende da necessidade de cada sistema. Esta classe tem a lógica da autenticação, caso a senha esteja no banco de dados é ela que recupera esta senha e compara com a senha informada pelo usuário. A classe `LoginContext` inicia a `LoginModule` com um `Subject`, um `CallbackHandler`, o `sharedState` dos `LoginModules` e opções (*options*) específicas do `LoginModule`.

O `Subject` *representa* o usuário ou serviço que está tentando se autenticar. O `CallbackHandler` é utilizado para interagir com o usuário, mas ele pode ser nulo. O `sharedState` é do tipo `Map` e é utilizado para compartilhar informações entre os `LoginModules`, ele é opcional. E as opções específicas são aquelas que foram definidas no arquivo de configuração, para o `JAASGenerico`, as opões são *debug* igual *true* e *digest* igual a MD5.

O método `login` chama o método `handler` da classe CallbackHandler, este método é responsável por interagir com o usuário, isso é, recuperar o *login* e senha dele por exemplo. Após recuperar o *login* e senha, o método `login` vai recuperar a senha armazenada, essa senha pode estar armazenada em um banco de dados, ou em diretório LDAP, etc. E por fim ele valida a senha enviada pelo usuário com a senha armazenada. Lembrando que no

exemplo genérico `JAASGenerico` foi colocado `digest=MD5`, isso significa que a senha armazenada está criptografa, então antes de compará-las, é necessário criptografar a senha enviada. Se o *login* informado existe e se senha está correta, o método `login` retorna *true*, caso contrário retorna *false*.

O método `commit` só é chamado após a execução do método `login` de cada `LoginModule`. Caso o método `login` tenha falhado ele deve retornar *false*, caso contrário ele popula o `Subject` com os `Principals` e com as `Credentials` e destrói os parâmetros já não mais necessários como o nome do usuário (*login*) e a senha. Se tudo ocorrer bem, ele guarda um *state* privado informando que o `commit` sucedeu bem e retorna *true*.

Caso seja necessário interromper o processo, deve ser chamado o método `abort`, ele é utilizado para eliminar todas as informações utilizadas, como o usuário e a senha. Se o método `login` ou `commit` retornar *false*, este método será chamado.

E por fim o método `logout`, ele remove os `Principals` e remove e destrói os `Credentials` do `Subject`. Se o `Subject` for `readOnly`, os *Credentials* não são removidos, são somente destruídos. Caso tudo ocorra bem retorna *true*, senão dispara um `LoginException`.

Para configuração genérica foi criada a classe `GenericoLoginModule`, logo abaixo está o diagrama de classe dela (Figura 5). No método `commit` além do *login*, foi adicionado o CPF e o cargo do usuário como principal na classe `Subject`.

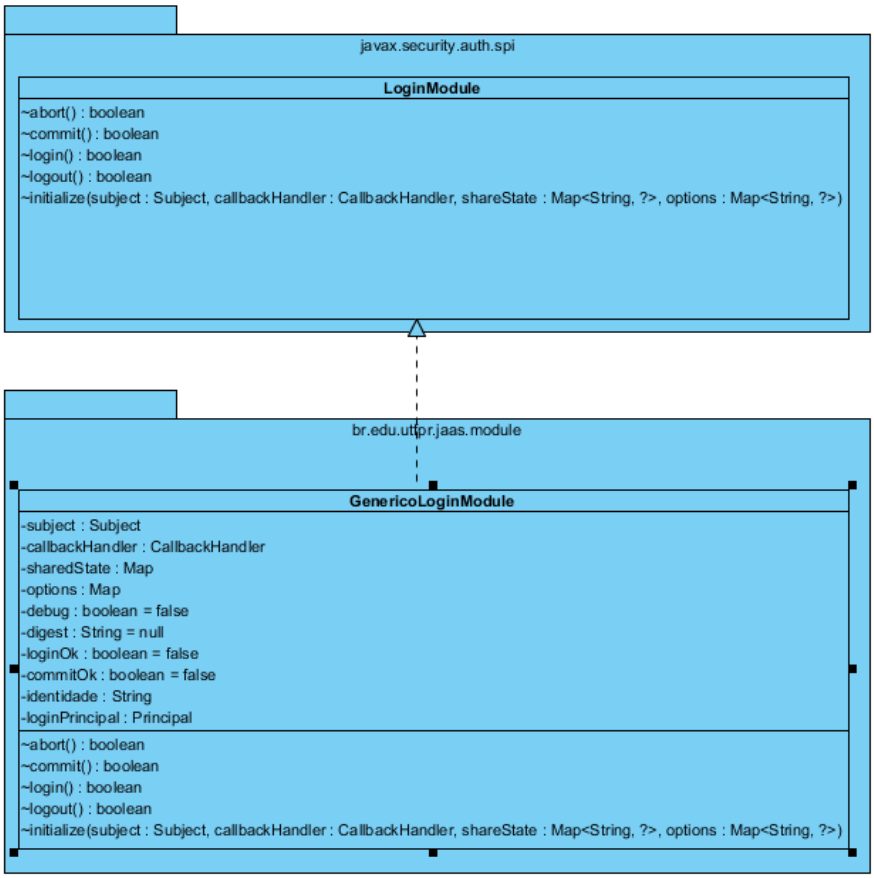

Figura 5 - Diagrama de Classe da classe GenericoLoginModule

Os métodos dela já foram explicados nesta subseção. Na Tabela 2 estão explicados seus atributos.

Tabela 2 - Atributos da classe GenericoLoginModule.

Atributos da classe GenericoLoginModule	
Nome	Descrição
subject	Representa o usuário ou serviço que está tentando autenticar-se.
callbackHandler	É utilizado para interagir com o usuário.
shareState	Utilizado para compartilhar informações entre os `LoginModules`, ele é opcional.
options	São as opções específicas que foram definidas no arquivo de configuração.
debug	Opção definida no arquivo de configuração, se ele for *true* o sistema exibe o log.
digest	Opção definida no arquivo de configuração, ele armazena o algoritmo de criptografia que foi utilizado na senha armazenada.

loginOk	Se for *true,* a autenticação no método `login` foi bem sucedida.
commitOk	Se for *true,* o método `commit` foi executado com sucesso.
identidade	Representa a identidade do `Subject`, ele pode ser o *login*, ou o cargo, ou o CPF.
loginPrincipal	Após o sucesso do processo de autenticação este `loginPrincipal` será adicionado ao `Subject`. Este `loginPrincipal` será inicializado com o valor do atributo `identidade`.

Abaixo é apresentado o diagrama de seqüência para o processo que acontece dentro da classe `GenericoLoginModule` em uma visão macro (Figura 6).

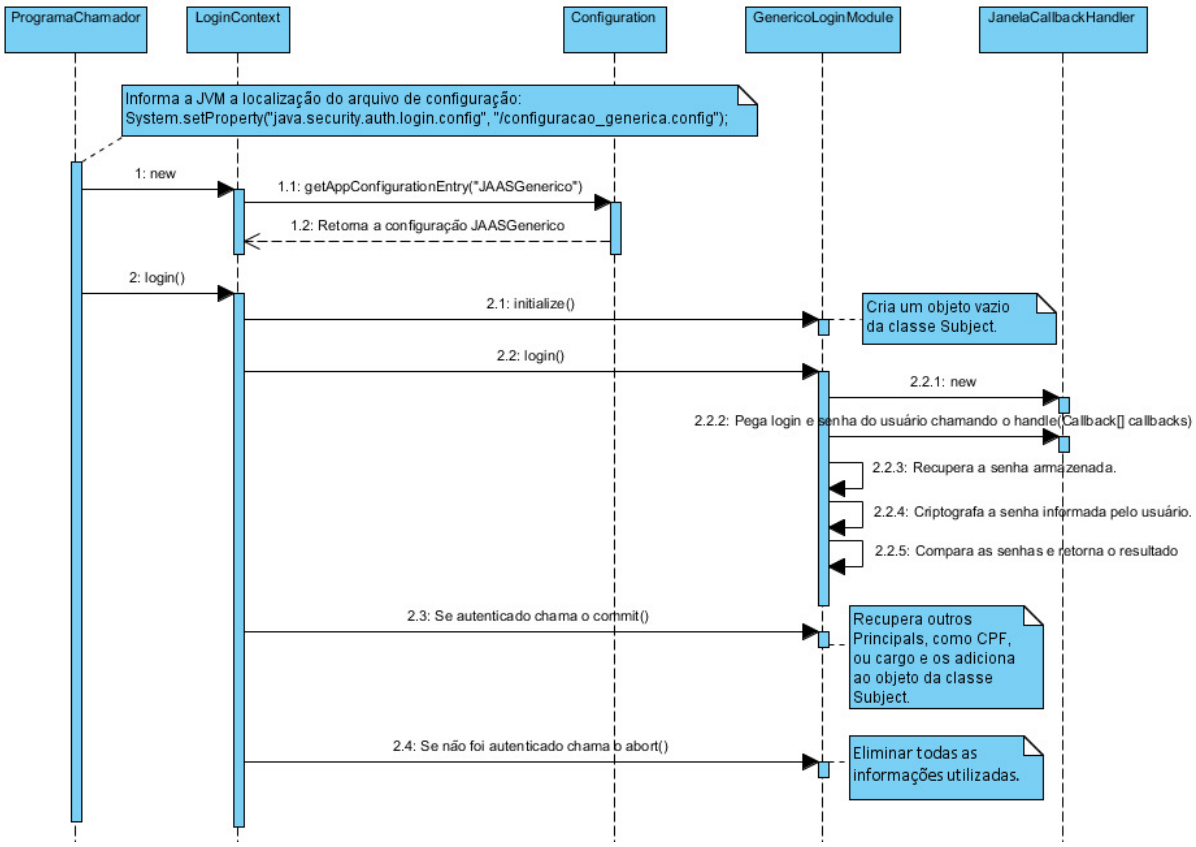

Figura 6 - Diagrama de Seqüência incluindo a classe JanelaCallbackHandler

Para este exemplo foi criado a classe `JanelaCallbackHandler`, ela é uma implementação da classe CallbackHandler. Ela mostra uma janela pedindo *login* e senha para o usuário.

3.2.4 Interface *CallbackHandler*

`CallbackHandler` é uma interface e possui somente o método `handle`. Quando o `LoginModule` necessita interagir com usuário deve ser criada uma classe que implementa esta interface. Esta nova classe pode requisitar o usuário e senha pelo console ou por uma janela, conforme necessidade. Como o `CallbackHandler` é passado como parâmetro para o `LoginContext`, cada aplicação que executar a classe `LoginContext` pode passar um `CallbackHandler` diferente e utilizar o mesmo `LoginModule`. Ao ser executado, o método `handle` recebe um `array` de `Callbacks`, este `array` pode ter, por exemplo, um `NameCallback` e um `PasswordCallback`. Após interagir com o usuário, cada `Callback` dentro do `array` é preenchido.

3.2.5 Interface *Callback*

`Callback` também é uma interface, além das classes `NameCallback` e `PasswordCallback`, há outras oferecidas pela plataforma que implementam esta interface, como `ChoiceCallback`, `ConfirmationCallback`, `LanguageCallback`, `TextOutputCallback`, etc.

A `NameCallback` pode ser utilizada para requisitar o *login* do usuário. A `PasswordCallback` é utilizada para requisitar a senha, ela possui o método `clearPassword`, ele substitui por espaços em branco a senha informada pelo usuário. O `TextOutputCallback` pode ser utilizado para mostrar uma mensagem na tela, ou como título da tela de login, etc.

Abaixo tem um exemplo de código utilizando estes três `Callbacks` explicados no parágrafo anterior e a classe `JanelaCallbackHandler`. O código fonte da classe `JanelaCallbackHandler` está no Apêndice H.

```
Callback [] callbacks = new Callback[3];
//Texto para utilizar como título na tela de login
callbacks[0] = new TextOutputCallback(TextOutputCallback.INFORMATION,"Tela
de Login");
//Texto para pedir o login ao usuário
callbacks[1] = new NameCallback("Login: ");
//Texto para pedir a senha ao usuário
callbacks[2] = new PasswordCallback("Senha: ", false);
```

```
JanelaCallbackHandler callbackHandler = new JanelaCallbackHandler();
callbackHandler.handle(callbacks);
```

O resultado do código acima é a Figura 7.

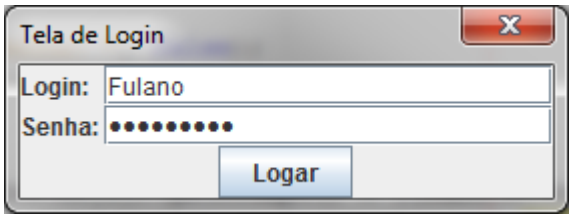

Figura 7 - Tela de Login (JanelaCallbackHandler)

3.2.6 Processo de autenticação

O primeiro passo é criar (1) as classes que implementam a `Principal`, `LoginModule` e `CallbackHandler` (se necessário) e o arquivo de configuração indicando o `LoginModule`.

No seguinte passo o programa chamador deve informar (2) qual arquivo de configuração será utilizado e criar (3) uma instância da classe `LoginContext`, passando pelo menos o nome da configuração a ser utilizada, nessa etapa é criado um `Subject` vazio. No quarto passo deve ser executado (4) o método `login` da classe recém instanciada.

Dentro do método `login` são executados os próximos passos, começando pela execução (5) do método `login` do `LoginModule`. Dentro do `LoginModule` é executado (6) o método `CallbackHandler.handle` para recuperar o usuário e senha. Tendo essas informações o `LoginModule` recupera (7) a senha do usuário desde um banco de dados ou de um diretório LDAP. No oitavo passo a senha é validada (8), em caso de sucesso o método retorna (9) *true*.

Após a execução do `LoginModule.login` e ainda dentro do `LoginContext.login` é executado (10) o método `LoginModule.commit`. O `commit` verifica o sucesso da execução do método `login`, em caso positivo são criados (11) os `Principals` e os *Credentials* para serem colocados dentro do `Subject` e o método retorna (12) *true*.

E por último, o resultado do processo de autenticação é retornado (13) ao programa chamador.

3.2.7 *Login Configuration Provider*

Como foi visto anteriormente, a definição do `LoginModule` *do sistema* está definido num arquivo com a extensão config, que é lido pelo *Provider* padrão da plataforma lê a configuração desde um arquivo. Entretando, é possível definir outro executando o comando `System.setProperty("login.configuration.provider", "package.NewProviderConfigFile");`.

3.3 JAAS *Authorization*

Processo que decide se um usuário ou sistema já autenticado tem permissão para acessar um determinado recurso ou executar uma ação. As permissões de acesso ou execução estão definidas em uma política de segurança, na próxima seção está descrito como definir esta política. Nesta política é possível definir o que pode ser executado e quem pode executar.

Tendo a política de segurança definida e o usuário já autenticado, o seguinte passo é recuperar o `Subject` chamando o método `getSubject()` da classe `LoginContext` e executar um dos métodos do `Subject`, `doAs` ou `doAsPrivileged`. Os dois métodos recebem como parâmetro uma classe que implementa a interface `PrivilegedAction` ou a `PrivilegedExceptionAction`, esta classe representa a ação e será executado o método `run` dela.

3.3.1 Definindo a política de segurança

Para definir as políticas de segurança será criado um arquivo com a extensão *policy*, será utilizado o nome *policy* para referir a este arquivo. Um exemplo de nome para este arquivo é `jaas_generico.policy`. Para informar ao sistema, em tempo de execução, qual arquivo inclui a política de segurança, é necessário executar o comando `System.setProperty("java.security.policy", "/jaas_generico.policy");`.

Definir um conjunto permissões no arquivo *policy* é definido como *grant*, a estrutura (ver Código 6) deste `grant` segundo a referência *Default Policy Implementation and Policy File Syntax* é (Oracle. 2011B):

```
grant signedBy "signer_names", codeBase "URL",
       principal principal_class_name "principal_name",
       principal principal_class_name "principal_name",
```

```
        ... {

      permission permission_class_name "target_name", "action",
          signedBy "signer_names";
      permission permission_class_name "target_name", "action",
          signedBy "signer_names";
      ...
  };
```

Código 6 - Estrutura modelo para definir a poítica de segurança.

`SignedBy`, `codeBase` e `principal` são opcionais. O `signedBy` define o apelido para um certificado armazenado em uma *key-store*. O `codeBase` define o código que será aplicado este `grant`. E o `principal` determina que este `grant` é aplicado a uma classe que implementa a interface `Principal`.

Para definir um *key-store* na *policy*, basta fazer como o Código 7. Escrever `keystore` e na frente colocar a localização dele:

```
keystore "http://foo.bar.com/blah/.keystore";
```

Código 7 - Definir key-store.

Cada linha que começa com `permission` representa uma permissão dada. O `principal_class_name` deve ser substituído pelo nome da classe que representa o tipo de permissão, por exemplo o `java.io.FilePermission`. `Target_name` pode ser o nome de um arquivo, uma pasta, etc. `Actions` representa a ação, por exemplo, se uma pessoa quer acessar um arquivo ou alterar, o `Actions` será "`read,write`".

Para dar permissão ao usuário Fulano acessar o arquivo TutorialJava.pdf foi colocado a seguinte permissão no arquivo *policy*. Este é o conteúdo do arquivo `jaas_generico.policy` (Código 8).

```
grant codebase "file:JAASGenerico.jar", Principal
br.edu.utfpr.jaas.common.LoginPrincipal "Fulano" {
      permission java.io.FilePermission "TutorialJava.pdf", "read";
};
```

Código 8 - Conteúdo do arquivo jaas_generico.policy.

Alguns exemplos retirados do *Default Policy Implementation and Policy File Syntax* (Oracle, 2011B):

Primeiro exemplo: Se o código está assinado por “Duke”, ele terá permissão de leitura e escrita para todos os arquivos na pasta /tmp.

```
grant signedBy "Duke" {
      permission java.io.FilePermission "/tmp/*", "read,write";
};
```

Código 9 - Exemplo 1 do arquivo policy.

Segundo Exemplo: Qualquer um tem esta permissão.

```
grant {
      permission java.util.PropertyPermission "java.vendor", "read";
}
```

Código 10 - Exemplo 2 do arquivo policy.

Terceiro Exemplo: utilizando `signedBy`, `codebase` e `principal`.

```
grant codebase "http://www.games.com",
      signedBy "Duke",
      principal javax.security.auth.x500.X500Principal "cn=Alice" {
      permission java.io.FilePermission "/tmp/games", "read, write";
};
```

Código 11 - Exemplo 3 do arquivo policy.

Quarto Exemplo: utilizando uma *keystore*, o texto "alice" representa um apelido no *keystore* e será substituído por `javax.security.auth.x500.X500Principal "cn=Alice"`.

```
keystore "http://foo.bar.com/blah/.keystore";

grant principal "alice" {
      permission java.io.FilePermission "/tmp/games", "read, write";
};
```

Código 12 - Exemplo 4 do arquivo policy.

3.3.2 Classe `Policy`

É uma classe abstrata que representa a política de segurança definida para um sistema.

3.3.3 Interface `PrivilegedAction` ou `PrivilegedExceptionAction`

Para executar uma ação no sistema é necessário criar uma classe que implemente uma das duas interfaces. O método `run` contém toda a lógica para esta ação.

Antes de falar da diferença entre as duas interfaces é necessário entender o que é uma *checked exception*. Todas as classes que são subclasses da `Exception` são *checked exception*. Quando é executado um método e este obriga o programador a tratar uma exceção utilizando `try/catch` ou adicionando-a na assinatura do método, esta pode ser uma *checked exception*.

A `PrivilegedAction` deve ser escolhida para um código que não disparará uma *checked exception* e a `PrivilegedExceptionAction` deve ser escolhida para um código que dispara uma *checked exception.*

Para o exemplo `JAASGenerico`, foi criado a classe `LerArquivoAction` (ver Código 13) que implementa `PrivilegedAction`, o método `run` dela lê o arquivo "TutorialJava.pdf".

```
package br.edu.utfpr.jaas.actions;

import java.io.File;
import java.security.PrivilegedAction;

public class LerArquivoAction implements PrivilegedAction {

    public Object run() {
        File f = new File("./TutorialJava.pdf");
        System.out.print("\nO arquivo ");
        if (!f.exists()){
            System.out.print("não ");
        System.out.println("foi localizado.");
         }
        return null;
    }
}
```

Código 13 - Código da classe LerArquivoAction

A Figura 8 mostra o diagrama de seqüência numa visão macro até o método `run` ser chamado.

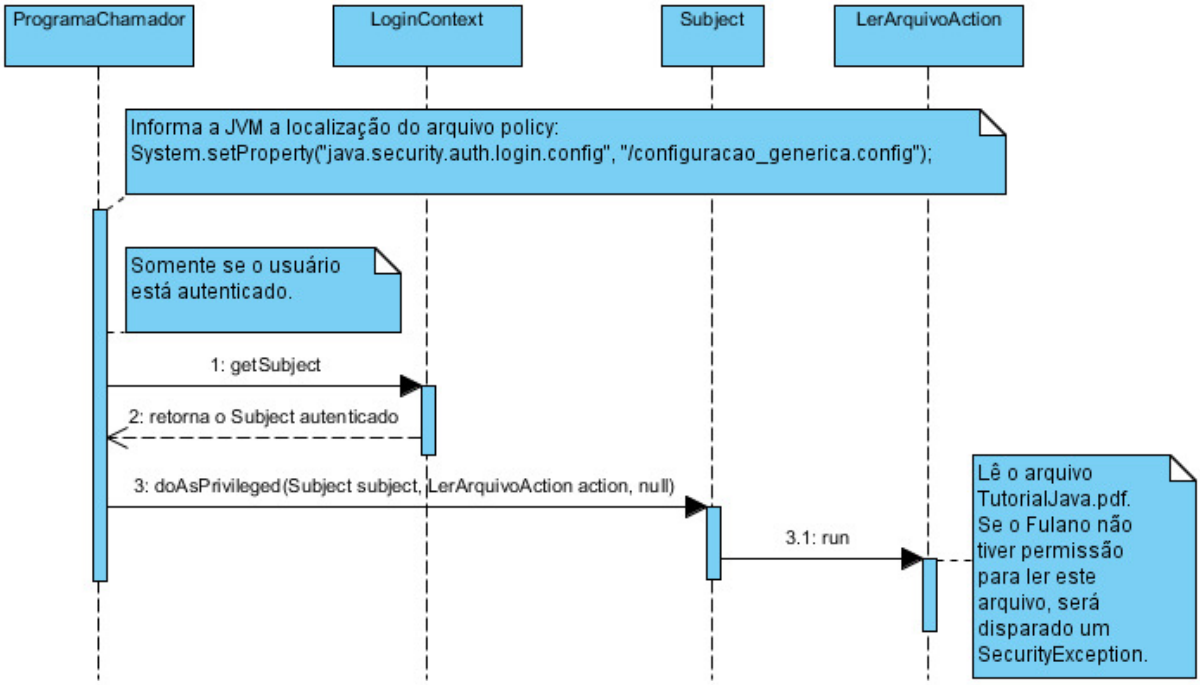

Figura 8 - Diagrama de Seqüência incluindo a classe LerArquivoAction

3.3.4 Classe `AuthPermission`

É uma classe que encapsula as permissões básicas requeridas pelo JAAS. Ela não tem uma lista das ações, tem simplesmente o nome. Se o nome existir então o programa tem a permissão, caso contrário ele não tem a permissão.

3.3.5 Classe `PrivateCredentialPermission`

Esta classe limita o acesso a cada objeto dentro do atributo `privateCredentials` da classe `Subject`. Para acessar um objeto dentro deste atributo é necessário adicionar o texto do Código 14 no arquivo *policy*:

```
grant {
     permission javax.security.auth.PrivateCredentialPermission
    "package.PrivateCredentialClass package.PrincipalClass \"O nome do
usuário\"",
          "read";
};
```

Código 14 - Permissão para acessar os objetos dentro do atributo privateCredentials.

3.3.6 *Security Manager*

Um código pode ser executado junto com um *Security Manager*, basta executar o comando `-Djava.security.manager` no console. Caso isso seja feito, o programa não tem mais acesso total a certos recursos e operações do sistema. Para recuperar o acesso é preciso definir algumas permissões no arquivo *policy*. Em seguida há alguns exemplos.

Se um `Subject` foi instanciado com o `readOnly` igual a *false* e o programa tenta mudar para *true*, ele deve chamar o método `setReadOnly()`, mas para que o sistema não dispare um `IllegalStateException` é necessário adicionar este texto no arquivo *policy*: `permission javax.security.auth.AuthPermission "setReadOnly"`.

Ainda na classe `Subject`, para poder alterar o `principals`, `publicCredentials` e `privateCredentials` é necessário ter permissão, para isso basta incluir mais três linhas no arquivo *policy*: `permission javax.security.auth.AuthPermission "modifyPrincipals"`, `permission javax.security.auth.AuthPermission "modifyPublicCredentials"` e `permission javax.security.auth.AuthPermission "modifyPrivateCredentials"`. Até mesmo para executar os métodos `getSubject(final AccessControlContext acc)`, `doAs` e `doAsPrivileged` é necessário inserir as permissões no arquivo *policy*: `permission javax.security.auth.AuthPermission "getSubject"`, `permission javax.security.auth.AuthPermission "doAs"` e `permission javax.security.auth.AuthPermission "doAsPrivileged"`.

Para simplesmente instanciar a classe `LoginContext` é necessário ter a permissão "`createLoginContext.<name>`", `name` é o nome do contexto. No exemplo acima o `name` é `JAASGenerico`: `permission javax.security.auth.AuthPermission "createLoginContext.JAASGenerico"`.

3.3.7 Processo de autorização

O processo de autorização só é executado após ter o usuário autenticado, isto é, após a execução com sucesso do método `LoginContext.login`.

Inicialmente, são definindos (1) a política de Segurança e criando (2) a classe que representa a ação, ou seja, uma classe que implemente a interface `PrivilegedAction` ou `PrivilegedExceptionAction`. Em seguida, o programa chamador informa (3) qual arquivo contém a política que ele seguirá, ele recupera (4) o `Subject` autenticado chamando o

método `LoginContext.getSubject` e executa (5) o método `doAs` ou o `doAsPrivileged` passando como parâmetro a classe que representa a ação, dentro destes métodos as permissões são associadas (6) ao `Subject`. E por fim é executado (7) o método `run` da classe que representa a ação, caso ele não tenha permissão para executar a ação, uma `SecurityException` será disparada.

3.3.8 *Policy Provider*

Igual ao *Provider* padrão para ler o arquivo de configuração do *login*, o *Policy Provider* padrão pega as definições da política de segurança de um arquivo. Mas também é possível definir outro executando o comando `System.setProperty("policy.provider", "package.NewProviderPolicyFile");`.

4. Tomcat e JAASRealm

Está seção descreve como configurar o `JAASRealm` no Tomcat. Tomcat é um servidor web para programas feitos em Java, ele pertence à Apache *Software Foundation*. O `JAASRealm` é um dos seis *Realms* disponibilizados pelo Tomcat.

Realm é um termo dado a um banco de dados ou a alguma estrutura que armazena uma lista de usuários válidos e suas senhas, e como um adicional, também armazena seus papéis (ou *roles*). Esses papéis são essenciais para definir quem tem acesso a um determinado recurso na aplicação. *Realm* é uma interface dentro do Tomcat que oferece meios para que uma aplicação recupere as informações para a autenticação, como usuários, senhas, papéis e permissões. O *Realm* oferece seis meios:

- `JDBCRealm`: Utiliza o Java *Database Connectivity* (JDBC) para recuperar as informações armazenadas em um banco de dados relacional.
- `DataSourceRealm`: Utiliza o Java *Naming Directory Interface* (JNDI) JDBC *DataSource* para recuperar as informações armazenadas em um banco de dados relacional.
- `JNDIRealm`: Utiliza o JNDI *provider* para recuperar as informações armazenadas em um diretório LDAP.
- `UserDatabaseRealm`: Recupera as informações contidas em um documento *Extensible Markup Language* (XML) (*conf/tomcat-users.xml*) utilizando o recurso JNDI.
- `MemoryReal`: Acessa informações que estão em documento XML (*conf/tomcat-users.xml*) mas que são jogadas na memória.
- `JAASRealm`: Acessas as informações para autenticação através da API JAAS.

4.1 Algumas Configurações Básicas

A seguir será mostrado um exemplo simples de configuração utilizando o `MemoryRealm`.

4.1.1 Definir *Roles* (Papéis) e Usuários

Uma das maneiras para definir os usuários é utilizando um XML, neste caso é o `tomcat-users.xml` na pasta *conf* do Tomcat. Dentro da *tag* `<tomcat_users>` são definidos

os papéis e os usuários. Outra maneira é utilizando um banco de dados ou um diretório LDAP. Neste exemplo é utilizado um XML.

A *tag* para definir um papel é a `<role>` e para um usuário é `<user>` (ver Código 15).

```
<tomcat-users>
        <role rolename="TestRole" />
        <user name = "Test" password="test" roles = "TestRole"/>
</tomcat-users>
```

Código 15 - Definir roles e users no Tomcat.

4.1.2 Habilitar a autenticação

Para que o Tomcat exija que o usuário se autentique antes de acessar um recurso restrito, basta utilizar a *tag* `<login-config>` (Código 16).

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

Código 16 - Tag para o Tomcat exigir autenricação.

Além da opção *BASIC* há outros três tipos de autenticação:

- *BASIC*: O container envia o pedido de autenticação utilizando uma tela de *login* do próprio navegador e transmite os dados do usuário codificados na base64.
- *DIGEST*: O container envia o pedido de autenticação utilizando uma tela de *login* do próprio navegador e transmite os dados do usuário criptografados.
- *CLIENT*-CERT: Igual aos dois anteriores, mas na transmissão das informações ele utiliza Certificados de Chave Pública.
- *FORM*: Permite que o formulário de *login* seja criado personalizado em HyperText Markup Language (HTML), ou seja, uma página web. Neste caso é preciso informar a página que tem o *login* e a página de erro (Código 17).

```
<login-config>
        <auth-method>FORM</auth-method>
         <form-login-config>
                <!--Página para Login-->
                <form-login-page>/formLogin.jsp</form-login-page>
                <!--Página de Erro-->
```

```
                <form-error-page>/errorLogin.jsp</form-error-page>
        </form-login-config>
</login-config>
```

Código 17 - Exemplo utilizando o método de autenticação, no Tomcat, FORM.

Esta informação é colocada no `web.xml` do seu projeto web e para este exemplo é utilizado o tipo FORM. Na hora de criar o HTML do `form` que receberá o usuário e senha, é necessário levar em conta três valores: (1) `j_security_check`, (2) `j_username` e (3) `j_password` (Código 18).

```
<form method="POST" action = "j_security_check">
        <input type="text" name="j_username"> <BR/>
        <input type="password" name="j_password"> <BR/>
        <input type="submit" value="Login">
</form>
```

Código 18 – Exemplo de form.

4.1.2 Definir as Restrições

Para especificar as restrições é utilizado a *tag* `<security-constraint>`. No Código 19 é mostrado um exemplo.

```
<security-constraint>
  <display-name>Exemplo para uma área protegida</display-name>
  <web-resource-collection>
    <web-resource-name>Area Protegida</web-resource-name>
    <url-pattern>/EasyTestJAAS.do</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
    <http-method>DELETE</http-method>
    <http-method>PUT</http-method>
    <http-method>TRACE</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>TestRole</role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

Código 19 - Definir restrições acesso para um recurso no Tomcat.

Dentro da *tag* `<web-resource-collection>` são definidos quais recursos são restringidos. Um recurso seria uma solicitação ao método *GET* do *Servlet* `EasyTestJAAS.do`. a *tag* `<url-pattern>` define os tipos de solicitações que serão restringidas. *GET*, *POST*,

DELETE, *PUT* e *TRACE* representam os tipos de solicitações HTML. As que estão dentro da *tag* `<http-method>` são permitidas para os usuários autenticados. Caso a o *GET* não estivesse definido, um usuário poderia fazer uma solicitação ao `EasyTestJAAS.do` utilizando *GET*.

Para definir quais papéis tem acesso a este recurso, é utilizado a *tag* `<auth-constraint>`.

É possível exigir que seja utilizada uma conexão segura entre o cliente e o servidor, algo como HTTPS, para isso há a *tag* `<transport-guarantee>` que é colocada dentro da *tag* `<user-data-constraint>`. Existem três valores possíveis para o `<transport-guarantee>`. *None* que quer dizer que não há proteção, *INTEGRAL* que as informações não podem sofrer mudanças durante o envio e *CONFIDENTIAL* as informações não podem ser vistas por ninguém durante a comunicação.

Para este exemplo é utilizado um exemplo mais simples (Código 20):

```
<security-constraint>
  <display-name> Exemplo para uma area protegida </display-name>
  <web-resource-collection>
    <web-resource-name>Área Protegida</web-resource-name>
    <url-pattern>/EasyTestJAAS.do</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>TestRole</role-name>
  </auth-constraint>
</security-constraint>
```

Código 20 – Restrições para a URL '/EasyTestJAAS.do'.

4.1.3 Configurar um `MemoryRealm`

O `MemoryRealm` é um dos tipos de *Realm* oferecido pelo Tomcat, para definir o tipo de *Realm*, é adicionado a *tag* `<Realm>` no arquivo `servlet.xml` na pasta *conf* do Tomcat (Código 21).

```
<Realm className="... Qual classe sera utilizada"
       ... outros parâmetros para este caso .../>
```

Código 21 – Como Definir o realm no Tomcat.

Para utilizar o `MemoryRealm` o `className` é `org.apache.catalina.realm.MemoryRealm`. Para cada tipo de *Realm* há um conjunto de atributos para serem inseridos na *tag Realm* depois do atributo `className`. Para o `MemoryRealm` há dois. O `digest` define qual criptografia foi utilizada para armazenar as senhas, se não foi utilizado nenhuma criptografia, este atributo não precisa ser definido. E O outro atributo é o `pathname`, que indica onde está o XML com os usuários e senha, se este não for definido é utilizado o padrão (`conf/tomcat-users.xml`). A configuração dos outros *Realms* não é feita neste trabalho, pois este não é foco dele. Na próxima subseção é mostrado como configurar o `JAASRealm`. Para este exemplo simples, o resultado ficou como o Código 22:

```
<Realm className="org.apache.catalina.realm.MemoryRealm" />
```

Código 22 - Utilizando MemoryRealm no Tomcat

O `MemoryRealm` já está configurado, a partir de agora se o usuário tenta acessar a página `/EasyTestJAAS.do`, o container exige que ele se autentique utilizando a página definida, *formLogin.jsp* (Figura 9 e 10).

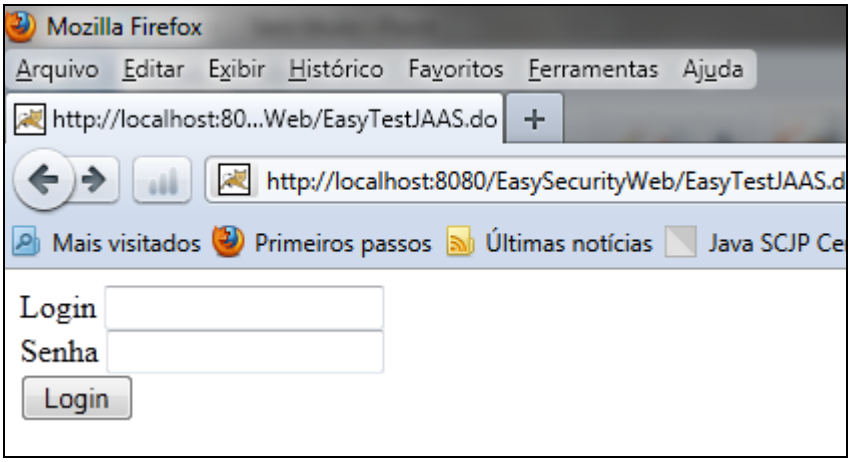

Figura 9 – Container pedindo para o usuário fornecer seu *login* e senha.

Figura 10 – O usuário *Test* está autenticado utilizando *MemoryRealm*.

Se a senha informada estiver errada, é mostrada a página definida (*errorLogin*.jsp), ver Figura 11.

Figura 11 – O usuário informou uma senha errada.

4.2 `JAASRealm`

`JAASRealm` é uma implementação da Interface *Realm* do Tomcat que utiliza o JAAS para autenticação. Para esta explicação do `JAASRealm` foi reaproveitado o que foi feito no exemplo que utilizou `MemoryRealm` com algumas modificações.

O primeiro passo é criar um *Login Module* customizado. Foi utilizado o exemplo dado em JavaTM *Authentication and Authorization Service* (JAAS) (Oracle, 2011G), a classe `SampleAcn.java`, com algumas alterações. O novo nome desta classe é `EasyLoginModule.java`.

4.2.1 Classe `EasyLoginModule`

Este *login module (módulo de login)* modificado deve ser criado num projeto separado ao seu projeto web, pois o jar dele deve ser colocado na pasta *lib* do Tomcat. O nome dele é `EasyLoginModule`.

O `EasyLoginModule` utiliza a classe `EasyUserPrincipal` no lugar da classe `SamplePrincipal`. Como o processo de autorização no Tomcat é feita por papéis, também foi criada a classe `EasyRolePrincipal`. A classe `EasyUserPrincipal` tem três atributos, `name` e `password` do tipo `String` e `roles` do tipo `Set`. E a classe `EasyRolePrincipal` tem apenas o atributo `name` do tipo `String`.

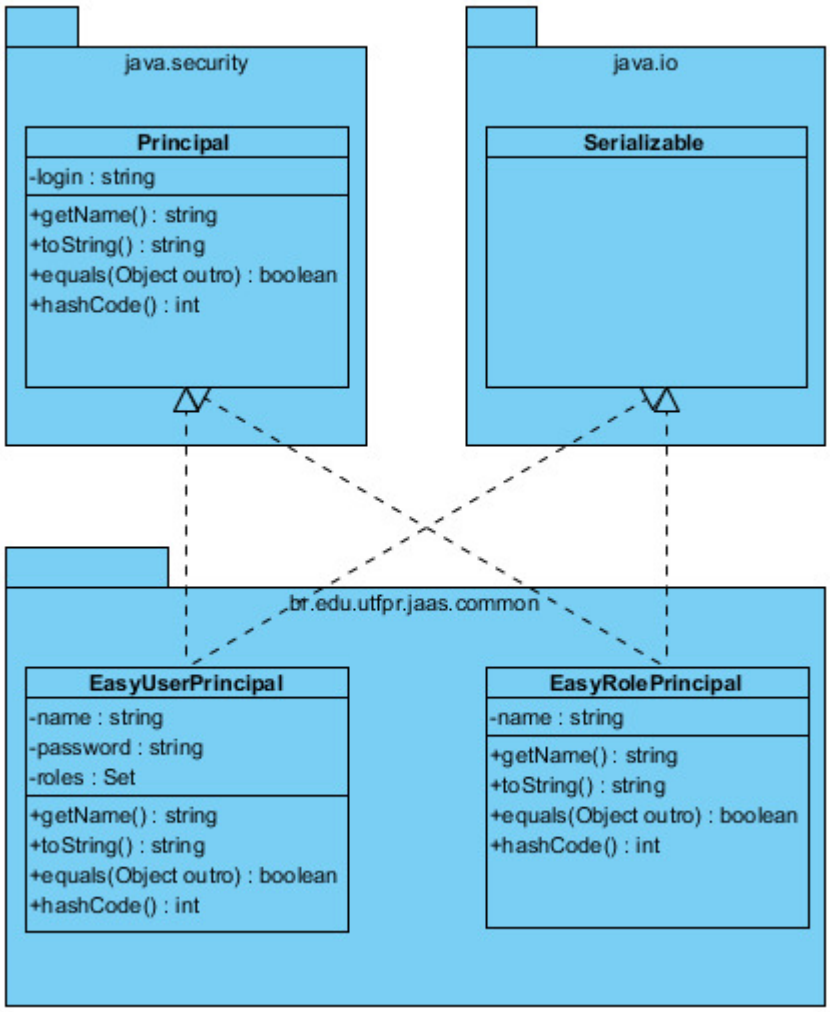

Figura 12 - Diagrama de Classes das classes EasyUserPrincipal e EasyRolePrincipal

O código foi alterado para autenticar o usuário *Test*, no lugar de *testUser*, com a senha *test*. Dentro do `Subject` também deve ser colocado o papel (*TestRole*) como `Principal`. Agora o jar deste projeto com estas novas classes `EasyLoginModule`, `EasyUserPrincipal` e `EasyRolePrincpal` deve ser colocado na pasta lib.

É necessário criar o arquivo de configuração do *login* para informar ao servidor qual *login module* será utilizado.

4.2.2 Criando o arquivo `easy_jaas.config`

Para este exemplo com o Tomcat, o nome da configuração é `EasySecurity`, o `LoginModule` é `EasyLoginModulel`e não tem o `digest=MD5` (ver Código 23).

```
EasySecurity {
  br.edu.utfpr.jaas.module.EasyLoginModule required
     debug=true;
};
```

Código 23 - Configuração do login para o exemplo EasySecurity.

O arquivo `easy_jaas. config` deve ser colocado na pasta *conf* do Tomcat. Para informar a JVM a localização do arquivo basta inserir o comando a no Código 24 no arquivo `catalina.bat` dentro da pasta *bin* do Tomcat.

```
set JAVA_OPTS=$JAVA_OPTS
-Djava.security.auth.login.config==$CATALINA_BASE/conf/easy_jaas.config
```

Código 24 – Informando o arquivo de configuração do login para o Tomcat.

O próximo passo seria configurar as restrições de acesso, mas como já foi feito no exemplo do `MemoryRealm`, não é necessário refazer.

No próximo passo será configurado o `JAASRealm`.

4.2.3 Configurando o JAASRealm

Como foi mencionado, esta configuração é feita no arquivo `servlet.xml`. Para este exemplo é utilizado a configuração no Código 25.

```
<Realm className="org.apache.catalina.realm.JAASRealm"
```

```
        appName="EasySecurity"
        userClassNames="br.edu.utfpr.jaas.common.EasyUserPrincipal"
        roleClassNames="br.edu.utfpr.jaas.common.EasyRolePrincipal"
        useContextClassLoader="false"/>
```

Código 25 - Utilizando JAASRealm no Tomcat.

O `appName` é o nome da configuração do *login* no arquivo `easy_jaas.config`. O `useContextClassLoader` igual a *false* quer dizer que é utilizado o *classloader* do container, caso contrário é utilizado a do contexto.

Para tornar a proposta mais interessante, os usuários e papeis poderiam estar armazenados em um banco de dados.

4.2.3 Recuperando usuários desde um banco de dados

O primeiro passo foi criar as tabelas no banco de dados, a tabela usuário com dois campos *name* e senha, a tabela *role* com o campo *name* e a tabela auxiliar para representar um relacionamento N por N das tabelas usuário e *roles*. Ver Figura 13.

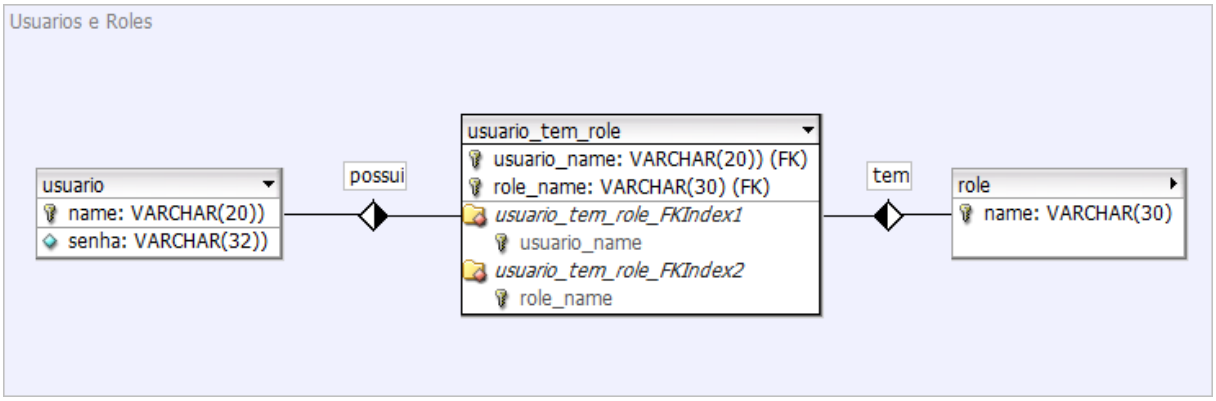

Figura 13 – Diagrama Relacional das tabelas usuário e *role*.

Depois de criado a tabela, inserido o usuário *Test* com a senha *test*, inserido o papel *TestRole* e feito o relacionamento entre estes dois registros, foi necessário criar a classe `UserDAO` com o método `retrieveUser(String name)` para recuperar os usuários com suas *roles*.

No seguinte passo foi alterada a classe `EasyLoginModule` para recuperar a senha e as *roles* do usuário utilizando o método `UserDAO.retrieveUser(String name)` e validar a senha do banco de dados com a fornecida pelo usuário. O método `commit` também foi

alterado para adicionar no `Subject` as *roles* recuperadas do banco de dados como `Principal`.

5. CONSIDERAÇÕES FINAIS

A API JAAS oferecida pela plataforma Java foi muito bem definida. Ela permite customização da segurança dos sistemas ao nível de autenticação e autorização. Por ser uma API muito abstrata, a sua implementação é um pouco confusa e difícil, mas com um pouco de experiência é possível criar implementações elegantes.

Além das implementações realizadas neste trabalho, há outras duas alternativas apresentadas pela API. Uma é a criação de um *provider* da política de segurança específico para o projeto web recuperando a política de um banco de dados. A outra é eliminar a dependência do servidor, ou seja, criar uma implementação do JAAS que seja independente do Tomcat e de qualquer outro servidor, isso poderia ser feito através do uso de filtros.

6. REFERÊNCIAS

APACHE. **Apache Tomcat 6.0 - Realm Configuration HOW-TO.** Apache. Disponível em: <http://tomcat.apache.org/tomcat-6.0-doc/realm-howto.html>. Acesso em: 19 Dez. 2010A

_____. **Apache Tomcat Configuration Reference - The Realm Component**. Apache. Disponível em: <http://tomcat.apache.org/tomcat-6.0-doc/config/realm.html>. Acesso em: 19 Dez. 2010B

_____. **Tomca API Documentation - JAASRealm**. Apache. Disponível em: <httphttp://tomcat.apache.org/tomcat-5.5-doc/catalina/docs/api/org/apache/catalina/realm/JAASRealm.html>. Acesso em: 19 Dez. 2010C

BASHAN, B.; SIERRA, K.; BATES, B. **Use a Cabeça**: Servlets & JSP: Castelo Rio de Janeiro: Alta Books, 2005. 534 p.

BBC News. **Sony investigating another hack**. BBC News. Disponível em: < http://www.bbc.co.uk/news/mobile/business-13636704>. Acesso em: 12 Jun. 2011.

CAMPOS, Fernando; NETO, Nery Signorini; CANTO, Wagner. **Web 2.0 Sob a Perspectiva da Segurança**. ITP – Instituto de Pesquisas Tecnológicas do Estado de São Paulo. Disponível em: < http://fernandocampos.pro.br/artigos/Web%202.0%20Sob%20a%20Perspectiva%20da%20Seguran%C3%A7a.pdf>. Acesso em: 3 Set. 2011.

CERT. **Cartilha de Segurança para Internet** – Parte I: Conceitos de Segurança. CERT. Disponível em: < http://cartilha.cert.br/conceitos/ >. Acesso em: 3 Set. 2011

FERREIRA, Aurélio Buarque de Holanda. **Novo Dicionário Eletrônico Aurélio**. Versão 5.0, CD-ROM. Rio de Janeiro: Positivo Informática Ltda, 2004.

GONG, Li; ELLISON, Gary; DAGEFORDE, Mary. **Inside Java™ 2 Platform Security:** Architecture, API Design, and Implementation, Second Edition. Boston: Prentice Hall, 2003.

LUFT, C. P. **Mini Dicionário Luft**. 5. ed. rev. e ampl. São Paulo: Ática e Scipione, 1990.

ORACLE. **Class Configuration** – JavaDoc. Oracle. Disponível em: < http://download.oracle.com/javase/6/docs/api/javax/security/auth/login/Configuration.html>. Acesso em: 06 Jun. 2011A.

_____. **Default Policy Implementation and Policy File Syntax**. Oracle. Disponível em: < http://download.oracle.com/javase/6/docs/technotes/guides/security/PolicyFiles.html>. Acesso em: 06 Jun. 2011B.

_____. **JAAS Authentication –** Tutorial. Oracle. Disponível em: < http://download.oracle.com/javase/6/docs/technotes/guides/security/jaas/tutorials/GeneralAcnOnly.html >. Acesso em: 04 Mar. 2011C.

_____. **JAAS Authorization –** Tutorial. Oracle. Disponível em: < http://download.oracle.com/javase/6/docs/technotes/guides/security/jaas/tutorials/GeneralAcnAndAzn.html**>.** Acesso em: 04 Mar. 2011D.

_____. **Java SE Documentation –** Security. Oracle. Disponível em: <http://download.oracle.com/javase/6/docs/technotes/guides/security/>. Acesso em: 04 Mar. 2011E.

_____. **JavaTM 2 Platform Security Architecture** – Reference Guide. Oracle. Disponível em: <http://download.oracle.com/javase/1.5.0/docs/guide/security/spec/security-spec.doc.html>. Acesso em: 04 Mar. 2011F.

_____. **JavaTM Authentication and Authorization Service (JAAS)** – Reference Guide. Oracle. Disponível em: < http://download.oracle.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html >. Acesso em: 04 Mar. 2011G.

_____. **JavaTM 2 Authentication and Authorization Service (JAAS)** – LoginModule Develorper's Guide. Oracle. Disponível em: < http://download.oracle.com/javase/1.5.0/docs/guide/security/jaas/JAASLMDevGuide.html> . Acesso em: 04 Mar. 2011H.

_____. **JavaTM 2 Platform Security Architecture** – Reference Guide. Oracle. Disponível em: <http://download.oracle.com/javase/1.5.0/docs/guide/security/spec/security-spec.doc.html>. Acesso em: 04 Mar. 2011I.

OWASP. **SQL Injection**. OWASP. Disponível em: <https://www.owasp.org/index.php/SQL_Injection>. Acesso em: 03 Set. 2011.

RIVEST, Ronald. **RFC 1321**: The MD5 Message-Digest Algorithm. Abril 1992.

APÊNDICES

APÊNDICE A – SQL para criar as tabelas e inserir os dados.

```
CREATE TABLE role (
  name VARCHAR(30) NOT NULL,
  PRIMARY KEY(name)
);

CREATE TABLE usuario (
  name VARCHAR(20) NOT NULL,
  senha VARCHAR(32) NOT NULL,
  PRIMARY KEY(name)
);

CREATE TABLE usuario_tem_role (
  usuario_name VARCHAR(20) NOT NULL,
  role_name VARCHAR(30) NOT NULL,
  PRIMARY KEY(usuario_name, role_name),
  FOREIGN KEY (usuario_name ) REFERENCES usuario ( name),
  FOREIGN KEY ( role_name ) REFERENCES role ( name)
);

INSERT INTO usuario VALUES('Test', 'test');
INSERT INTO role VALUES ('TestRole');
INSERT INTO usuario_tem_role VALUES ('Test','TestRole');
```

APÊNDICE B – Classe EasyRolePrincipal.

```
package br.edu.utfpr.jaas.common;

import java.security.Principal;

/**
 * <p> Esta classe implementa a interface <code>Principal</code> e a
representa
 * a role (papel) do usuário no sistema.
 *
 */
public class EasyRolePrincipal implements Principal, java.io.Serializable {

    /**
     * O nome da role (papel)
     */
    private String name;

    public EasyRolePrincipal() {
    }

    /**
     * Cria um objeto da classe EasyRolePrincipal com o nome da role.
     *
     * @param name (representa a role do usuário).
     * @exception NullPointerException Se o <code>name</code>
     *          é <code>null</code>.
     */
    public EasyRolePrincipal(String name) {
            if (name == null)
                throw new NullPointerException("O name não pode ser
nulo.");

            this.name = name;
    }

    /**
     * Return name.
     * @return name desta role.
     */
    public String getName() {
        return name;
    }

    /**
     * Return a <code>String</code> representando este
<code>EasyRolePrincipal</code>.
     *
     * @return a <code>String</code> representando este
<code>EasyRolePrincipal</code>.
     */
    public String toString() {
        return("EasyRolePrincipal:  " + name);
    }

    /**
     * só retorna true e as roles forem iguais.
     *
     * @param o é um objeto que será comparado com este>.
```

```
     * @return só retorna true e as roles forem iguais.
     */
    public boolean equals(Object o) {
    if (o == null)
        return false;

        if (this == o)
            return true;

        if (!(o instanceof EasyRolePrincipal))
            return false;
        EasyRolePrincipal that = (EasyRolePrincipal)o;

    if (this.getName().equals(that.getName()))
        return true;
    return false;
    }

    /**
     * Return um hash code para este <code>EasyRolePrincipal</code>.
     *
     * @return um hash code para este <code>EasyRolePrincipal</code>.
     */
    public int hashCode() {
       return name.hashCode();
    }
}
```

APÊNDICE C – Classe EasyUserPrincipal.

```
package br.edu.utfpr.jaas.common;

import java.security.Principal;
import java.util.Set;

/**
 * <p> Esta classe implementa a Interface <code>Principal</code> e
representa o
 * usuário com seus papéis, login e senha.
 *
 */
public class EasyUserPrincipal implements Principal, java.io.Serializable {

    /**
     * O Login do usuário.
     */
    private String name;

    /**
     * A senha do usuário.
     */
    private String password;

    /**
     * Os papéis (roles) do usuário.
     */
    private Set<EasyRolePrincipal> roles;

    /**
     * Cria um objeto da classe EasyUserPrincipal.
     */
    public EasyUserPrincipal(){

    }

    /**
     * Cria um objeto da classe EasyUserPrincipal com o login do usuário.
     *
     * @param name (representa o login do usuário).
     * @exception NullPointerException Se o <code>name</code>
     *          é <code>null</code>.
     */
    public EasyUserPrincipal(String name) {
            if (name == null)
                throw new NullPointerException("O login não pode ser
nulo");

            this.name = name;
    }

    /**
     * Return o login do <code>EasyUserPrincipal</code>.
     *
     * @return o login do <code>EasyUserPrincipal</code>
     */
    public String getName() {
        return name;
    }
```

```
    /**
     * Pega a senha do usuário.
     * @return o password
     */
    public String getPassword() {
                return password;
        }

    /**
     * Altera a senha do usuário.
     * @param password
     */
        public void setPassword(String password) {
                this.password = password;
        }

        /**
         * Pega os papéis do usuário.
         * @return
         */
        public Set<EasyRolePrincipal> getRoles() {
                return roles;
        }

        /**
         * Altera os papéis do usuário
         * @param roles
         */
        public void setRoles(Set<EasyRolePrincipal> roles) {
                this.roles = roles;
        }

        /**
         * Altera o login do usuário.
         * @param name
         */
        public void setName(String name) {
                this.name = name;
        }

    /**
     * Return a <code>String</code> representando este
<code>EasyUserPrincipal</code>.
     *
     * @return a <code>String</code> representando este
<code>EasyUserPrincipal</code>.
     */
    public String toString() {
        return("EasyPrincipal:  " + name);
    }

    /**
     * só retorna true e os logins forem iguais.
     *
     * @param o é um objeto que será comparado com este>.
     * @return só retorna true e os logins forem iguais.
     */
    public boolean equals(Object o) {
            if (o == null)
                return false;
```

```
                if (this == o)
                    return true;

                if (!(o instanceof EasyUserPrincipal))
                    return false;
                EasyUserPrincipal that = (EasyUserPrincipal)o;

            if (this.getName().equals(that.getName()))
                return true;
            return false;
    }

    /**
     * Return um hash code para este <code>EasyUserPrincipal</code>.
     *
     * @return um hash code para este <code>EasyUserPrincipal</code>.
     */
    public int hashCode() {
        return name.hashCode();
    }
}
```

APÊNDICE D – Classe UserDAO

```
package br.edu.utfpr.jaas.jdbc.dao;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.HashSet;
import java.util.logging.Level;
import java.util.logging.Logger;

import br.edu.utfpr.jaas.common.EasyRolePrincipal;
import br.edu.utfpr.jaas.common.EasyUserPrincipal;

public class UserDAO{
        /**
         * Conexao com o banco.
         */
        private Connection con = null;

        /**
         * Recupera o usuario que possua o name passado como parametro e
também
         * recuperará sua roles.
         * @param name
         * @return
         */
        public EasyUserPrincipal getUser( String name ) {
        PreparedStatement ps = null;
        ResultSet rs = null;
        EasyUserPrincipal usuario = null;

        try {
            StringBuffer cmd =  new StringBuffer();
            cmd.append("SELECT * FROM usuario INNER JOIN
usuario_tem_role");
            //cmd.append("SELECT * FROM usuario");
            cmd.append(" ON name = usuario_name");
            cmd.append(" WHERE name = ?");
            ps = getConnection().prepareStatement(cmd.toString());
            ps.setString(1, name);
            rs = ps.executeQuery();
            if(rs.next()){
               usuario = new EasyUserPrincipal();
               usuario = populate( rs );
               HashSet<EasyRolePrincipal> roles = new
HashSet<EasyRolePrincipal>();
               EasyRolePrincipal role1 = new
EasyRolePrincipal(rs.getString("role_name"));
               roles.add(role1);
               while(rs.next()){
                       EasyRolePrincipal role = new
EasyRolePrincipal(rs.getString("role_name"));
               }
               usuario.setRoles(roles);
            }
        } catch( SQLException e ){
            e.printStackTrace();
        } finally {
```

```
            try{
                rs.close();
                ps.close();
                closeConection();
            } catch( SQLException e ) {
                e.printStackTrace();
            } finally {
                return usuario;
            }
        }
    }

    public EasyUserPrincipal populate( ResultSet rs ) throws SQLException {
        EasyUserPrincipal usuario = new EasyUserPrincipal();

        usuario.setName( rs.getString( "name" ) );
        usuario.setPassword( rs.getString( "senha" ) );
        return usuario;
    }

    public Connection getConnection(){
                if(con == null)
                try {

                    Class.forName("org.postgresql.Driver");
                    this.con =
DriverManager.getConnection("jdbc:postgresql://localhost:5432/postgres","po
stgres","postgres");

                } catch (ClassNotFoundException e1) {

Logger.getLogger(ConexaoPostgresql.class.getName()).log(Level.SEVERE,
e1.toString(), e1);
                } catch (SQLException e) {
                    //Caso a conexao nao seja bem sucedida, SQLException
                    //fornece algumas informacoes sobre a causa do erro
                    StringBuffer msg = new StringBuffer("Informacoes de
SQLException:");
                    while (e != null){
                        msg.append("[SQLState: ");
                        msg.append(e.getSQLState());
                        msg.append("][Message:  ");
                        msg.append(e.getMessage());
                        msg.append("][Vendor:   ");
                        msg.append(e.getErrorCode());
                        msg.append("]");
                        e = e.getNextException();
                    }

Logger.getLogger(ConexaoPostgresql.class.getName()).log(Level.SEVERE,
msg.toString(), e);
                }
                return this.con;
        }

    /**
         * fecha a conexao com o banco
         */
        public void closeConection() {
        if(con != null){
            try {
```

```
                this.con.close();
                con = null;
            } catch (SQLException e) {
                //Caso a conexao seja bem sucedida, SQLException
                //fornece algumas informacoes sobre a causa do erro
                StringBuffer msg = new StringBuffer("Informacoes de
SQLException:");
                while (e != null){
                    msg.append("[SQLState: ");
                    msg.append(e.getSQLState());
                    msg.append("][Message:  ");
                    msg.append(e.getMessage());
                    msg.append("][Vendor:   ");
                    msg.append(e.getErrorCode());
                    msg.append("]");
                    e = e.getNextException();
                }

Logger.getLogger(ConexaoPostgresql.class.getName()).log(Level.SEVERE,
msg.toString(), e);
            }
        }
        }
}
```

APÊNDICE E – formLogin.jsp

```
<html>
 <head>
 </head>
 <body>
       <form method="POST" action = "j_security_check">
              Login <input type="text" name="j_username"> <BR/>
              Senha <input type="password" name="j_password"> <BR/>
              <input type="submit" value="Login">
       </form>
 </body>
</html>
```

APÊNDICE F – errorLogin.jsp

```
<html>
 <head>
 </head>
 <body>
        <%
        if (request.getUserPrincipal() != null) {
                out.println("<h1>Não foi possível autenticar o
usu&eacute;rio " + request.getUserPrincipal().getName() + "</h1>");
        } else {
                out.println("<h1>Usu&aacute;rio ou senha
inv&aacute;lida.</h1>");
        }
        %>
 </body>
</html>
```

APÊNDICE G – Servlet EasyTestJAAS

```
package br.edu.utfpr.jaas.servlet;

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

/**
 *
 * @author Carlos Samuel
 * A url deste servlet é EasyTestJAAS.do, ele mostra na tela o usuário
 * autenticado.
 */
public class EasyTestJAAS extends HttpServlet {

    /**
     * Processa a chamada dos dois métodos HTTP<code>GET</code> e
<code>POST</code>.
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Testando a Segurança</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Usuário " +
request.getUserPrincipal().getName() + " logado</h1>");
            out.println("</body>");
            out.println("</html>");
        } finally {
            out.close();
        }
    }

    /**
     *
     * Processa a chamada do método  HTTP <code>GET</code>.
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
response)
    throws ServletException, IOException {
        processRequest(request, response);
```

```
    }

    /**
     * Processa a chamada do método  HTTP <code>POST</code>.
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
response)
    throws ServletException, IOException {
        processRequest(request, response);
    }
}
```

APÊNDICE H – Classe JanelaCallbackHandler

```
package br.edu.utfpr.jaas.callback.gui;

import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;

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.TextOutputCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JPasswordField;
import javax.swing.JTextField;

public class JanelaCallbackHandler implements CallbackHandler {
    java.awt.Frame parentFrame;
    private static final int PASSWORD_WIDTH = 25;
    private PasswordCallback passwordCallback;
    private JPasswordField passwordField;
    private NameCallback loginCallback;
    private JTextField loginField;

    public JanelaCallbackHandler() {
    }

    /**
     * This method sets the frame to which the dialogs will be modal.
     */
    public void setParentFrame(java.awt.Frame pf) {
        parentFrame = pf;
    }

    public void handle(Callback[] callbacks)
                    throws UnsupportedCallbackException {
        String title = null;

        final JTextField filename = new JTextField(PASSWORD_WIDTH);
        JButton ok = new JButton("Logar");

        GridBagLayout gb = new GridBagLayout();
        GridBagConstraints c = new GridBagConstraints();
        JPanel p = new JPanel(gb);

        for (int i = 0; i < callbacks.length; i++) {
                if (callbacks[i] instanceof TextOutputCallback) {
                        title = ((TextOutputCallback)
callbacks[i]).getMessage();
                } else if (callbacks[i] instanceof NameCallback) {
                        loginCallback = (NameCallback) callbacks[i];

                        loginField = new JTextField(PASSWORD_WIDTH);
                        c.fill = GridBagConstraints.BOTH;
                        c.gridwidth = 1;
                        JLabel l = new JLabel(loginCallback.getPrompt());
                        gb.setConstraints(l, c);
```

```
                        p.add(l);

                        c.gridwidth = GridBagConstraints.REMAINDER;
                        gb.setConstraints(loginField, c);
                        p.add(loginField);
                } else if (callbacks[i] instanceof PasswordCallback) {
                        passwordCallback = (PasswordCallback) callbacks[i];

                        passwordField = new JPasswordField(PASSWORD_WIDTH);

                        c.fill = GridBagConstraints.BOTH;
                        c.gridwidth = 1;
                        JLabel l = new
JLabel(passwordCallback.getPrompt());
                        gb.setConstraints(l, c);
                        p.add(l);

                        c.gridwidth = GridBagConstraints.REMAINDER;
                        gb.setConstraints(passwordField, c);
                        p.add(passwordField);
                } else {
                        throw new
UnsupportedCallbackException(callbacks[i]);
                }
        }

        c.fill = GridBagConstraints.EAST;
        c.gridwidth = GridBagConstraints.REMAINDER;
        gb.setConstraints(ok, c);
        p.add(ok);

        final JDialog d = new JDialog(parentFrame, title, true);
        final boolean[] modalResult = new boolean[] { false };
        d.setResizable(false);
        d.getContentPane().add(p);

        ok.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent a) {
                modalResult[0] = true;
                d.setVisible(false);
                if (passwordCallback != null && passwordField.getText() !=
null
                                && passwordField.getText().length() > 0) {

passwordCallback.setPassword(passwordField.getText()
                                        .toCharArray());
                }

                if (loginCallback != null && loginField.getText() != null
                                && loginField.getText().length() > 0) {
                        loginCallback.setName(loginField.getText());
                }
                System.exit(0);
            }
        });

        java.awt.Dimension g = java.awt.Toolkit.getDefaultToolkit()
                        .getScreenSize();

        d.pack();
        d.setLocation((g.width - d.getWidth()) / 2,
```

```
                            (g.height - d.getHeight()) / 2);
        d.show();
    }
}
```

APÊNDICE I – Classe GenericoCallbackHandler

```
package br.edu.utfpr.jaas.module;

import br.edu.utfpr.jaas.common.CPFPrincipal;
import br.edu.utfpr.jaas.common.CargoPrincipal;
import br.edu.utfpr.jaas.common.LoginPrincipal;
import java.lang.reflect.InvocationTargetException;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.Principal;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;
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.TextOutputCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.FailedLoginException;
import javax.security.auth.login.LoginException;
import javax.security.auth.spi.LoginModule;

/**
 *
 * @author Carlos Samuel
 */
public class GenericoLoginModule  implements LoginModule {


    /**
     * Representa o usuário ou serviço que está tentando autenticar-se.
     */
    private Subject subject;
    /**
     * É utilizado para interagir com o usuário
     */
    private CallbackHandler callbackHandler;
    /**
     * Utilizado para compartilhar informações entre os LoginModules,
     * ele é opcional
     */
    private Map sharedState;
    /**
     * São as opções específicas que foram definidas no arquivo de
configuração
     */
    private Map options;


    /**
     * Opção definida no arquivo de configuração, se for true o sistema
     * exibirá um log.
     */
    private boolean debug = false;
    /**
     * Opção definida no arquivo de configuração, ele armazena o algoritmo
de
```

```
     * criptografia que foi utilizado na senha armazenada.
     */
    private String digest;

    // O estado da autenticação
    /**
     * Se for true, a autenticação no método login foi bem sucedida.
     */
    private boolean loginOk = false;//método login
    /**
     * Se for true, o método commit foi executado com sucesso.
     */
    private boolean commitOk = false;//Método commit

    // Informações do usuario
    /**
     * Representa a identidade do Subject, ele pode ser o login, ou o
cargo, ou o CPF.
     */
    private String identidade;
    /**
     * Após o sucesso do processo de autenticação este loginPrincipal será
     * adicionado ao Subject. Este loginPrincipal será inicializado com o
valor
     * do atributo identidade.
     */
    private Principal loginPrincipal;

    private static String CLASS_LOGIN_PRINCIPAL_NAME =
"br.edu.utfpr.jaas.common.LoginPrincipal";
    private static String CLASS_CARGO_PRINCIPAL_NAME  =
"br.edu.utfpr.jaas.common.CargoPrincipal";
    private static String CLASS_CPF_PRINCIPAL_NAME =
"br.edu.utfpr.jaas.common.CPFPrincipal";
    private static String CLASS_JANELA_CALLBACK_HANDLER_NAME  =
"br.edu.utfpr.jaas.callback.gui.JanelaCallbackHandler";
    private static String classPrincipalToBeUsed =
CLASS_LOGIN_PRINCIPAL_NAME;
    private String classCallbackHandlerName =
CLASS_JANELA_CALLBACK_HANDLER_NAME;


    private CargoPrincipal cargoPrincipal = new
CargoPrincipal("administrador");
    private CPFPrincipal cPFPrincipal = new CPFPrincipal("12312312345");

    public void initialize(Subject subject, CallbackHandler
callbackHandler, Map<String, ?> sharedState, Map<String, ?> options) {
        this.subject = subject;
        this.callbackHandler = callbackHandler;
        this.sharedState = sharedState;
        this.options = options;

        // Pegando os valores dos options
        if((String)options.get("debug") != null){
            debug = "true".equalsIgnoreCase((String)options.get("debug"));
        }


        if((String)options.get("digest") != null){
            digest = ((String)options.get("digest"));
```

```
        }

        String auxClassPrincipalName =
(String)options.get("classPrincipalToBeUsed");
        if(auxClassPrincipalName != null &&
auxClassPrincipalName.length()>0 ){
                this.classPrincipalToBeUsed = auxClassPrincipalName;
        }

        String auxClassCallbackHandler =
(String)options.get("classCallbackHandlerName");
        if(auxClassCallbackHandler != null &&
auxClassCallbackHandler.length()>0 ){
                this.classCallbackHandlerName = auxClassPrincipalName;
        }
    }

    public boolean login() throws LoginException {
        printLog("Inicia o método login.");
        // prompt for a user name and password
        if (callbackHandler == null){
            if(classCallbackHandlerName != null &&
classCallbackHandlerName.length()>0){
                try {
                    Class clCallbackHandler =
Class.forName(classCallbackHandlerName);
                    // get the constructor with one parameter
                    java.lang.reflect.Constructor
constructorCallbackHandler = clCallbackHandler.getConstructor(new
Class[]{});
                    callbackHandler = (CallbackHandler)
constructorCallbackHandler.newInstance(new Object[]{});
                } catch (InstantiationException ex) {

Logger.getLogger(GenericoLoginModule.class.getName()).log(Level.SEVERE,
null, ex);
                } catch (IllegalAccessException ex) {

Logger.getLogger(GenericoLoginModule.class.getName()).log(Level.SEVERE,
null, ex);
                } catch (IllegalArgumentException ex) {

Logger.getLogger(GenericoLoginModule.class.getName()).log(Level.SEVERE,
null, ex);
                } catch (InvocationTargetException ex) {

Logger.getLogger(GenericoLoginModule.class.getName()).log(Level.SEVERE,
null, ex);
                }catch (NoSuchMethodException ex) {

Logger.getLogger(GenericoLoginModule.class.getName()).log(Level.SEVERE,
null, ex);
                } catch (SecurityException ex) {

Logger.getLogger(GenericoLoginModule.class.getName()).log(Level.SEVERE,
null, ex);
                } catch (ClassNotFoundException ex) {

Logger.getLogger(GenericoLoginModule.class.getName()).log(Level.SEVERE,
null, ex);
                }
```

```
            } else {
                throw new LoginException("Error: no CallbackHandler
available " +
                        "to garner authentication information from the
user");
            }
        }

        Callback [] callbacks = new Callback[3];
        callbacks[0] = new
TextOutputCallback(TextOutputCallback.INFORMATION,"Tela de Login");
        callbacks[1] = new NameCallback("Login: ");
        callbacks[2] = new PasswordCallback("Senha: ", false);


        char [] senha = null;

        try {
            callbackHandler.handle(callbacks);
            printLog("Login: "+((NameCallback)callbacks[1]).getName());
            identidade = ((NameCallback)callbacks[1]).getName();
            ((PasswordCallback)callbacks[2]).getPassword();
            printLog("Senha: "+ new
String(((PasswordCallback)callbacks[2]).getPassword()));
            senha = ((PasswordCallback)callbacks[2]).getPassword();
            if (senha == null) {
                senha = new char[0];
            }


        } catch (java.io.IOException ioe) {
            throw new LoginException(ioe.toString());
        } catch (UnsupportedCallbackException uce) {
            throw new LoginException("Error: " +
uce.getCallback().toString() +
            " not available to garner authentication information " +
            "from the user");
        }

        // print debugging information
        if (debug) {
            System.out.println("\t\t[GenericoLoginModule] Login informado:
" +
                    identidade);
        }

        // Verificando login e senha
        boolean loginCorreto = false;
        //UserDAO userDAO = new UserDAO();
        //this.easyUserPrincipal = userDAO.getUser(username);
        //if (this.easyUserPrincipal != null) {
        //    usernameCorrect = true;
        //}
        //String senhaArmazenada = easyUserPrincipal.getPassword();

        if(identidade != null && identidade.length()>0){
            loginCorreto = true;
        }
        String senhaArmazenada =
"732002CEC7AEB7987BDE842B9E00EE3B";//senha1234 em MD5
```

```
        String auxSenha = new String(senha);


        if(digest != null && digest.length()>0){
            try {
                MessageDigest md = MessageDigest.getInstance(digest);
                byte [] hash = md.digest(new String (senha).getBytes());
                auxSenha = new String (hexCodes(hash));
            } catch (NoSuchAlgorithmException ex) {

Logger.getLogger(GenericoLoginModule.class.getName()).log(Level.SEVERE,
null, ex);
            }
        }

        if (loginCorreto && auxSenha.equals(senhaArmazenada)) {
            loginOk = true;
            if (debug){
                System.out.println("\t\t[GenricoLoginModule] " +
                        "Processo de autenticação executado com sucesso.");
            }
        } else { //Processo de autenticação falhou
            identidade = null;
            ((PasswordCallback)callbacks[2]).clearPassword();
            if (debug){
                System.out.println("\t\t[GenricoLoginModule] " +
                        "Processo de autenticação falhou");
            }

            if (!loginCorreto) {
                    throw new FailedLoginException("Login Incorreto");
            } else {
                    throw new FailedLoginException("Senha Incorreta");
            }
        }
        return loginOk;
    }

    public boolean commit() throws LoginException {
        if (loginOk == false) {
            commitOk = false;
        } else {
            Class clPrincipal;
            try {
                clPrincipal = Class.forName(classPrincipalToBeUsed);
                java.lang.reflect.Constructor constructorPrincipal =
clPrincipal.getConstructor(new Class[] {String.class});

                loginPrincipal = (Principal)
constructorPrincipal.newInstance(new Object[]{identidade});
                subject.getPrincipals().add(loginPrincipal);
                subject.getPrincipals().add(cargoPrincipal);
                subject.getPrincipals().add(cPFPrincipal);


                if (debug){
                    System.out.println("\t\t[GenricoLoginModule] " +
                            "Principals adicionado ao Subject.");
                }
```

```
                commitOk = true;
            } catch (ClassNotFoundException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            } catch (SecurityException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            } catch (NoSuchMethodException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            } catch (IllegalArgumentException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            } catch (InstantiationException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            } catch (IllegalAccessException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            } catch (InvocationTargetException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            }
        }
        return commitOk;
    }

    public boolean abort() throws LoginException {
        boolean result = false;
        if (loginOk == false) {
            identidade = null;//Limpar
        } else if (loginOk == true && commitOk == false) {
            result = true;
            loginOk = false;
            identidade = null;//Limpar
        } else {
            result = true;
            logout();
        }
        return result;
    }

    public boolean logout() throws LoginException {

        subject.getPrincipals().remove(loginPrincipal);
        subject.getPrincipals().remove(cargoPrincipal);
        subject.getPrincipals().remove(cPFPrincipal);
        loginOk = false;
        commitOk = false;
        identidade = null;//Limpar
        return true;
    }


    private static char[] hexCodes(byte[] text) {

        char[] hexOutput = new char[text.length * 2];
        String hexString;
        for (int i = 0; i < text.length; i++) {
            hexString = "00" + Integer.toHexString(text[i]);
```

```
            hexString.toUpperCase().getChars(
                    hexString.length() - 2, hexString.length(), hexOutput,
i * 2);
         }
         return hexOutput;

    }

    private void printLog(String message){
       if (debug){
            System.out.println("\t\t[GenricoLoginModule] " + message);
         }
    }
}
```

APÊNDICE J – Classes LoginPrincipal, CargoPrincipal, CPFPrincipal

```
/*
 * Esta classe implementa a interface principal e representa o login do
usuário
 * (Subject) de um sistema. este login é uma identidade do usuário.
 */

package br.edu.utfpr.jaas.common;

import java.io.Serializable;
import java.security.Principal;

/**
 * @author Carlos Samuel
 */
public class LoginPrincipal implements Principal, Serializable {

    /**
     * serial version padrão
     */
    private static final long serialVersionUID = 1L;
    /**
     * Representa o login.
     */
    private String login;

    public LoginPrincipal(String login) {
            if (login == null)
                throw new NullPointerException("O login não pode ser
nulo!");
            this.login = login;
    }

    /**
     * Implementando o método abstrato da interface Principal para retornar
o
     * name.
     */
    public String getName() {
       return this.login;
    }

    /**
     * Retorna o name;
     */
    public String toString() {
        return this.getName();
    }

    /**
     * Se o name do objeto outro é igual ao name deste objeto, então
retorna true,
     * caso contrário retorna false.
     *
     * @param um objeto do tipo <code>LoginPrincipal</code>.
     *
     * @return true se os names são iguais.
     */
    public boolean equals(Object outro) {
        boolean isEqual = false;
```

```
        if (outro != null && outro instanceof LoginPrincipal){
                isEqual =
this.getName().equals(((LoginPrincipal)outro).getName());
        }
        return isEqual;
    }

    /**
     * Retorna o hash code do retorno do getName().
     *
     * @return hash code do retorno do getName().
     */
    public int hashCode() {
        return this.getName().hashCode();
    }
}
```

```
/*
 * Esta classe implementa a interface principal e representa o cargo do
usuário
 * (Subject)de um sistema. este cargo será um identidade do usuário.
 */

package br.edu.utfpr.jaas.common;

import java.io.Serializable;
import java.security.Principal;

/**
 * @author Carlos Samuel
 */
public class CargoPrincipal implements Principal, Serializable {

    /**
     * serial version padrão
     */
    private static final long serialVersionUID = 1L;
    /**
     * Representa o cargo.
     */
    private String cargo;

    public CargoPrincipal(String cargo) {
            if (cargo == null)
                throw new NullPointerException("O cargo não pode ser
nulo!");
            this.cargo = cargo;
    }

    /**
     * Implementando o método abstrato da interface Principal para retornar
o
     * name.
     */
    public String getName() {
       return this.cargo;
    }

    /**
```

```
     * Retorna o name;
     */
    public String toString() {
        return this.getName();
    }

    /**
     * Se o name do objeto outro é igual ao name deste objeto, então
retorna true,
     * caso contrário retorna false.
     *
     * @param um objeto do tipo <code>LoginPrincipal</code>.
     *
     * @return true se os names são iguais.
     */
    public boolean equals(Object outro) {
        boolean isEqual = false;
        if (outro != null && outro instanceof CargoPrincipal){
                isEqual =
this.getName().equals(((CargoPrincipal)outro).getName());
        }
        return isEqual;
    }

    /**
     * Retorna o hash code do retorno do getName().
     *
     * @return hash code do retorno do getName().
     */
    public int hashCode() {
        return this.getName().hashCode();
    }
}
```

```
/*
 * Esta classe implementa a interface principal e representa o CPF do
usuário
 * (Subject) de um sistema. este cargo será um identidade do usuário.
 */

package br.edu.utfpr.jaas.common;

import java.io.Serializable;
import java.security.Principal;

/**
 * @author Carlos Samuel
 */
public class CPFPrincipal implements Principal, Serializable {

    /**
     * serial version padrão
     */
    private static final long serialVersionUID = 1L;
    /**
     * Representa o CPF.
     */
    private String cpf;
```

```
    public CPFPrincipal(String cpf) {
            if (cpf == null)
                throw new NullPointerException("O cargo não pode ser
nulo!");
            this.cpf = cpf;
    }

    /**
     * Implementando o método abstrato da interface Principal para retornar
o
     * name.
     */
    public String getName() {
       return this.cpf;
    }

    /**
     * Retorna o name;
     */
    public String toString() {
        return this.getName();
    }

    /**
     * Se o name do objeto outro é igual ao name deste objeto, então
retorna true,
     * caso contrário retorna false.
     *
     * @param um objeto do tipo <code>LoginPrincipal</code>.
     *
     * @return true se os names são iguais.
     */
    public boolean equals(Object outro) {
        boolean isEqual = false;
        if (outro != null && outro instanceof CPFPrincipal){
                isEqual =
this.getName().equals(((CPFPrincipal)outro).getName());
        }
        return isEqual;
    }

    /**
     * Retorna o hash code do retorno do getName().
     *
     * @return hash code do retorno do getName().
     */
    public int hashCode() {
        return this.getName().hashCode();
    }
}
```

ANEXOS

ANEXO A – Classe EasyLoginModule. Esta classe é SampleLoginModule alterada.

Para achar as alterações, procure por Código Alterado.

```
/*
 * @(#)SampleLoginModule.java   1.18 00/01/11
 *
 * Copyright 2000-2002 Oracle and/or its affiliates. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or
 * without modification, are permitted provided that the following
 * conditions are met:
 *
 * -Redistributions of source code must retain the above copyright
 * notice, this  list of conditions and the following disclaimer.
 *
 * -Redistribution in binary form must reproduct the above copyright
 * notice, this list of conditions and the following disclaimer in
 * the documentation and/or other materials provided with the
 * distribution.
 *
 * Neither the name of Oracle and/or its affiliates. or the names of
 * contributors may be used to endorse or promote products derived
 * from this software without specific prior written permission.
 *
 * This software is provided "AS IS," without a warranty of any
 * kind. ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND
 * WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY
 * EXCLUDED. SUN AND ITS LICENSORS SHALL NOT BE LIABLE FOR ANY
 * DAMAGES OR LIABILITIES  SUFFERED BY LICENSEE AS A RESULT OF  OR
 * RELATING TO USE, MODIFICATION OR DISTRIBUTION OF THE SOFTWARE OR
 * ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE
 * FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT, INDIRECT,
 * SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER
 * CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF
 * THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN
 * ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
 *
 * You acknowledge that Software is not designed, licensed or
 * intended for use in the design, construction, operation or
 * maintenance of any nuclear facility.
 */

package br.edu.utfpr.jaas.module;

import java.lang.reflect.InvocationTargetException;
import java.security.Principal;
import java.util.Iterator;
import java.util.Map;

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 javax.security.auth.spi.LoginModule;

import br.edu.utfpr.jaas.common.EasyRolePrincipal;
import br.edu.utfpr.jaas.common.EasyUserPrincipal;
import br.edu.utfpr.jaas.jdbc.dao.UserDAO;

/**
 * <p> This sample LoginModule authenticates users with a password.
 *
 */
public class EasyLoginModule implements LoginModule {

    // initial state
    private Subject subject;
    private CallbackHandler callbackHandler;
    private Map sharedState;
    private Map options;

    // configurable option
    private boolean debug = false;

    // the authentication status
    private boolean succeeded = false;
    private boolean commitSucceeded = false;

    // username and password
    private String username;
    private char[] password;


//#########################################################################
//######################### Início Código Alterado ########################
//#########################################################################

    private EasyUserPrincipal userPrincipal;
    private EasyRolePrincipal rolePrincipal;

//#########################################################################
//######################### Fim Código Alterado ###########################
//#########################################################################

    /**
     * Initialize this <code>LoginModule</code>.
     *
     * <p>
     *
     * @param subject the <code>Subject</code> to be authenticated. <p>
     *
     * @param callbackHandler a <code>CallbackHandler</code> for
communicating
     *          with the end user (prompting for user names and
     *          passwords, for example). <p>
     *
     * @param sharedState shared <code>LoginModule</code> state. <p>
     *
     * @param options options specified in the login
     *          <code>Configuration</code> for this particular
```

```
     *          <code>LoginModule</code>.
     */
    public void initialize(Subject subject, CallbackHandler
callbackHandler,
            Map sharedState, Map options) {

    this.subject = subject;
    this.callbackHandler = callbackHandler;
    this.sharedState = sharedState;
    this.options = options;

    // initialize any configured options
    debug = "true".equalsIgnoreCase((String)options.get("debug"));
    }

    /**
     * Authenticate the user by prompting for a user name and password.
     *
     * <p>
     *
     * @return true in all cases since this <code>LoginModule</code>
     *      should not be ignored.
     *
     * @exception FailedLoginException if the authentication fails. <p>
     *
     * @exception LoginException if this <code>LoginModule</code>
     *      is unable to perform the authentication.
     */
    public boolean login() throws LoginException {


            // prompt for a user name and password
            if (callbackHandler == null)
                throw new LoginException("Error: no CallbackHandler
available " +
                    "to garner authentication information from the user");

                    Callback[] callbacks = new Callback[2];
                    callbacks[0] = new NameCallback("user name: ");
                    callbacks[1] = new PasswordCallback("password: ",
false);


                    try {
                        callbackHandler.handle(callbacks);
                        username = ((NameCallback)callbacks[0]).getName();
                        char[] tmpPassword =
((PasswordCallback)callbacks[1]).getPassword();
                        if (tmpPassword == null) {
                        // treat a NULL password as an empty password
                        tmpPassword = new char[0];
                        }
                        password = new char[tmpPassword.length];
                        System.arraycopy(tmpPassword, 0,
                            password, 0, tmpPassword.length);
                        ((PasswordCallback)callbacks[1]).clearPassword();

                    } catch (java.io.IOException ioe) {
                        throw new LoginException(ioe.toString());
                    } catch (UnsupportedCallbackException uce) {
```

```
                        throw new LoginException("Error: " +
uce.getCallback().toString() +
                        " not available to garner authentication information
" +
                        "from the user");
                    }

                    // print debugging information
                    if (debug) {
                        System.out.println("\t\t[EasyLoginModule] " +
                                "user entered user name: " +
                                username);
                        System.out.print("\t\t[EasyLoginModule] " +
                                "user entered password: ");
                        for (int i = 0; i < password.length; i++)
                        System.out.print(password[i]);
                        System.out.println();
                    }

                    // verify the username/password
                    boolean usernameCorrect = false;
                    boolean passwordCorrect = false;

//##########################################################################
//######################### Início Código Alterado ###########################
//##########################################################################

                    //Pegar a senha do banco de dados
                    UserDAO userDAO = new UserDAO();
                    this.userPrincipal = (EasyUserPrincipal)
userDAO.getUser(username);
                    //Verifica se exist o usuário
                    if (this.userPrincipal != null) {
                        usernameCorrect = true;
                    }
                    String auxPassword = new String(password);
                    //Verifica a senha
                    if (usernameCorrect &&
auxPassword.equals(userPrincipal.getPassword())) {

//##########################################################################
//######################### Fim Código Alterado ##############################
//##########################################################################

                        // authentication succeeded!!!
                        passwordCorrect = true;
                        if (debug)
                        System.out.println("\t\t[EasyLoginModule] " +
                                "authentication succeeded");
                        succeeded = true;
                        return true;
                    } else {

                        // authentication failed -- clean out state
                        if (debug)
                        System.out.println("\t\t[EasyLoginModule] " +
                                "authentication failed");
                        succeeded = false;
                        username = null;
                        for (int i = 0; i < password.length; i++)
                        password[i] = ' ';
```

```
                        password = null;
                        if (!usernameCorrect) {
                                throw new FailedLoginException("User Name
Incorrect");
                        } else {
                                throw new FailedLoginException("Password
Incorrect");
                        }
                    }
    }

    /**
     * <p> This method is called if the LoginContext's
     * overall authentication succeeded
     * (the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL
LoginModules
     * succeeded).
     *
     * <p> If this LoginModule's own authentication attempt
     * succeeded (checked by retrieving the private state saved by the
     * <code>login</code> method), then this method associates a
     * <code>SamplePrincipal</code>
     * with the <code>Subject</code> located in the
     * <code>LoginModule</code>.  If this LoginModule's own
     * authentication attempted failed, then this method removes
     * any state that was originally saved.
     *
     * <p>
     *
     * @exception LoginException if the commit fails.
     *
     * @return true if this LoginModule's own login and commit
     *      attempts succeeded, or false otherwise.
     */
    public boolean commit() throws LoginException {
            if (succeeded == false) {
                return false;
            } else {
        // add a Principal (authenticated identity)
        // to the Subject


//#########################################################################
//######################## Início Código Alterado ##########################
//#########################################################################


                        //Adiciona ao Subject o login como identidade.
                        if
(!subject.getPrincipals().contains(userPrincipal))
                subject.getPrincipals().add(userPrincipal);

                        //Os papeis também devem ser adicionados ao Subject
como identidade
                Iterator<EasyRolePrincipal> iteratorUserRoles =
userPrincipal.getRoles().iterator();

                while (iteratorUserRoles.hasNext()) {
                                EasyRolePrincipal role =
iteratorUserRoles.next();
                        if (!subject.getPrincipals().contains(role))
```

```
                                subject.getPrincipals().add(role);
                }

//#########################################################################
//######################### Fim Código Alterado ###########################
//#########################################################################

                if (debug) {
                System.out.println("\t\t[EasyLoginModule] " +
                        "added Principal to Subject");
                }

                // in any case, clean out state
                username = null;
                for (int i = 0; i < password.length; i++)
                password[i] = ' ';
                password = null;

                commitSucceeded = true;
                return true;
            }
        }

        /**
         * <p> This method is called if the LoginContext's
         * overall authentication failed.
         * (the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL
LoginModules
         * did not succeed).
         *
         * <p> If this LoginModule's own authentication attempt
         * succeeded (checked by retrieving the private state saved by the
         * <code>login</code> and <code>commit</code> methods),
         * then this method cleans up any state that was originally saved.
         *
         * <p>
         *
         * @exception LoginException if the abort fails.
         *
         * @return false if this LoginModule's own login and/or commit attempts
         *      failed, and true otherwise.
         */
        public boolean abort() throws LoginException {
        if (succeeded == false) {
            return false;
        } else if (succeeded == true && commitSucceeded == false) {
            // login succeeded but overall authentication failed
            succeeded = false;
            username = null;
            if (password != null) {
            for (int i = 0; i < password.length; i++)
                password[i] = ' ';
            password = null;
            }
            userPrincipal = null;
        } else {
            // overall authentication succeeded and commit succeeded,
            // but someone else's commit failed
            logout();
        }
        return true;
```

```
    }

    /**
     * Logout the user.
     *
     * <p> This method removes the <code>EasyUserPrincipal</code>
     * that was added by the <code>commit</code> method.
     *
     * <p>
     *
     * @exception LoginException if the logout fails.
     *
     * @return true in all cases since this <code>LoginModule</code>
     *          should not be ignored.
     */
    public boolean logout() throws LoginException {

    subject.getPrincipals().remove(userPrincipal);
    succeeded = false;
    succeeded = commitSucceeded;
    username = null;
    if (password != null) {
        for (int i = 0; i < password.length; i++)
        password[i] = ' ';
        password = null;
    }
    userPrincipal = null;
    return true;
    }
}
```