Plataforma do Aplicativo JBoss Enterprise 5

Guia de Segurança

para uso com a Plataforma do Aplicativo JBoss Enterprise 5

Edição 5.1.0

Last Updated: 2017-10-25

Plataforma do Aplicativo JBoss Enterprise 5 Guia de Segurança

para uso com a Plataforma do Aplicativo JBoss Enterprise 5
Edição 5.1.0

Anil Saldhana

Jaikiran Pai

Marcus Moyses

Peter Skopek

Stephan Mueller

Jared Morgan
Engineering Content Services
jmorgan@redhat.com

Joshua Wulf
Engineering Content Services
jwulf@redhat.com

Nota Legal

Resumo

O Guia de Segurança é destinado aos Administradores de Sistema e Desenvolvedores. Ele explica como implementar segurança na Plataforma do Aplicativo JBoss Enterprise 5 e suas liberações do patch. Além disso, este guia cobre a Segurança do Aplicativo JBoss Enterprise: uma introdução à Autenticação do Java e Serviço de Autorização, o Modelo de Segurança e Extensão da Arquitetura, gerenciamento e configuração dos Domínios de Segurança, substituição de senhas de texto por máscaras em arquivos de configuração, além do uso do SSL para proteger a Invocação do Método Remoto dos EJBs.

Índice

PARTE I. VISÃO GERAL DA SEGURANÇA

A Segurança é a parte fundamental de qualquer aplicativo empresarial. Você deve estar apto à restringir o acesso a seus aplicativos e controlar quais operações os usuários do aplicativo podem executar.

A especificação *Java Enterprise Edition* (Java EE) define um modelo de segurança baseado na função simples para *Enterprise Java Beans* (EJBs) e para os componentes da web. O framework *JBoss Security Extension* (JBossSX) manuseia a segurança da plataforma e fornece suporte para ambos modelo de segurança do Java EE declarativo e integração de segurança personalizada através da camada proxy de segurança.

A implantação padrão do modelo de segurança declarativo é baseado nos assuntos e modelos de logon do Serviço de Autorização e Autenticação do Java - *Java Authentication and Authorization Service* (JAAS). A camada proxy de segurança permite proteção personalizada que não pode ser descrita usando o modelo declarativo a ser adicionado a um EJB numa maneira que é independente do objeto comercial do EJB.

NOTA

Os modelos de segurança de especificação servlet e o EJB, assim como o JAAS, estão descritos em maiores detalhes na Parte I, “Visão Geral da Segurança”.

CAPÍTULO 1. VISÃO GERAL DE PROTEÇÃO DECLARATIVA DO JAVA EE

Ao invés de incorporar a segurança em seu componente comercial, o modelo de segurança do Java EE é declarativo: você descreve as funções de segurança e permissões num descritor XML padrão. Isto isola a segurança a partir do código de nível comercial uma vez que a segurança tende mais a ser uma função de onde o componente é implantado do que um aspecto inerente de lógica comercial do componente.

Por exemplo, considere um componente de caixa eletrônico usado para acessar uma conta bancária. As solicitações de segurança, funções e permissões do componente irão variar independentemente de como você acessa a conta bancária. Como você acessa sua informação de conta pode também variar dependendo de qual banco a conta é gerenciada ou onde o caixa eletrônico está localizado.

A proteção do aplicativo Java EE é baseada na especificação das solicitações de segurança através dos descritores de implantação do Java EE padrão. Você pode proteger o acesso aos EJBs e componentes da web num aplicativo empresarial pelo uso dos descritores de implantação **`ejb-jar.xml`** e **`web.xml`**. As seguintes seções descrevem o propósito e uso de vários elementos de segurança.

1.1. REFERÊNCIAS DE SEGURANÇA

Ambos os Enterprise Java Beans (EJBs) e servlets podem declarar um ou mais elementos <security-role-ref>. A Figura 1.1, “O elemento <security-role-ref>” descreve o elemento <security-role-ref>, seus elementos filho e atributos.

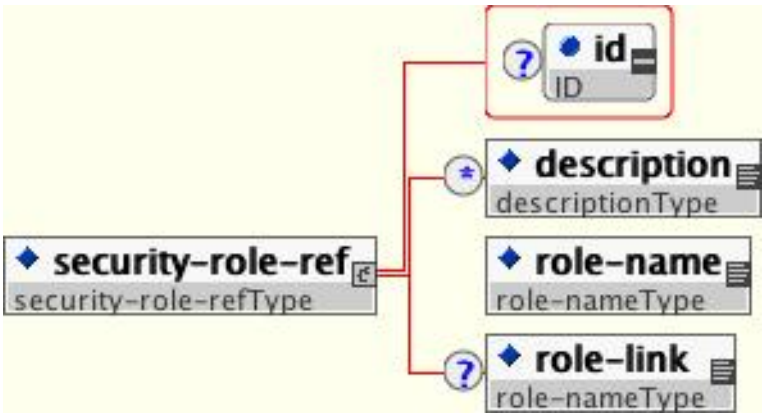


Figura 1.1. O elemento <security-role-ref>

Este elemento declara que um componente está usando o valor do atributo **`role-nameType`** do elemento <role-name> como um argumento ao método **`isCallerInRole(String)`**. Um componente pode verificar se o chamador está na função declarada pelo elemento <role-name> ou <security-role-ref>. O valor do elemento <role-name> deve conectar-se ao elemento através do elemento <role-link>. O uso típico do **`isCallerInRole`** é executar uma checagem de segurança que não pode ser definida pelo uso dos elementos <method-permissions> baseados na função.

O Exemplo 1.1, “fragmento do descritor ejb-jar.xml” descreve o uso do <security-role-ref> num arquivo **`ejb-jar.xml`**.

Exemplo 1.1. fragmento do descritor ejb-jar.xml

```
<!-- A sample ejb-jar.xml fragment -->
<ejb-jar>
```

```
  <enterprise-beans>
    <session>
      <ejb-name>ASessionBean</ejb-name>
      ...
      <security-role-ref>
          <role-name>TheRoleICheck</role-name>
          <role-link>TheApplicationRole</role-link>
      </security-role-ref>
    </session>
  </enterprise-beans>
  ...
</ejb-jar>
```

O Exemplo 1.2, “fragmento do descritor web.xml” apresenta o uso do <security-role-ref> num arquivo **web.xml**.

NOTA

Este fragmento é apenas uma amostra. Nas implantações, os elementos nesta seção devem conter os nomes da função e conexões relevantes à implantação EJB.

Exemplo 1.2. fragmento do descritor web.xml

```
<web-app>
    <servlet>
        <servlet-name>AServlet</servlet-name>
        ...
        <security-role-ref>
            <role-name>TheServletRole</role-name>
            <role-link>TheApplicationRole</role-link>
        </security-role-ref>
    </servlet>
    ...
</web-app>
```

1.2. IDENTIDADE DE SEGURANÇA

Um Enterprise Java Bean (EJB) pode especificar a identidade que outro EJB deve usar quando invocando métodos em componentes usando o elemento <security-identity>. A Figura 1.2, “O elemento de identidade de segurança” descreve o elemento <security-identity>, seus elementos filho, e atributos.

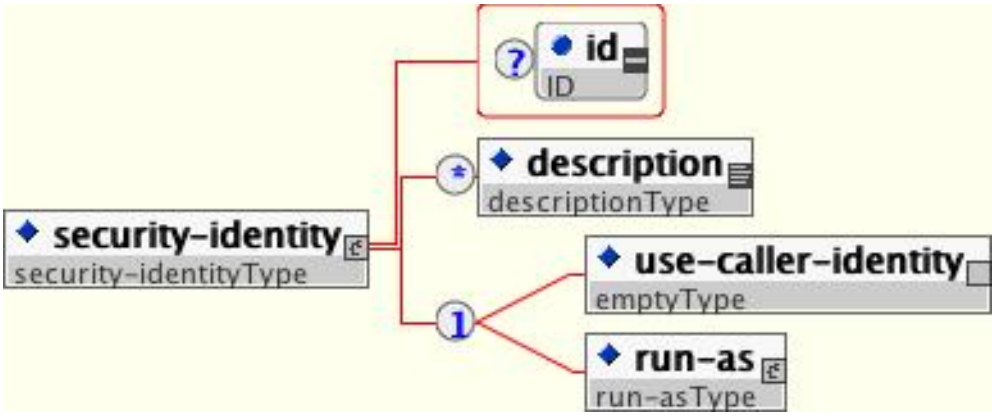


Figura 1.2. O elemento de identidade de segurança

A identidade de invocação pode ser a do chamador atual ou pode ser uma função específica. O assembler do aplicativo usa o elemento <security-identity> com um elemento filho <use-caller-identity>. Isto indica que a identidade do chamador atual deve ser propagada como uma identidade de segurança para as invocações do método realizadas pelo EJB. A propagação da identidade do chamador é o padrão usado na ausência de uma declaração do elemento <security-identity>.

Alternativamente, o assembler do aplicativo pode usar o elemento filho <run-as> ou <role-name> para especificar que uma função de segurança específica suprida pelo valor do elemento role-name> deve ser usada como uma identidade de segurança para invocações de método realizadas pelo EJB.

Perceba que isto não altera a identidade do chamador conforme visto pelo método **`EJBContext.getCallerPrincipal()`**. Ao invés disto, as funções de segurança do chamador são configuradas à função única especificada pelo valor do elemento <run-as> ou <role-name>.

Um caso de uso do elemento <run-as> é prevenir os clientes externos de acessarem EJBs internos. Você pode configurar este comportamento determinando os elementos <method-permission> do EJB interno, que restringe o acesso à uma função nunca determinada a um cliente externo. Os EJBs, que devem usar em troca EJBs internos, são então configurados com um <run-as> ou <role-name> igual à função restringida. O seguinte fragmento do descritor descreve uma amostra do uso do elemento <security-identity>.

```
<!-- A sample ejb-jar.xml fragment -->
<ejb-jar>
    <enterprise-beans>
        <session>
            <ejb-name>ASessionBean</ejb-name>
            <!-- ... -->
            <security-identity>
                <use-caller-identity/>
            </security-identity>
        </session>
        <session>
            <ejb-name>RunAsBean</ejb-name>
            <!-- ... -->
            <security-identity>
                <run-as>
                    <description>A private internal role</description>
                    <role-name>InternalRole</role-name>
                </run-as>
            </security-identity>
        </session>
```

```
    </enterprise-beans>
    <!-- ... -->
</ejb-jar>
```

Quando você usa um <run-as> para determinar uma função específica para chamadas de saída, um principal nomeado **anonymous** é determinado a todas as chamadas de saída. Caso você deseje que outro principal seja associado com a chamada, você deverá associar um <run-as-principal> com o bean no arquivo **jboss.xml**. O seguinte fragmento associa um principal nomeado **internal** com o **RunAsBean** com a amostra anterior.

```
<session>
    <ejb-name>RunAsBean</ejb-name>
    <security-identity>
        <run-as-principal>internal</run-as-principal>
    </security-identity>
</session>
```

O elemento <run-as> encontra-se também disponível nas definições do servlet num arquivo **web.xml**. A amostra seguinte apresenta como determinar o **InternalRole** de função a um servlet:

```
<servlet>
    <servlet-name>AServlet</servlet-name>
    <!-- ... -->
    <run-as>
        <role-name>InternalRole</role-name>
    </run-as>
</servlet>
```

As chamadas a partir deste servlet estão associadas com o **principal** anônimo. O elemento está disponível no arquivo **jboss-web.xml** para determinar um principal específico juntamente com a função **run-as**. O seguinte fragmento apresenta como associar um principal nomeado **internal** ao servlet acima.

```
<servlet>
    <servlet-name>AServlet</servlet-name>
    <run-as-principal>internal</run-as-principal>
</servlet>
```

1.3. FUNÇÕES DE SEGURANÇA

O nome de função de segurança referenciado tanto pelo elemento <security-role-ref> ou <security-identity> deve mapear uma das funções declaradas do aplicativo. Um assembler de aplicativo define as funções de segurança lógica pela declaração dos elementos <security-role>. O valor do atributo role-name é um nome de função do aplicativo lógico, tal como Administrator, Architect ou Sales_Manager.

As especificações do Java EE descrevem que é importante estar ciente que as funções de segurança no descritor da implantação são usadas para definir a visualização da segurança lógica de um aplicativo. As funções definidas nos descritores de implantação do Java EE não devem ser confundidas com os grupos de usuários, principais e outros conceitos que existem no ambiente operacional da empresa alvo. As funções do descritor de implantação são construções de aplicativos com os nomes específicos de domínio. Por exemplo, o aplicativo do banco pode usar nomes de função tais como Bank_Manager, Teller ou Customer.

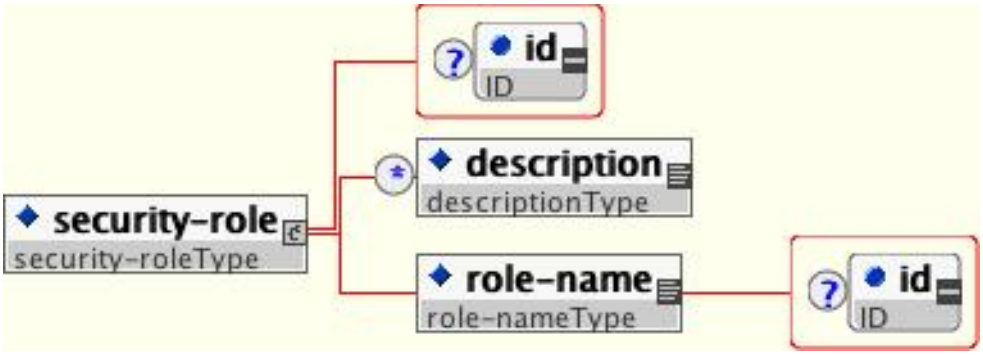


Figura 1.3. elemento <security-role>

No JBoss, o elemento <security-role> é apenas usado para mapear os valores <security-role-ref> ou <role-name> à função lógica que a função do componente referencia. As funções determinadas do usuário são uma função dinâmica do gerenciador de segurança do aplicativo, conforme será visto quando discutirmos em maiores detalhes a implantação do JBossSX.

O JBoss não requer a definição dos elementos <security-role> com o objetivo de declarar as permissões do método. No entanto, a especificação dos elementos <security-role> continua a ser uma prática recomendada para garantia de portabilidade através dos servidores do aplicativo e para a manutenção do descritor de implantação. O Exemplo 1.3, “fragmento do descritor ejb-jar.xm” descreve o uso do <security-role> num arquivo **`ejb-jar.xml`**.

Exemplo 1.3. fragmento do descritor ejb-jar.xm

```
<!-- A sample ejb-jar.xml fragment -->
<ejb-jar>
    <!-- ... -->
    <assembly-descriptor>
        <security-role>
            <description>The single application role</description>
            <role-name>TheApplicationRole</role-name>
        </security-role>
    </assembly-descriptor>
</ejb-jar>
```

O Exemplo 1.4, “amostra do fragmento do descritor web.xml” apresenta o uso do <security-role> num arquivo **`web.xml`**.

Exemplo 1.4. amostra do fragmento do descritor web.xml

```
<!-- A sample web.xml fragment -->
<web-app>
    <!-- ... -->
    <security-role>
        <description>The single application role</description>
        <role-name>TheApplicationRole</role-name>
    </security-role>
</web-app>
```

1.4. PERMISSÕES DO MÉTODO EJB

Um assembler do aplicativo pode usar a declaração do elemento <method-permission> para determinar as funções que permitem invocar a página principal do EJB e métodos de interface remota.

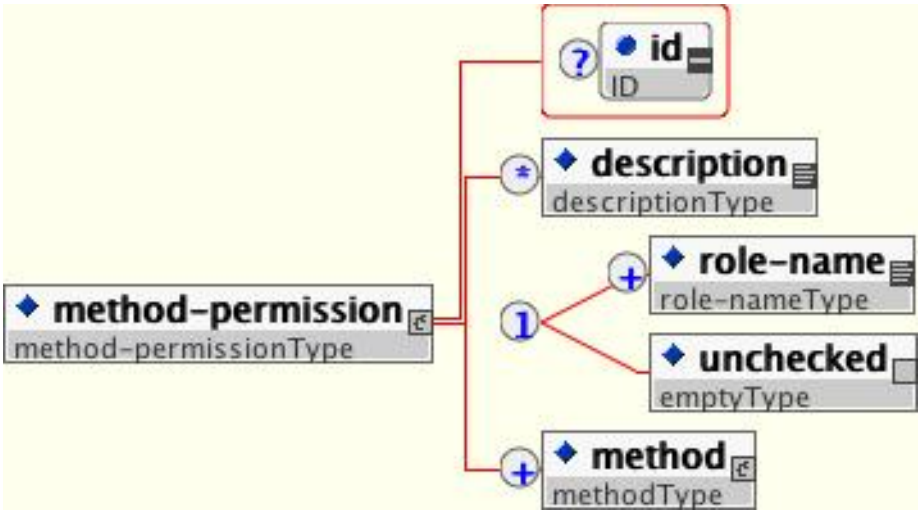


Figura 1.4. O elemento <method-permission>

Cada elemento <method-permission> contém um ou mais elementos <role-name>. O <role-name> define funções lógicas que são permitidas para acesso aos métodos EJB como identidade pelos elementos filho <method>. Você pode especificar um elemento <unchecked> ao invés do elemento <role-name> para declarar que qualquer usuário autenticado pode acessar os métodos identificados pelos elementos filho do método. Além disso, você pode declarar que ninguém deve possuir acesso a um método que possui o elemento exclude-list. Caso um EJB possua métodos que não foram declarados como acessíveis pela função usando um elemento <method-permission>, o padrão dos métodos EJB é a exclusão dos mesmos para uso. Isso é equivalente à padronização dos métodos no **`exclude-list`**.

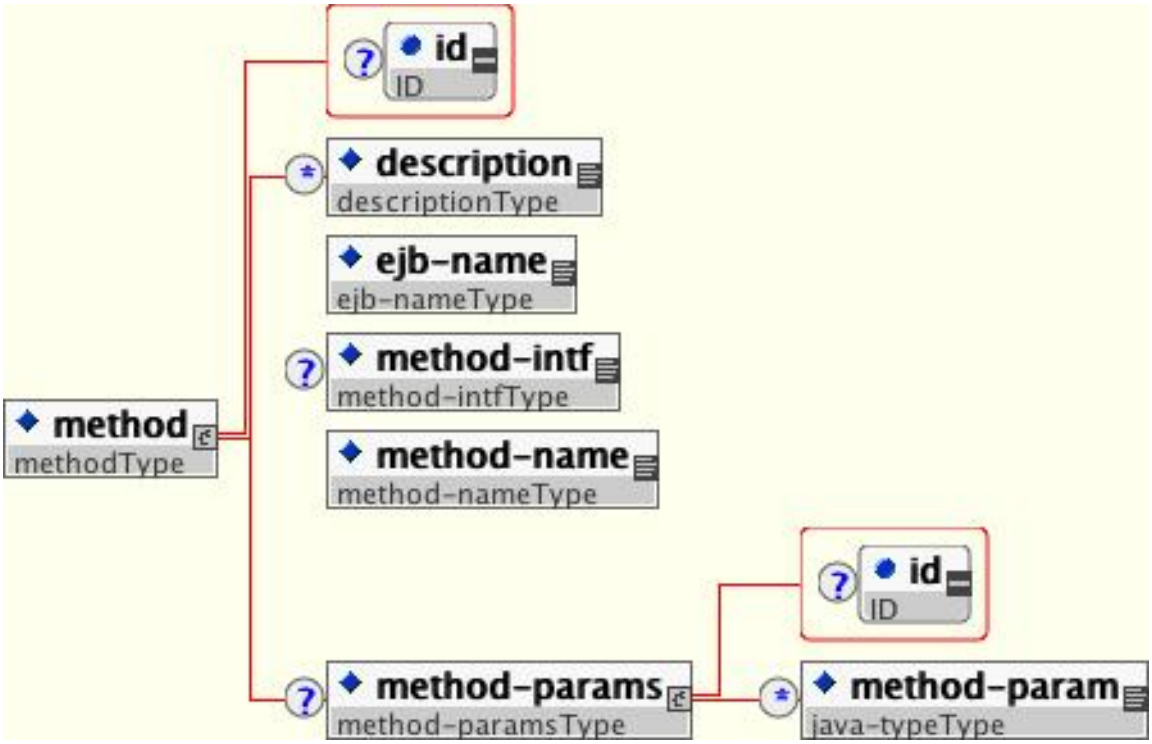


Figura 1.5. elemento <method>

Existem três estilos suportados das declarações do elemento do método.

O primeiro é usado para referenciação de todos os métodos da interface do componente e página principal do bean enterprise nomeado:

```
<method>
    <ejb-name>EJBNAME</ejb-name>
    <method-name>*</method-name>
</method>
```

O segundo estilo é usado para referenciação de um método específico da página principal ou interface do componente do bean enterprise nomeado:

```
<method>
    <ejb-name>EJBNAME</ejb-name>
    <method-name>METHOD</method-name>
                </method>
```

Caso existam métodos múltiplos com o mesmo nome sobrecarregado, este estilo refere-se a todos os métodos sobrecarregados.

O terceiro estilo é usado para referir-se a um método específico com um conjunto de métodos com um nome sobrecarregado:

```
<method>
    <ejb-name>EJBNAME</ejb-name>
    <method-name>METHOD</method-name>
    <method-params>
        <method-param>PARAMETER_1</method-param>
        <!-- ... -->
        <method-param>PARAMETER_N</method-param>
    </method-params>
</method>
```

O método deve ser definido na página principal ou interface remota do bean enterprise especificado. Os valores do elemento <method-param> são um nome inteiramente qualificado do tipo de parâmetro correspondente. Caso existam métodos múltiplos com a mesma assinatura sobrecarregada, a permissão é aplicada a todos os métodos sobrecarregados de combinação.

O elemento <method-intf> pode ser usado para diferenciar métodos com o mesmo nome e assinatura definidos em ambas interfaces remotas e página principal de um bean enterprise.

O Exemplo 1.5, “uso do elemento <method-permission>” fornece amostras completas do uso do elemento <method-permission>.

Exemplo 1.5. uso do elemento <method-permission>

```
<ejb-jar>
    <assembly-descriptor>
        <method-permission>
            <description>The employee and temp-employee roles may
access any
                method of the EmployeeService bean </description>
            <role-name>employee</role-name>
            <role-name>temp-employee</role-name>
            <method>
                <ejb-name>EmployeeService</ejb-name>
```

```
                <method-name>*</method-name>
            </method>
        </method-permission>
        <method-permission>
            <description>The employee role may access the
findByPrimaryKey,
                getEmployeeInfo, and the updateEmployeeInfo(String)
method of
                the AardvarkPayroll bean </description>
            <role-name>employee</role-name>
            <method>
                <ejb-name>AardvarkPayroll</ejb-name>
                <method-name>findByPrimaryKey</method-name>
            </method>
            <method>
                <ejb-name>AardvarkPayroll</ejb-name>
                <method-name>getEmployeeInfo</method-name>
            </method>
            <method>
                <ejb-name>AardvarkPayroll</ejb-name>
                <method-name>updateEmployeeInfo</method-name>
                <method-params>
                    <method-param>java.lang.String</method-param>
                </method-params>
            </method>
        </method-permission>
        <method-permission>
            <description>The admin role may access any method of the
                EmployeeServiceAdmin bean </description>
            <role-name>admin</role-name>
            <method>
                <ejb-name>EmployeeServiceAdmin</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>
        <method-permission>
            <description>Any authenticated user may access any method
of the
                EmployeeServiceHelp bean</description>
            <unchecked/>
            <method>
                <ejb-name>EmployeeServiceHelp</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>
        <exclude-list>
            <description>No fireTheCTO methods of the EmployeeFiring
bean may be
                used in this deployment</description>
            <method>
                <ejb-name>EmployeeFiring</ejb-name>
                <method-name>fireTheCTO</method-name>
            </method>
        </exclude-list>
    </assembly-descriptor>
</ejb-jar>
```

1.5. ANOTAÇÕES DE SEGURANÇA DO ENTERPRISE BEAN

Os beans Enterprise usam Anotações para passar informações ao implantador sobre a segurança e outros aspectos do aplicativo. O implantador pode montar a própria política de segurança do bean enterprise, caso especificado nas anotações ou no descritor de implantação.

Quaisquer valores de método especificados no descritor da implantação substituem o valor da anotação. Caso o valor do método não seja especificado no descritor da implantação, esses valores determinados usando as anotações serão usados. A substituição da granulação é baseado no método.

As anotações que endereçam a segurança e podem ser usadas nos beans enterprise incluem o seguinte:

@DeclareRoles

Declara cada função de segurança no código. Por favor, refira-se ao *Java EE 5 Tutorial* Declaring Security Roles Using Annotations para maiores informações sobre a configuração das funções.

@RolesAllowed, @PermitAll, and @DenyAll

Especifica as permissões do método para anotações. Refira-se ao *Java EE 5 Tutorial* Specifying Method Permissions Using Annotations para maiores informações sobre a configuração das permissões do método de anotação.

@RunAs

Configura a identidade de segurança propagada de um componente. Por favor, refira-se ao *Java EE 5 Tutorial* Configuring a Component's Propagated Security Identity para maiores informações sobre a configuração das identidades usando a segurança propagada.

1.6. RESTRIÇÕES DE SEGURANÇA DO CONTEÚDO DA WEB

A segurança é definida pelas funções que permitem o acesso ao conteúdo por um padrão de URL que identifica o conteúdo protegido num aplicativo da web. Este conjunto de informação é declarado pelo uso do elemento `web.xml` security-constraint.

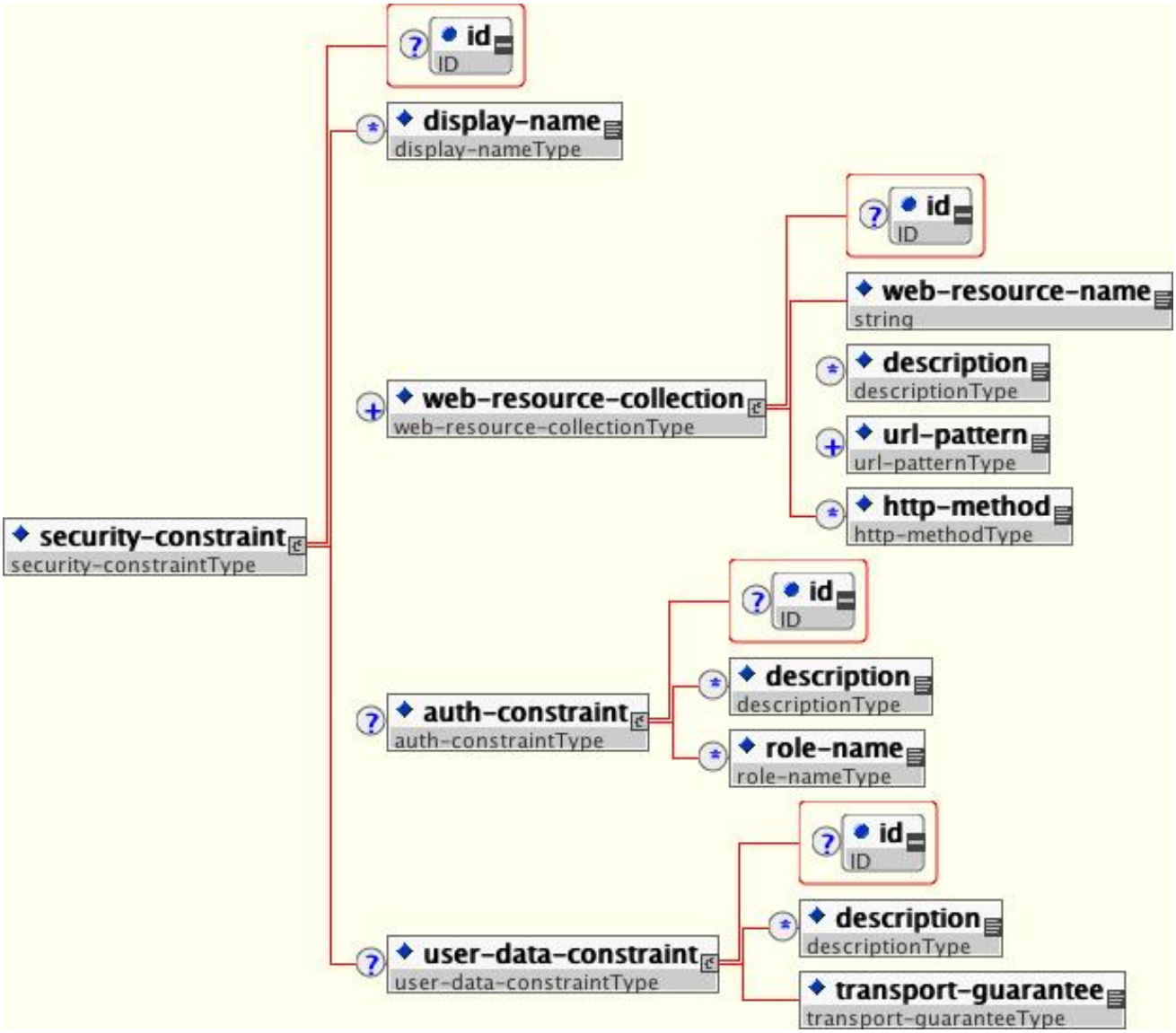


Figura 1.6. elemento <security-constraint>

O conteúdo a ser protegido é declarado usando um ou mais elementos <web-resource-collection>. Cada elemento <web-resource-collection> contém uma série opcional de elementos <url-pattern> seguidos por uma série de elementos <http-method>. O valor do elemento <url-pattern> especifica um URL padrão relacionado que solicita a combinação de um URL para que a solicitação corresponda a uma tentativa de acesso ao conteúdo protegido. O valor do elemento <http-method> especifica um tipo de solicitação HTTP para permissão.

O elemento <user-data-constraint> opcional especifica as solicitações para a camada de transporte do cliente à conexão do servidor. A solicitação pode ser para a integridade do conteúdo (prevenindo as violações de dados no processo de comunicação) ou para confidencialidade (prevenindo a leitura enquanto em trânsito). O valor do elemento <transport-guarantee> especifica o grau pelo qual a comunicação entre o cliente e o servidor deve ser protegida. Os valores dos mesmos são **`NONE`**, **`INTEGRAL`** e **`CONFIDENTIAL`**. O valor **`NONE`** significa que o aplicativo não requer quaisquer garantias de transporte. O valor **`INTEGRAL`** significa que o aplicativo requer que os dados sejam enviados entre o cliente e o servidor a ser enviado de tal maneira, sendo que eles não possam ser alterados em trânsito. O valor **`CONFIDENTIAL`** significa que o aplicativo requer que os dados sejam transmitidos de maneira que previnam outras entidades de observar os conteúdos de transmissão. Na maioria dos casos, a presença do aviso **`INTEGRAL`** ou **`CONFIDENTIAL`** indica que o uso do SSL é requerido.

O elemento <login-config> opcional é usado para configurar o método de autenticação que deve ser usado, o nome realm que deve ser usado pelo aplicativo e os atributos que são necessários pelo mecanismo de logon do formulário.

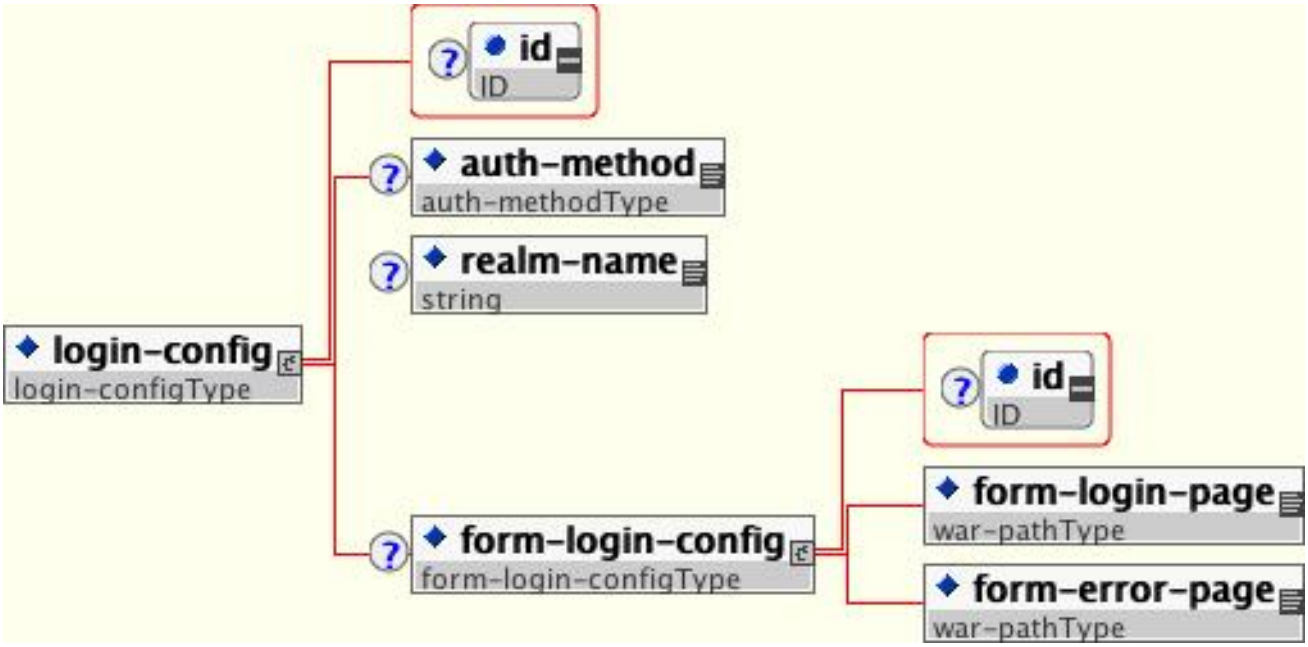


Figura 1.7. elemento <login-config>

O elemento <auth-method> filho especifica o mecanismo de autenticação pelo aplicativo da web. O usuário deve realizar a autenticação usando o mecanismo configurado, como pré-requisito para ganho de acesso a quaisquer recursos da web que são protegidos por uma restrição de autorização. Os valores <auth-method> legais são **`BASIC`**, **`DIGEST`**, **`FORM`** e **`CLIENT-CERT`**. O elemento filho <realm-name> especifica o nome realm para uso no HTTP básico e autorização de resumo. O elemento filho <form-login-config> especifica o logon assim como as páginas de erro que devem ser usadas no logon baseado no formulário. O **`form-login-config`** e seus elementos filhos são ignorados, caso o valor <auth-method> não seja **`FORM`**.

O Exemplo 1.6, “fragmento do descritor web.xml” indica que qualquer URL sob o caminho **`/restricted`** do aplicativo da web requer uma função **`AuthorizedUser`**. Não existe garantia de transporte solicitada e método de autenticação usado para obtenção da identidade do usuário de autenticação HTTP BÁSICA.

Exemplo 1.6. fragmento do descritor web.xml

```
<web-app>
    <!-- ... -->
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Secure Content</web-resource-name>
            <url-pattern>/restricted/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>AuthorizedUser</role-name>
        </auth-constraint>
        <user-data-constraint>
            <transport-guarantee>NONE</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
    <!-- ... -->
```

```
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>The Restricted Zone</realm-name>
    </login-config>
    <!-- ... -->
    <security-role>
        <description>The role required to access restricted content
</description>
        <role-name>AuthorizedUser</role-name>
    </security-role>
</web-app>
```

1.7. ATIVAÇÃO DA AUTENTICAÇÃO BASEADA NO FORMULÁRIO

A autenticação baseada no formulário fornece flexibilidade na definição de uma página JSP/HTML personalizada para logon e uma página separada pela qual os usuários são direcionados, caso um erro ocorra durante o logon.

A autenticação baseada no formulário é definida pela inclusão do **`<auth-method>FORM</auth-method>`** no elemento <login-config> do descritor da implantação, **`web.xml`**. O logon e as páginas de erro estão também definidas no <login-config>, conforme abaixo:

```
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/login.html</form-login-page>
    <form-error-page>/error.html</form-error-page>
  </form-login-config>
</login-config>
```

Quando um aplicativo da web com autenticação baseada no formulário é implantada, o recipiente da web usa o **`FormAuthenticator`** para direcionar usuários à página apropriada. A Plataforma do Aplicativo JBoss Enterprise mantém o pool de sessão de forma que a informação de autenticação não precisa estar presente a cada solicitação. Quando o **`FormAuthenticator`** receber uma solicitação, ele consulta o **`org.apache.catalina.session.Manager`** por uma sessão existente. Caso a sessão não existir, a nova sessão é criada. Em seguida, o **`FormAuthenticator`** verifica os credenciais da sessão.

NOTA

Cada sessão é identificada por uma ID de sessão, sendo esta uma sequência de 16 bytes gerada de valores aleatórios. Esses valores são restaurados a partir do **`/dev/urandom`** (Linux) por padrão e aplicados hash com o MD5. As checagens são executadas na criação da ID da sessão como garantia de que a ID criada é única.

Uma vez verificada, a ID da sessão é determinada como parte de uma cookie e então retornada ao cliente. Esta cookie é esperada em solicitações de clientes subsequentes e é usada para identificar a sessão do usuário.

A cookie passada ao cliente é o par do valor do nome com diversos atributos opcionais. O atributo do identificador é chamado **`JSESSIONID`**. O seu valor é uma sequência hexa da ID da sessão. Esta cookie é configurada para ser não-persistente. Isto significa que ela será deletada ao lado do cliente quando o

navegador for encerrado. No lado do servidor, as sessões expiram após 60 segundos de inatividade, sendo que os objetos de sessão do período e suas informações de credencial são deletados.

Vamos dizer que um usuário tenta acessar o aplicativo da web que está protegido com uma autenticação baseada no formulário. O **`FormAuthenticator`** aplica o cache na solicitação, cria uma nova sessão se for necessário, e redireciona o usuário à página de logon definida no **`login-config`**. (No código da amostra anterior, a página de logon é **`login.html`**.) Em seguida, o usuário insere seu nome e senha de usuário no formulário HTML fornecido. O nome e senha do usuário são passados ao **`FormAuthenticator`** através da ação do formulário **`j_security_check`**.

O **`FormAuthenticator`** autentica o nome e senha do usuário em relação ao realm anexado ao contexto do aplicativo da web. O realm é o **`JBossWebRealm`**, na Plataforma do Aplicativo JBoss Enterprise. O **`FormAuthenticator`** restaura as solicitações salvas do cache e redireciona o usuário à solicitação original, quando a autenticação for bem sucedida.

NOTA

O servidor reconhece as solicitações de autenticação do formulário apenas quando o URI encerrar por **`/j_security_check`** e que pelo menos os parâmetros **`j_username`** e **`j_password`** existam.

1.8. ATIVAÇÃO DA SEGURANÇA DECLARATIVA

Os elementos de segurança do Java EE mencionados até agora, descrevem as solicitações de segurança apenas da perspectiva do aplicativo. O implantador do aplicativo mapeia as funções do domínio do aplicativo num ambiente de implantação, uma vez que os elementos de segurança do Java EE declaram funções lógicas. As especificações do Java EE omitem esses detalhes específicos do servidor do aplicativo.

Você deverá especificar um gerenciador de segurança que implanta o modelo de segurança do Java EE usando os descritores da implantação específica do servidor do JBoss, com o objetivo de mapear as funções do aplicativo no ambiente de implantação. Maiores informações sobre a configuração de segurança podem ser encontradas no Exemplo 12.11, “Módulo de Logon Personalizado do JndiUserAndPass”.

CAPÍTULO 2. INTRODUÇÃO AO JAAS

O framework JBossSX é baseado no JAAS API. Você deve entender os elementos básicos do JAAS API antes de você poder entender os detalhes de implantação do JBossSX. As seguintes seções fornecem uma introdução ao JAAS para prepará-lo à discussão sobre a arquitetura JBossSX descrita adiante neste guia.

O JAAS 1.0 API consiste de um conjunto de pacotes Java designados para autenticação e autorização do usuário. O API implementa a versão Java do framework de Módulos de Autenticação Pugláveis - Pluggable Authentication Modules (PAM) e extende a arquitetura de controle de acesso à Plataforma Java 2 para suportar autorização baseada no usuário.

O JAAS foi liberado primeiramente como um pacote de extensão para o JDK 1.3 e vinculado com o JDK 1.5. Esta introdução baseia-se na autenticação de capacidades do JAAS para implantar o modelo de segurança J2EE baseado na função declarativa, uma vez que o framework do JBossSX apenas utiliza este procedimento.

A autenticação do JAAS é executada de modo puglável. Isto permite que os aplicativos Java mantenham-se independentes das tecnologias de autenticação e permite que o gerenciador de segurança JBossSX trabalhe em diferentes infra-estruturas de segurança. A integração com a infra-estrutura de segurança é alcançada sem a alteração da implantação do gerenciador de segurança JBossSX. Você precisa apenas alterar a configuração da autenticação que a pilha JAAS usa.

2.1. CLASSES PRINCIPAIS DO JAAS

As classes principais JAAS podem ser divididas em três categorias: comum, autenticação e autorização. A seguinte lista apresenta as classes de autenticação e comum uma vez que as classes específicas costumavam a implantar a funcionalidade do JBossSX descrita neste capítulo.

Segue abaixo as classes comuns:

- **`Subject`** (**`javax.security.auth.Subject`**)
- **`Principal`** (**`java.security.Principal`**)

Elas são as classes de autenticação:

- **`Callback`** (**`javax.security.auth.callback.Callback`**)
- **`CallbackHandler`** (**`javax.security.auth.callback.CallbackHandler`**)\n\t\n
- **`Configuration`** (**`javax.security.auth.login.Configuration`**)
- **`LoginContext`** (**`javax.security.auth.login.LoginContext`**)
- **`LoginModule`** (**`javax.security.auth.spi.LoginModule`**)

2.1.1. Classes do Assunto e Principal

Os aplicativos devem ser primeiramente autenticados na fonte do solicitante com o objetivo de autorizar o acesso aos recursos. O framework JAAS define o termo assunto para representar uma fonte do solicitante. A classe **`Subject`** é a classe central no JAAS. O **`Subject`** representa a informação para a entidade única, tal como uma pessoa ou serviço. Ele abrange os principais da entidade, credenciais públicos e credenciais privados. Os JAAS APIs usam a interface **`java.security.Principal`** do Java 2 existente para representar um principal, que é basicamente um nome digitado.

Um assunto é populado com as identidades ou principais durante o processo de autenticação. Um assunto pode possuir diversos principais. Por exemplo, uma pessoa pode possuir o principal de nome (John Doe), um principal de número de segurança social (123-45-6789) e um principal de nome de usuário (johnd), sendo que todos ajudam a distinguir o assunto de outros assuntos. Segue abaixo dois métodos disponíveis para restauração dos principais associados com um assunto.

```
public Set getPrincipals() {...}
public Set getPrincipals(Class c) {...}
```

O **`getPrincipals()`** retorna todos os principais contidos no assunto. O **`getPrincipals(Class c)`** retorna apenas os principais que são instâncias de classe **`c`** ou uma se suas sub-classes. Um conjunto vazio é retornado caso o assunto não possua principais de combinação.

Perceba que a interface **`java.security.acl.Group`** é uma sub-interface do **`java.security.Principal`**, portanto uma instância no conjunto dos principais pode representar o agrupamento lógico de outros principais ou grupos de principais.

2.1.2. Autenticação do Assunto

A Autenticação do Assunto requer um logon JAAS. O procedimento de logon consiste das seguintes etapas:

1. O aplicativo instancia um **`LoginContext`** e passa o nome da configuração de logon, além de um **`CallbackHandler`** para popular os objetos **`Callback`**, conforme solicitado pelos **`LoginModule`**s da configuração.

2. O **`LoginContext`** consulta um **`Configuration`** para carregar todos os **`LoginModules`** incluídos na configuração de logon nomeada. Caso tal configuração nomeada não existir, a configuração **`other`** será usada como padrão.

3. O aplicativo invoca o método **`LoginContext.login`**.

4. O método de logon invoca todos os **`LoginModule`**s carregados. Uma vez que cada **`LoginModule`** tenta autenticar o assunto, ele invoca o método de manuseio no **`CallbackHandler`** associado para obter informações solicitadas para o processo de autenticação. A informação requerida é passada para manuseio do método na forma de um array de objetos **`Callback`**. Caso bem sucedido, os **`LoginModule`**s associam principais e credenciais relevantes com o assunto.

5. O **`LoginContext`** retorna o status de autenticação para o aplicativo. O sucesso da operação é representado pelo retorno do método de logon. A falha é representada através de um LoginException sendo lançado através do método de logon.

6. Caso a autenticação seja bem sucedida, o aplicativo restaura o assunto autenticado usando o método **`LoginContext.getSubject`**.

7. Após o escopo da autenticação do assunto ser completado, todos os principais e informações relacionadas associadas com o assunto pelo método de logon podem ser removidos pela invocação do método **`LoginContext.logout`**.

A classe **`LoginContext`** fornece os métodos básicos de autenticação dos assuntos e oferece uma maneira de desenvolver um aplicativo que é independente da tecnologia de autenticação subjacente. O **`LoginContext`** consulta um **`Configuration`** para determinar os serviços de autenticação configurados para um aplicativo em particular. As classes **`LoginModule`** representam os serviços de

autenticação. Portanto, você pode conectar-se a diferentes módulos de logon em um aplicativo sem alterar o próprio aplicativo. O seguinte código apresenta as etapas solicitadas por um aplicativo para autenticar o assunto.

```
CallbackHandler handler = new MyHandler();
LoginContext lc = new LoginContext("some-config", handler);

try {
    lc.login();
    Subject subject = lc.getSubject();
} catch(LoginException e) {
    System.out.println("authentication failed");
    e.printStackTrace();
}

// Perform work as authenticated Subject
// ...

// Scope of work complete, logout to remove authentication info
try {
    lc.logout();
} catch(LoginException e) {
    System.out.println("logout failed");
    e.printStackTrace();
}

// A sample MyHandler class
class MyHandler
    implements CallbackHandler
{
    public void handle(Callback[] callbacks) throws
        IOException, UnsupportedCallbackException
    {
        for (int i = 0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof NameCallback) {
                NameCallback nc = (NameCallback)callbacks[i];
                nc.setName(username);
            } else if (callbacks[i] instanceof PasswordCallback) {
                PasswordCallback pc = (PasswordCallback)callbacks[i];
                pc.setPassword(password);
            } else {
                throw new UnsupportedCallbackException(callbacks[i],
                                                       "Unrecognized
Callback");
            }
        }
    }
}
```

Os desenvolvedores integram uma tecnologia de autenticação pela criação de uma implantação da interface **`LoginModule`**. Isto permite que um administrador conecte-se a diferentes tecnologias de autenticação em um aplicativo. Você pode formar uma cadeia de múltiplos **`LoginModule`**s para permitir que mais de uma tecnologia de autenticação participe no processo de autenticação. Por exemplo, um **`LoginModule`** pode executar a autenticação baseada no nome/senha do usuário, enquanto outro pode ser a interface para dispositivos de hardware tais como leituras de cartões e autenticadores biométricos.

O ciclo de vida de um **LoginModule** é dirigido pelo objeto **LoginContext** pelo qual o cliente crie e imprime o método de logon. O processo consiste em duas fases. Segue abaixo as etapas do processo:

- O **LoginContext** cria cada **LoginModule** configurado usando o construtor não-arg.
- Cada **LoginModule** é inicializado por uma chamada para inicializar cada método. Garante-se que o argumento **Subject** seja não-nulo. A assinatura do método inicializar é a seguinte: **public void initialize(Subject subject, CallbackHandler callbackHandler, Map sharedState, Map options)**.
- O método **login** é chamado para iniciar o processo de autenticação. Por exemplo, a implantação do método pode pedir a um usuário pelo nome e senha do usuário e verificar a informação sobre os dados armazenados no serviço de nomeação, tais como o NIS ou LDAP. Implantações alternativas podem servir como interface para cartões smart e dispositivos biométricos, ou simplesmente extrair a informação do usuário a partir do sistema de operação subjacente. A validação da identidade do usuário por cada **LoginModule** é considerada fase 1 da autenticação do JAAS. A assinatura do método de **login** é o **boolean login() throws LoginException**. O **LoginException** indica falha. O valor de retorno verdadeiro indica que o método é bem sucedido, onde o valor de retorno negativo indica que o módulo de logon deve ser ignorado.
- Caso a autenticação de visão geral do **LoginContext** seja bem sucedida, o **commit** é invocado em cada **LoginModule**. Caso a fase 1 suceda para um **LoginModule**, o método de confirmação continua na fase 2 e associa os principais relevantes, credenciais públicos e/ou credenciais privados com o assunto. Caso a fase 1 falhar para o **LoginModule**, o **commit** removerá qualquer estado de autenticação armazenado anteriormente, tal como nomes e senhas de usuário. A assinatura do método **commit** é a seguinte: **boolean commit() throws LoginException**. A falha em completar a fase de confirmação é indicada pelo lançamento de um **LoginException**. O retorno como verdadeiro indica que o método é bem sucedido, onde o retorno como falso indica que o módulo de logon deve ser ignorado.
- Caso a autenticação da visão geral do **LoginContext** falhar, o método **abort** será invocado em cada **LoginModule**. O método **abort** remove ou destrói qualquer estado de autenticação criado pelos métodos de logon ou inicialização. A assinatura do método **abort** é o **boolean abort() throws LoginException**. A falha em completar a fase **abort** é indicada pelo lançamento de um **LoginException**. O retorno como verdadeiro indica que o método foi bem sucedido, onde o retorno como falso indica que o módulo de logon deve ser ignorado.
- Para remover o estado de autenticação após o logon com êxito, o aplicativo invoca o **logout** no **LoginContext**. Este, por sua vez resulta numa invocação de método **logout** em cada **LoginModule**. O método **logout** remove os principais e credenciais originalmente associados com o assunto durante a operação **commit**. Os credenciais devem ser destruídos na remoção. A assinatura do método **logout** é a seguinte: **boolean logout() throws LoginException**. A falha em completar o processo de saída é indicada pelo lançamento de um **LoginException**. O retorno como verdadeiro indica que o método foi bem sucedido, onde o retorno como falso indica que o módulo de logon deve ser ignorado.

Quando o **LoginModule** necessitar comunicar-se com o usuário para obtenção de informação da autenticação, ele usará um objeto **CallbackHandler**. Os aplicativos implementam a interface CallbackHandler e passam-as ao **LoginContext**, que envia a informação de autenticação diretamente aos módulos de logon subjacentes.

Os módulos de logon usam ambos **CallbackHandler** para obterem entradas dos usuários, tais como uma senha ou PIN de cartão smart, além de fornecerem informação aos usuários tais como informação

de status. Quando permitindo que o aplicativo especifique o **`CallbackHandler`**, os **`LoginModule`**s subjacentes continuam independentes das diversas maneiras em que os aplicativos interagem com os usuários. Por exemplo, uma implantação do **`CallbackHandler`** para um aplicativo GUI pode exibir uma janela para solicitar a entrada do usuário. Por outro lado, uma implantação **`CallbackHandler`** para um ambiente sem GUI, tal como o servidor do aplicativo, pode simplesmente obter informação do credencial pelo uso de um API do servidor do aplicativo. A interface callbackhandler possui um método para implantação:

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

A interface **`Callback`** é a última classe de autenticação que iremos observar. Ela é uma interface de marcação pela qual diversas implantações padrões são fornecidas, incluindo o **`NameCallback`** e **`PasswordCallback`** usado numa amostra anterior. O **`LoginModule`** usa um **`Callback`** para obter informação solicitada pelo mecanismo de autenticação. Os **`LoginModule`**s passam um array dos **`Callback`**s diretamente ao método **`CallbackHandler.handle`** durante a fase de logon da autenticação. Caso um **`callbackhandler`** não entender como usar um objeto **`Callback`** passado a um método de manuseio, ele lança um **`UnsupportedCallbackException`** para abortar a chamada de logon.

CAPÍTULO 3. MODELO DE SEGURANÇA DO JBOSS

Assim como o resto da arquitetura do JBoss, a segurança no nível mais inferior é definida como um conjunto de interfaces pelas quais implementações alternadas podem ser fornecidas.

- **`org.jboss.security.AuthenticationManager`**
- **`org.jboss.security.RealmMapping`**
- **`org.jboss.security.SecurityProxy`**
- **`org.jboss.security.AuthorizationManager`**
- **`org.jboss.security.AuditManager`**
- **`org.jboss.security.MappingManager`**

A Figura 3.1, “Relações da Interface do Modelo de Segurança para os Elementos do Recipiente EJB do Servidor JBoss. ” apresenta o diagrama de classe das interfaces de segurança e sua relação à arquitetura do recipiente EJB.

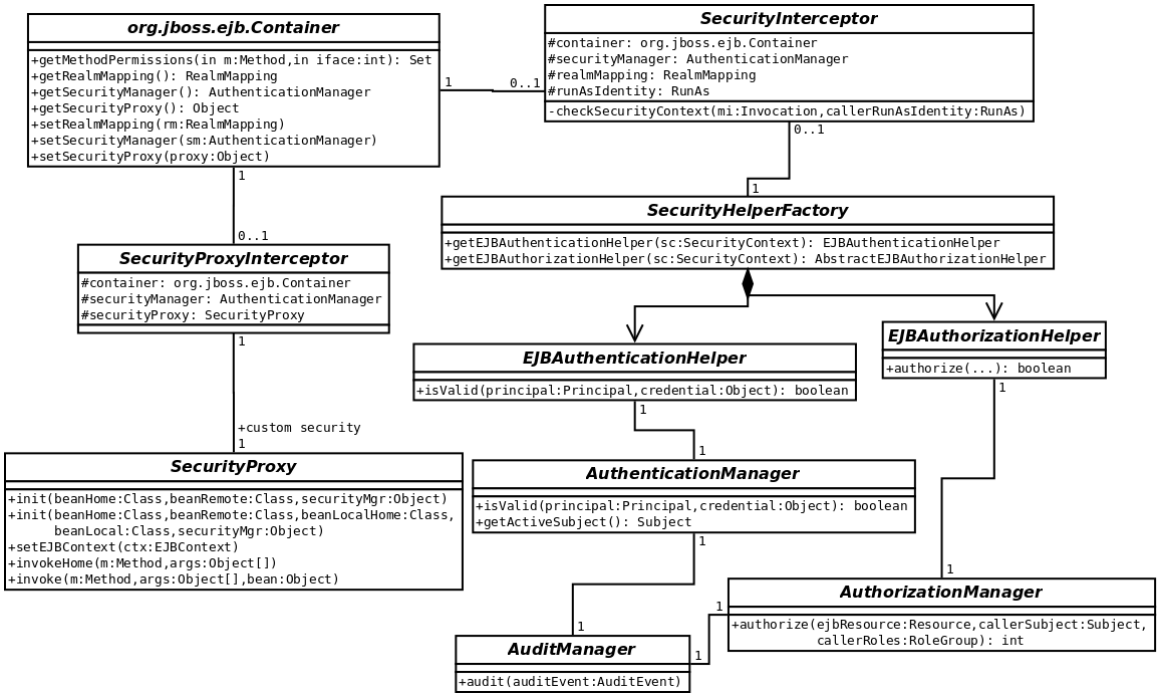


Figura 3.1. Relações da Interface do Modelo de Segurança para os Elementos do Recipiente EJB do Servidor JBoss.

A camada do Recipiente EJB é representada pelas classes - **`org.jboss.ejb.Container`**, **`org.jboss.SecurityInterceptor`** e **`org.jboss.SecurityProxyInterceptor`**. As demais classes são interfaces e classes fornecidas pelo subsistema de segurança.

As duas interfaces solicitadas para a implementação do modelo de segurança J2EE são:

- **`org.jboss.security.AuthenticationManager`**
- **`org.jboss.security.AuthorizationManager`**

As funções das interface de segurança apresentadas na Figura 3.1, “Relações da Interface do Modelo de Segurança para os Elementos do Recipiente EJB do Servidor JBoss. ” encontram-se resumidas abaixo.

Funções da Interface de Segurança

AuthenticationManager

Esta interface é responsável pela validação de credenciais associados com os *Principals*. Os Principais são identidades, tais como nomes de usuários, números de funcionário e números de segurança social. Os *Credentials* são prova de identidade, tais como senhas, teclas de sessão e assinaturas digitais. O método **`isValid`** é invocado para determinar se é que uma identidade de usuário e as credenciais associadas conhecidas conforme conhecidas no ambiente operacional são prova válida da identidade do usuário.

AuthorizationManager

Esta interface é responsável pelo controle de acesso mandatório pelas especificações do Java EE. A implementação desta interface fornece a habilidade de empilhar um conjunto de Fornecedores de Política úteis para a autorização conectável.

SecurityProxy

Esta interface descreve as solicitações para um plugin **`SecurityProxyInterceptor`** personalizado. O **`SecurityProxy`** permite que a externalização de segurança personalizada realize checagem baseada em métodos para ambos os métodos de página principal EJB e interface remota.

AuditManager

Esta interface é responsável em fornecer um atalho de auditoria para os eventos de segurança.

MappingManager

Esta interface é responsável em fornecer o mapeamento do Principal, Função e Atributos. A implementação do AuthorizationManager pode chamar internamente o gerenciador do mapeamento para mapear as funções antes de executar o controle de acesso.

SecurityDomain

Esta é uma extensão das interfaces **`AuthenticationManager`**, RealmMapping e **`SubjectSecurityManager`**. O **`SecurityDomain`** é a maneira recomendada para implementar segurança nos componentes, devido às vantagens oferecidas pelo JAAS Subject e o aumento de suporte oferecido ao aplicativo de estilo ASP e implementações de recurso. O **`java.security.KeyStore`** e as interfaces **`com.sun.net.ssl.KeyManagerFactory`** e **`com.sun.net.ssl.TrustManagerFactory`** estão incluídos nesta classe.

RealmMapping

Esta interface é responsável pelo mapeamento principal e função. O método **`getPrincipal`** leva uma identidade de usuário conforme conhecida no ambiente operacional e retorna a identidade do domínio do aplicativo. O método **`doesUserHaveRole`** que valida aquela identidade do usuário no ambiente de operação, é determinado para indicar a função daquele domínio do aplicativo.

Perceba que as interfaces **`AuthenticationManager`**, **`RealmMapping`** e **`SecurityProxy`** não possuem associações às classes relacionadas ao JAAS. Embora o framework do JBossSX seja inteiramente dependente no JAAS, as interfaces de segurança básicas requeridas para implementação

do modelo de segurança do Java EE não o são. O framework do JBossSX é simplesmente uma implementação das interfaces do plug-in de segurança baseadas no JAAS.

O diagrama do componente na Figura 3.2, “Classes de Implementação do Framework do JBossSX e Camada do Recipiente EJB do Servidor do JBoss.” ilustra esta situação. O problema desta arquitetura de plug-in é que você está livre para substituir o JAAS baseado nas classes de implementação do JBossSX com sua própria implementação de gerenciador de segurança baseado no não-JAAS. Você poderá verificar como isto funciona quando observando o JBossSX MBeans disponível para a configuração do JBossSX na Figura 3.2, “Classes de Implementação do Framework do JBossSX e Camada do Recipiente EJB do Servidor do JBoss.”.

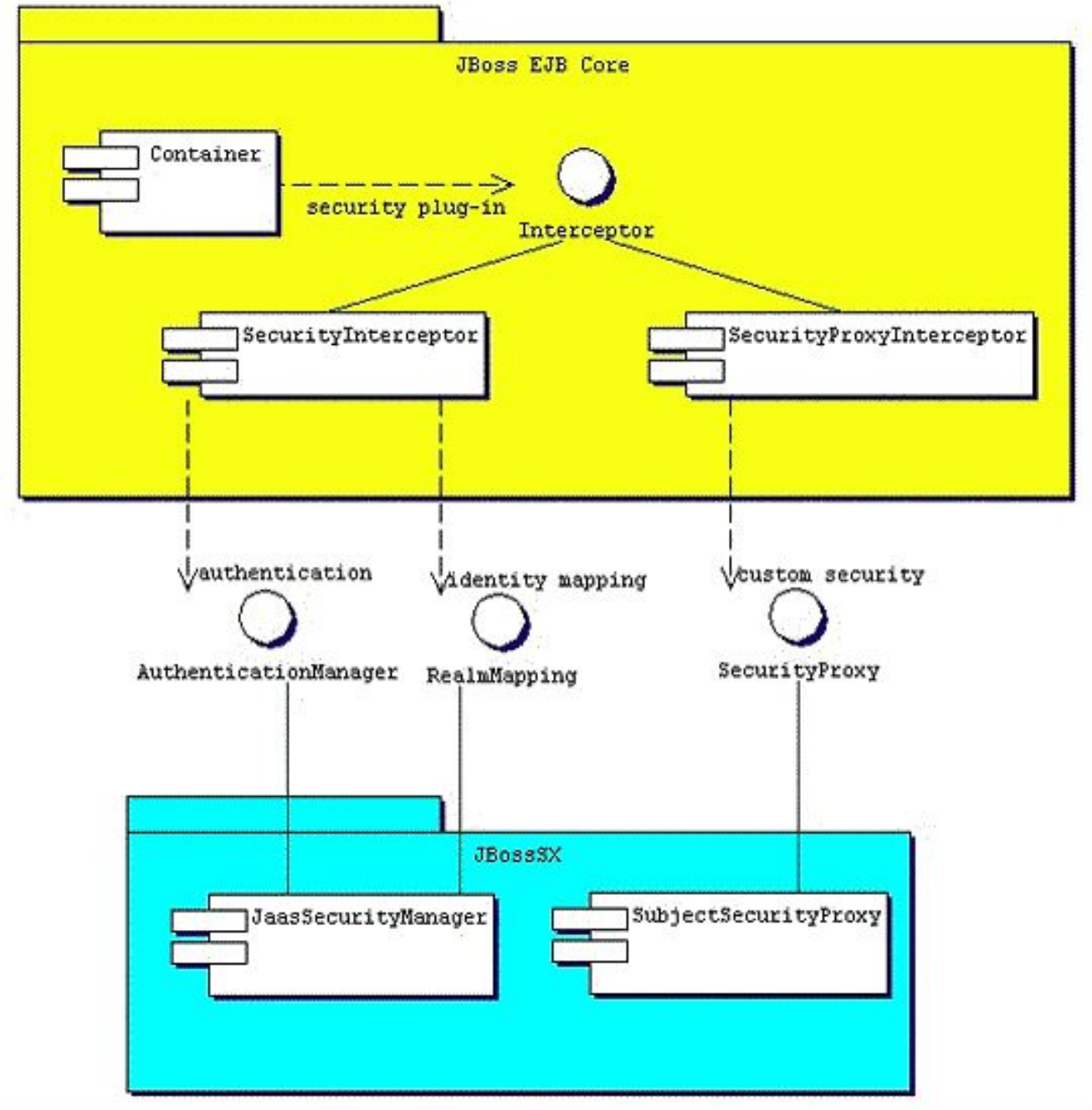


Figura 3.2. Classes de Implementação do Framework do JBossSX e Camada do Recipiente EJB do Servidor do JBoss.

3.1. ATIVAÇÃO DE SEGURANÇA DECLARATIVA REVISITADA

Anteriormente neste capítulo, a discussão do modelo de segurança padrão do Java EE terminava com

uma solicitação de uso do descritor de implementação específico do servidor para ativar a segurança. Os destalhes desta configuração são apresentados aqui. A Figura 3.3, “jboss.xml and jboss-web.xml Security Element Subsets.” apresenta o EJB específico do JBoss e os elementos relacionados à segurança do descritor de implantação do aplicativo da web.

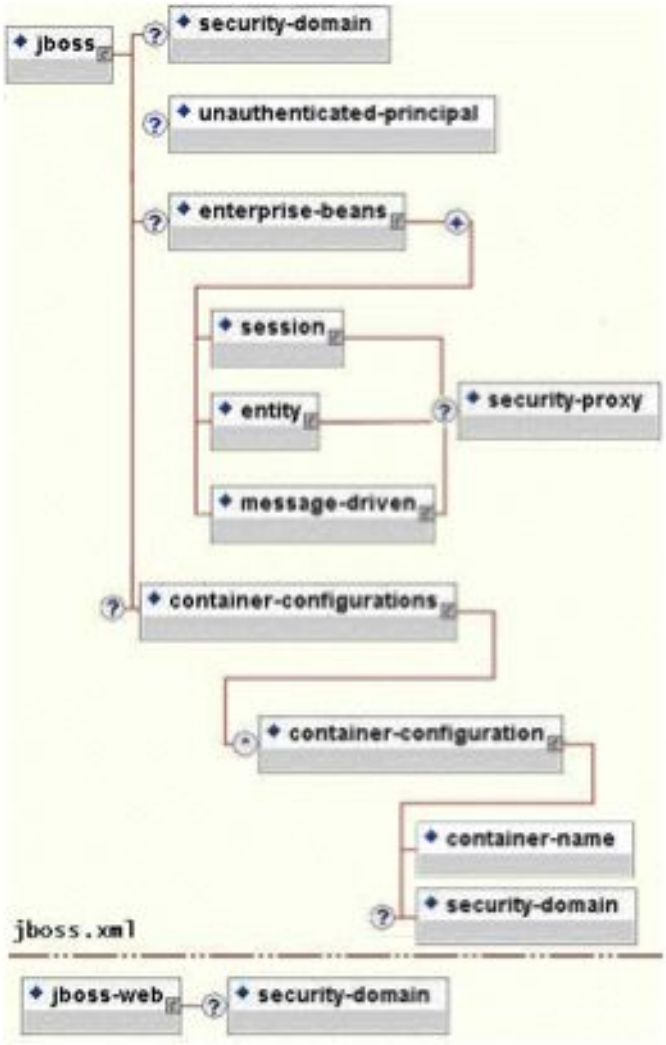


Figura 3.3. jboss.xml and jboss-web.xml Security Element Subsets.

O valor de um elemento <security-domain> especifica o nome JNDI da implementação da interface de segurança que o JBoss usa para o EJB e os recipientes da web. Este é um objeto que implementa ambas as interfaces **`AuthenticationManager`** e **`RealmMapping`**. Ele define qual domínio de segurança é especificado para todas as unidades de implantação, quando especificado como um elemento de nível superior. Este é o uso típico uma vez que a mistura dos gerenciadores de segurança com uma unidade de implantação complica a operação entre componentes e administração.

Você precisa especificar o <security-domain> à nível de configuração do recipiente, com o objetivo de especificar o domínio de segurança para um EJB individual. Isto substituirá qualquer elemento <security-domain> de nível superior.

O elemento <unauthenticated-principal> especifica o nome em uso para o objeto **`Principal`** retornado pelo método **`EJBContext.getUserPrincipal`** quando um usuário não-autenticado invoca um EJB. Perceba que isto não requer permissões especiais a um chamador não-autenticado. O seu propósito inicial é permitir servlets desprotegidos e páginas JSP para invocar EJBs desprotegidos e permitir que o EJB de destino obtenha um **`Principal`** não nulo para o chamador usando o método **`getUserPrincipal`**. Este é um requerimento da especificação do J2EE.

O elemento <security-proxy> identifica uma implementação de proxy de segurança personalizada. Ele permite as checagens de segurança por solicitação fora do escopo do modelo de segurança declarativo EJB, sem a incorporação da lógica de segurança na implementação EJB. Você pode usar uma implementação da interface **`org.jboss.security.SecurityProxy`**. Alternativamente, você pode usar uma interface comum que usa um objeto para implementar métodos na página principal, remota, página principal local ou interfaces locais do EJB. A instância deve ser quebrada automaticamente numa implementação que delega as invocações de métodos ao objeto, caso a classe gerada não implementar a interface **`SecurityProxy`**. O **`org.jboss.security.SubjectSecurityProxy`** é um exemplo de implementação **`SecurityProxy`** usado pela instalação padrão.

Observe um exemplo simples do **`SecurityProxy`** personalizado no contexto de um bean de sessão sem estado trivial. O **`SecurityProxy`** personalizado valida que ninguém invoque o método **`echo`** do bean com quatro letras como seu argumento. Esta checagem não é possível com a segurança baseada na função. Neste caso, você não pode definir uma função **`FourLetterEchoInvoker`**, uma vez que o contexto de segurança é um argumento de método e não uma propriedade do chamador. O código para o **`SecurityProxy`** personalizado é gerado no Exemplo 3.1, “Implementação EchoSecurityProxy Personalizada. ”

Exemplo 3.1. Implementação EchoSecurityProxy Personalizada.

```
package org.jboss.book.security.ex1;

import java.lang.reflect.Method;
import javax.ejb.EJBContext;

import org.apache.log4j.Category;

import org.jboss.security.SecurityProxy;

/** A simple example of a custom SecurityProxy implementation
* that demonstrates method argument based security checks.
* @author Scott.Stark@jboss.org
* @version $Revision: 1.4 $
*/
public class EchoSecurityProxy implements SecurityProxy
{
Category log = Category.getInstance(EchoSecurityProxy.class);
Method echo;

public void init(Class beanHome, Class beanRemote,
Object securityMgr)
throws InstantiationException
{
log.debug("init, beanHome="+beanHome
+ ", beanRemote="+beanRemote
+ ", securityMgr="+securityMgr);
// Get the echo method for equality testing in invoke
try {
Class[] params = {String.class};
echo = beanRemote.getDeclaredMethod("echo", params);
} catch(Exception e) {
String msg = "Failed to find an echo(String) method";
log.error(msg, e);
throw new InstantiationException(msg);
}
```

```
}

public void setEJBContext(EJBContext ctx)
{
log.debug("setEJBContext, ctx="+ctx);
}

public void invokeHome(Method m, Object[] args)
throws SecurityException
{
// We don't validate access to home methods
}

public void invoke(Method m, Object[] args, Object bean)
throws SecurityException
{
log.debug("invoke, m="+m);
// Check for the echo method
if (m.equals(echo)) {
// Validate that the msg arg is not 4 letter word
String arg = (String) args[0];
if (arg == null || arg.length() == 4)
throw new SecurityException("No 4 letter words");
}
// We are not responsible for doing the invoke
}
}
```

O **EchoSecurityProxy** confere se o método a ser invocado na instância do bean corresponde ao método **echo(String)** carregado no método init. Caso haja um correspondente, o argumento do método é obtido e seu comprimento comparado com 4 ou nulo. Ambos os casos resultam num **SecurityException** sendo lançado.

Não há dúvida de que isto é um exemplo planejado, mas apenas em seu próprio aplicativo. Uma solicitação comum é que aplicativos devem executar checagens de segurança baseadas no valor dos argumentos do método. O motivo do exemplo é demonstrar como a segurança personalizada acima do escopo do modelo de segurança declarativo padrão pode ser introduzido independente de uma implementação de bean. Isto permite que a especificação e a codificação das solicitações de segurança sejam delegadas às especificações de segurança. Uma vez que a camada proxy pode ser realizada independente da implementação do bean, a segurança pode ser alterada para combinar as solicitações do ambiente de implementação.

O descritor **jboss.xml** associado que instala o **EchoSecurityProxy** como um proxy personalizado para o **EchoBean** é gerado no Exemplo 3.2, “descritor jboss.xml ”.

Exemplo 3.2. descritor jboss.xml

```
<jboss>
<security-domain>java:/jaas/other</security-domain>

<enterprise-beans>
<session>
<ejb-name>EchoBean</ejb-name>
<security-proxy>org.jboss.book.security.ex1.EchoSecurityProxy</security-
```

```
proxy>
</session>
</enterprise-beans>
</jboss>
```

Agora teste o proxy personalizado pela rodagem de um cliente que tenta invocar o método **`EchoBean.echo`** com os argumentos **`Hello`** e **`Four`**, conforme ilustrado no seguinte fragmento:

```
public class ExClient
{
public static void main(String args[])
throws Exception
{
Logger log = Logger.getLogger("ExClient");
log.info("Looking up EchoBean");

InitialContext iniCtx = new InitialContext();
Object ref = iniCtx.lookup("EchoBean");
EchoHome home = (EchoHome) ref;
Echo echo = home.create();

log.info("Created Echo");
log.info("Echo.echo('Hello') = "+echo.echo("Hello"));
log.info("Echo.echo('Four') = "+echo.echo("Four"));
}
}
```

A primeira chamada deve ser bem sucedida, enquanto que a segunda deve falhar uma vez que **`Four`** é uma palavra de quatro letras. Execute o cliente usando Ant a partir do diretório das amostras:

```
[examples]$ ant -Dchap=security -Dex=1 run-example
run-example1:
...
[echo] Waiting for 5 seconds for deploy...
[java] [INFO,ExClient] Looking up EchoBean
[java] [INFO,ExClient] Created Echo
[java] [INFO,ExClient] Echo.echo('Hello') = Hello
[java] Exception in thread "main" java.rmi.AccessException:
SecurityException; nested exception is:
[java] java.lang.SecurityException: No 4 letter words
...
[java] Caused by: java.lang.SecurityException: No 4 letter words
...
\t
```

O resultado é a chamada do método **`echo('Hello')`** bem sucedida e a chamada do método **`echo('Four')`**\tresultando numa exceção confusa, também esperada. O resultado acima está truncado para adaptação neste livro. A parte principal da exceção é que o **`SecurityException("No 4 letter words")`** gerado pelo **`EchoSecurityProxy`** foi lançado para abortar a tentativa de invocação, conforme o desejado.

CAPÍTULO 4. ARQUITETURA DE EXTENSÃO DE SEGURANÇA DO JBOSS

A discussão anterior da camada geral de segurança do JBoss declara que o *JBoss Security Extension framework* (JBossSX) é uma implantação das interfaces da camada de segurança. Este é o propósito principal do framework do JBossSX. O framework apresenta uma boa oferta de possibilidades de personalização para integração em infra-estruturas existentes. Uma infra-estrutura de segurança pode ser qualquer coisa de um banco de dados ou um servidor LDAP para um pacote de software de segurança sofisticado. A flexibilidade de integração é atingida usando o *Pluggable Authentication Model* (PAM) disponível no framework JAAS.

A parte principal do framework do JBOssSX é a implantação do **`org.jboss.security.plugins.JaasSecurityManager`**. Esta é a implantação padrão das interfaces **`AuthenticationManager`** e **`RealmMapping`**. A Figura 4.1, “Relação entre o valor do descritor <security-domain> de implantação, recipiente do componente e JaasSecurityManager.” apresenta como o **`JaasSecurityManager`** integra-se ao EJB e camadas de recipientes da web, baseados no elemento <security-domain> do descritor de implantação do componente correspondente.

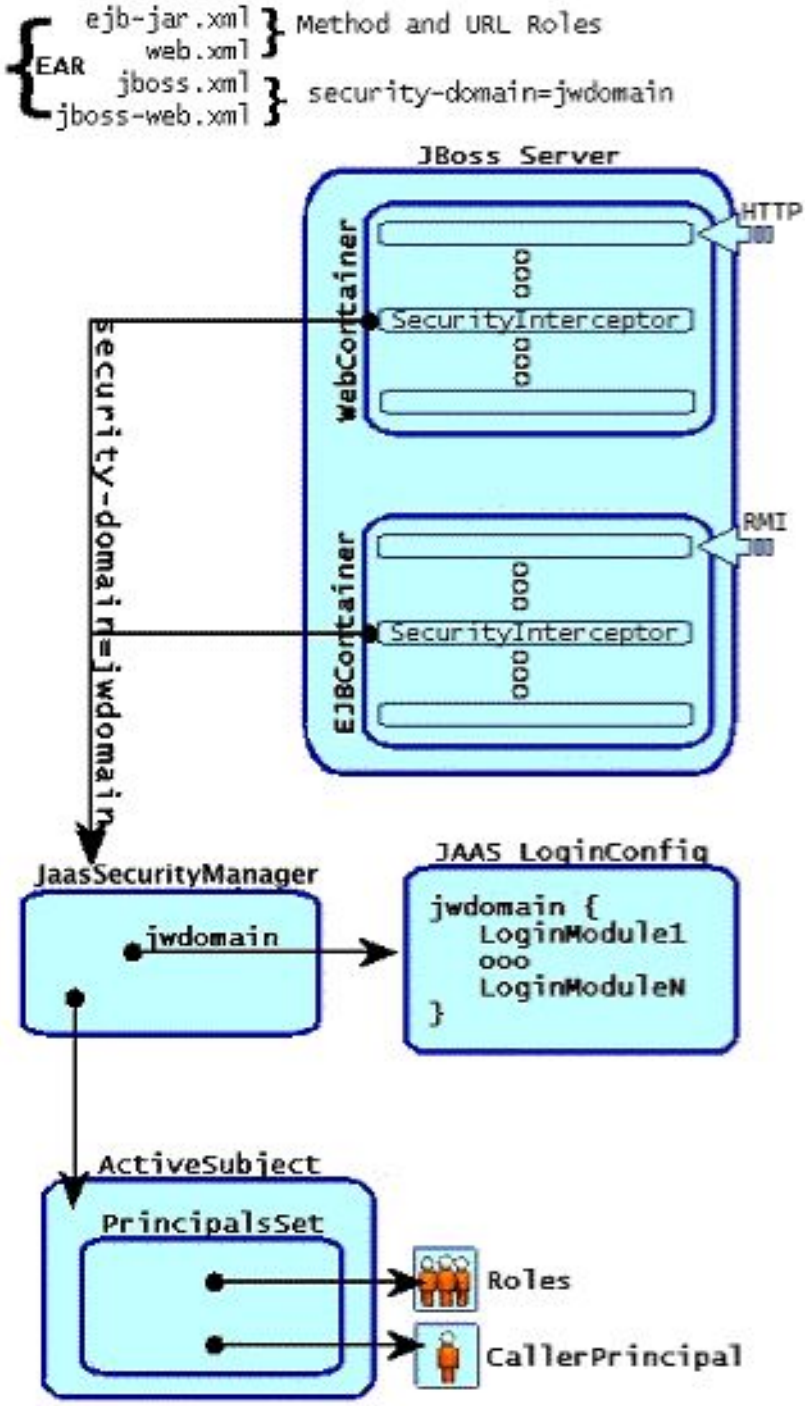


Figura 4.1. Relação entre o valor do descritor <security-domain> de implantação, recipiente do componente e JaasSecurityManager.

A Figura 4.1, “Relação entre o valor do descritor <security-domain> de implantação, recipiente do componente e JaasSecurityManager.” representa um aplicativo empresarial que contém ambos EJBs e conteúdo da web protegidos sob o **`jwdomain`** de domínio de segurança. Os recipientes EJB e da web possuem um interceptor de arquitetura que reforça o modelo de segurança do recipiente. No período de implantação, o valor do elemento <security-domain> nos descritores **`jboss.xml`** e **`jboss-web.xml`** é usado para obter uma instância do gerenciador de segurança associada com o recipiente. Em seguida, o interceptor de segurança usa o gerenciador de segurança para executar sua função. Quando um componente protegido for solicitado, o interceptor de segurança delega checagens de segurança para proteger a instância do gerenciador de segurança associado com o recipiente.

A implementação JBossSX **`JaasSecurityManager`** executa checagens de segurança baseadas na informação associada com a instância que resulta da execução dos módulos de logon JAAS

configurados sob o nome de combinação do valor do elemento <security-domain>. Nós descreveremos mais a fundo a implementação **`JaasSecurityManager`** e seu uso na próxima seção.

4.1. COMO O JAASSECURITYMANAGER USA O JAAS

O **`JaasSecurityManager`** usa os pacotes JAAS para implementar o **`AuthenticationManager`** e o comportamento de interface **`RealmMapping`**. Basicamente, seu comportamento deriva de uma execução das instâncias do módulo de logon que estão configuradas sob o nome que coincide ao domínio de segurança pelo qual o **`JaasSecurityManager`** foi determinado. Os módulos de logon implantam a autenticação do principal do domínio de segurança e o comportamento do mapeamento de função. Portanto, você pode usar o **`JaasSecurityManager`** através de diferentes domínios de segurança simplesmente pela conexão de diferentes configurações de módulo do logon pelos domínios.

Para ilustração dos detalhes do uso do **`JaasSecurityManager`** do processo de autenticação do JAAS, você passará pela invocação do cliente de uma invocação do método da página principal do EJB. O pré-requisito da configuração é que o EJB seja implantado no servidor do JBoss e seus métodos de interface da página principal sejam protegidos usando os elementos <method-permission> no descritor, além de ser determinado um domínio de segurança nomeado **`jwdomain`**, usando o descritor **`jboss.xml`** do elemento <security-domain>.

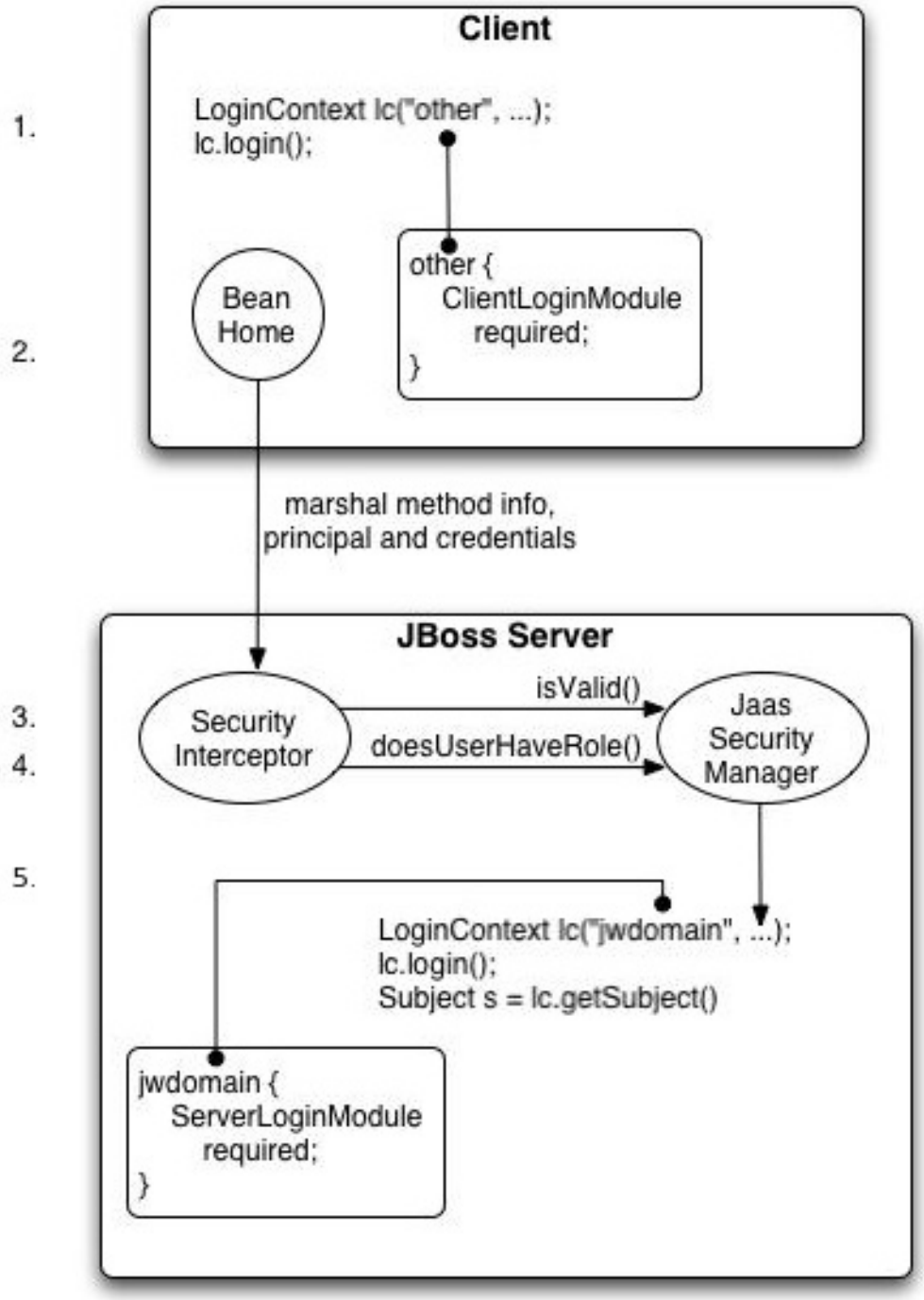


Figura 4.2. Etapas de Autenticação do Método da Página Principal do EJB e Invocação da Autorização.

A Figura 4.2, “Etapas de Autenticação do Método da Página Principal do EJB e Invocação da Autorização.” fornece uma visualização do cliente à comunicação do servidor. Segue abaixo as etapas numeradas:

1. O cliente deve executar um logon JAAS para estabelecer o principal e os credenciais para a autenticação, além de ser etiquetado *Logon ao Lado do Cliente* na figura. Este é o procedimento de como os clientes estabelecem suas identidades no JBoss. O suporte para apresentação da informação de logon através das propriedades **`InitialContext`** do JNDI é fornecido através de uma configuração alternativa.

   O logon JAAS implica na criação de uma instância **`LoginContext`** e passagem do nome da configuração a ser usada. O nome da configuração é **`other`**. Este logon de uma única vez

associa o principal do logon e as credenciais com todas as invocações do método EJB subsequentes. Perceba que o processo pode não ser autenticado pelo usuário. A natureza do logon ao lado do cliente depende da configuração do módulo de logon que o cliente usa. Nesta amostra, a entrada da configuração do logon ao lado do cliente **`other`** é configurada para uso do módulo **`ClientLoginModule`** (um **`org.jboss.security.ClientLoginModule`**). Este é o módulo ao lado do cliente padrão que simplesmente efetua o bind no nome do usuário e senha para a camada de invocação EJB do JBoss para uma autenticação mais tarde no servidor. A identidade do cliente não é autenticada no cliente.

2. O cliente obtém uma interface da página principal do EJB e tenta criar um bean. Esse evento é etiquetado como a *Invocação do Método de Página Principal*. Isto resulta no envio da invocação do método da interface de página principal ao servidor do JBoss. A invocação inclui os argumentos do método passados pelo cliente, juntamente com a identidade do usuário e credenciais do logon do JAAS ao lado do cliente, executados na Etapa 1.

3. No lado do servidor, o interceptor de segurança primeiramente requer a autenticação do usuário invocando a chamada, que uma vez ao lado do cliente, envolve o logon do JAAS.

4. O domínio de segurança pelo qual o EJB é protegido determina a escolha dos módulos de logon. O nome do domínio de segurança é usado como o nome da entrada da configuração do logon passado ao construtor **`LoginContext`**. O domínio de segurança EJB é o **`jwdomain`**. Caso o logon do JAAS autenticar o usuário, o JAAS **`Subject`** e contém o seguinte em seu **`PrincipalsSet`**:

    - O **`java.security.Principal`** que corresponde à identidade do cliente, como é conhecida no ambiente de segurança da implantação.

    - O **`java.security.acl.Group`** nomeado **`Roles`** que contém o nome de função a partir do domínio do aplicativo pelo qual o usuário é determinado. Os objetos **`org.jboss.security.SimplePrincipal`** são usados para representar os nomes de função. O **`SimplePrincipal`** é uma implantação baseada na sequência simples do **`Principal`**. Essas funções são usadas para validar as funções determinadas aos métodos no **`ejb-jar.xml`** e na implantação do método **`EJBContext.isCallerInRole(String)`**.

    - Um **`java.security.acl.Group`** opcional nomeado **`CallerPrincipal`**, que contém um **`org.jboss.security.SimplePrincipal`** único que corresponde à identidade do chamador do domínio do aplicativo. O membro do grupo único **`CallerPrincipal`** será o valor retornado pelo método **`EJBContext.getCallerPrincipal()`**. O propósito do mapeamento é permitir um **`Principal`** também conhecido no ambiente de segurança opcional para mapear um **`Principal`** com o nome conhecido pelo aplicativo. Na ausência de um mapeamento **`CallerPrincipal`**, o principal do ambiente de segurança implantado é usado como o valor do método **`getCallerPrincipal`**. Quer dizer, o principal operacional é o mesmo que o principal do domínio do aplicativo.

5. O passo final da checagem do interceptor de segurança é verificar que o usuário autenticado possui permissão para invocar o método solicitado. Isto é etiquetado como *Autorização ao lado do servidor* na Figura 4.2, “Etapas de Autenticação do Método da Página Principal do EJB e Invocação da Autorização.”. As seguintes etapas descrevem a execução da autorização:

    - Obtém os nomes das funções permitidas para acesso ao método EJB a partir do container. Os nomes da função são determinados pelo descritor **`ejb-jar.xml`** dos elementos <role-name> de todos os elementos <method-permission> contendo o método invocado.

    - Caso nenhuma função tenha sido determinada ou o método seja especificado num

elemento exclude-list, o acesso ao método será negado. Do contrário, o método **doesUserHaveRole** é invocado no gerenciador de segurança pelo interceptor de segurança para verificar se o chamador possui um dos nomes de função determinados. Esse método interage através dos nomes de função e checa se o grupo **Roles** do Assunto do usuário autenticado contém um **SimplePrincipal** com o nome de função determinado. O acesso é permitido caso qualquer nome de função seja membro do grupo **Roles**. O acesso é negado caso nenhum dos nomes de funções forem membros.

- Caso o EJB for configurado por um proxy de segurança personalizada, a invocação de método é delegada também. Caso o proxy de segurança desejar negar acesso ao chamador, ele lançará um **java.lang.SecurityException**. Caso nenhum **SecurityException** for lançado, o acesso ao método é permitido e a invocação de método passa ao próximo interceptor do recipiente. Perceba que o **SecurityProxyInterceptor** manuseia esta checagem e este interceptor não é apresentado.

- O Tomcat gerencia elementos de restrição de segurança e verificação de função para as conexões do JBoss Web.

  Quando uma solicitação receber uma conexão da web, o Tomcat checa as restrições de segurança definidas no **web.xml** que coincide com o recurso solicitado e o método HTTP acessado.

  Caso uma restrição existir, o Tomcat chama o JaasSecurityManager para executar a autenticação principal, que em troca garante que as funções do usuário sejam associadas com o objeto do principal.

  A verificação da função é executada inteiramente pelo Tomcat, incluindo as checagens dos parâmetros, tais como **allRoles** e se é que o **STRICT_MODE** é usado ou não.

Cada invocação do método EJB protegido, ou acesso do conteúdo da web protegido, requer a autenticação e autorização do chamador uma vez que a informação de segurança é manuseada como um atributo sem estado da solicitação que deve ser apresentada e validada em cada solicitação. Isto pode ser uma operação cara caso o logon do JAAS envolver a comunicação cliente-para-servidor. Devido a isto, o **JaasSecurityManager** suporta a noção de um cache de autenticação que é usado para armazenar a informação do principal e credencial a partir dos logons anteriores bem sucedidos. Você pode especificar a instância do cache de autenticação para uso como parte da configuração **JaasSecurityManager** conforme descrito na seção seguinte do serviço MBean associado. Na ausência de qualquer cache de usuário definido, o cache padrão que mantém a informação do credencial para o período de tempo configurável será usado.

4.2. O JAASSECURITYMANAGERSERVICE MBEAN

O serviço do MBean **JaasSecurityManagerService** administra os gerenciadores de segurança. Embora seu nome inicie com *Jaas*, os gerenciadores de segurança que isto manuseia não precisam usar JAAS em suas implantações. O nome surgiu a partir do fato que a implantação do gerenciador de segurança é um **JaasSecurityManager**. A função principal do **JaasSecurityManagerService** é externalizar a implantação do gerenciador de segurança. Você pode alterar a implantação do gerenciador de segurança fornecendo uma implantação alternativa das interfaces **AuthenticationManager** e **RealmMapping**.

A segunda função de fundamento do **JaasSecurityManagerService** é fornecer uma implantação **javax.naming.spi.ObjectFactory** do JNDI para permitir um gerenciamento simples sem código do nome JNDI para mapeamento da implantação do gerenciador de segurança. A segurança é ativada

pela especificação do nome JNDI da implantação de segurança através do elemento descritor da implantação <security-domain>.

Quando você especifica o nome JNDI, é necessário ter um bind de objeto disponível para uso. O **JaasSecurityManagerService** gerencia a associação das instâncias do gerenciador de segurança para nomes pela efetuação de bind na próxima referência de sistema de nomeação (entre o mesmo), como o JNDI ObjectFactory sob o nome de **java:/jaas**. Tudo isto, para simplificar a configuração do nome JNDI para bindings do gerenciador de segurança. Isto permite uma convenção de nomeação do **java:/jaas/XYZ** do formulário como o valor para o elemento <security-domain> e a instância do gerenciador de segurança para o domínio de segurança ser criado, conforme isto seja necessário.

O gerenciador de segurança para o **XYZ** de domínio é criado na primeira busca relacionada à efetuação de bind ao **java:/jaas/XYZ**, criando uma instância da classe especificada pelo atributo **SecurityManagerClassName** usando um construtor que leva o nome do domínio de segurança.

IMPORTANTE

Na versões anteriores à versão 5.0 da Plataforma do Aplicativo Enterprise, o prefixo "***java:/jaas***, em cada elemento do descritor da implantação **<security-domain>**, era solicitado para efetuação do bind correta no nome JNDI do domínio de segurança para os binds do gerenciador de segurança.

O prefixo ***java:/jaas*** não é solicitado pela declaração do domínio de segurança a partir da Plataforma do Aplicativo JBoss Enterprise 5. O prefixo **java:/jaas** ainda é suportado e é mantido para a compatibilidade inversa.

Por exemplo, considere o seguinte trecho de configuração de segurança do seguinte recipiente:

```
<jboss>
    <!-- Configure all containers to be secured under the "customer"
security domain -->
    <security-domain>customer</security-domain>
    <!-- ... -->
</jboss>
```

Qualquer busca de nome **customer** retornará uma instância de gerenciador de segurança associada com o domínio de segurança nomeado **customer**. Este gerenciador de segurança implementará as interfaces de segurança e será do tipo especificado pelo atributo **JaasSecurityManagerServiceSecurityManagerClassName**.

O MBean do **JaasSecurityManagerService** é configurado por padrão para uso na distribuição do JBoss padrão e você normalmente poderá usar a configuração padrão assim como ela é. Os atributos configuráveis do **JaasSecurityManagerService** incluem:

SecurityManagerClassName

O nome da classe que fornece a implantação do gerenciador de segurança. A implantação deve suportar ambas as interfaces **org.jboss.security.AuthenticationManager** e **org.jboss.security.RealmMapping**. O padrão é **org.jboss.security.plugins.JaasSecurityManager**, caso não tenha sido especificado.

CallbackHandlerClassName

O nome da classe que fornece a implantação **javax.security.auth.callback.CallbackHandler** usada pelo **JaasSecurityManager**.

NOTA

Você pode substituir o manuseador usado pela implantação padrão **`JaasSecurityManager`**, caso a implantação padrão (**`org.jboss.security.auth.callback.SecurityAssociationHandler`**) não atender suas necessidades. A maioria das implantações irão considerar o manuseador padrão suficiente.

SecurityProxyFactoryClassName

O nome da classe que fornece a implantação **`org.jboss.security.SecurityProxyFactory`**. O padrão é **`org.jboss.security.SubjectSecurityProxyFactory`**, caso não seja especificado.

AuthenticationCacheJndiName

Especifica a localização da política do cache credencial de segurança. Isto é tratado primeiramente como uma localização **`ObjectFactory`** capaz de retornar as instâncias baseando-se no <security-domain>. Isto é realizado anexando o nome do domínio de segurança ao nome quando observando o **`CachePolicy`** para o domínio. Caso isto falhe, a localização é tratada como um **`CachePolicy`** único para todos os domínios de segurança. A política de cache de período limitado é usada como padrão.

DefaultCacheTimeout

Especifica o intervalo da política de cache de período limitado em segundos. O valor padrão é 1800 segundos (30 minutos). O valor de uso para o intervalo é uma média entre as operações de autenticações frequentes e do período pelo qual a informação de credencial pode permanecer fora de sync em relação ao armazenamento da informação de segurança. Caso você deseje desativar o cache dos credenciais de segurança, configure isto para 0 com o objetivo de forçar a autenticação a ocorrer todo o tempo. Isto não possui efeito caso o **`AuthenticationCacheJndiName`** tenha o valor padrão alterado.

DefaultCacheResolution

Especifica a resolução da política de cache de período limitado padrão em segundos. Isto controla o intervalo pelo qual o carimbo de tempo atual do cache é atualizado e deve ser inferior que **`DefaultCacheTimeout`**, com o objetivo do intervalo ser significativo. A resolução padrão é de 60 segundos (1 minuto). Isto não tem efeito caso o **`AuthenticationCacheJndiName`** tenha o valor padrão alterado.

DefaultUnauthenticatedPrincipal

Especifica o principal para uso para usuários não-autenticados. Esta configuração facilita a configuração de permissões padrões para usuários que não foram autenticados.

O **`JaasSecurityManagerService`** também suporta um número de operações úteis. Elas incluem o esvaziamento de qualquer cache de autenticação do domínio de segurança no período de rodagem e qualquer um dos métodos da interface do gerenciador de segurança.

O esvaziamento do cache de autenticação do domínio de segurança pode ser usado para remover todos os credenciais quando o armazenamento subjacente for atualizado e você desejar o estado do armazenamento a ser usado imediatamente. A assinatura da operação do MBean é a seguinte: **`public void flushAuthenticationCache(String securityDomain)`**.

Isto pode ser invocado de forma programática usando o seguinte trecho de código:

```
MBeanServer server = ...;
String jaasMgrName = "jboss.security:service=JaasSecurityManager";
ObjectName jaasMgr = new ObjectName(jaasMgrName);
Object[] params = {domainName};
String[] signature = {"java.lang.String"};
server.invoke(jaasMgr, "flushAuthenticationCache", params, signature);
```

A obtenção da lista dos usuários ativos fornece um instantâneo das chaves **`Principals`** no cache de autenticação do domínio de segurança que não estão expiradas. A assinatura da operação do MBean é a seguinte: **`public List getAuthenticationCachePrincipals(String securityDomain)`**.

Isto pode ser invocado de forma programática usando o seguinte trecho de código:

```
MBeanServer server = ...;
String jaasMgrName = "jboss.security:service=JaasSecurityManager";
ObjectName jaasMgr = new ObjectName(jaasMgrName);
Object[] params = {domainName};
String[] signature = {"java.lang.String"};
List users = (List) server.invoke(jaasMgr,
"getAuthenticationCachePrincipals",
                                  params, signature);
```

O gerenciador de segurança possui alguns métodos de acesso adicionais.

```
public boolean isValid(String securityDomain, Principal principal, Object
credential);
public Principal getPrincipal(String securityDomain, Principal principal);
public boolean doesUserHaveRole(String securityDomain, Principal
principal,
                                Object credential, Set roles);
public Set getUserRoles(String securityDomain, Principal principal, Object
credential);
```

Eles fornecem acesso ao **`AuthenticationManager`** correspondente e ao método da interface do domínio de segurança associado nomeado pelo argumento **`securityDomain`**.

4.3. O JAASSECURITYDOMAIN MBEAN

O **`org.jboss.security.plugins.JaasSecurityDomain`** é uma extensão do **`JaasSecurityManager`** que adiciona a noção de um **`KeyStore`**, um JSSE **`KeyManagerFactory`** e um **`TrustManagerFactory`** para suporte do SSL e outros casos de uso de criptografia. Os atributos configuráveis adicionais do **`JaasSecurityDomain`** incluem:

KeyStoreType

O tipo de implantação **`KeyStore`**. Este é o tipo de argumento passado ao método de criação **`java.security.KeyStore.getInstance(String type)`**. O padrão é **`JKS`**.

KeyStoreURL

Um URL para a localização do banco de dados **`KeyStore`**. Ele é usado para obter um **`InputStream`** para inicializar o **`KeyStore`**. Caso a sequência não conter um URL de nome/valor, o valor será tratado como um arquivo.

KeyStorePass

A senha associada com os conteúdos do banco de dados do **KeyStore**. O **KeyStorePass** é também usado na combinação com os atributos **Salt** e **IterationCount** para criar uma chave secreta do PBE usado com as operações de codificação/decodificação. O formato do valor do atributo **KeyStorePass** é um dos seguintes:

- A senha de texto simples para o **KeyStore**. O valor **toCharArray()** da sequência é usado sem qualquer manipulação.
- O comando a ser executado para obter a senha de texto plano. O formato é **{EXT}...** onde o **...** é a mesma linha de comando passada ao método **Runtime.exec(String)** para executar o comando específico da plataforma. O primeiro resultado da linha de comando é usado como senha.
- A classe a ser criada para obter a senha de texto simples. O formato é **{CLASS}classname[:ctorarg]**, onde o **[:ctorarg]** é uma sequência opcional que será passada ao construtor quando instanciando o **classname**. A senha é obtida a partir do nome de classe pela invocação de um método **toCharArray()**, caso encontrado. Do contrário, o método **toString()** será usado.

Salt

O valor do salt **PBEParameterSpec**.

IterationCount

O valor de contagem de interação **PBEParameterSpec**.

TrustStoreType

O tipo de implantação **TrustStore**. Este é o tipo de argumento passado ao método de criação **java.security.KeyStore.getInstance(String type)**. O padrão é **JKS**.

TrustStoreURL

Um URL para a localização do banco de dados **TrustStore**. Isto é usado para obter um **InputStream** para inicializar o **KeyStore**. Caso a sequência não seja um URL de valor, ele será usado como um arquivo.

TrustStorePass

A senha associada com os conteúdos do banco de dados do armazenamento. O **TrustStorePass** é uma senha simples e não possui as mesmas opções de configuração como o **KeyStorePass**.

ManagerServiceName

Determina a sequência do nome do objeto de um MBean do serviço do gerenciador de segurança. Ele é usado para registrar os padrões para registro do **JaasSecurityDomain** como um gerenciador de segurança sob o **java:/jaas/<domain>**, onde o **<domain>** é o nome passado ao construtor do MBean. O padrão do nome é **jboss.security:service=JaasSecurityManager**.

PARTE II. SEGURANÇA DO APLICATIVO

CAPÍTULO 5. VISÃO GERAL

A segurança do Aplicativo

Autenticação

O processo pelo qual o servidor determina se é que um usuário deve estar apto a acessar um sistema ou uma operação.

Autorização

Processo pelo qual o servidor determina se é que o usuário autenticado possui permissão de acesso aos privilégios específicos ou recursos no sistema ou operação.

Mapeamento

Processo pelo qual o servidor associa usuários autenticados com os perfis de autorização pré-definidos.

Auditoria

Processo pelo qual o servidor monitora a autenticação e autorização dos eventos de segurança.

As políticas de autenticação, autorização e mapeamento são configuradas com a granulação à nível do aplicativo usando o conceito de um domínio de segurança na Plataforma do Aplicativo JBoss Enterprise 5.

Domínio de Segurança

O conjunto de políticas de autenticação, autorização e mapeamento que são definidas no XML e estão disponíveis aos aplicativos no período de rodagem usando a Nomeação do Java e Interface do Diretório (JNDI - Java Naming and Directory Interface).

O domínio de segurança pode ser definido no perfil do servidor num descritor de implantação do aplicativo.

A auditoria para o EJB3 e o recipiente da web são configurados independemente entre si e operados à nível do servidor.

A Plataforma do Aplicativo JBoss Enterprise usa o framework do modelo conectável para implementar segurança, fornecendo a separação da implementação de segurança e o design do aplicativo. O framework do módulo conectável fornece flexibilidade na implantação do aplicativo, configuração e desenvolvimento futuro. Os módulos de implantação da funcionalidade de segurança são conectados ao framework e fornecem funções de segurança aos aplicativos. Os aplicativos conectam-se ao framework de segurança através dos domínios de segurança.

Os aplicativos podem ser implantados em novos cenários com a configuração de segurança alterada pela associação dos mesmos com um domínio de segurança diferente. Os novos métodos de segurança podem ser implantados conectando o Jboss, internamente personalizado ou módulos de terceiros ao framework. Os aplicativos ganham funcionalidade de segurança daquele módulo sem nenhuma re-codificação do aplicativo, apenas pela reconfiguração do domínio de segurança.

O Capítulo 6, *Esquema do Domínio de Segurança* descreve a configuração do XML do domínio de segurança.

O Capítulo 7, *Autenticação* contém informação detalhada sobre a política de autenticação do domínio de segurança.

O Capítulo 8, *Autorização* contém informação detalhada sobre a política de autorização do domínio de segurança.

O Capítulo 9, *Mapeamento* contém informação detalhada sobre a política de mapeamento do domínio de segurança.

CAPÍTULO 6. ESQUEMA DO DOMÍNIO DE SEGURANÇA

O esquema do domínio de segurança é construído usando o XML.

O XSD que define a estrutura do domínio de segurança está declarado no arquivo security-beans_1_0.xsd. A localização do arquivo varia dependendo da versão da Plataforma do Aplicativo JBoss Enterprise em uso.

5.1

`/schema/security-beans_1_0.xsd` dentro do **`jboss-as/lib/jbosssx.jar/`**

5.0, 5.0.1

`/schema/security-beans_1_0.xsd` dentro do **`jboss-as/common/lib/jbosssx.jar`**

O esquema é o mesmo independente de qual servidor da Plataforma do Aplicativo do JBoss Enterprise estar em uso.

Exemplo 6.1. Esquema do Domínio de Segurança

```
<application-policy xmlns="urn:jboss:security-beans:1.0" name="">
  <authentication>
    <login-module code="" flag"
[required|requisite|sufficient|optional]" extends="">
      <module-option name=""></module-option>
    </login-module>
  </authentication>
  <authorization>
    <policy-module code="" flag="
[required|requisite|sufficient|optional]"/>
  </authorization>
  <mapping>
    <mapping-module code="" type"" />
  </mapping>
</application-policy>
```

Descritores do Elemento do Domínio de Segurança

<application-policy>

Os elementos do domínio de segurança estão contidos no elemento <application-policy>, independente de como ele é implantado no sistema. O elemento usa o ML namespace conforme declarado no atributo ***`xmlns`***.

O atributo ***name*** configura o nome do domínio de segurança referenciado por um aplicativo. O nome do domínio de segurança está vinculado no JNDI sob o contexto **`java:/jaas`** e é acessado pelos aplicativos através da referência em seus descritores de implantação.

O elemento <application-policy> pode conter um número de elementos filhos que determinam o comportamento para o domínio de segurança e todos os aplicativos que usam o mesmo. Esses elementos estão descritos em maiores detalhes na Seção 6.1, “<authentication>”, Seção 6.2, “<authorization>” e Seção 6.3, “<mapping>”.

6.1. <AUTHENTICATION>

O elemento <authentication> contém os seguintes elementos filhos.

<authentication>

Esse elemento contém os elementos <login-module> que controlam quais módulos de autenticação são usados quando autenticando usuários que conectam-se através do aplicativo.

Quando os elementos <login-module> estiverem presentes, eles formam um grupo coletivo de solicitações que devem ser realizados antes da autenticação ser confirmada. Esse grupo coletivo é chamado *stack*.

<login-module>

Este elemento usa o atributo ***code*** para especificar qual implantação do módulo de logon um aplicativo pode usar e o atributo ***flag*** para informar o aplicativo como analisar cada módulo de logon presente na pilha. O atributo ***flag*** suporta os seguintes valores:

requeridos

O módulo deve suceder para que uma autenticação seja bem sucedida. Caso quaisquer <login-module> falharem, a autenticação também falhará. Os módulos de logon restantes na pilha são chamados independente do resultado de autenticação.

solicitação

É necessário que o módulo seja bem sucedido. Caso ele suceda, a autenticação continua na parte inferior da pilha. Caso o módulo falhe, o controle retorna imediatamente ao aplicativo.

suficiência

Não há necessidade de que o módulo de logon suceda. Caso ele seja bem sucedido, o controle retorna imediatamente ao aplicativo. Caso o módulo falhe, a autenticação continua na parte inferior do aplicativo.

opcional

Não é necessário que o módulo de logon suceda. A autenticação continua a proceder na parte inferior da pilha independente de módulo de logon suceder ou falhar.

Cada <login-module> contém um conjunto de elementos <module-option> que definem futuramente as configurações solicitadas pela implantação do módulo de logon.

<module-option>

Cada módulo de logon possui o próprio conjunto de configuração. O atributo do nome especifica a propriedade solicitada pelo módulo de logon e o valor é declarado no CDATA do elemento <module-option>. As opções de módulo dependem em qual módulo de logon você escolher. A Seção 12.1, “Módulos de Uso” descreve as opções de módulos em maiores detalhes.

6.2. <AUTHORIZATION>

<authorization>

O elemento contém os elementos <policy-module> que definem o módulo de política usado para autorizar usuários e se é que o módulo é requerido ou não:

Quando os elementos <policy-module> estiverem presentes, eles formarão uma grupo coletivo de solicitações que devem ser realizadas antes da autorização ser verificada. Este grupo coletivo é chamado *stack*.

<policy-module>

Este elemento usa o atributo ***code*** para especificar qual implementação do módulo da política um aplicativo pode utilizar e o atributo ***flag*** para informar o aplicativo de como analisar cada módulo de política presente na pilha da política. O atributo ***flag*** suporta os seguintes valores:

requeridos

O módulo deve suceder para a autorização ser bem sucedida. Caso qualquer <policy-module> solicitado falhar, a tentativa de autorização falhará. Os módulos restantes na pilha são chamados independente do resultado do módulo.

solicitação

É necessário que o módulo seja bem sucedido. Caso ele suceda, a autorização continua na parte inferior da pilha. Caso ele falhe, o controle retorna imediatamente ao aplicativo.

suficiência

Não é necessário que o módulo de logon suceda. Caso ele seja bem sucedido, o controle retorna imediatamente ao aplicativo. Caso ele falhe, a autorização continua na parte inferior da pilha.

opcional

Não é necessário que o módulo de logon seja bem sucedido. A autorização continua na parte inferior da pilha independente do módulo suceder ou falhar.

6.3. <MAPPING>

<mapping>

Este elemento contém os elementos <mapping-module> que são usados para definir os parâmetros dos elementos do módulo de mapeamento.

Quando os elementos <mapping-module> estiverem presentes, eles formarão um grupo coletivo de solicitações que devem ser realizadas antes do mapeamento ser bem sucedido. Esse grupo coletivo é chamado *stack*.

<mapping-module>

Esses elementos usam o atributo ***code*** para especificar qual implantação do módulo de mapeamento um aplicativo pode usar e o atributo ***flag*** para informar o aplicativo como analisar cada módulo de mapeamento presente na pilha da política. O atributo flag suporta os seguintes valores:

requeridos

O módulo deve suceder para o mapeamento ser bem sucedido. Caso qualquer <mapping-module> falhar, a autenticação falhará. Os módulos restantes na pilha são chamados independente do resultado da autenticação.

solicitação

É necessário que o módulo seja bem sucedido. Caso ele suceda, o mapeamento continua na parte inferior da pilha. Caso o módulo falhe, o controle retorna imediatamente ao aplicativo.

suficiência

Não é necessário que o módulo seja bem sucedido. Caso suceda, o controle retorna imediatamente ao aplicativo. Caso o módulo falhe, o mapeamento continua na parte inferior da pilha.

opcional

O módulo não precisa ser bem sucedido. A autenticação continua na parte inferior da pilha, independente do módulo ser bem sucedido ou falhar.

CAPÍTULO 7. AUTENTICAÇÃO

As seguintes amostras descrevem maneiras que você pode usar as políticas de aplicativo num domínio de segurança.

A única política de autenticação é declarada nas amostras para maior clareza, no entanto você pode incluir os elementos <authorization> e <mapping> no mesmo <application-policy>. Refira-se à Seção 6.1, “<authentication>” para uma informação detalhada sobre o elemento <authentication>.

Exemplo 7.1. Política de autenticação da pilha de logon único

Esta amostra descreve uma configuração de domínio de segurança única nomeada **`jmx-console`** que usa um módulo de logon único, **`UsersRolesLoginModule`** (refira-se à Seção 12.1.5, “UsersRolesLoginModule”).

O módulo de logon é fornecido pelo usuário e as propriedades de arquivos do diretório **`jboss-as/server/$PROFILE/conf/props`**.

Nesta instância, o <login-module> deve suceder ou a autenticação falhará.

```
<application-policy xmlns="urn:jboss:security-beans:1.0 name="jmx-
console">
   <authentication>
       <login-module
code="org.jboss.security.auth.spi.UsersRolesLoginModule"
flag="required">
           <module-option name="usersProperties">props/jmx-console-
users.properties</module-option>
           <module-option name="rolesProperties">props/jmx-console-
roles.properties</module-option>
       </login-module>
   </authentication>
</application-policy>
```

Exemplo 7.2. Política da autenticação da pilha de logon múltiplo

Esta amostra descreve a configuração do domínio de segurança nomeado **`web-console`** que usa dois módulos de logon na pilha de módulo de logon de autenticação.

Um <login-module> obtém os credenciais de logon usando o **`LdapLoginModule`** (referia-se à Seção 12.1.1, “LdapLoginModule”), onde o outro <login-module> obtém as credenciais de autenticação usando o **`BaseCertLoginModule`** (refira-se à Seção 12.1.7, “BaseCertLoginModule”).

Nesta instância, ambos os módulos são marcados como suficientes, portanto apenas um deles deve suceder para que a autenticação ocorra com êxito.

```
<application-policy xmlns="urn:jboss:security-beans:1.0 name="web-
console">
    <authentication>
    <!-- LDAP configuration -->
    <login-module code="org.jboss.security.auth.spi.LdapLoginModule"
         flag="sufficient" />
    <!-- database configuration -->
```

```
    <login-module
code="org.jboss.security.auth.spi.BaseCertLoginModule"
                        flag="sufficient" />

    </authentication>
</application-policy>
```

7.1. MANUSEADORES DE CHAMADA DE RETORNO PERSONALIZADA

A implementação dos manuseadores de chamada de retorno em procedimentos de autenticação permite que um módulo de logon autentique um usuário independente do método de autenticação do aplicativo do cliente.

Você pode implementar os manuseadores de chamada de retorno usando os seguintes métodos:

- Especificando o atributo CallbackHandlerClassName na definição JaasSecurityManagerService MBean do **`conf/jboss-service.xml`**.
- Injetando a instância do manuseador da chamada de retorno no JNDISecurityManagement bean do **`deploy/security/security-jboss-beans.xml`**.

Procedimento 7.1. Define o manuseador de chamada de retorno usando os atributos

Este procedimento descreve como especificar um manuseador da chamada de retorno no arquivo de configuração do **`jboss-service.xml`**.

1. **Abra o arquivo de configuração**
   Navegue ao ***`$JBOSS_HOME`*`/server/`*`$PROFILE`*`/conf/`**

   Abra o arquivo **`jboss-service.xml`**.

   Por padrão, o arquivo **`jboss-service.xml`** contém a configuração no Exemplo 7.3, “configuração padrão do jboss-service”

   Exemplo 7.3. configuração padrão do jboss-service

```
<?xml version="1.0" encoding="UTF-8"?>
...

<!--
=====================================================================
== -->
<!-- Security
-->
<!--
=====================================================================
== -->

<!-- JAAS security manager and realm mapping -->
  <mbean
code="org.jboss.security.plugins.JaasSecurityManagerService"
name="jboss.security:service=JaasSecurityManager">
  <!-- A flag which indicates whether the SecurityAssociation
```

```
server mode
  is set on service creation. This is true by default since the
  SecurityAssociation should be thread local for multi-threaded
server
  operation.-->
  <attribute name="ServerMode">true</attribute>

  <attribute
name="SecurityManagerClassName">org.jboss.security.plugins.JaasSec
urityManager</attribute>

  <attribute
name="DefaultUnauthenticatedPrincipal">anonymous</attribute>

  <!-- DefaultCacheTimeout: Specifies the default timed cache
policy timeout
  in seconds.
  If you want to disable caching of security credentials, set this
to 0 to
  force authentication to occur every time. This has no affect if
the
  AuthenticationCacheJndiName has been changed from the default
value.-->

  <attribute name="DefaultCacheTimeout">1800</attribute>

  <!-- DefaultCacheResolution: Specifies the default timed cache
policy
  resolution in seconds. This controls the interval at which the
cache
  current timestamp is updated and should be less than the
DefaultCacheTimeout
  in order for the timeout to be meaningful. This has no affect if
the
  AuthenticationCacheJndiName has been changed from the default
value.-->

  <attribute name="DefaultCacheResolution">60</attribute>

  <!-- DeepCopySubjectMode: This set the copy mode of subjects
done by the
  security managers to be deep copies that makes copies of the
subject
  principals and credentials if they are cloneable. It should be
set to
  true if subject include mutable content that can be corrupted
when
  multiple threads have the same identity and cache flushes/logout
clearing
  the subject in one thread results in subject references
affecting other
  threads.-->

  <attribute name="DeepCopySubjectMode">false</attribute>
```

```
  </mbean>

...
```

2. **Anexe o atributo**
   Para definir o manuseador da chamada de retorno padrão, anexe um elemento <attribute> como um filho do elemento <mbean> e especifique o nome inteiramente qualificado do manuseador da chamada de retorno. Refira-se ao Exemplo 7.4, “chamador da chamada de retorno anexado do jboss-service” para uma amostra do elemento <attribute>, com o manuseador da chamada de retorno especificada.

Exemplo 7.4. chamador da chamada de retorno anexado do jboss-service

```
<?xml version="1.0" encoding="UTF-8"?>
...

<!--
=====================================================================
== -->
<!-- Security
-->
<!--
=====================================================================
== -->

<!-- JAAS security manager and realm mapping -->
  <mbean
code="org.jboss.security.plugins.JaasSecurityManagerService"
name="jboss.security:service=JaasSecurityManager">
  <!-- A flag which indicates whether the SecurityAssociation
server mode
  is set on service creation. This is true by default since the
  SecurityAssociation should be thread local for multi-threaded
server
  operation.-->
  <attribute name="ServerMode">true</attribute>

  <attribute
name="SecurityManagerClassName">org.jboss.security.plugins.JaasSec
urityManager</attribute>

  <attribute
name="DefaultUnauthenticatedPrincipal">anonymous</attribute>

  <!-- DefaultCacheTimeout: Specifies the default timed cache
policy timeout
  in seconds.
  If you want to disable caching of security credentials, set this
to 0 to
  force authentication to occur every time. This has no affect if
the
  AuthenticationCacheJndiName has been changed from the default
value.-->
```

```
  <attribute name="DefaultCacheTimeout">1800</attribute>

  <!-- DefaultCacheResolution: Specifies the default timed cache
policy
  resolution in seconds. This controls the interval at which the
cache
  current timestamp is updated and should be less than the
DefaultCacheTimeout
  in order for the timeout to be meaningful. This has no affect if
the
  AuthenticationCacheJndiName has been changed from the default
value.-->

  <attribute name="DefaultCacheResolution">60</attribute>

  <!-- DeepCopySubjectMode: This set the copy mode of subjects
done by the
  security managers to be deep copies that makes copies of the
subject
  principals and credentials if they are cloneable. It should be
set to
  true if subject include mutable content that can be corrupted
when
  multiple threads have the same identity and cache flushes/logout
clearing
  the subject in one thread results in subject references
affecting other
  threads.-->

  <attribute name="DeepCopySubjectMode">false</attribute>

  <attribute
name="CallbackHandlerClassName">org.jboss.security.plugins.
[Custom_Callback_Handler_Name]</attribute>

  </mbean>

...
```

3. **Reinicie o servidor**
   Você acabou de configurar o arquivo **`jboss-service.xml`** para uso com o manuseador de chamada de retorno.

   Reinicie o servidor para garantir de que a nova política de segurança fará efeito.

Procedimento 7.2. Defina o manuseador da chamada de retorno usando a injeção

Este procedimento descreve como injetar uma instância do manuseador da chamada de retorno de segurança no JNDISecurityManagement bean.

1. **Crie uma instância de chamada de retorno personalizada**
   Você deve criar uma instância do manuseador da chamada de retorno personalizada e registrá-la.

2. **Abra o arquivo de configuração**
   Navigue ao **`$JBOSS_HOME/server/$PROFILE/deploy/security/`**

   Abra o arquivo **`security-jboss-beans.xml`**.

   Por padrão, o arquivo **`security-jboss-beans.xml`** contém a configuração do JNDIBasedSecurityManagement bean no Exemplo 7.5, “configuração padrão do security-jboss-beans”

   Exemplo 7.5. configuração padrão do security-jboss-beans

   ```
   <!-- JNDI Based Security Management -->
   <bean name="JBossSecuritySubjectFactory"
   class="org.jboss.security.integration.JBossSecuritySubjectFactory"
   />
   ```

3. **Anexe a propriedade de injeção**
   Para injetar o manuseador da chamada de retorno, anexe um elemento <property> como filho do elemento <mbean> do JNDIBasedSecurityManagement. Especifique o manuseador da chamada de retorno usando os elementos <property> e <inject> descritos no Exemplo 7.4, “chamador da chamada de retorno anexado do jboss-service”.

   Exemplo 7.6. manuseador da chamada de retorno do security-jboss-beans

   ```
   <bean name="JBossSecuritySubjectFactory"
   class="org.jboss.security.integration.JBossSecuritySubjectFactory"
   >
      <property name="securityManagement">
         <inject bean="JNDIBasedSecurityManagement" />
      </property>
   </bean>
   ```

4. **Reinicie o servidor**
   Você acabou de configurar o arquivo **`security-jboss-beans.xml`** para injeção de seu manuseador da chamada de retorno personalizada.

   Reinicie o servidor para garantir de que a nova política de segurança fará efeito.

CAPÍTULO 8. AUTORIZAÇÃO

A autorização relata o tipo de componentes que você deseja proteger, ao invés da camada que reside no mesmo.

O domínio de segurança não solicita claramente uma política de autorização. Caso uma política de autorização não seja especificada, o ***jboss-web-policy*** padrão e a autorização ***jboss-ejb-policy***, configurados no **jboss-as/server/*$PROFILE*/deploy/security/security-policies-jboss-beans.xml**, serão utilizados.

Caso você escolha especificar uma política de autorização, ou criar um arquivo de descritor de implementação padrão com uma política de autorização válida, essas configurações substituem as configurações padrões no **security-policies-jboss-beans.xml**.

Os usuários podem fornecer políticas de autorização que implementam o comportamento padrão. A configuração do comportamento padrão permite que pilhas de controle de autorização a serem conectadas no **jboss.xml** (for EJBs) e **jboss-web.xml** (para WAR).

A substituição da autorização padrão para os componentes da WEB ou EJB é fornecida para o Contrato de Autorização Java para Recipientes - Java Authorization Contract for Containers (JACC) e Linguagem Marcada de Controle de Acesso Extensivo - Extensible Access Control Markup Language (XACML), além dos módulos padrões que implementam o comportamento de especificação.

Refira-se à Seção 6.2, “<authorization>” para maiores informações sobre o esquema do elemento <authorization>.

Procedimento 8.1. Configuração das políticas de autorização para todos os componentes EJB e WAR

Você pode substituir a autorização para todos os componentes EJBs e Web ou para um componente particular.

Este procedimento descreve como definir o controle de autorização JACC para todos os componentes EJB e WAR. A amostra define os módulos de política para os aplicativos EJB e Web: **jboss-web-policy** e **jboss-ejb-policy**.

1. **Abra o bean de política de segurança**
   Navegue ao **$JBOSS_HOME/server/*$PROFILE*/deploy/security**

   Abra o arquivo **security-policies-jboss-beans.xml**.

   Por padrão, o arquivo **security-policies-jboss-beans.xml** contém a configuração no Exemplo 8.1, “security-policies-jboss-beans.xml”.

   Exemplo 8.1. security-policies-jboss-beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">

   <application-policy xmlns="urn:jboss:security-beans:1.0"
name="jboss-web-policy" extends="other">
      <authorization>
         <policy-module
code="org.jboss.security.authorization.modules.DelegatingAuthoriza
```

```
tionModule" flag="required"/>
      </authorization>
   </application-policy>

   <application-policy xmlns="urn:jboss:security-beans:1.0"
name="jboss-ejb-policy" extends="other">
      <authorization>
         <policy-module
code="org.jboss.security.authorization.modules.DelegatingAuthoriza
tionModule" flag="required"/>
      </authorization>
   </application-policy>

</deployment>
```

2. **Altere as definições da política do aplicativo**
   Para configurar uma política de autorização única para cada componente usando o JACC, anexe cada atributo **<policy-module> *code*** com o nome do módulo de autorização JACC.

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns="urn:jboss:bean-deployer:2.0">

   <application-policy xmlns="urn:jboss:security-beans:1.0"
name="jboss-web-policy" extends="other">
      <authorization>
         <policy-module
code="org.jboss.security.authorization.modules.JACCAuthorizationModu
le" flag="required"/>
      </authorization>
   </application-policy>

   <application-policy xmlns="urn:jboss:security-beans:1.0"
name="jboss-ejb-policy" extends="other">
      <authorization>
         <policy-module
code="org.jboss.security.authorization.modules.JACCAuthorizationModu
le" flag="required"/>
      </authorization>
   </application-policy>

   <application-policy xmlns="urn:jboss:security-beans:1.0"
name="jacc-test" extends="other">
      <authorization>
         <policy-module
code="org.jboss.security.authorization.modules.JACCAuthorizationModu
le" flag="required"/>
      </authorization>
   </application-policy>

</deployment>
```

3. **Reinicie o servidor**
   Você configurou o arquivo **security-policy-jboss-beans.xml** com a autorização JACC ativada para cada política do aplicativo.

Reinicie o servidor para garantir que a nova política de segurança tenha efeito.

Configuração de autorização para componentes EJB e WEB específicos

Caso aplicativos solicitem políticas de segurança granular, você pode declarar políticas de segurança de autorização múltiplas para cada política de aplicativo. Os novos domínios de segurança podem herdar configurações base a partir de outros domínios de segurança e substituir configurações específicas tais como o módulo de política de autorização.

Procedimento 8.2. Configuração das políticas de autorização para domínios de segurança específicos

Você pode substituir a autorização para um componente particular.

Este procedimento descreve como herdar as configurações de outras definições de domínio de segurança e especificar diferentes políticas de autorização por domínio de segurança.

Neste procedimento, os dois domínios de segurança são definidos. O domínio de segurança ***test-domain*** usa o módulo de logon **UsersRolesLoginModule** e usa a autorização JACC. O domínio de segurança ***test-domain-inherited*** herda a informação do módulo de logon a partir do ***test-domain***, além de especificar que a autorização XACML deve ser usada.

1. **Abra a política de segurança**
   Você pode especificar as configurações do domínio de segurança no arquivo **jboss-as/server/$PROFILE/conf/login-config.xml** ou criar um arquivo descritor de implantação contendo as configurações. Escolha o descritor de implantação caso deseje empacotar as configurações do domínio de segurança com seu aplicativo.

   - **Localize e abra o login-config.xml**
     Navegue ao arquivo **login-config.xml** do perfil do servidor sendo utilizado e abra o arquivo para edição.

     $JBOSS_HOME/jboss-as/server/*$PROFILE*/conf/login-config.xml

   - **Crie o descritor jboss-beans.xml**
     Crie o descritor ***[prefix]*-jboss-beans.xml** substituindo *[prefix]* por um nome significativo (por exemplo: **test-war-jboss-beans.xml**)

     Salve este arquivo no diretório **/deploy** do perfil do servidor que você está configurando.

     jboss-as/server/*$PROFILE*/deploy/*[prefix]*-jboss-beans.xml

2. **Especifique o domínio de segurança do teste-domínio**
   No arquivo de destino escolhido no passo 1, especifique o domínio de segurança ***test-domain***. Este domínio contém a informação de autenticação, incluindo a definição<login-module> e a definição do módulo de política de autorização JACC.

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">

   <application-policy xmlns="urn:jboss:security-beans:1.0"
name="test-domain">
     <authentication>
        <login-module code =
"org.jboss.security.auth.spi.UsersRolesLoginModule"
```

```
            flag = "required">
            <module-option name =
"unauthenticatedIdentity">anonymous</module-option>
            <module-option
name="usersProperties">u.properties</module-option>
            <module-option
name="rolesProperties">r.properties</module-option>
         </login-module>
      </authentication>
      <authorization>
         <policy-module
code="org.jboss.security.authorization.modules.JACCAuthorizationModu
le" flag="required"/>
      </authorization>
   </application-policy>


</deployment>
```

3. **Anexe o domínio de segurança teste-domínio-herdado**
   Anexe a definição da política do aplicativo ***`test-domain-inherited`*** após a política do aplicativo ***`test-domain`***.

   Configure o atributo ***`extends`*** para **`other`**, de forma que a informação do módulo de logon é herdada.

   Especifique o módulo de autorização XACML no elemento **`<policy-module>`**.

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">

   <application-policy xmlns="urn:jboss:security-beans:1.0"
name="test-domain">
     <authentication>
        <login-module code =
"org.jboss.security.auth.spi.UsersRolesLoginModule"
            flag = "required">
            <module-option name =
"unauthenticatedIdentity">anonymous</module-option>
            <module-option
name="usersProperties">u.properties</module-option>
            <module-option
name="rolesProperties">r.properties</module-option>
         </login-module>
      </authentication>
      <authorization>
         <policy-module
code="org.jboss.security.authorization.modules.JACCAuthorizationModu
le" flag="required"/>
      </authorization>
   </application-policy>

   <application-policy xmlns="urn:jboss:security-beans:1.0"
name="test-domain-inherited" extends="other">
      <authorization>
```

```
        <policy-module
code="org.jboss.security.authorization.modules.XACMLAuthorizationMod
ule" flag="required"/>
      </authorization>
   </application-policy>

</deployment>
```

4. **Reinicie o servidor**
   Você precisa configurar o arquivo de destino com os domínios de segurança que usam diferentes métodos de autorização.

   Reinicie o servidor para garantir que a nova política de segurança tenha efeito.

8.1. DELEGAÇÃO DE MÓDULO

O Procedimento 8.1, “Configuração das políticas de autorização para todos os componentes EJB e WAR” e Procedimento 8.2, “Configuração das políticas de autorização para domínios de segurança específicos” descrevem as amostras mais simples que apresentam como a autenticação básica pode ser configurada nos domínios de segurança.

Você pode usar a delegação do módulo de autorização com um descritor de implantação (**`*-jboss-beans.xml`**) para especificar as políticas de autorização à autenticação padrão de sua implementação, uma vez que a autorização relata o tipo de componente (não a camada) que você protege.

A classe **`org.jboss.security.authorization.modules.AuthorizationModuleDelegate`** fornece um número de sub-classes que permitem você implementar a delegação do módulo:

- **`AbstractJACCModuleDelegate`**
- **`WebPolicyModuleDelegate`**
- **`EJBPolicyModuleDelegate`**
- **`WebXACMLPolicyModuleDelegate`**
- **`WebJACCPolicyModuleDelegate`**
- **`EJBXACMLPolicyModuleDelegate`**
- **`EJBJACCPolicyModuleDelegate`**

Você pode criar seu próprio módulo de delegação, contanto que módulo estenda a classe **`org.jboss.security.authorization.modules.AuthorizationModuleDelegate`**.

Você pode declarar os módulos de delegação com o elemento <module-option> de sua política <authorization> para implementar o módulo de delegação. Cada módulo possui prefixo do componente que ele é relacionado, conforme apresentado no Exemplo 8.2, “Declaração do Módulo de Delegação”.

Exemplo 8.2. Declaração do Módulo de Delegação

```
<application-policy xmlns="urn:jboss:security-beans:1.0" name="test-
domain" extends="other">
<authorization>
<policy-module code="xxx.yyy.MyAuthorizationModule" flag="required">
```

```
<module-option
name="delegateMap">web=xxx.yyy.mywebauthorizationdelegate,ejb=xxx.yyy.my
ejbauthorizationdelegate</module-option>
</policy-module>
</authorization>
</application-policy>
```

CAPÍTULO 9. MAPEAMENTO

É possível mapear funções ao nível de implementação a partir de derivações à nível do domínio de segurança (tal como o nível EAR), na Plataforma do Aplicativo JBoss Enterprise 5.

Isto é atingido a partir da derivação da classe **`org.jboss.security.mapping.providers.DeploymentRolesMappingProvider`** conforme valor para o atributo ***`code`*** no elemento <mapping-module>. Além disso, o atributo ***`type`*** deve ser configurado para **`role`**. Refira-se à Seção 6.3, “<mapping>” para maiores informações sobre o esquema do elemento <mapping>.

Você pode forçar a interpretação da função adicional para declarar princípios especificados para uma implantação em particular (war, ear, ejb-jar etc), pela configuração do elemento de configuração de mapeamento com o parâmetro baseado na função.

IMPORTANTE

O elemento <rolemapping> contia o elemento <mapping-module> e a declaração da classe nas versões anteriores às da Plataforma do Aplicativo JBoss Enterprise 5. O <rolemapping> foi preterido e substituído pelo elemento <mapping>.

Exemplo 9.1. Declaração <mapping-module>

```
<application-policy name="test-domain">
<authentication>
...
</authentication>
<mapping>
<mapping-module
code="org.jboss.security.mapping.providers.DeploymentRolesMappingProvide
r" type="role"/>
</mapping>
...
</application-policy>
```

Uma vez que o domínio de segurança é configurado corretamente, você pode anexar o grupo do elemento <security-role> como um elemento filho do <assembly-descriptor> para o arquivo **`WEB-INF/jboss-web.xml`** (**`.war`** ou **`.sar`**).

Exemplo 9.2. Declaração <security-role>

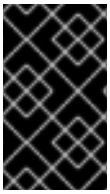
```
<assembly-descriptor>
   ...
   <security-role>
      <role-name>Support</role-name>
      <principal-name>Mark</principal-name>
      <principal-name>Tom</principal-name>
   </security-role>
   ...
</assembly-descriptor>
```

Uma função de segurança relacionada aos princípios de Suporte é implementada juntamente à informação de função de segurança base contida no **`WEB-INF/jboss-web.xml`**.

CAPÍTULO 10. AUDITORIA

Certas organizações governamentais autorizam auditoria em aplicativos empresariais para garantir que os componentes de uma implantação sejam rastreáveis e operem com seus parâmetros de design. Além disso, as regulamentações e padrões governamentais requerem controles de auditoria juntamente com a auditoria do aplicativo padrão.

Os administradores de sistema podem ativar a auditoria do evento de segurança para monitorar constantemente a operação do domínio de segurança e aplicativos EJB e WEB implantados.



IMPORTANTE

A auditoria do evento de segurança pode introduzir um impacto de desempenho nos servidores que gerenciam os grandes volumes de evento. A auditoria é desativada por padrão e pode ser configurada para ser disponibilizada por demanda.

A ativação da auditoria do evento de segurança difere-se entre os componentes EJB e Web. O Procedimento 10.1, “Ativação do recurso de auditoria de segurança” descreve os passos mínimos para ativação do serviço de auditoria para EJBs em sua implantação. O Procedimento 10.2, “Ativação da auditoria de segurança para recipientes da Web” descreve como ativar a auditoria de evento de segurança para recipientes da Web.

IMPORTANTE

A auditoria do evento do recipiente da web pode expor a informação confidencial do usuário. Os administradores devem certificar-se que os procedimentos de proteção dos dados apropriados, tais como senha hash, são implantados quando configurando a auditoria de segurança para eventos de recipiente da Web.

Procedimento 10.1. Ativação do recurso de auditoria de segurança

1. **Abra o arquivo de configuração log4j**
   Navegue ao **`$JBOSS_HOME/server/$PROFILE/conf/`**

   Abra o arquivo **`jboss-log4j.xml`** usando um editor de texto.

2. **Descomente a categoria de auditoria de segurança**
   Por padrão, a definição da categoria do Provedor de Auditoria de Segurança é comentada no arquivo **`jboss-log4j.xml`**. Descomente a definição de categoria apresentada no Exemplo 10.1, “Categoria do Provedor de Auditoria de Segurança log4j”.

   Exemplo 10.1. Categoria do Provedor de Auditoria de Segurança log4j

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Limit the verbose MC4J EMS (lib used by admin-console)
categories -->
<category name="org.mc4j.ems">
  <priority value="WARN"/>
</category>

<!-- Show the evolution of the DataSource pool in the logs
[inUse/Available/Max]
<category
```

```
name="org.jboss.resource.connectionmanager.JBossManagedConnectionP
ool">
  <priority value="TRACE"/>
</category>
   -->

<!-- Category specifically for Security Audit Provider -->
<category
name="org.jboss.security.audit.providers.LogAuditProvider"
additivity="false">
<priority value="TRACE"/>
  <appender-ref ref="AUDIT"/>
</category>


<!-- Limit the org.jboss.serial (jboss-serialization) to INFO as
its DEBUG is verbose -->
<category name="org.jboss.serial">
  <priority value="INFO"/>
</category>
```

3. **Descomente o anexo de auditoria**
   Por padrão, a definição AUDITORIA é comentada no arquivo **`jboss-log4j.xml`**. Descomente a definição do anexo apresentada no Exemplo 10.1, “Categoria do Provedor de Auditoria de Segurança log4j”.

Exemplo 10.2. Categoria do Provedor de Auditoria de Segurança log4j

```
...
<!--  Emit events as JMX notifications
<appender name="JMX"
class="org.jboss.monitor.services.JMXNotificationAppender">
  <errorHandler
class="org.jboss.logging.util.OnlyOnceErrorHandler"/>
  <param name="Threshold" value="WARN"/>
  <param name="ObjectName"
value="jboss.system:service=Logging,type=JMXNotificationAppender"/
>
  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="%d %-5p [%c] %m"/>
  </layout>
</appender>
-->

<!-- Security AUDIT Appender -->
<appender name="AUDIT"
class="org.jboss.logging.appender.DailyRollingFileAppender">
  <errorHandler
class="org.jboss.logging.util.OnlyOnceErrorHandler"/>
  <param name="File" value="${jboss.server.log.dir}/audit.log"/>
  <param name="Append" value="true"/>
  <param name="DatePattern" value="'.'yyyy-MM-dd"/>
  <layout class="org.apache.log4j.PatternLayout">
```

```
    <param name="ConversionPattern" value="%d %-5p [%c] (%t:%x)
%m%n"/>
  </layout>
</appender>

   <!-- ================ -->
   <!-- Limit categories -->
   <!-- ================ -->

<!-- Limit the org.apache category to INFO as its DEBUG is verbose
-->
<category name="org.apache">
  <priority value="INFO"/>
</category>
...
```

4. **Salve e reinicie o servidor**
   Você ativou o serviço de auditoria para sua implantação, conforme configurado no arquivo **`jboss-log4j.xml`**.

   Reinicie o servidor para garantir que a nova política de segurança seja aplicada.

5. **Verifique se a auditoria de segurança está operando corretamente**
   Uma vez que o serviço de auditoria seja configurado e implantado, as entradas de log de auditoria verificarão se o serviço de auditoria e a invocação EJB foram procedidos com sucesso.

   O arquivo **`audit.log`** está localizado no diretório **`jboss-as/server/$PROFILE/log/`**.

   Uma invocação EJB processada com êxito parece-se com o seguinte resultado **`audit.log`**.

   Exemplo 10.3. Entrada de log da Invocação EJB com êxito

```
2008-12-05 16:08:26,719 TRACE
[org.jboss.security.audit.providers.LogAuditProvider] (http-
127.0.0.1-8080-2:)
[Success]policyRegistration=org.jboss.security.plugins.JBossPolicy
Registration@76ed4518; Resource:=
[org.jboss.security.authorization.resources.EJBResource:contextMap
=
{policyRegistration=org.jboss.security.plugins.JBossPolicyRegistra
tion@76ed4518}:method=public abstract
org.jboss.test.security.interfaces.RunAsServiceRemote
org.jboss.test.security.interfaces.RunAsServiceRemoteHome.create()
throws
java.rmi.RemoteException,javax.ejb.CreateException:ejbMethodInterf
ace=Home:ejbName=RunAs:ejbPrincipal=jduke:MethodRoles=Roles(identi
tySubstitutionCaller,):securityRoleReferences=null:callerSubject=S
ubject:
        Principal: [roles=[identitySubstitutionCaller,
extraRunAsRole],principal=runAsUser]
        Principal:
Roles(members:extraRunAsRole,identitySubstitutionCaller)
:callerRunAs=[roles=[identitySubstitutionCaller,
extraRunAsRole],principal=runAsUser]:callerRunAs=[roles=
```

```
[identitySubstitutionCaller,
extraRunAsRole],principal=runAsUser]:ejbRestrictionEnforcement=fal
se:ejbVersion=null];Source=org.jboss.security.plugins.javaee.EJBAu
thorizationHelper;Exception:=;
```

Uma invocação EJB processada sem êxito parece-se ao seguinte resultado **audit.log**.

Exemplo 10.4. Entrada de log da Invocação EJB processada sem êxito

```
[Error]policyRegistration=org.jboss.security.plugins.JBossPolicyRe
gistration@76ed4518;Resource:=
[org.jboss.security.authorization.resources.EJBResource:contextMap
=
{policyRegistration=org.jboss.security.plugins.JBossPolicyRegistra
tion@76ed4518}:method=public java.security.Principal
org.jboss.test.security.ejb3.SimpleStatelessSessionBean.invokeUnav
ailableMethod():ejbMethodInterface=Remote:ejbName=SimpleStatelessS
essionBean:ejbPrincipal=UserA:MethodRoles=Roles(<NOBODY>,):securit
yRoleReferences=null:callerSubject=Subject:
        Principal: UserA
        Principal: Roles(members:RegularUser,Administrator)
:callerRunAs=null:callerRunAs=null:ejbRestrictionEnforcement=false
:ejbVersion=null];Source=org.jboss.security.plugins.javaee.EJBAuth
orizationHelper;Exception:=Authorization Failed: ;
```

Procedimento 10.2. Ativação da auditoria de segurança para recipientes da Web

1. **Ative a auditoria da segurança EJB**
   Você deve ativar a segurança descrita no Procedimento 10.1, “Ativação do recurso de auditoria de segurança”.

2. **Ative a auditoria no realm do servidor**
   A auditoria do recipiente deve ser primeiramente ativada no realm do servidor do arquivo **server.xml**.

   O arquivo **server.xml** está localizado no diretório ***jboss-as*/server/*$PROFILE*/deploy/jbossweb.sar/**.

   O elemento **<Realm>** deve possuir o atributo ***enableAudit="true"*** configurado, conforme o Exemplo 10.5, “server.xml audit activation”.

   Exemplo 10.5. server.xml audit activation

   ```
   <Realm className="org.jboss.web.tomcat.security.JBossWebRealm"
   certificatePrincipal="org.jboss.security.auth.certs.SubjectDNMappi
   ng" allRolesMode="authOnly"
   enableAudit="true"/>
   ```

3. **Especifique a propriedade do sistema dos níveis de auditoria**

Os níveis de auditoria para aplicativos da Web devem ser especificados usando a propriedade do sistema org.jboss.security.web.audit no script **`run.bat`** (Microsoft Windows) ou **`run.sh`** (Linux).

Alternativamente, você pode especificar a propriedade de sistema no arquivo **`jboss-as/server/$PROFILE/deploy/properties-service.xml`**.

- **Linux**
  Adicione a propriedade de sistema no arquivo **`jboss-as/bin/run.sh`**

```
## Specify the Security Audit options
#System Property setting to configure the web audit:
#* off = turn it off
#* headers = audit the headers
#* cookies = audit the cookie
#* parameters = audit the parameters
#* attributes = audit the attributes
#* headers,cookies,parameters = audit the headers,cookie and
#                               parameters
#* headers,cookies = audit the headers and cookies
JAVA_OPTS="$JAVA_OPTS -
Dorg.jboss.security.web.audit=headers,cookies,parameter"
```

- **Microsoft Windows**
  Adicione a propriedade de sistema ao arquivo **`jboss-as/bin/run.bat`**

```
rem Specify the Security Audit options
rem System Property setting to configure the web audit
rem * off = turn it off
rem * headers = audit the headers
rem * cookies = audit the cookie
rem * parameters = audit the parameters
rem * attributes = audit the attributes
rem * headers,cookies,parameters = audit the headers,cookie and
rem   parameters
rem * headers,cookies = audit the headers and cookies
set JAVA_OPTS=%JAVA_OPTS% " -
Dorg.jboss.security.web.audit=headers,cookies,parameter"
```

- **properties-service.xml**
  Atualize o MBean de classe **`SystemPropertiesService`** no arquivo **`jboss-as/server/$PROFILE/deploy/properties-service.xml`** e declare a propriedade java como um <attribute>. Você pode descomentar o bloqueio do sistema de operação relevante da amostra do código abaixo.

```
...

<mbean code="org.jboss.varia.property.SystemPropertiesService"
name="jboss:type=Service,name=SystemProperties">

   <!-- Linux Attribute Declaration -->
   <!--
   <attribute name="Properties">JAVA_OPTS="$JAVA_OPTS -
Dorg.jboss.security.web.audit=headers,cookies,parameter"
```

```
   </attribute>
   -->

   <!-- Windows Attribute Declaration -->
   <!--
   <attribute name="Properties">JAVA_OPTS=%JAVA_OPTS% " -
Dorg.jboss.security.web.audit=headers,cookies,parameter"
   </attribute>
   -->

</mbean>

...
```

4. **Verifique se a auditoria de segurança está operando corretamente**
   Uma propriedade do sistema é especificada no arquivo, as entradas de log da auditoria verificarão se a invocação da Web foi procedida com êxito.

   O arquivo **audit.log** está localizado no diretório **jboss-as/server/*$PROFILE*/log/**.

   Uma invocação da Web procedida com êxito parece-se com o seguinte resultado **audit.log**.

   Exemplo 10.6. Entrada do log da Invocação da Web processada com êxito

```
2008-12-05 16:08:38,997 TRACE
[org.jboss.security.audit.providers.LogAuditProvider] (http-
127.0.0.1-8080-17:)
[Success]policyRegistration=org.jboss.security.plugins.JBossPolicy
Registration@76ed4518;Resource:=
[org.jboss.security.authorization.resources.WebResource:contextMap
=
{policyRegistration=org.jboss.security.plugins.JBossPolicyRegistra
tion@76ed4518,securityConstraints=
[Lorg.apache.catalina.deploy.SecurityConstraint;@6feeae6,
resourcePermissionCheck=true},canonicalRequestURI=/restricted/get-
only/x,request=[/web-constraints:cookies=null:headers=user-
agent=Jakarta Commons-
HttpClient/3.0,authorization=host=localhost:8080,]
[parameters=],CodeSource=null];securityConstraints=SecurityConstra
int[RestrictedAccess - Get
Only];Source=org.jboss.security.plugins.javaee.WebAuthorizationHel
per;resourcePermissionCheck=true;Exception:=;
```

   Uma invocação EJB processada sem êxito parece-se ao seguinte resultado **audit.log**.

   Exemplo 10.7. Entrada do log da Invocação da Web processada sem êxito

```
2008-12-05 16:08:41,561 TRACE
[org.jboss.security.audit.providers.LogAuditProvider] (http-
127.0.0.1-8080-4:)
[Failure]principal=anil;Source=org.jboss.web.tomcat.security.JBoss
WebRealm;request=[/jaspi-web-basic:cookies=null:headers=user-
agent=Jakarta Commons-
HttpClient/3.0,authorization=host=localhost:8080,][parameters=]
```

```
[attributes=];2008-12-05 16:07:30,129 TRACE
[org.jboss.security.audit.providers.LogAuditProvider]
(WorkerThread#1[127.0.0.1:55055]:)
```

CAPÍTULO 11. IMPLANTAÇÃO DOS DOMÍNIOS DE SEGURANÇA

Domínio de Segurança Modular

O método de implantação do domínio de segurança, onde a declaração do domínio de segurança está inclusa no *deployment descriptor*. Um domínio de segurança modular possui a forma do **`*-jboss-beans.xml`**. Ele está incluído no diretório **`META-INF`** do EJB Jars ou no diretório **`WEB-INF`** do aplicativo da web (WAR).

Descritor de Implementação

Um arquivo de configuração XML declarativa que descreve as configurações da implementação de um aplicativo. A maneira em que um aplicativo é implantado pode ser alterado com este arquivo, eliminando a necessidade de realizar alterações ao código adjacente do aplicativo.

Existem duas maneiras de implantar um domínio de segurança na Plataforma do Aplicativo JBoss Enterprise:

1. Declare o domínio de segurança no arquivo **`jboss-as/server/`*`$PROFILE`*`/conf/login-config.xml`**.

2. Crie e implemente o domínio de segurança modular.

Procedimento 11.1. Configuração do Domínio de Segurança Modular

Siga este procedimento para configurar um descritor de implementação do domínio de segurança modular com dois domínios para o EJB e os aplicativos da web.

Cada domínio usa o **`UsersRolesLoginModule`** para a política de autorização, no entanto você não está limitado a este módulo de logon quando criando um domínio de segurança modular. Refira-se à Seção 12.1, “Módulos de Uso” para módulos de logon adicionais lançados com a Plataforma do Aplicativo JBoss Enterprise.

1. **Crie um descritor de implantação**
   Você deve criar um arquivo do descritor de implantação para conter uma configuração do domínio de segurança.

   Caso você já tenha criado um descritor de implantação para seu aplicativo, você pode pular este procedimento e proceder ao passo 2.

   Este filename possui o formulário ***`[domain_name]`*`-jboss-beans.xml`**. Enquanto o *domain_name* é arbitrário, você deve escolher um nome que é significativo ao aplicativo para garantir o nome do descritor de implantação seja único por todo o perfil do servidor.

   O arquivo deve conter a declaração XML padrão e um elemento **`<deployment>`** corretamente configurado.

   ```
   <?xml version="1.0" encoding="UTF-8"?>

   <deployment xmlns="urn:jboss:bean-deployer:2.0">


   </deployment>
   ```

2. **Defina políticas de aplicativo**
   Os domínios de segurança individuais são definidos com o elemento <deployment>.

   Segue abaixo dois domínios de segurança especificados. Cada política de autenticação usa o mesmo módulo de login e parâmetros de módulo.

   NOTA

   Outros módulos de logon estão disponíveis para uso com a Plataforma do Aplicativo Enterprise. Para maiores informações sobre os módulos de logon disponíveis, refira-se à Seção 12.1, “Módulos de Uso”.

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <application-policy xmlns="urn:jboss:security-beans:1.0"
name="web-test">
    <authentication>
      <login-module
code="org.jboss.security.auth.spi.UsersRolesLoginModule"
flag="required">
        <module-option
name="unauthenticatedIdentity">anonymous</module-option>
        <module-option name="usersProperties">u.properties</module-
option>
        <module-option name="rolesProperties">r.properties</module-
option>
      </login-module>
    </authentication>
  </application-policy>

  <application-policy xmlns="urn:jboss:security-beans:1.0"
name="ejb-test">
    <authentication>
      <login-module
code="org.jboss.security.auth.spi.UsersRolesLoginModule"
flag="required">
        <module-option
name="unauthenticatedIdentity">anonymous</module-option>
        <module-option name="usersProperties">u.properties</module-
option>
        <module-option name="rolesProperties">r.properties</module-
option>
      </login-module>
    </authentication>
  </application-policy>

</deployment>
```

3. **Implemente ou empacote o descritor de implantação**
   Mova o arquivo do descritor de implantação para o diretório **`jboss-as/server/$PROFILE/deploy`** do perfil do servidor solicitado em sua instalação.

Caso você estiver distribuindo seu aplicativo a uma audiência mais ampla, empacote o descritor de implantação no diretório **`META-INF`** do EJB Jar ou no diretório **`WEB-INF`** de seu aplicativo da web (WAR).

CAPÍTULO 12. MÓDULOS DE LOGON

12.1. MÓDULOS DE USO

A Plataforma do Aplicativo JBoss Enterprise inclui diversos módulos úteis de logon agrupados à maioria das necessidades de gerenciamento do usuário. A Plataforma do Aplicativo JBoss Enterprise pode ler informação do usuário a partir de banco de dados relacionais, um servidor do Protocolo de Acesso do Diretório de Carga Leve - Lightweight Directory Access Protocol (LDAP) ou arquivos simples. Adicionados aos módulos de logon principais, o JBoss fornece outros módulos de logon que fornecem a informação do usuário para as mais personalizadas necessidades no JBoss.

12.1.1. LdapLoginModule

O **`LdapLoginModule`** é uma implantação **`LoginModule`** que autentica o servidor do Protocolo de Acesso ao Diretório de Peso Leve (LDAP - Lightweight Directory Access Protocol), caso seu nome de usuário e credenciais estejam armazenados num servidor LDAP que é acessado usando um provedor LDAP da Interface do Diretório e Nomeação do Java (JNDI).

NOTA

Este módulo de logon suporta também a identidade não-autenticada e o empilhamento da senha.

A informação de conectividade do LDAP é fornecida como opções de configuração que são passadas através do objeto do ambiente usado para criar o contexto inicial do JNDI. As propriedades JNDI do LDAP padrão usadas incluem o seguinte:

java.naming.factory.initial

O nome da classe de implantação **`InitialContextFactory`**. O padrão é a implantação do provedor LDAP do Sun **`com.sun.jndi.ldap.LdapCtxFactory`**.

java.naming.provider.url

O URL do LDAP para o servidor LDAP.

java.naming.security.authentication

O nível do protocolo para uso. Os valores disponíveis incluem o **none**, **`simple`** e **`strong`**. Caso a propriedade seja indefinida, o comportamento é determinado pelo provedor do serviço.

java.naming.security.protocol

O protocolo de transporte para uso de acesso de segurança. Configure a opção de configuração ao tipo do provedor do serviço (por exemplo: o SSL). Caso a propriedade seja indefinida, o comportamento será determinado pelo provedor do serviço.

java.naming.security.principal

Especifica a identidade do Principal para autenticação do chamador ao serviço. Ele é construído a partir de outras propriedades conforme descrito abaixo.

java.naming.security.credentials

Especifica os credenciais do Principal para autenticação do chamador ao serviço. Os credenciais podem levar o modelo de uma senha com hash, uma senha de texto simples, uma chave ou um

certificado. Caso a propriedade for indefinida, o comportamento é determinado pelo provedor do serviço.

As opções da configuração do módulo do logon suportado incluem o seguinte:

principalDNPrefix

O prefixo adicionado ao nome do usuário para formar o *distinguished name* do usuário. Consulte **`principalDNSuffix`** para maiores informações.

principalDNSuffix

O sufixo adicionado ao nome do usuário quando formando o nome distinguido do usuário. Ele é útil caso você solicite pelo nome do usuário e não queira que o usuário insira o todo o nome distinguido. O uso desta propriedade e **`principalDNSuffix`** leva o **`userDN`** a ser formado como **`principalDNPrefix + username + principalDNSuffix`**.

useObjectCredential

O valor que indica o credencial deve ser obtido como um **`Object`** opaco usando o tipo do **`org.jboss.security.auth.callback.ObjectCallback`** do **`Callback`** ao invés de uma senha **`char[]`** usando um **`PasswordCallback`** do JAAS. Isto permite a passagem da informação sem-credencial para o servidor LDAP. Os valores disponíveis são **`true`** e **`false`**.

rolesCtxDN

Ajustado, o nome distinguido do contexto para busca das funções do usuário.

userRolesCtxDNAttributeName

Nome de um atributo no objeto do usuário que contém o nome distinguido do contexto para busca das funções do usuário. Isto difere-se do **`rolesCtxDN`**, de forma que o contexto de busca para funções do usuário pode ser único para cada usuário.

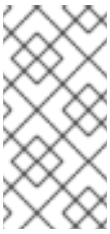
roleAttributeID

Nome do atributo contendo as funções do usuário. O padrão é **`roles`**, caso não seja especificado.

roleAttributeIsDN

Aviso indicando se é que o roleAttributeID contém todo o nome distinguido de um objeto de função, ou o nome de função. O nome de função é obtido do valor do atributo roleNameAttributeId do nome do contexto pelo nome distinguido.

Caso verdadeiro, o atributo da função representa o nome distinguido de um objeto de função. Caso seja falso, o nome de função é obtido a partir do valor de um **`roleAttributeID`**. O padrão é **`false`**.



NOTA

Em certos esquemas de diretório (ex.: MS ActiveDirectory), os atributos de função no objeto do usuário são armazenados como Nomes Distinguidos para objetos de função ao invés de nomes simples. O roleAttributeIsDN deve ser configurado para **`true`**, para implantações que usam este tipo de esquema.

roleNameAttributeID

O nome do atributo de um contexto direcionado pelo valor do nome distinguido roleCtxDN que contém o nome de função. Caso a propriedade roleAttributeIsDN seja configurada para **`true`**, esta propriedade será usada para encontrar o atributo **`name`** do objeto de função. O padrão é **`group`**.

uidAttributeID

Nome do atributo no objeto contento as funções do usuário que correspondem à ID do usuário. Isto é usado para localizar as funções do usuário. O padrão é **`uid`**, caso não seja especificado.

matchOnUserDN

Aviso que especifica se é que a busca por funções do usuário devem coincidir com todo o nome distinguido do usuário. Caso **`true`**, o **`userDN`** completo será usado como valor de combinação. Caso **`false`**, apenas o nome do usuário será usado como valor de combinação em relação ao atributo uidAttributeName. O valor padrão é **`false`**.

unauthenticatedIdentity

O nome principal a ser determinado sem conter qualquer informação de autenticação. Este comportamento é herdado a partir da super-classe **`UsernamePasswordLoginModule`**.

allowEmptyPasswords

Um aviso indicando se as senhas nulas (comprimento 0) devem ser passadas ao servidor LDAP. Uma senha vazia é tratada como um logon anônimo por alguns servidores LDAP e isto pode não ser um recurso desejado. Para rejeição de senhas nulas, configure isto para **`false`**. Caso configurado para **`true`**, o servidor LDAP validará a senha nula. O padrão é **`true`**.

A autenticação do usuário é executada pela conexão ao servidor LDAP, baseado nas opções de configuração do módulo de logon. A conexão ao servidor LDAP é realizada pela criação de um **`InitialLdapContext`** com um ambiente composto de propriedades JNDI do LDAP descritas anteriormente nesta seção.

O Context.SECURITY_PRINCIPAL é configurado ao nome distinguido do usuário obtido pelo manuseador da chamada de retorno em combinação com os valores de opção principalDNPrefix e principalDNSuffix. Além disso, a propriedade Context.SECURITY_CREDENTIALS é tanto configurada à senha **`String`** ou o credencial **`Object`** dependendo da opção useObjectCredential.

Uma vez que a autenticação tenha sido bem sucedida (a instância **`InitialLdapContext`** é criada), as funções do usuário são solicitadas pela execução de uma busca da localização **`rolesCtxDN`** com os atributos determinados para os valores de opção roleAttributeName e uidAttributeName. Os nomes de função são obtidos pela invocação do método **`toString`** nos atributos de função do conjunto do resultado de busca.

Exemplo 12.1. Política da Autenticação do Módulo do Logon LDAP

Esta política de autenticação descreve como usar os parâmetros na política de autenticação do domínio de segurança.

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

```
            <module-option name="java.naming.provider.url">
ldap://ldaphost.jboss.org:1389/
            </module-option>
            <module-option name="java.naming.security.authentication">
simple
            </module-option>
            <module-option name="principalDNPrefix">uid=</module-option>
<module-option name="principalDNSuffix">
,ou=People,dc=jboss,dc=org
            </module-option>
            <module-option name="rolesCtxDN">
ou=Roles,dc=jboss,dc=org
            </module-option>
            <module-option name="uidAttributeID">member</module-option>
            <module-option name="matchOnUserDN">true</module-option>
            <module-option name="roleAttributeID">cn</module-option>
            <module-option name="roleAttributeIsDN">false </module-option>
         </login-module>
      </authentication>
</application-policy>
```

As opções ***java.naming.factory.initial***, ***java.naming.factory.url*** e ***java.naming.security*** da configuração testLDAP <login-module> indicam as condições:

- A implantação do provedor JNDI do LDAP Sun será usada.
- O servidor LDAP está localizado no **ldaphost.jboss.org** do host na porta 1389.
- O método de autenticação simples do LDAP será usado para conexão ao servidor LDAP.

O módulo do logon tenta conectar-se ao servidor LDAP usando um Nome Distinguido (DN) representando o usuário pelo qual está tentando autenticar. Este DN é construído a partir do principalDNPrefix passado, o nome do usuário e o principalDNSuffix, conforme descritos acima. No Exemplo 12.2, “Amostra do Arquivo LDIF”, o nome do usuário **jsmith** mapearia para **uid=jsmith,ou=People,dc=jboss,dc=org**.

Nome Distinguido (DN)

Num Protocolo de Acesso ao Diretório de Carga Leve (LDAP), o nome distinguido identifica unicamente um objeto num diretório. Cada nome distinguido deve possuir um nome e localização únicos a partir de todos os objetos, que é atingido usando um número de pares de valor do atributo (AVPs). Os AVPs definem informações tais como nomes comuns, unidade de organização, entre outras coisas. A combinação desses valores resulta numa sequência única solicitada pelo LDAP.

NOTA

A amostra assume que o servidor LDAP autentica os usuários usando o atributo **userPassword** da entrada do usuário (**theduke** nesta amostra). A maioria dos servidores LDAP operam desta maneira. No entanto, caso o servidor LDAP manuseia autenticações de forma diferente, você deverá certificar-se que o LDAP é configurado de acordo com suas solicitações do ambiente de produção.

Uma vez que a autenticação é bem sucedida, as funções pelas quais a autorização será baseada são

restauradas pela execução da busca da sub-árvore do **rolesCtxDN** para entradas cujos uidAttributeID coincidem com o usuário. Caso o matchOnUserDN seja verdadeiro, a busca será baseada no DN completo do usuário. Do contrário, a busca será baseada no nome do usuário atual inserido. Nesta amostra, a busca está sob o **ou=Roles,dc=jboss,dc=org** para quaisquer entradas que possuem o atributo **member** ao **uid=jduke,ou=People,dc=jboss,dc=org**. A busca localizaria o **cn=JBossAdmin** sob a entrada das funções.

A busca retorna o atributo especificado na opção roleAttributeID. Nesta amostra, o atributo é o **cn**. O valor retornado seria o **JBossAdmin**, de forma que o usuário **jsmith** é determinado à função do **JBossAdmin**.

O servidor LDAP local normalmente fornece serviços de identidade e localização, mas não está apto a usar os serviços de autorização. Isto é devido às funções do aplicativo nem sempre mapearem bem os grupos LDAP e os administradores LDAP normalmente hesitarem em permitir os dados específicos do aplicativo externo nos servidores LDAP centrais. O módulo de autenticação LDAP é normalmente emparelhado com outro módulo de logon, que pode fornecer funções mais úteis ao aplicativo sendo desenvolvido.

O arquivo do Formato de Inter-alteração dos Dados LDAP (LDIF) representando a estrutura do diretório que esses dados operam, está apresentado no Exemplo 12.2, “Amostra do Arquivo LDIF”.

Formato de Inter-alteração dos Dados LDAP (LDIF)

O formato de inter-alteração de dados planos usados para representar o conteúdo do diretório LDAP e solicitações de atualização. O conteúdo do diretório é representado como uma gravação para cada solicitação de atualização e objeto. O conteúdo consiste na adição, renomeação, remoção, modificação das solicitações.

Exemplo 12.2. Amostra do Arquivo LDIF

```
dn: dc=jboss,dc=org
objectclass: top
objectclass: dcObject
objectclass: organization
dc: jboss
o: JBoss

dn: ou=People,dc=jboss,dc=org
objectclass: top
objectclass: organizationalUnit
ou: People

dn: uid=jsmith,ou=People,dc=jboss,dc=org
objectclass: top
objectclass: uidObject
objectclass: person
uid: jsmith
cn: John
sn: Smith
userPassword: theduke

dn: ou=Roles,dc=jboss,dc=org
objectclass: top
objectclass: organizationalUnit
ou: Roles
```

```
dn: cn=JBossAdmin,ou=Roles,dc=jboss,dc=org
objectclass: top
objectclass: groupOfNames
cn: JBossAdmin
member: uid=jsmith,ou=People,dc=jboss,dc=org
description: the JBossAdmin group
```

12.1.2. Empilhamento da Senha

Os módulos de logon podem ser encadeados numa pilha, com cada módulo de logon fornecendo ambos componentes de autorização e autenticação. Isto funciona em muitos casos, mas às vezes a autenticação e autorização são divididas em múltiplos armazenamentos de gerenciamento do usuário.

A Seção 12.1.1, “LdapLoginModule” descreve como combinar um LDAP e o banco de dados relacional, permitindo um usuário a ser autenticado por qualquer sistema. No entanto, considere o caso onde os usuários são gerenciados num servidor LDAP central, mas as funções específicas do aplicativo são armazenadas no banco de dados do aplicativo. A opção do módulo de pilha da senha captura essa relação.

Para uso da pilha da senha, cada módulo de logon deve configurar o atributo <module-option> **`password-stacking`** para **`useFirstPass`**. Caso um módulo anterior configurado para a pilha da senha tenha autenticado o usuário, todos os demais módulos de pilha considerarão o usuário autenticado e apenas tentarão fornecer um conjunto de funções para a etapa de autorização.

Quando a opção **`password-stacking`** for configurada para **`useFirstPass`**, este módulo observa primeiro por um nome e senha compartilhados sob os nomes de propriedade javax.security.auth.login.name e javax.security.auth.login.password respectivamente no mapa do estado compartilhado do módulo de logon.

Caso encontradas, essas propriedades são usadas como o nome e senha principal. Caso não encontradas, a senha e nome principal são configurados por esse módulo de logon e armazenados sob os nomes de propriedade javax.security.auth.login.name e javax.security.auth.login.password respectivamente.

NOTA

Quando usando a pilha da senha, configure todos os módulos a serem solicitados. Isto garante que todos os módulo são considerados e possuem a chance de contribuir funções ao processo de autorização.

Exemplo 12.3. Amostra da Pilha da Senha

Essa amostra apresenta como a pilha de senha pode ser usada.

```
<application-policy name="todo">
   <authentication>
      <login-module code="org.jboss.security.auth.spi.LdapLoginModule"
flag="required">
      <!-- LDAP configuration -->
         <module-option name="password-stacking">useFirstPass</module-
option>
      </login-module>
```

```
      <login-module
code="org.jboss.security.auth.spi.DatabaseServerLoginModule"
flag="required">
      <!-- database configuration -->
         <module-option name="password-stacking">useFirstPass</module-
option>
      </login-module>
   </authentication>
</application-policy>
```

12.1.3. Aplicação do Hash na Senha

A maioria dos módulos de logon devem comparar uma senha suprida por um cliente com uma senha armazenada num sistema de gerenciamento do usuário. Esses módulos normalmente trabalham com as senhas de texto plano, mas podem ser configurados para suportar as senhas com hash para prevenir textos planos de serem armazenados ao lado do servidor.

Exemplo 12.4. Aplicação do Hash na Senha

A seguinte configuração do módulo de logon determina aos usuários não-autenticados o **nobody** de nome principal e contém o based64-encoded, hashes MD5 das senhas num arquivo **`usersb64.properties`**. O arquivo **`usersb64.properties`** pode fazer parte de um classpath de implantação ou ser salvo no diretório **`/conf`**.

```
<policy>
   <application-policy name="testUsersRoles">
      <authentication>
         <login-module
code="org.jboss.security.auth.spi.UsersRolesLoginModule"
flag="required">
            <module-option
name="usersProperties">usersb64.properties</module-option>
            <module-option name="rolesProperties">test-users-
roles.properties</module-option>
            <module-option
name="unauthenticatedIdentity">nobody</module-option>
            <module-option name="hashAlgorithm">MD5</module-option>
            <module-option name="hashEncoding">base64</module-option>
         </login-module>
      </authentication>
   </application-policy>
</policy>
```

hashAlgorithm

O nome do algoritmo **`java.security.MessageDigest`** para aplicar o hash na senha. Não existe padrão, de forma que a opção deve ser especificada para ativar o hash. Os valores típicos são **`MD5`** e **`SHA`**.

hashEncoding

A sequência especifica um dos três tipos de codificação: **`base64`**, **`hex`** ou **`rfc2617`**. O padrão é **`base64`**.

hashCharset

O conjunto de caractere de codificação usado para converter a senha de texto simples a um array de byte. A codificação do padrão da plataforma é o padrão.

hashUserPassword

Especifica que o algorismo hash deve ser aplicado à senha que o usuário submete. A senha do usuário com hash é comparada ao valor no módulo de logon, que espera-se ser um hash da senha. O padrão é **`true`**.

hashStorePassword

Especifica que o algorismo hash deve ser aplicado à senha armazenada ao lado do servidor. Isto é usado para resumir a autenticação, onde o usuário submete um hash da senha do usuário com os tokens específicos de solicitação a partir do servidor a ser comparado. O algorismo hash (para resumir, isto seria **`rfc2617`**) é utilizado para computar o hash ao lado do servidor, que deve coincidir o valor com hash enviado a partir do cliente.

Caso você precise gerar senhas em código, a classe **`the org.jboss.security.Util`** fornece um método de ajuda estático que efetuará o hash numa senha usando a codificação especificada.

```
String hashedPassword = Util.createPasswordHash("MD5",
 Util.BASE64_ENCODING, null, null, "password");
```

O OpenSSL fornece uma forma alternativa de gerar senhas com hash.

```
echo -n password | openssl dgst -md5 -binary | openssl base64
```

Em ambos os casos, a senha de texto deve efetuar o hash para **`X03MO1qnZdYdgyfeuILPmQ==`**. Este valor deve ser armazenado num armazenamento do usuário.

12.1.4. Identidade não-autenticada

Perceba que todas as solicitações são recebidas num formato autenticado. O **`unauthenticated identity`** é uma opção de configuração do módulo de logon que determina uma identidade específica (convidado, por exemplo) para solicitações que são feitas com a informação não associada. Isto pode ser usado para permitir os servlets não-protegidos para invocar os métodos no EJBs que não requerem uma função específica. Tal principal não possui funções associadas e pode apenas associar tanto os métodos EJB ou EJBs não assegurados que são associados com a restrição de permissão não selecionada.

- **unauthenticatedIdentity**: define o nome principal que deve ser determinado para solicitações que não contém a informação de autenticação.

12.1.5. UsersRolesLoginModule

O **`UsersRolesLoginModule`** é um módulo de logon que suporta usuários múltiplos e funções de usuários carregadas de arquivos de propriedades Java. O arquivo de mapeamento nome-para-senha do usuário é chamado **`users.properties`** e o arquivo de mapeamento de nome-para-funções é chamado **`roles.properties`**.

As opções de configuração do módulo de logon suportadas incluem o seguinte:

usersProperties

Nome do recurso (arquivo) de propriedades contendo o nome do usuário para mapeamentos de senha. O padrão é **`<filename_prefix>-users.properties`**.

rolesProperties

Nome do recurso (arquivo) de propriedades contendo o nome de usuário para mapeamentos de funções. O padrão é **`<filename_prefix>-roles.properties`**.

O modelo de logon suporta o empilhamento da senha, a aplicação do hash na senha e a identidade não-autenticada.

Os arquivos de propriedades são localizados durante a inicialização usando o carregador de classe do contexto de segmentação do método iniciar. Isto significa que esses arquivos podem ser posicionados no JAR de implantação do Java EE, o diretório de configuração do JBoss ou qualquer diretório do servidor do JBoss ou classpath do sistema. O propósito primário deste módulo de logon é testar facilmente as configurações de segurança de usuários múltiplos e funções usando os arquivos de propriedade com o aplicativo.

Exemplo 12.5. UserRolesLoginModule

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">

   <!-- ejb3 test application-policy definition -->
   <application-policy xmlns="urn:jboss:security-beans:1.0" name="ejb3-
sampleapp">
      <authentication>
         <login-module
code="org.jboss.security.auth.spi.UsersRolesLoginModule"
flag="required">
            <module-option name="usersProperties">ejb3-sampleapp-
users.properties</module-option>
            <module-option name="rolesProperties">ejb3-sampleapp-
roles.properties</module-option>
         </login-module>
      </authentication>
   </application-policy>

</deployment>
```

O arquivo **`ejb3-sampleapp-users.properties`** no Exemplo 12.5, “UserRolesLoginModule” usa o formato **`username=password`** com cada entrada de usuário numa linha separada:

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

O arquivo **`ejb3-sampleapp-roles.properties`** referenciado no Exemplo 12.5, “UserRolesLoginModule” usa o **`username=role1,role2,`** padrão com o valor do nome do grupo opcional. Por exemplo:

```
username1=role1,role2,...
username1.RoleGroup1=role3,role4,...
username2=role1,role3,...
```

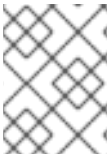
O nome da propriedade user name.XXX padrão presente no **ejb3-sampleapp-roles.properties** é usado para determinar as funções do nome do usuário para o grupo nomeado em particular onde a porção **XXX** do nome da propriedade é o nome do grupo. O formulário user name=... é uma abreviação para user name.Roles=..., onde o nome do grupo **Roles** é um nome padrão que o **JaasSecurityManager** espera conter as funções que definem permissões aos usuários.

Segue abaixo as definições equivalentes para o nome de usuário **jduke**:

```
jduke=TheDuke,AnimatedCharacter
jduke.Roles=TheDuke,AnimatedCharacter
```

12.1.6. DatabaseServerLoginModule

O **DatabaseServerLoginModule** é o módulo do logon na Conectividade do Banco de Dados do Java - Java Database Connectivity (JDBC) que suporta a autenticação e o mapeamento da função. Use este módulo de logon caso você possua o nome do usuário, senha e informação da função armazenada num banco de dados relacional.



NOTA

Este módulo suporta o empilhamento da senha, a aplicação do hash na senha e a identidade não-autenticada.

O **DatabaseServerLoginModule** é baseado em duas tabelas lógicas:

```
Table Principals(PrincipalID text, Password text)
Table Roles(PrincipalID text, Role text, RoleGroup text)
```

A tabela **Principals** associa o **PrincipalID** do usuário com a senha válida e a tabela **Roles** associa o **PrincipalID** do usuário com os próprios conjuntos de funções. As funções usadas para permissões do usuário devem estar contidas em filas com um valor de coluna **RoleGroup** do **Roles**.

As tabelas são lógicas de forma que você pode especificar a solicitação SQL que o módulo de logon usa. A única solicitação é que o **java.sql.ResultSet** possua a mesma estrutura lógica às das tabelas **Principals** e **Roles** descritas anteriormente. Os nomes atuais das tabelas e colunas não são relevantes uma vez que os resultados são acessados baseando-se no índice da coluna.

Para melhor entendimento desta noção, considere um banco de dados com duas tabelas, **Principals** e **Roles**, conforme já declarado. As seguintes declarações preenchem as tabelas com os seguintes dados:

- O **PrincipalIDjava** com o **Password** do **echoman** na tabela **Principals**.
- O **PrincipalIDjava** com a função nomeada **Echo** no **RolesRoleGroup** da tabela **Roles**.
- O **PrincipalIDjava** com a função nomeada **caller_java** no **CallerPrincipalRoleGroup** da tabela **Roles**.

```
INSERT INTO Principals VALUES('java', 'echoman')
INSERT INTO Roles VALUES('java', 'Echo', 'Roles')
INSERT INTO Roles VALUES('java', 'caller_java', 'CallerPrincipal')
```

As opções de logon suportadas incluem o seguinte:

dsJndiName

O nome do JNDI para o **DataSource** do banco de dados contendo as tabelas **Principals** e **Roles**. O padrão é **java:/DefaultDS**, caso não seja especificado.

principalsQuery

A consulta de declaração preparada é equivalente a: **select Password from Principals where PrincipalID=?**. Essa será a declaração preparada utilizada, caso não seja especificada.

rolesQuery

A consulta de declaração preparada é equivalente a: **select Role, RoleGroup from Roles where PrincipalID=?**. Essa será a declaração preparada utilizada, caso não seja especificada.

ignorePasswordCase

Um aviso boolean indicando se a comparação de senha deve ignorar o caso. Isto pode ser útil para a codificação de senha com hash onde o caso da senha com não é significativo.

principalClass

Uma opção que especifica uma classe de implementação **Principal**. Isto deve suportar um construtor levando um argumento de sequência para o nome principal.

Uma amostra da configuração **DatabaseServerLoginModule** pode ser construída conforme abaixo:

```
CREATE TABLE Users(username VARCHAR(64) PRIMARY KEY, passwd VARCHAR(64))
CREATE TABLE UserRoles(username VARCHAR(64), userRoles VARCHAR(32))
```

Uma entrada **login-config.xml** correspondente parece-se com o seguinte:

```
<policy>
   <application-policy name="testDB">
      <authentication>
         <login-module
code="org.jboss.security.auth.spi.DatabaseServerLoginModule"
flag="required">
            <module-option name="dsJndiName">java:/MyDatabaseDS</module-
option>
            <module-option name="principalsQuery">select passwd from
Users username where username=?</module-option>
            <module-option name="rolesQuery">select userRoles, 'Roles'
from UserRoles where username=?</module-option>
         </login-module>
      </authentication>
   </application-policy>
</policy>
```

12.1.7. BaseCertLoginModule

O **BaseCertLoginModule** autentica usuários baseados nos certificados X509. Um caso típico de uso para o módulo de logon é a autenticação **CLIENT-CERT** na camada da web.

Este módulo de logon executa apenas autenticação. Você deverá combinar isto com outro módulo de logon capaz de adquirir as funções de autorização para definir completamente o acesso à web protegida ou componente EJB. As duas sub-classes deste módulo de logon, **`CertRolesLoginModule`** e **`DatabaseCertLoginModule`** estendem o comportamento para obter as funções de autorização de ambos banco de dados ou arquivo de propriedades.

O **`BaseCertLoginModule`** precisa de um **`KeyStore`** para executar a validação do usuário. Isto é obtido através de uma implantação **`org.jboss.security.SecurityDomain`**. Normalmente, a implantação **`SecurityDomain`** é configurada usando o **`org.jboss.security.plugins.JaasSecurityDomain`** MBean conforme apresentado neste fragmento de configuração **`jboss-service.xml`**:

```
<mbean code="org.jboss.security.plugins.JaasSecurityDomain"
name="jboss.ch8:service=SecurityDomain">
   <constructor>
      <arg type="java.lang.String" value="jmx-console"/>
   </constructor>
   <attribute name="KeyStoreURL">resource:localhost.keystore</attribute>
   <attribute name="KeyStorePass">unit-tests-server</attribute>
</mbean>
```

A configuração cria um domínio de segurança com o **`jmx-console`** do nome, com uma implantação **`SecurityDomain`** disponível através do JNDI sob o **`java:/jaas/jmx-console`** do nome. O domínio de segurança segue o padrão de nomeamento do domínio de segurança do JBossSX.

Procedimento 12.1. Proteção dos Aplicativos da Web com os Certificados e Autorização baseada na Função

Este procedimento descreve como proteger um aplicativo da web, tal como o **`jmx-console.war,`** usando os certificados de cliente e a autorização baseada da função.

1. **Declaração dos Recursos e Funções**
   Modifique o **`web.xml`** para declarar os recursos a serem protegidos com as funções permitidas e domínio de segurança a serem usados para a autenticação e autorização.

```
<?xml version="1.0"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

 ...
   <!-- A security constraint that restricts access to the HTML JMX
console to users with the role JBossAdmin. Edit the roles to what
you want and uncomment the WEB-INF/jboss-web.xml/security-domain
element to enable secured access to the HTML JMX console. -->
   <security-constraint>
      <web-resource-collection>
         <web-resource-name>HtmlAdaptor</web-resource-name>
         <description>An example security config that only allows
users with the role JBossAdmin to access the HTML JMX console web
application
         </description>
         <url-pattern>/*</url-pattern>
      </web-resource-collection>
```

```
      <auth-constraint>
         <role-name>JBossAdmin</role-name>
      </auth-constraint>
   </security-constraint>

   <login-config>
      <auth-method>BASIC</auth-method>
      <realm-name>JBoss JMX Console</realm-name>
   </login-config>

   <security-role>
      <role-name>JBossAdmin</role-name>
   </security-role>
</web-app>
```

2. **Especificação do Domínio de Segurança do JBoss**
   Especifique o domínio de segurança solicitado no **`jboss-web.xml`**.

```
<jboss-web>
   <security-domain>jmx-console</security-domain>
</jboss-web>
```

3. **Especificação da Configuração do Módulo de Logon**
   Define a configuração do módulo de logon para o domínio de segurança do jmx-console que você acabou de especificar. Isto é realizado no arquivo **`conf/login-config.xml`**.

```
<application-policy name="jmx-console">
   <authentication>
      <login-module
code="org.jboss.security.auth.spi.BaseCertLoginModule"
flag="required">
         <module-option name="password-
stacking">useFirstPass</module-option>
         <module-option name="securityDomain">jmx-console</module-
option>
      </login-module>
      <login-module
code="org.jboss.security.auth.spi.UsersRolesLoginModule"
flag="required">
         <module-option name="password-
stacking">useFirstPass</module-option>
         <module-option name="usersProperties">jmx-console-
users.properties</module-option>
         <module-option name="rolesProperties">jmx-console-
roles.properties</module-option>
      </login-module>
   </authentication>
</application-policy>
```

O Procedimento 12.1, “Proteção dos Aplicativos da Web com os Certificados e Autorização baseada na Função” que apresenta o **`BaseCertLoginModule`** é usado para autenticação do cert do cliente e o **`UsersRolesLoginModule`** é apenas usado para a autorização devido à opção **`password-stacking=useFirstPass`**. Ambos os **`localhost.keystore`** e **`jmx-console-roles.properties`** requerem uma entrada que mapeia o associado principal ao cert do cliente.

Por padrão, o principal é criado usando o nome distinguido do certificado do cliente, tal como o DN especificado no Exemplo 12.6, “Amostra de Certificado”.

Exemplo 12.6. Amostra de Certificado

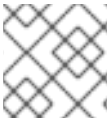
```
[conf]$ keytool -printcert -file unit-tests-client.export
Owner: CN=unit-tests-client, OU=JBoss Inc., O=JBoss Inc., ST=Washington,
C=US
Issuer: CN=jboss.com, C=US, ST=Washington, L=Snoqualmie Pass,
EMAILADDRESS=admin
@jboss.com, OU=QA, O=JBoss Inc.
Serial number: 100103
Valid from: Wed May 26 07:34:34 PDT 2004 until: Thu May 26 07:34:34 PDT
2005
Certificate fingerprints:
         MD5:  4A:9C:2B:CD:1B:50:AA:85:DD:89:F6:1D:F5:AF:9E:AB
         SHA1:
DE:DE:86:59:05:6C:00:E8:CC:C0:16:D3:C2:68:BF:95:B8:83:E9:58
```

O **`localhost.keystore`** precisa do certificado no Exemplo 12.6, “Amostra de Certificado” armazenado com um alias do **`CN=unit-tests-client, OU=JBoss Inc., O=JBoss Inc., ST=Washington, C=US`**. O **`jmx-console-roles.properties`** também precisa de uma entrada para a mesma entrada. Uma vez que o DN contém caracteres que são normalmente tratados como delimitadores, você deverá fugir do problema com caracteres usando uma barra invertida ('\'), conforme ilustrado abaixo.

```
# Um arquivo roles.properties de amostra para uso com o
UsersRolesLoginModule
CN\=unit-tests-client,\ OU\=JBoss\ Inc.,\ O\=JBoss\ Inc.,\
ST\=Washington,\ C\=US=JBossAdmin
admin=JBossAdmin
```

12.1.8. IdentityLoginModule

O **`IdentityLoginModule`** é um módulo de logon\nsimples que associa o nome do usuário de código rígido a qualquer assunto autenticado em relação ao módulo. Ele cria uma instância **`SimplePrincipal`** usando o nome especificado pela opção **`principal`**.



NOTA

Este módulo suporta o empilhamento de senha.

Este módulo de logon é útil quando você precisar fornecer uma identidade fixa a um serviço, e em ambientes de desenvolvimento quando você desejar testar a segurança associada com as funções associadas e um principal gerado.

As opções de configuração do módulo de logon incluem:

principal

Este é o nome a ser usado no **`SimplePrincipal`**, sendo que todos os usuários são autenticados como tal. O padrão do nome principal é **`guest`**, caso nenhuma opção do principal seja especificada.

funções

Esta é a lista de vírgula delimitada das funções que serão determinadas ao usuário.

Segue abaixo uma amostra da entrada de configuração XMLLoginConfig. A entrada autentica todos os usuários como o principal nomeado **jduke** e determina os nomes de função determinados do **TheDuke** e **AnimatedCharacter**:

```
<policy>
   <application-policy name="testIdentity">
      <authentication>
         <login-module
code="org.jboss.security.auth.spi.IdentityLoginModule" flag="required">
            <module-option name="principal">jduke</module-option>
            <module-option
name="roles">TheDuke,AnimatedCharacter</module-option>
         </login-module>
      </authentication>
   </application-policy>
</policy>
```

12.1.9. RunAsLoginModule

O **RunAsLoginModule** (**org.jboss.security.auth.spi.RunAsLoginModule**) é um módulo de ajuda que envia uma execução como função na pilha durante a fase de logon de autenticação, além de lançar a execução como função tanto na fase de confirmação ou abortação.

O propósito deste módulo de logon é fornecer uma função dos módulos de logon que devem acessar os recursos protegidos, com o objetivo de executar a própria autenticação (por exemplo, o módulo de logon que acessa um EJB protegido). O **RunAsLoginModule** deve ser configurado antes dos módulos de logon que requerem uma execução como função estabelecida.

A única opção de configuração do módulo de logon é a seguinte:

roleName

Nome da função para uso como a execução durante a fase de logon. Caso não especificado, o padrão do **nobody** será usado.

12.1.10. Criação do RunAsIdentity

Com o objetivo da Plataforma do Aplicativo JBoss Enterprise proteger o acesso aos métodos EJB, a identidade do usuário deve ser conhecida no momento em que a chamada do método é realizada.

A identidade do usuário no servidor é representada por uma instância **javax.security.auth.Subject** ou uma instância **org.jboss.security.RunAsIdentity**. Ambas as classes armazenam um ou mais principais que representam a identidade e uma lista de funções que a identidade possui. No caso do **javax.security.auth.Subject**, uma lista de credenciais é também armazenada.

Na seção <assembly-descriptor> do descritor de implantação **ejb-jar.xml**, você especifica uma ou mais funções que um usuário deve acessar para os diversos métodos EJB. Uma comparação destas listas mostra se o usuário possui uma das funções necessárias para acessar o método EJB.

Exemplo 12.7. Criação do org.jboss.security.RunAsIdentity

No arquivo **`ejb-jar.xml`**, você especifica um elemento <security-identity> com uma função <run-as> definida como o filho do elemento <session>.

```
<session>
   ...
   <security-identity>
      <run-as>
         <role-name>Admin</role-name>
      </run-as>
   </security-identity>
   ...
</session>
```

Esta declaração significa que a função "Admin" RunAsIdentity deve ser criada.

Para nomear o principal da função Admin, você deverá definir um elemento **`<run-as-principal>`** no arquivo **`jboss-web.xml`**.

```
<session>
   ...
   <security-identity>
      <run-as-principal>John</run-as-principal>
   </security-identity>
   ...
</session>
```

O elemento **`<security-identity>`** em ambos os arquivos **`ejb-jar.xml`** e **`jboss-web.xml`** são analisados no período de implantação. O nome da função **`<run-as>`** e o nome **`<run-as-principal>`** são armazenados na classe **`org.jboss.metadata.SecurityIdentityMetaData`**.

Exemplo 12.8. Determinação de funções múltiplas a um RunAsIdentity

Você pode determinar funções ao RunAsIdentity pelo mapeamento de funções aos principais no descritor de implantação **`jboss-web.xml`** do grupo do elemento **`<assembly-descriptor>`**.

```
<assembly-descriptor>
   ...
   <security-role>
      <role-name>Support</role-name>
      <principal-name>John</principal-name>
      <principal-name>Jill</principal-name>
      <principal-name>Tony</principal-name>
   </security-role>
   ...
</assembly-descriptor>
```

No Exemplo 12.7, “Criação do org.jboss.security.RunAsIdentity ”, o **`<run-as-principal>`** da "Marca" foi criado. A configuração nesta amostra estende a função "Admin" pela adição da função "Suporte". A nova função contém principais extras, incluindo o principal definido originalmente "John".

> O elemento **<security-role>** em ambos os arquivos **ejb-jar.xml** e **jboss.xml** são analisados no período de implantação. O **<role-name>** e os dados **<principal-name>** são armazenados na classe **org.jboss.metadata.SecurityIdentityMetaData**.

12.1.11. ClientLoginModule

O **ClientLoginModule** (**org.jboss.security.ClientLoginModule**) é uma implantação do **LoginModule** para uso dos clientes JBoss, com o objetivo de estabelecer a identidade do chamador e credenciais. Basicamente, isto determina o **org.jboss.security.SecurityAssociation.principal** ao valor do **NameCallback** preenchido pelo **callbackhandler** e o **org.jboss.security.SecurityAssociation.credential** para o valor do **PasswordCallback** preenchido pelo **callbackhandler**.

O **ClientLoginModule** é o único mecanismo suportado por um cliente para estabelecer o chamador do segmento atual. Ambos os aplicativos do cliente autônomo e os ambientes do servidor (atuando como clientes do JBoss EJB onde o ambiente de segurança não foi configurado para uso do JBossSX) devem usar o **ClientLoginModule**.

Perceba que este módulo de logon não executa qualquer autenticação. Ele apenas copia a informação de logon fornecida à mesma na camada de invocação do EJB do servidor do JBoss para subsequentes autenticações no servidor. Caso você precise executar a autenticação ao lado do cliente dos usuários, você precisará configurar outro módulo de logon, além do **ClientLoginModule**.

As opções do módulo de logon suportadas incluem o seguinte:

multi-threaded

O valor que especifica a maneira que os segmentos de logon conectam-se às fontes de armazenamento credencial e principal. Cada segmento de logon possui o próprio armazenamento credencial e principal e cada segmento separado deve executar o próprio logon, quando configurado para verdadeiro. Isto é útil em ambientes onde as identidades do usuário múltiplo são ativas em segmentos separados. A identidade de logon e as credenciais são variáveis aplicáveis a todos os segmentos no VM, quando configurado para falso. A configuração padrão é **false**.

password-stacking

Ativa a autenticação ao lado dos clientes usando outros módulos de logon tais como o **LdapLoginModule**. Quando a opção **password-stacking** for configurada para **useFirstPass**, o módulo primeiramente observa o nome de usuário compartilhado e senha usando o **javax.security.auth.login.name** e **javax.security.auth.login.password** respectivamente no mapa do estado compartilhado do módulo de logon. Isto permite que um módulo configurado anteriormente à estabelecer um nome e senha de usuário do JBoss válidos.

restore-login-identity

O valor que especifica se é que o principal e credencial **SecurityAssociation** vistos na entrada do método **login()** estão salvos e restaurados tanto quando abortando ou finalizando. Isto é necessário se você alterar as identidades e restaurar a identidade do chamador original. A informação principal e credencial são salvas e restauradas quando anulando ou finalizando, caso configurado para **true**. Caso configurado para **SecurityAssociation**, a anulação ou finalização esvaziarão o **SecurityAssociation**.

12.2. MÓDULOS PERSONALIZADOS

Caso os módulos de logon incluídos no framework do JBossSX não funcionarem no seu ambiente de segurança, você pode gravar sua própria implantação do módulo de logon personalizado. O **`JaasSecurityManager`** requer um padrão de uso particular do conjunto de principais **`Subject`**. Você deve entender os recursos da classe de informação da classe do Sujeito JAAS e o uso esperado desses recursos para gravar um módulo de logon que funciona com o **`JaasSecurityManager`**.

Esta seção examina essas solicitações e introduz duas implantações **`LoginModule`** básicas abstratas que podem ajudá-lo a implantar os módulos de logon personalizados.

Você pode obter a informação associada com o **`Subject`** pelo uso dos seguintes métodos:

```
java.util.Set getPrincipals()
java.util.Set getPrincipals(java.lang.Class c)
java.util.Set getPrivateCredentials()
java.util.Set getPrivateCredentials(java.lang.Class c)
java.util.Set getPublicCredentials()
java.util.Set getPublicCredentials(java.lang.Class c)
```

O JBossSX selecionou a escolha mais lógica: os conjuntos de principais obtidos através do **`getPrincipals()`** e **`getPrincipals(java.lang.Class)`** para as identidades e funções do **`Subject`**. Segue abaixo o uso padrão:

- As identidades do usuário (por exemplo: nome do usuário, número de segurança social e ID do empregado) conforme armazenado com os objetos **`java.security.Principal`** no conjunto **`SubjectPrincipals`**. A implementação **`Principal`** que representa a identidade do usuário deve basear-se nas comparações e igualdade do nome do principal. Uma implantação útil está disponível como a classe **`org.jboss.security.SimplePrincipal`**. Outras instâncias **`Principal`** podem ser adicionadas ao conjunto **`SubjectPrincipals`** conforme seja necessário.

- As funções determinadas do usuário estão também armazenadas no conjunto **`Principals`** e estão agrupadas nos conjuntos de função nomeados usando as instâncias **`java.security.acl.Group`**. A interface **`Group`** define uma coleção de **`Principal`**s e/ou **`Group`**s, além de ser uma sub-interface do **`java.security.Principal`**.

- Qualquer número de conjuntos de funções podem ser determinados a um **`Subject`**.

- O framework do JBossSX usa dois conjuntos de funções bem conhecidos pelos nomes de **`Roles`** e **`CallerPrincipal`**.

  - O grupo **`Roles`** é uma coleção de **`Principal`**s para funções nomeadas conforme o domínio do aplicativo pelo qual o **`Subject`** é autenticado. Este conjunto de função é usado por métodos como o **`EJBContext.isCallerInRole(String)`**, em que os EJBs podem usar para verificar se o chamador atual pertence à função do domínio do aplicativo nomeado. A lógica do interceptor de segurança que executa as checagens de permissão do método usa também este conjunto de função.

  - O **`CallerPrincipalGroup`** consiste da identidade **`Principal`** única determinada ao usuário no domínio do aplicativo. O método **`EJBContext.getCallerPrincipal()`** usa o **`CallerPrincipal`** para permitir que o domínio do aplicativo realize o mapeamento a partir da identidade do ambiente de operação à identidade do usuário adequada ao aplicativo. Caso o **`Subject`** não possua um **`CallerPrincipalGroup`**, a identidade de aplicativo é o mesmo como a identidade de ambiente operacional.

12.2.1. Suporte Padrão de Uso do Assunto

O JBossSX inclui os módulos de logon que preenchem o **`Subject`** autenticado com o padrão de modelo que reforça o uso do **`Subject`** correto, com o objetivo de simplificar a implantação correta do uso padrão descrito na Seção 12.2, “Módulos Personalizados”.

AbstractServerLoginModule

O módulo mais genérico dos dois é a classe **`org.jboss.security.auth.spi.AbstractServerLoginModule`**.

Ele fornece uma implantação da interface **`javax.security.auth.spi.LoginModule`** e oferece métodos abstratos para as tarefas chave específicas à infra-estrutura de segurança do ambiente. Os detalhes chave da classe estão descritos no Exemplo 12.9, “Fragmento da Classe AbstractServerLoginModule”. Os comentários do JavaDoc destacam as responsabilidades das sub-classes.

IMPORTANTE

A variável da instância **`loginOk`** é experimental. Ela deve ser configurada para **`true`**, caso o logon for bem sucedido ou **`false`** por qualquer sub-classes que substituem o método de logon. Caso esta variável for incorretamente usada, o método de confirmação não atualizará corretamente o assunto.

O rastreamento da fase de logon permite que módulos de logon sejam encadeados com os avisos de controle. Os avisos de controle não requerem que os módulos de logon sejam bem sucedidos como parte do processo de autenticação.

Exemplo 12.9. Fragmento da Classe AbstractServerLoginModule

```
package org.jboss.security.auth.spi;
/**
 *  This class implements the common functionality required for a JAAS
 *  server-side LoginModule and implements the JBossSX standard
 *  Subject usage pattern of storing identities and roles. Subclass
 *  this module to create your own custom LoginModule and override the
 *  login(), getRoleSets(), and getIdentity() methods.
 */
public abstract class AbstractServerLoginModule
    implements javax.security.auth.spi.LoginModule
{
    protected Subject subject;
    protected CallbackHandler callbackHandler;
    protected Map sharedState;
    protected Map options;
    protected Logger log;

    /** Flag indicating if the shared credential should be used */
    protected boolean useFirstPass;
    /**
     * Flag indicating if the login phase succeeded. Subclasses that
     * override the login method must set this to true on successful
     * completion of login
     */
    protected boolean loginOk;
```

```
    // ...
    /**
     * Initialize the login module. This stores the subject,
     * callbackHandler and sharedState and options for the login
     * session. Subclasses should override if they need to process
     * their own options. A call to super.initialize(...)  must be
     * made in the case of an override.
     *
     * <p>
     * The options are checked for the  <em>password-stacking</em>
parameter.
     * If this is set to "useFirstPass", the login identity will be
taken from the
     * <code>javax.security.auth.login.name</code> value of the
sharedState map,
     * and the proof of identity from the
     * <code>javax.security.auth.login.password</code> value of the
sharedState map.
     *
     * @param subject the Subject to update after a successful login.
     * @param callbackHandler the CallbackHandler that will be used to
obtain the
     * the user identity and credentials.
     * @param sharedState a Map shared between all configured login
module instances
     * @param options the parameters passed to the login module.
     */
    public void initialize(Subject subject,
                           CallbackHandler callbackHandler,
                           Map sharedState,
                           Map options)
    {
        // ...
    }


    /**
     *  Looks for javax.security.auth.login.name and
     *  javax.security.auth.login.password values in the sharedState
     *  map if the useFirstPass option was true and returns true if
     *  they exist. If they do not or are null this method returns
     *  false.
     *  Note that subclasses that override the login method
     *  must set the loginOk var to true if the login succeeds in
     *  order for the commit phase to populate the Subject. This
     *  implementation sets loginOk to true if the login() method
     *  returns true, otherwise, it sets loginOk to false.
     */
    public boolean login()
        throws LoginException
    {
        // ...
    }

    /**
```

```
     *  Overridden by subclasses to return the Principal that
     *  corresponds to the user primary identity.
     */
    abstract protected Principal getIdentity();

    /**
     *  Overridden by subclasses to return the Groups that correspond
     *  to the role sets assigned to the user. Subclasses should
     *  create at least a Group named "Roles" that contains the roles
     *  assigned to the user.  A second common group is
     *  "CallerPrincipal," which provides the application identity of
     *  the user rather than the security domain identity.
     *
     *  @return Group[] containing the sets of roles
     */
    abstract protected Group[] getRoleSets() throws LoginException;
}
```

UsernamePasswordLoginModule

O segundo módulo de logon base abstrato apto para os módulos de logon personalizados é o **`org.jboss.security.auth.spi.UsernamePasswordLoginModule`**.

Este módulo de logon futuramente simplificará a implantação do módulo de logon reforçando um nome de usuário baseado na sequência assim como uma identidade de usuário e uma senha **`char[]`** conforme as credenciais de autenticação. Ele suporta também o mapeamento de usuários anônimos (indicados pelo nome do usuário e senha nulos) para um principal sem funções. Os detalhes chave da classe estão destacadas no fragmento da classe seguinte. Os comentários do JavaDoc detalham as responsabilidades das sub-classes.

Exemplo 12.10. Fragmento da Classe UsernamePasswordLoginModule

```
package org.jboss.security.auth.spi;

/**
 *  An abstract subclass of AbstractServerLoginModule that imposes a
 *  an identity == String username, credentials == String password
 *  view on the login process. Subclasses override the
 *  getUsersPassword() and getUsersRoles() methods to return the
 *  expected password and roles for the user.
 */
public abstract class UsernamePasswordLoginModule
    extends AbstractServerLoginModule
{
    /** The login identity */
    private Principal identity;
    /** The proof of login identity */
    private char[] credential;
    /** The principal to use when a null username and password are seen
*/
    private Principal unauthenticatedIdentity;

    /**
     * The message digest algorithm used to hash passwords. If null then
     * plain passwords will be used. */
```

```
    private String hashAlgorithm = null;

    /**
     *  The name of the charset/encoding to use when converting the
     * password String to a byte array. Default is the platform's
     * default encoding.
     */
     private String hashCharset = null;

    /** The string encoding format to use. Defaults to base64. */
    private String hashEncoding = null;

    // ...

    /**
     *  Override the superclass method to look for an
     *  unauthenticatedIdentity property. This method first invokes
     *  the super version.
     *
     *  @param options,
     *  @option unauthenticatedIdentity: the name of the principal to
     *  assign and authenticate when a null username and password are
     *  seen.
     */
    public void initialize(Subject subject,
                           CallbackHandler callbackHandler,
                           Map sharedState,
                           Map options)
    {
        super.initialize(subject, callbackHandler, sharedState,
                         options);
        // Check for unauthenticatedIdentity option.
        Object option = options.get("unauthenticatedIdentity");
        String name = (String) option;
        if (name != null) {
            unauthenticatedIdentity = new SimplePrincipal(name);
        }
    }

    // ...

    /**
     *  A hook that allows subclasses to change the validation of the
     *  input password against the expected password. This version
     *  checks that neither inputPassword or expectedPassword are null
     *  and that inputPassword.equals(expectedPassword) is true;
     *
     *  @return true if the inputPassword is valid, false otherwise.
     */
    protected boolean validatePassword(String inputPassword,
                                       String expectedPassword)
    {
        if (inputPassword == null || expectedPassword == null) {
            return false;
        }
        return inputPassword.equals(expectedPassword);
```

```
    }

    /**
     *  Get the expected password for the current username available
     * via the getUsername() method. This is called from within the
     * login() method after the CallbackHandler has returned the
     * username and candidate password.
     *
     * @return the valid password String
     */
    abstract protected String getUsersPassword()
        throws LoginException;
}
```

Módulos de Logon de Sub-classificação

A escolha da sub-classificação do **`AbstractServerLoginModule`** versus **`UsernamePasswordLoginModule`** simplesmente baseia-se no fato do nome do usuário e credenciais baseados na sequência a serem usados ou não pela tecnologia de autenticação que você está gravando para o módulo de logon. Caso a semântica baseada na sequência for válida, o **`UsernamePasswordLoginModule`** será sub-classificado. Do contrário, o **`AbstractServerLoginModule`** será sub-classificado.

Etapas de Sub-classificação

As etapas, pelas quais seu módulo de logon personalizado deve executar, dependem de qual classe de módulo de logon básico você escolher. Quando gravando o módulo de logon personalizado, que integra sua infra-estrutura de segurança, você deve iniciar pela sub-classificação do **`AbstractServerLoginModule`** ou **`UsernamePasswordLoginModule`** para garantir que seu módulo fornece a informação **`Principal`** autenticada pela forma esperada pelo gerenciador de segurança do JBossSX.

Você deverá substituir o seguinte na sub-classificação do **`AbstractServerLoginModule`**:

- **`void initialize(Subject, CallbackHandler, Map, Map)`**: caso você possua opções de personalização para análise.
- **`boolean login()`**: para executar a atividade da autenticação. Certifique-se de configurar a variável da instância **`loginOk`** para verdadeiro caso o logon for bem sucedido e falso caso ele falhar.
- **`Principal getIdentity()`**: para retornar o objeto **`Principal`** para o usuário autenticado pela etapa **`log()`**.
- **`Group[] getRoleSets()`**: para retornar pelo menos um **`Group`** nomeado **`Roles`** que contém as funções determinadas ao **`Principal`** autenticado durante o **`login()`**. O segundo **`Group`** comum é nomeado **`CallerPrincipal`** e fornece a identidade do aplicativo do usuário ao invés da identidade do domínio de segurança.

Você deverá substituir o seguinte, quando sub-classificando o **`UsernamePasswordLoginModule`**:

- **`void initialize(Subject, CallbackHandler, Map, Map)`**: caso você possua opções de personalização para análise.
- **`Group[] getRoleSets()`**: para retornar pelo menos um **`Group`** nomeado **`Roles`** que contém

as funções determinadas ao **`Principal`** autenticado durante o **`login()`**. O segundo **`Group`** comum é nomeado **`CallerPrincipal`** e fornece a identidade do aplicativo do usuário ao invés da identidade do domínio de segurança.

- **`String getUsersPassword()`**: para retornar a senha esperada pelo nome de usuário atual disponível através do método **`getUsername()`**. O método **`getUsersPassword()`** é chamado a partir do **`login()`** após o **`callbackhandler`** retornar o nome do usuário e senha do candidato.

12.2.2. Amostra do LoginModule Personalizado

A seguinte informação o ajudará a criar uma amostra do Módulo de Logon que estende o **`UsernamePasswordLoginModule`** e obtém uma senha de usuário e nomes de função a partir da observação do JNDI.

No final desta seção, você terá criado um módulo de logon do contexto JNDI personalizado que retornará uma senha de usuário, caso você execute uma busca no contexto usando o nome do **`password/<username>`** do formulário (onde o **`<username>`** é o usuário atual sendo autenticado). Similarmente, uma busca do **`roles/<username>`** do formulário retorna funções de usuário requeridas.

O Exemplo 12.11, “Módulo de Logon Personalizado do JndiUserAndPass” apresenta o código de fonte para o módulo de logon personalizado do **`JndiUserAndPass`**.

Perceba que isto estende o JBoss **`UsernamePasswordLoginModule`**, tudo o que o **`JndiUserAndPass`** realiza é obter a senha e funções do usuário a partir do armazenamento JNDI. O **`JndiUserAndPass`** não interage com as operações **`LoginModule`** do JAAS.

Exemplo 12.11. Módulo de Logon Personalizado do JndiUserAndPass

```
package org.jboss.book.security.ex2;

import java.security.acl.Group;
import java.util.Map;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.login.LoginException;

import org.jboss.security.SimpleGroup;
import org.jboss.security.SimplePrincipal;
import org.jboss.security.auth.spi.UsernamePasswordLoginModule;

/**
 *  An example custom login module that obtains passwords and roles
 *  for a user from a JNDI lookup.
 *
 *  @author Scott.Stark@jboss.org
 *  @version $Revision: 1.4 $
*/
public class JndiUserAndPass
    extends UsernamePasswordLoginModule
{
    /** The JNDI name to the context that handles the password/username
lookup */
```

```
    private String userPathPrefix;
    /** The JNDI name to the context that handles the roles/ username
lookup */
    private String rolesPathPrefix;

    /**
     * Override to obtain the userPathPrefix and rolesPathPrefix
options.
     */
    public void initialize(Subject subject, CallbackHandler
callbackHandler,
                           Map sharedState, Map options)
    {
        super.initialize(subject, callbackHandler, sharedState,
options);
        userPathPrefix = (String) options.get("userPathPrefix");
        rolesPathPrefix = (String) options.get("rolesPathPrefix");
    }

    /**
     *  Get the roles the current user belongs to by querying the
     * rolesPathPrefix + '/' + super.getUsername() JNDI location.
     */
    protected Group[] getRoleSets() throws LoginException
    {
        try {
            InitialContext ctx = new InitialContext();
            String rolesPath = rolesPathPrefix + '/' +
super.getUsername();

            String[] roles = (String[]) ctx.lookup(rolesPath);
            Group[] groups = {new SimpleGroup("Roles")};
            log.info("Getting roles for user="+super.getUsername());
            for(int r = 0; r < roles.length; r ++) {
                SimplePrincipal role = new SimplePrincipal(roles[r]);
                log.info("Found role="+roles[r]);
                groups[0].addMember(role);
            }
            return groups;
        } catch(NamingException e) {
            log.error("Failed to obtain groups for
                        user="+super.getUsername(), e);
            throw new LoginException(e.toString(true));
        }
    }

    /**
     * Get the password of the current user by querying the
     * userPathPrefix + '/' + super.getUsername() JNDI location.
     */
    protected String getUsersPassword()
        throws LoginException
    {
        try {
            InitialContext ctx = new InitialContext();
            String userPath = userPathPrefix + '/' +
```

```
super.getUsername();
            log.info("Getting password for user="+super.getUsername());
            String passwd = (String) ctx.lookup(userPath);
            log.info("Found password="+passwd);
            return passwd;
        } catch(NamingException e) {
            log.error("Failed to obtain password for
                        user="+super.getUsername(), e);
            throw new LoginException(e.toString(true));
        }
    }
}
```

Os detalhes do armazenamento JNDI podem ser encontrados no **`org.jboss.book.security.ex2.service.JndiStore`** MBean. Este serviço vincula um **`ObjectFactory`** que retorna um **`javax.naming.Context`** proxy ao JNDI. O proxy manuseia as operações de busca realizadas pela checagem do prefixo do nome de busca em relação ao **`password`** e **`roles`**.

Quando o nome iniciar pelo **`password`**, a senha do usuário é solicitada. Quando o nome iniciar por **`roles`**, as funções do usuário serão solicitadas. A implementação da amostra sempre retorna uma senha de **`theduke`** e um array de nomes de funções igual ao **`{"TheDuke", "Echo"}`**, independente do nome de usuário. Você pode experimentar este procedimento com outras implementações.

O código de amostra inclui um bean de sessão simples para testar o módulo de logon personalizado. Para construção, implantação e rodagem da amostra, execute o seguinte comando no diretório de amostras:

```
[examples]$ ant -Dchap=security -Dex=2 run-example
...
run-example2:
     [echo] Waiting for 5 seconds for deploy...
     [java] [INFO,ExClient] Login with user name=jduke, password=theduke
     [java] [INFO,ExClient] Looking up EchoBean2
     [java] [INFO,ExClient] Created Echo
     [java] [INFO,ExClient] Echo.echo('Hello') = Hello
```

A escolha do uso do módulo de logon personalizado para a autenticação ao lado do servidor do usuário é determinado pela configuração de logon para o domínio de segurança da amostra. O descritor EJB JAR **`META-INF/jboss.xml`** determina o domínio de segurança.

```
<?xml version="1.0"?>
<jboss>
   <security-domain>security-ex2</security-domain>
</jboss>
```

O descritor **`META-INF/login-config.xml`** do SAR define a configuração do módulo de logon.

```
<application-policy name = "security-ex2">
   <authentication>
      <login-module code="org.jboss.book.security.ex2.JndiUserAndPass"
flag="required">
         <module-option
```

```
name="userPathPrefix">/security/store/password</module-option>
         <module-option name =
"rolesPathPrefix">/security/store/roles</module-option>
      </login-module>
   </authentication>
</application-policy>
```

PARTE III. CRIPTOGRAFIA E SEGURANÇA

CAPÍTULO 13. PROTOCOLO DE SENHA REMOTA DE SEGURANÇA

O protocolo da Senha Remota de Segurança (Secure Remote Password - SRP) é uma implementação de introdução de troca de chave pública descrita na Solicitação do Grupo de Trabalho Padrão para Comentários - Internet Standards Working Group Request For Comments 2945 (RFC2945). O resumo do RFC2945 declara:

> Este documento descreve um mecanismo de autenticação de rede potente criptográfica conhecido como protocolo da Senha Remota de Segurança - Secure Remote Password (SRP). Este mecanismo é útil para negociação de conexões de segurança usando uma senha de usuário fornecida, enquanto eliminando os problemas de segurança tradicionalmente associados com as senhas reutilizáveis. Além disso, o sistema executa uma troca da chave de segurança no processo de autenticação, permitindo camadas de segurança (privacidade e/ou proteção de integridade) a serem ativadas durante a sessão. Os servidores de chave confiável e infra-estruturas de certificado não são solicitados e os clientes não precisam armazenar ou gerenciar quaisquer chaves de longo termo. O SPR oferece ambas vantagens de implantação e segurança sobre técnicas de respostas de desafio existentes, fazendo disto uma substituição ideal onde a autenticação de senha é necessária.

A especificação RFC2945 completa pode ser obtida a partir do http://www.rfc-editor.org/rfc.html. Informações adicionais sobre o algoritmo SRP e sua história podem ser encontradas no http://www-cs-students.stanford.edu/~tjw/srp/.

Algoritmos como o *Diffie-Hellman* e *RSA* são conhecidos como algoritmos de troca de chave pública. O conceito de algoritmo de chave pública é que você possui duas chaves, uma pública que está disponível para todos e uma privada de apenas seu conhecimento. Quando alguém desejar lhe enviar alguma informação criptografada, a informação é criptografada usando a chave pública. Apenas você está apto a criptografar a informação usando a chave privada. Compare isto com uma senha compartilhada mais tradicional baseada nos esquemas de criptografia que requerem que o remetente e destinatário saibam da senha compartilhada. Os algoritmos de chave pública eliminam a necessidade de compartilhar senhas.

O JBossSX framework inclui uma implementação do SRP que consiste dos seguintes elementos:

- Uma implementação do protocolo de introdução do SRP que é independente de qualquer protocolo de cliente/servidor particular.
- Uma implementação RMI de protocolo de introdução como uma implementação SRP do servidor/cliente padrão.
- Uma implementação JAAS **`LoginModule`** ao lado do cliente que usa uma implementação para uso de autenticação dos clientes num modo de segurança.
- Um JMX MBean para gerenciamento da implantação do servidor RMI. O MBean permite que a implementação do servidor RMI seja conectável ao JMX framework e que externaliza a configuração do armazenamento de informação de verificação. Isto também estabiliza um cache de autenticação que é vinculado ao JNDI namespace do servidor do JBoss.
- Uma implementação JAAS **`LoginModule`** ao lado do servidor que usa um cache de autenticação gerenciado pelo SRP JMX MBean.

A Figura 13.1, “Os componentes do JBossSX do framework do cliente-servidor SRP.” descreve os componentes chaves envolvidos na implementação do framework do cliente/servidor SRP.

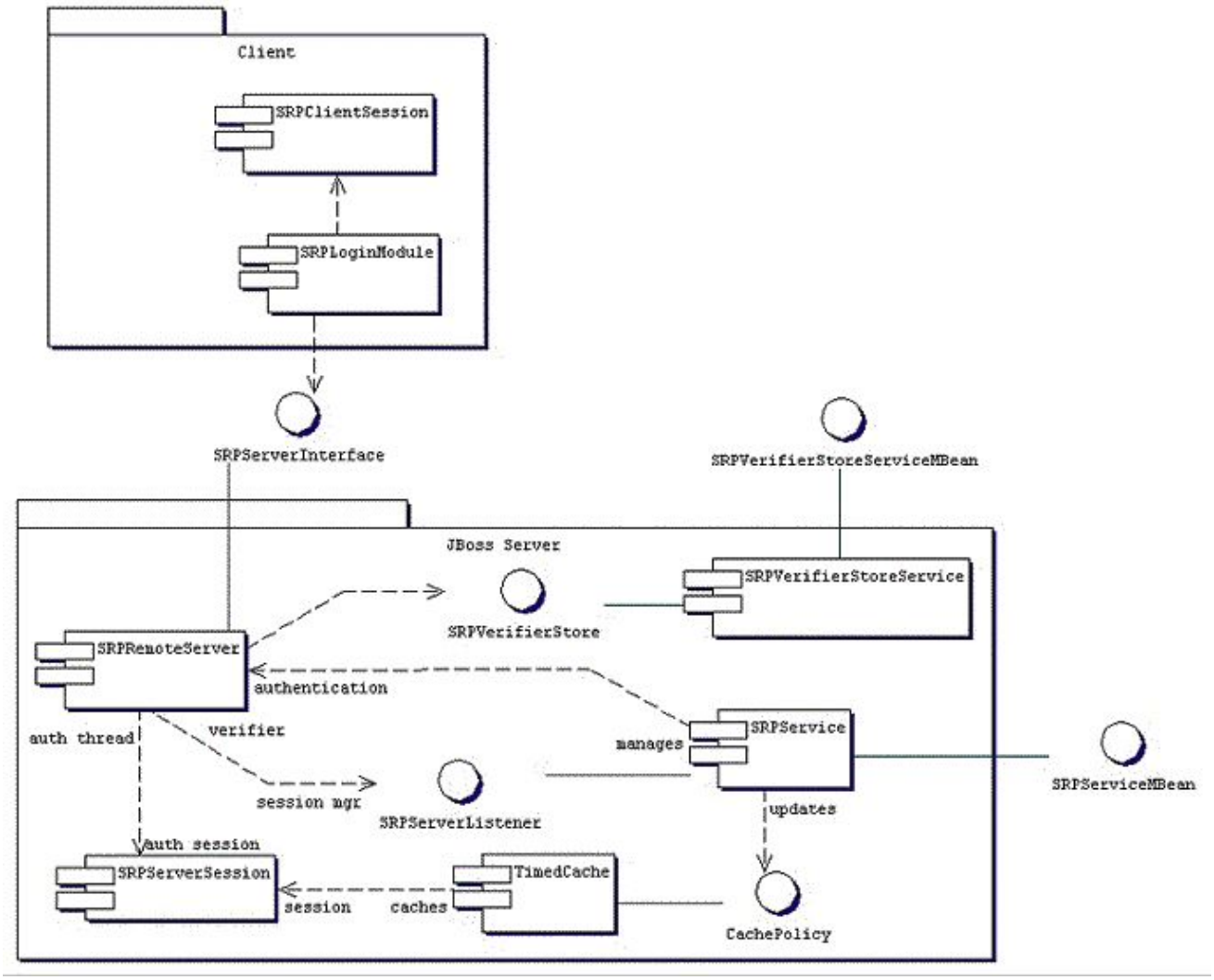


Figura 13.1. Os componentes do JBossSX do framework do cliente-servidor SRP.

O SRP aparece como uma implementação **`LoginModule`** do JAAS personalizado, ao lado do cliente, que comunica-se com o servidor de autenticação através de um **`org.jboss.security.srp.SRPServerInterface`** proxy. O cliente ativa a autenticação usando o SRP pela criação de uma entrada de configuração de logon que inclui o **`org.jboss.security.srp.jaas.SRPLoginModule`**. Este módulo suporta as seguintes opções de configuração:

principalClassName

Valor constante configurado para **`org.jboss.security.srp.jaas.SRPPrincipal`**.

srpServerJndiName

O nome do JNDI do objeto usado para comunicar-se com o servidor da autenticação do SRP. Caso ambas opções **`srpServerJndiName`** e **`srpServerRmiUrl`** forem especificadas, o **`srpServerJndiName`** leve prioridade sobre o **`srpServerRmiUrl`**.

srpServerRmiUrl

A sequência URL do protocolo RMI para a localização do **`SRPServerInterface`** proxy usado para comunicar-se com o servidor de autenticação SRP.

externalRandomA

Aviso que especifica se é que um componente aleatório da chave pública do cliente "A" deve vir do retorno de chamada do usuário. Isto pode ser usado para inserir um número qualquer criptográfico a partir de um token de hardware. O recurso é ativado, caso ele seja configurado para **`true`**.

hasAuxChallenge

O aviso que especifica se é que uma sequência será enviada ao servidor com um desafio adicional para validação do servidor. Caso a sessão do cliente suportar uma codificação de criptografia, a criptografia temporária será criada usando a chave privada da sessão e o objeto de desafio enviado com um **`javax.crypto.SealedObject`**. O recurso será desativado, caso configurado para **`true`**.

multipleSessions

O sinalizador especifica se é que o cliente gerado pode possuir sessões ativas múltiplas de logon. O recurso será ativado, caso ativado para **`true`**.

Outras opções passadas que não combinam com uma das opções anteriores nomeadas são tratadas como uma propriedade de uso para o ambiente passado ao construtor **`InitialContext`**. Isto é útil caso a interface do servidor SRP não esteja disponível no **`InitialContext`** padrão.

O **`SRPLoginModule`** e **`ClientLoginModule`** padrão podem ser configurados para permitir que credenciais sejam usadas para validação de acesso para os componentes do Java EE de segurança. Uma amostra de configuração de logon pode ser encontrada no Exemplo 13.1, “Entrada de Configuração de Segurança”.

Exemplo 13.1. Entrada de Configuração de Segurança

```
srp {
    org.jboss.security.srp.jaas.SRPLoginModule required
    srpServerJndiName="SRPServerInterface"
    ;

    org.jboss.security.ClientLoginModule required
    password-stacking="useFirstPass"
    ;
};
```

Existem dois MBeans que gerenciam os objetos que coletivamente formam o servidor SRP. O serviço primário é o **`org.jboss.security.srp.SRPService`** MBean. O outro MBean é o **`org.jboss.security.srp.SRPVerifierStoreService`**.

O **`org.jboss.security.srp.SRPService`** é responsável por expor uma versão RMI acessível do SRPServerInterface assim como uma atualização do cache da sessão de autenticação SRP.

Os atributos SRPService MBean configuráveis incluem o seguinte:

JndiName

Especifica o nome pelo qual o SRPServerInterface proxy deve estar disponível. Esta é a localização onde o **`SRPService`** realiza o bind no proxy do dinâmico serializável para o **`SRPServerInterface`**. O valor padrão é **`srp/SRPServerInterface`**.

VerifierSourceJndiName

Especifica o nome da implementação **`SRPVerifierSource`** que o **`SRPService`** deve usar. O

padrão do nome JNDI de fonte é **`srp/DefaultVerifierSource`**.

AuthenticationCacheJndiName

Especifica o nome pelo qual a implementação da autenticação **`org.jboss.util.CachePolicy`** usada para informação de autenticação de cache o bind é aplicado. O padrão do cache JNDI de autenticação é **`srp/AuthenticationCache`**.

ServerPort

O portal RMI para o **`SRPRemoteServerInterface`**. O valor padrão é 10099.

ClientSocketFactory

Nome da classe de implementação **`java.rmi.server.RMIClientSocketFactory`** personalizada opcional usada durante a exportação do **`SRPServerInterface`**. O valor padrão é **`RMIClientSocketFactory`**.

ServerSocketFactory

Nome da classe de implementação **`java.rmi.server.RMIServerSocketFactory`** personalizada opcional usada durante a exportação do **`SRPServerInterface`**. O valor padrão é **`RMIServerSocketFactory`**.

AuthenticationCacheTimeout

O intervalo da política de cache (em segundos). O valor padrão é 1800 (30 minutos).

AuthenticationCacheResolution

Especifica a resolução da política de cache (em segundos). Isto controla o intervalo entre as checagens para intervalos. O valor padrão é 60 (1 minuto).

RequireAuxChallenge

Configurado caso um cliente fornecer um desafio auxiliar como parte da fase de verificação. Isto fornece controle sobre a configuração **`SRPLoginModule`** usada pelo cliente independente da opção **`useAuxChallenge`** estar ativada ou não.

OverwriteSessions

Especifica se é que uma autenticação do usuário com êxito para uma sessão existente deve ser substituída ou não na sessão atual. Isto controla o comportamento do cache da sessão SRP do servidor quando clientes não tiverem ativado a sessão múltipla por modo de usuário. Caso configurado para **`false`**, a segunda tentativa de autenticação do usuário será concluída com êxito, no entanto a sessão SRP resultante não substituirá o estado da sessão SRP anterior. O valor padrão é **`false`**.

VerifierStoreJndiName

Especifica a localização da implementação do armazenamento da informação de senha SRP que deve ser fornecida e disponibilizada através do JNDI.

O **`org.jboss.security.srp.SRPVerifierStoreService`** é um serviço MBean de amostra que aplica o bind na implementação da interface **`SRPVerifierStore`**, que usa um arquivo dos objetos serializados como um armazenamento de persistência. Mesmo que não isto não seja realístico para um

ambiente de produção, ele permite testes para o protocolo SRP e fornece uma amostra de solicitações para um serviço **`SRPVerifierStore`**.\n

Os atributos **`SRPVerifierStoreService`** MBean incluem o seguinte:

JndiName

O nome JNDI pelo qual a implementação **`SRPVerifierStore`** deve estar disponível. O padrão é **`srp/DefaultVerifierSource`**, caso não seja especificado.

StoreFile

A localização do arquivo de armazenamento do objeto serializado do verificador de senha do usuário. Isto pode ser tanto um URL ou um nome de recurso a ser encontrado no classpath. O padrão é **`SRPVerifierStore.ser`**, caso não seja especificado.

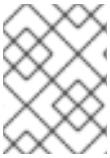
O **`SRPVerifierStoreService`** MBean também suporta as operações ***`addUser`*** e ***`delUser`*** para adição e remoção de usuários. As assinaturas são as seguintes:

```
public void addUser(String username, String password) throws IOException;
public void delUser(String username) throws IOException;
```

Uma configuração de amostra destes serviços é apresentada no Exemplo 13.2, “A interface SRPVerifierStore ” .

13.1. ENTENDENDO O ALGORITMO

O recurso do algoritmo SRP é que ele permite a autenticação mútua do cliente e servidor usando senhas de texto simples sem um canal de comunicação de segurança.



NOTA

A informação adicional no algoritmo SRP e seu histórico podem ser encontrados no http://srp.stanford.edu/.

Existem seis etapas executadas para completar a autenticação:

1. O **`SRPLoginModule`** ao lado do cliente restaura a partir do serviço de nomeação da instância SRPServerInterface para o servidor de autenticação remoto.

2. O **`SRPLoginModule`** seguinte ao lado do cliente solicita que os parâmetros SRP associem-se com o nome do usuário na tentativa do logon. Existe um número de parâmetros envolvidos no algoritmo SRP que devem ser escolhidos quando a senha do usuário é primeiro transformada na forma de verificador usada pelo algoritmo SRP. Ao invés da codificação rígida aos parâmetros (que poderia ser feita com um risco mínimo de segurança), a implantação do JBossSX permite que um usuário restaure esta informação como parte de seu protocolo Exchange. A chamada **`getSRPParameters(username)`** restaura os parâmetros do SRP para o nome do usuário gerado.

3. O **`SRPLoginModule`** ao lado do cliente inicia uma sessão SRP pela criação de um objeto **`SRPClientSession`** usando o nome do usuário do logon, senha de texto simples e parâmetros SRP obtidos na etapa 2. O cliente então cria um número aleatório A que será usado para construir a chave de sessão SRP primária. Em seguida, o cliente inicializa ao lado do

servidor da sessão SRP pela invocação do método **`SRPServerInterface.init`** e passa o nome do usuário e cliente gerados no número aleatório **`A`**. O servidor retorna seu próprio número aleatório **`B`**. Esta etapa corresponde à troca das chaves públicas.

4. O **`SRPLoginModule`** ao lado do cliente obtém uma chave de sessão SRP privada que é gerada como um resultado de trocas de mensagens anteriores. Isto é salvo como um credencial privado no **`Subject`** do logon. A resposta do desafio do servidor **`M2`** a partir da etapa 4 é verificada pela invocação do método **`SRPClientSession.verify`**. Caso isto suceda, a autenticação mútua do cliente para o servidor, e do servidor para o cliente são completadas. O **`SRPLoginModule`** ao lado do cliente seguinte cria um **`M1`** de desafio para o servidor pela invocação do método **`SRPClientSession.response`**, passando o número aleatório **`B`** do servidor como um argumento. Este desafio é enviado ao servidor através do método **`SRPServerInterface.verify`** e a resposta do servidor é salva como um **`M2`**. Esta etapa corresponde a uma troca de desafios. Nessas alturas, o servidor verificou que o usuário é quem ele disse ser.

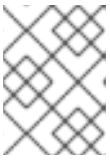
5. O **`SRPLoginModule`** ao lado do cliente salva o nome do usuário do logon e o desafio do **`M1`** no mapa sharedState do **`LoginModule`**. Isto é usado como nome e credenciais do Principal pelo **`ClientLoginModule`** do JBoss padrão. O desafio do **`M1`** é usado no lugar da senha como prova de identidade em quaisquer métodos de invocação nos componentes do Java EE. O desafio do **`M1`** é um hash forte de criptografia associado com a sessão SRP. Não é possível obter a senha do usuário através desta própria intercepção através de terceiros.

6. No final deste protocolo de autenticação, o SRPServerSession é posicionado no cache de autenticação do SRPService para uso subsequente pelo **`SRPCacheLoginModule`**.

Mesmo que o SRP possua propriedades interessantes, ele continua sendo um componente no framework do JBoss e possui algumas limitações pelas quais você deve estar ciente. Os problemas de nota incluem o seguinte:

- A maneira pela qual o JBoss desanexa o protocolo de transporte do método a partir do recipiente pode permitir que um usuário rastreio o desafio **`M1`** do SRP e use efetivamente o desafio para realizar solicitações como do nome do usuário associado. Os interceptores personalizados podem ser usados para prevenir o problema, pela criptografia do desafio usando a chave de sessão do SRP.

- O SRPService mantém um cache de sessões SRP que entram em intervalo após um período configurável. Após elas entrarem no intervalo, qualquer acesso do componente Java EE falhará uma vez que não há mecanismo para renegociação transparente dos credenciais de autenticação SRP. Você deve tanto configurar o período para intervalo do cache bem longo ou manusear a re-autenticação em seu código na falha.



NOTA

O SRPService suporta as durações do intervalo até 2,147,483,647 segundos ou aproximadamente 68 anos.

- Por padrão, pode haver apenas uma sessão SRP para um nome de usuário gerado por padrão. A sessão é classificada como com estado, uma vez que a sessão SRP negociada produz uma chave de sessão privada que pode ser usada para criptografia e descriptografia entre o cliente e servidor. O JBoss suporta múltiplas sessões de SRP por usuário, no entanto não é possível criptografar dados com uma chave de sessão e descriptografá-la com outra.

Você deverá configurar o domínio de segurança pelo qual os componentes são protegidos para usar o **`org.jboss.security.srp.jaas.SRPCacheLoginModule`**, com o objetivo de usar a autenticação

SPR de ponta-à-ponta. O **SRPCacheLoginModule** possui uma opção de configuração única nomeada **cacheJndiName** que determina a localização JNDI da instância **CachePolicy** de autenticação SRP. Isto deve corresponder ao valor de atributo **AuthenticationCacheJndiName** do **SRPService** MBean.

O **SRPCacheLoginModule** autentica os credenciais do usuário pela obtenção do desafio do cliente a partir do objeto **SRPServerSession** no cache de autenticação e comparação do mesmo ao desafio passado como credenciais do usuário. A Figura 13.2, “SRPCacheLoginModule com o Cache de Sessão SRP” ilustra a operação da implantação do método SRPCacheLoginModule.login.

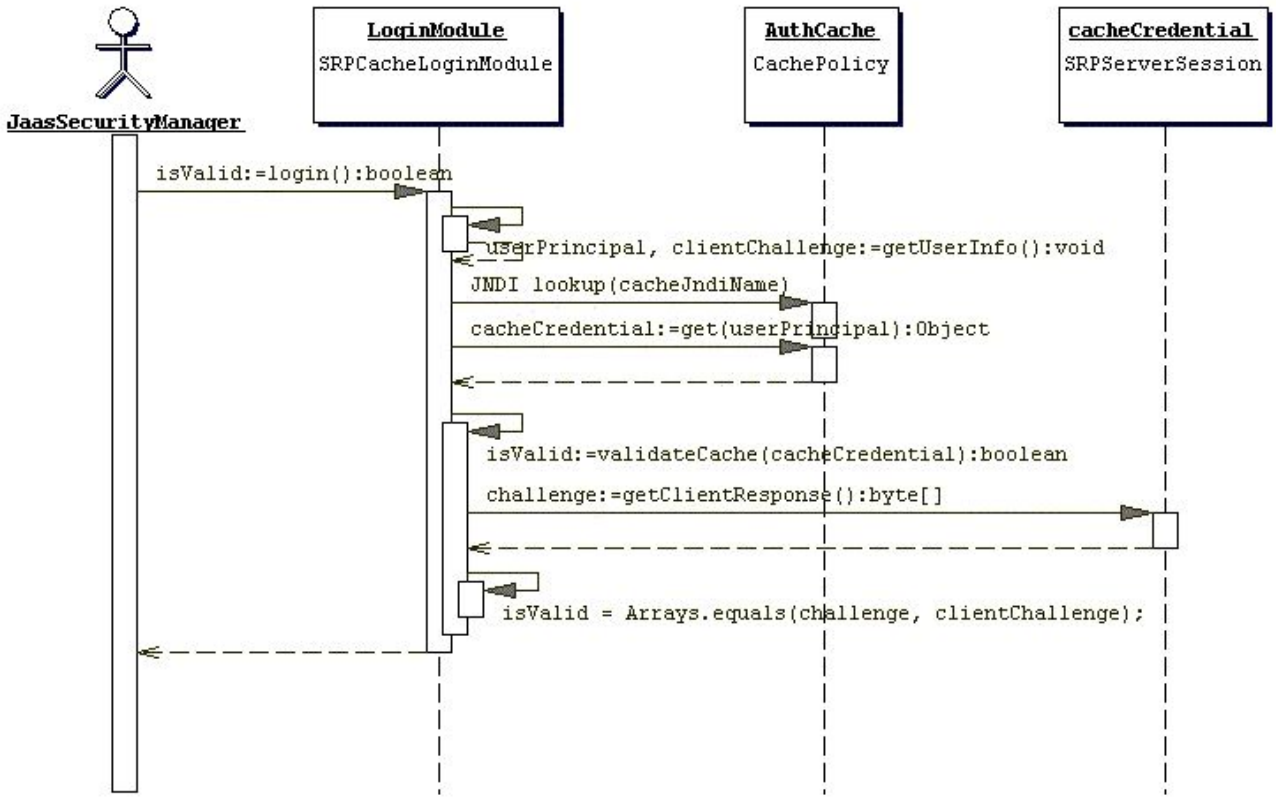


Figura 13.2. SRPCacheLoginModule com o Cache de Sessão SRP

13.2. CONFIGURAÇÃO DA INFORMAÇÃO DE SENHA REMOTA DE SEGURANÇA

Você deve criar um serviço MBean que fornece uma implementação da interface **SRPVerifierStore** que integra-se com seus armazenamentos de informação de segurança existentes. Informações sobre a interface **SRPVerifierStore** podem ser encontradas no Exemplo 13.2, “A interface SRPVerifierStore ”.

NOTA

A implementação padrão da interface **SRPVerifierStore** não é recomendada para um ambiente de segurança de produção uma vez que isto requer todas as informações hash de senha a serem disponibilizadas como um arquivo de objetos serializados.

Exemplo 13.2. A interface SRPVerifierStore

```
package org.jboss.security.srp;
```

```
import java.io.IOException;
import java.io.Serializable;
import java.security.KeyException;

public interface SRPVerifierStore
{
    public static class VerifierInfo implements Serializable
    {

        public String username;


        public byte[] salt;
        public byte[] g;
        public byte[] N;
    }


    public VerifierInfo getUserVerifier(String username)
        throws KeyException, IOException;

    public void setUserVerifier(String username, VerifierInfo info)
        throws IOException;


     public void verifyUserChallenge(String username, Object
auxChallenge)
         throws SecurityException;
}
```

A função primária de uma implementação **`SRPVerifierStore`** é fornecer acesso ao objeto **`SRPVerifierStore.VerifierInfo`** para o nome do usuário gerado. O método **`getUserVerifier(String)`** é chamado pelo **`SRPService`** no início de uma sessão SRP do usuário para obter os parâmetros necessários pelo algorítimo SRP. Os elementos dos objetos são os seguintes:

user name

O nome ou id do usuário usado para o logon.

verifier

O hash unidirecional de senha ou PIN que o usuário insere como prova de identidade. A classe **`org.jboss.security.Util`** possui um método **`calculateVerifier`** que executa o algoritmo hash de senha. A senha resultante leva a forma de **`H(salt | H(username | ':' | password))`**, onde **`H`** é a função hash de segurança SHA, conforme definido pelo RFC2945. O nome do usuário é convertido a partir da sequência a um **`byte[]`**, usando a codificação UTF-8.

salt

Número alternado usado para aumentar a dificuldade do ataque de dicionário de força bruta na verificação do banco de dados de senha, no evento em que o banco de dados estiver comprometido. O valor deve ser gerado a partir de um algoritmo de número aleatório quando a senha de texto simples existente estiver com hash.

g

O gerador primitivo do algoritmo SRP. Ele pode ser um parâmetro fixo conhecido ao invés de uma configuração por usuário. A classe de utilidade **`org.jboss.security.srp.SRPConf`** fornece diversas configurações para o **`g`**, incluindo o padrão útil obtido através do **`SRPConf.getDefaultParams().g()`**.

N

Os módulos de segurança primordial do algoritmo SRP. Isto pode ser um parâmetro fixo bem conhecido ao invés de uma configuração por usuário. A classe da utilidade **`org.jboss.security.srp.SRPConf`** fornece diversas configurações para o **`N`**, incluindo um bom padrão que pode ser obtido através do **`SRPConf.getDefaultParams().N()`**.

Procedimento 13.1. Integração do Armazenamento de Senha Existente

Leia este procedimento para melhor entendimento das etapas envolvidas na integração do seu armazenamento de senha existente.

1. **Crie o Armazenamento da Informação de Senha com Hash**
   Caso suas senhas já tenham sido armazenadas em um formulário com hash irreversível, este procedimento pode apenas ser feito baseando-se no usuário (por exemplo, como parte de um procedimento de atualização).

   Você pode implementar o **`setUserVerifier(String, VerifierInfo)`** como um método **`noOp`**, ou um método que lança uma exceção declarando que o armazenamento é de leitura apenas.

2. **Crie uma Interface SRPVerifierStore**
   Você deve criar uma implementação de interface **`SRPVerifierStore`** personalizada que entende como obter o **`VerifierInfo`** a partir do armazenamento criado.

   O **`verifyUserChallenge(String, Object)`** pode ser usado para integrar um token de hardware existente baseado em esquemas como SafeWord ou Radius no algoritmo SRP. Este método de interface é chamado apenas quando uma configuração **`SRPLoginModule`** de cliente especifica a opção **`hasAuxChallenge`**.

3. **Crie o JNDI MBean**
   Você deverá criar um MBean que expõe a interface **`SRPVerifierStore`** disponível ao JNDI e expõe quaisquer parâmetros configuráveis requeridos.

   O **`org.jboss.security.srp.SRPVerifierStoreService`** permitirá que você implemente isto, embora você possa implementar também o MBean usando a implementação do arquivo de propriedades Java do **`SRPVerifierStore`** (refira-se à Seção 13.3, “Amostra da Senha Remota de Segurança”).

13.3. AMOSTRA DA SENHA REMOTA DE SEGURANÇA

A amostra apresentada nesta seção demonstra a autenticação ao lado do cliente do usuário através do SRP assim como o acesso protegido subsequente a um EJB simples usando o desafio da sessão SRP como credencial do usuário. O código de teste implanta um EJB JAR que inclui um SAR para a configuração da configuração do módulo do logon ao lado do servidor e serviços SRP.

A configuração do módulo do logon ao lado do servidor é dinamicamente instalada usando o **`SecurityConfig`** MBean. Uma implantação personalizada da interface **`SRPVerifierStore`** é

também usada na amostra. A interface usa um armazenamento em memória que é gerado a partir do arquivo de propriedades Java, ao invés do armazenamento do objeto serializado conforme usado pelo **`SRPVerifierStoreService`**.

O serviço personalizado é o **`org.jboss.book.security.ex3.service.PropertiesVerifierStore`**. Segue abaixo os conteúdos do JAR que contém a amostra dos serviços EJB e SRP.

```
[examples]$ jar tf output/security/security-ex3.jar
META-INF/MANIFEST.MF
META-INF/ejb-jar.xml
META-INF/jboss.xml
org/jboss/book/security/ex3/Echo.class
org/jboss/book/security/ex3/EchoBean.class
org/jboss/book/security/ex3/EchoHome.class
roles.properties
users.properties
security-ex3.sar
```

Os itens relacionados ao SRP chave nesta amostra são a configuração dos serviços SRP MBean e as configurações do módulo de logon SRP. O descritor **`jboss-service.xml`** do **`security-ex3.sar`** está descrito no Exemplo 13.3, “O Descritor security-ex3.sar jboss-service.xml”.

As configurações do módulo de logon ao lado do servidor e do cliente de amostra estão descritas no Exemplo 13.4, “Configuração JAAS padrão ao lado do cliente.” e Exemplo 13.5, “A configuração XMLLoginConfig ao lado do servidor” gerado.

Exemplo 13.3. O Descritor security-ex3.sar jboss-service.xml

```
<server>
    <!-- The custom JAAS login configuration that installs
         a Configuration capable of dynamically updating the
         config settings -->

    <mbean code="org.jboss.book.security.service.SecurityConfig"
           name="jboss.docs.security:service=LoginConfig-EX3">
        <attribute name="AuthConfig">META-INF/login-
config.xml</attribute>
        <attribute
name="SecurityConfigName">jboss.security:name=SecurityConfig</attribute>
    </mbean>

    <!-- The SRP service that provides the SRP RMI server and server
side
         authentication cache -->
    <mbean code="org.jboss.security.srp.SRPService"
           name="jboss.docs.security:service=SRPService">
        <attribute name="VerifierSourceJndiName">srp-test/security-
ex3</attribute>
        <attribute name="JndiName">srp-
test/SRPServerInterface</attribute>
        <attribute name="AuthenticationCacheJndiName">srp-
test/AuthenticationCache</attribute>
        <attribute name="ServerPort">0</attribute>
```

```
<depends>jboss.docs.security:service=PropertiesVerifierStore</depends>
    </mbean>

    <!-- The SRP store handler service that provides the user password
verifier
         information -->
    <mbean code="org.jboss.security.ex3.service.PropertiesVerifierStore"
           name="jboss.docs.security:service=PropertiesVerifierStore">
        <attribute name="JndiName">srp-test/security-ex3</attribute>
    </mbean>
</server>
```

Os serviços de amostra são o **ServiceConfig**, o **PropertiesVerifierStore** e o **SRPService** MBeans. Perceba que o atributo **JndiName** do **PropertiesVerifierStore** é igual ao atributo **VerifierSourceJndiName** do **SRPService**. Além disso, o **SRPService** depende do **PropertiesVerifierStore**. Isto é solicitado uma vez que o **SRPService** precisa de uma implantação da interface **SRPVerifierStore** para acesso à informação da verificação da senha do usuário.

Exemplo 13.4. Configuração JAAS padrão ao lado do cliente.

```
srp {
    org.jboss.security.srp.jaas.SRPLoginModule required
    srpServerJndiName="srp-test/SRPServerInterface"
    ;

    org.jboss.security.ClientLoginModule required
    password-stacking="useFirstPass"
    ;
};
```

A configuração do módulo de logon ao lado do cliente faz uso do **SRPLoginModule** com um valor de opção **srpServerJndiName** que corresponde ao valor do atributo **SRPService** JndiName do componente do servidor do JBoss (**srp-test/SRPServerInterface**). O **ClientLoginModule** deve também ser configurado com o valor **password-stacking="useFirstPass"** para propagar os credenciais de autenticação do usuário gerado pelo **SRPLoginModule** à camada de invocação EJB.

Exemplo 13.5. A configuração XMLLoginConfig ao lado do servidor

```
<application-policy name="security-ex3">
    <authentication>
        <login-module
code="org.jboss.security.srp.jaas.SRPCacheLoginModule"
                      flag = "required">
            <module-option name="cacheJndiName">srp-
test/AuthenticationCache</module-option>
        </login-module>
        <login-module
code="org.jboss.security.auth.spi.UsersRolesLoginModule"
                      flag = "required">
            <module-option name="password-
stacking">useFirstPass</module-option>
```

```
            </login-module>
        </authentication>
    </application-policy>
```

Existem dois problemas a serem destacados sobre a configuração do módulo de logon ao lado do servidor:

1. A opção de configuração **`cacheJndiName=srp-test/AuthenticationCache`** informa o **`SRPCacheLoginModule`** sobre a localização do **`CachePolicy`** que contém o **`SRPServerSession`** para usuários que foram autenticados em relação ao **`SRPService`**. Este valor corresponde ao **`SRPService`** de valor do atributo **`AuthenticationCacheJndiName`**.

2. A configuração inclui um **`UsersRolesLoginModule`** com a opção de configuração **`password-stacking=useFirstPass`**. Você deve usar um segundo módulo de logon com o **`SRPCacheLoginModule`**, uma vez que o SRP é apenas uma tecnologia de autenticação. Para determinação de funções do principal que em troca determina as permissões associadas, um segundo módulo de logon deve ser configurado para aceitar os credenciais de autenticação validados pelo **`SRPCacheLoginModule`**.

O **`UsersRolesLoginModule`** aumenta a autenticação SRP pelo arquivo de propriedades baseado na autorização. As funções do usuário são obtidas a partir do arquivo **`roles.properties`** no EJB JAR.

Rode a amostra do cliente 3 pela execução do seguinte comando a partir do diretório das amostras do livro:

```
[examples]$ ant -Dchap=security -Dex=3 run-example
...
run-example3:
     [echo] Waiting for 5 seconds for deploy...
     [java] Logging in using the 'srp' configuration
     [java] Created Echo
     [java] Echo.echo()#1 = This is call 1
     [java] Echo.echo()#2 = This is call 2
```

No diretório **`examples/logs`**, o arquivo **`ex3-trace.log`** contém um rastro detalhado ao lado do cliente do algoritmo SRP. Os rastros apresentam passo-a-passo da construção das chaves públicas, desafios, chave de sessão e verificação.

Observe que o cliente leva um bom tempo para execução, relativo às outras amostras simples. O motivo disto é a construção da chave pública do cliente. Isto envolve a criação de um número alternado extremamente criptografado e este processo demora mais quando é primeiramente executado. As tentativas de autenticação subsequentes com o mesmo VM são muito mais rápidas.

Perceba que o **`Echo.echo()#2`** falha com uma exceção de autenticação. O código do cliente é suspenso por 15 segundos após realizar a primeira chamada para demonstrar o comportamento da expiração do cache **`SRPService`**. O intervalo da política de cache **`SRPService`** é configurado para 10 segundos para forçar este problema. Você deve configurar o intervalo do cache corretamente ou manusear a re-autenticação na falha, conforme descrito na Seção 13.3, “Amostra da Senha Remota de Segurança”.

CAPÍTULO 14. GERENCIADOR DE SEGURANÇA DO JAVA

Gerenciador de Segurança do Java

O Gerenciador de Segurança do Java é uma classe que gerencia o limite externo da área restrita da Máquina Virtual do Java (JVM), controlando como a execução do código com o JVM pode interagir com os recursos fora do JVM. Quando o Gerenciador de Segurança do Java é ativado, o Java API checa com o gerenciador de segurança pela aprovação antes de executar uma porção de operações potencialmente perigosas.

O Gerenciador de Segurança usa uma política de segurança para determinar se é que a ação gerada será permitida ou negada.

Política de Segurança

Um conjunto de permissões definidas para classes diferentes de código. O Gerenciador de Segurança compara ações solicitadas por aplicativos em relação à política de segurança. Caso uma ação seja permitida pela política, o Gerenciador de Segurança permitirá que a ação seja efetuada. Caso a ação não seja permitida pela política, o Gerenciador de Segurança irá negar aquela ação. A política de segurança pode definir permissões baseadas na localização do código ou na assinatura do código.

O Gerenciador de Segurança e a política de segurança são configurados usando as opções da Máquina Virtual **`java.security.manager`** e **`java.security.policy`**.

Opções relacionadas ao Gerenciador de Segurança

java.security.manager

Usa um gerenciador de segurança, opcionalmente especificando qual gerenciador de segurança a ser usado. Caso nenhum argumento seja fornecido com esta opção, o gerenciador de segurança JDK padrão, **`java.lang.SecurityManager`**, será usado. Forneça o classname inteiramente qualificado de uma subclasse do **`java.lang.SecurityManager`** com esta opção, caso deseje usar outra implementação de segurança.

java.security.policy

Especifica um arquivo de política para argumentar ou substituir a política de segurança para a MV. Esta opção possui duas formas:

`java.security.policy=`*`policyFileURL`*

O arquivo de política referenciado *policyFileURL* irá *argumentar* a política de segurança padrão configurada pela MV.

`java.security.policy==`*`policyFileURL`*

O arquivo de política referenciado pelo *policyFileURL* irá *substiruir* a política de MV.

O valor *policyFileURL* pode ser um URL ou um caminho de arquivo.

A Plataforma do JBoss Enterprise não ativa o Gerenciador de Segurança por padrão. Use o Gerenciador de Segurança para configurar a Plataforma, conforme descrito na Seção 14.1, “Uso do Gerenciador de Segurança” .

14.1. USO DO GERENCIADOR DE SEGURANÇA

A Plataforma do Aplicativo JBoss Enterprise pode usar o Gerenciador de Segurança ou um gerenciador de segurança personalizado. Por favor refira-se às Opções relacionadas ao Gerenciador de Segurança para maiores informações sonre o Gerenciador de Segurança personalizado.

O arquivo da política de segurança deve ser especificado quando a Plataforma é configurada para uso de um gerenciador de segurança. O arquivo da política de segurança **`jboss-as/bin/server.policy.cert`** é adicionado como ponto inicial.

Por favor refira-se à Seção 14.3, “Gravação da Política de Segurança para a Plataforma do Aplicativo do JBoss Enterprise” para maiores informações sobre a gravação da política de segurança.

Arquivo de Configuração

O arquivo **`run.conf`** (Linux) ou **`run.conf.bat`** (Windows) é usado para configurar o Gerenciador de Segurança e a política de segurança. Este arquivo é encontrado no diretório **`jboss-as/bin`**.

Este arquivo é usado para configurar as opções à nível do sevidor e aplica-se a todos os perfis do servidor. A configuração do Gerenciador de Segurança e política de segurança envolve a configuração específica do perfil. Você pode eleger uma cópia do **`run.conf`** global ou arquivo **`run.conf.bat`** a partir do **`jboss-as/bin/`** para o perfil do servidor (por exemplo: **`jboss-as/server/production/run.conf`**) e realizar as alterações da configuração. O arquivo de configuração no perfil do servidor possui precedência sobre o arquivo **`run.conf`** / **`run.conf.bat`** global quando o perfil do servidor é iniciado.

Procedimento 14.1. Ativação do Gerenciador de Segurança

Este procedimento configura a Plataforma do Aplicativo JBoss Enterprise para inicializar o Gerenciador de Segurança do Java ativado.

As ações de edição do arquivo neste procedimento refere-se ao arquivo **`run.conf`** (Linux) ou **`run.conf.bat`** (Windows) no diretório do perfil do servidor, caso existir algum, ou no **`jboss-as/bin`**. Refira-se ao Arquivo de Configuração para maiores detalhes sobre a localização deste arquivo.

1. **Especificação do diretório da página principal do JBoss.**
   Edite o arquivo **`run.conf`** (Linux) ou **`run.conf.bat`** (Windows). Adicione a opção **`jboss.home.dir`**, especificando o caminho ao diretório **`jboss-as`** de sua instalação.

   Linux

   ```
   JAVA_OPTS="$JAVA_OPTS -Djboss.home.dir=/path/to/jboss-eap-5.1/jboss-as"
   ```

   Windows

   ```
   JAVA_OPTS="%JAVA_OPTS% -Djboss.home.dir=c:\path\jboss-eap-5.1\jboss-as"
   ```

2. **Especificação do diretório de página principal**
   Adicione a opção **`jboss.server.home.dir`**, especificando o caminho para seu perfil de servidor.

   Linux

   ■

```
JAVA_OPTS="$JAVA_OPTS -Djboss.server.home.dir=path/to/jboss-eap-
5.1/jboss-as/server/production"
```

Windows

```
JAVA_OPTS="%JAVA_OPTS% -Djboss.server.home.dir=c:\path\to\jboss-eap-
5.1\jboss-as\server\production"
```

3. **Especificação do Manuseador do Protocolo**
   Adicione a opção **`java.protocol.handler.pkgs`**, especificando o manuseador stub do JBoss.

   Linux

   ```
   JAVA_OPTS="$JAVA_OPTS -
   Djava.protocol.handler.pkgs=org.jboss.handlers.stub"
   ```

   Windows

   ```
   JAVA_OPTS="%JAVA_OPTS% -
   Djava.protocol.handler.pkgs=org.jboss.handlers.stub"
   ```

4. **Especificando a política de segurança para uso**
   Adicione a variável **`$POLICY`**, especificando a política de segurança em uso. Adicione a definição da variável antes da linha que ativa o Gerenciador de Segurança.

   Exemplo 14.1. Use a política de segurança incluída na Plataforma

   ```
   POLICY="server.policy.cert"
   ```

5. **Ativação do Gerenciador de Segurança**
   Descomente a linha seguinte pela remoção do # inicial:

   Linux

   ```
   #JAVA_OPTS="$JAVA_OPTS -Djava.security.manager -
   Djava.security.policy=$POLICY"
   ```

   Windows

   ```
   #JAVA_OPTS="%JAVA_OPTS% -Djava.security.manager -
   Djava.security.policy=%POLICY%"
   ```

   Resultado:

   A Plataforma do Aplicativo do JBoss Enterprise está configurada para iniciar com o Gerenciador de Segurança ativado.

6. **Opcional: Importação da chave de assinatura do JBoss da Red Hat**

A política de segurança incluída concede permissões ao código assinado do JBoss. Você deverá importar a chave de assinatura do JBoss para o armazenamento da chave **cacerts**, caso você use a política incluída.

O seguinte comando assume que o **JAVA_HOME** da variável do ambiente seja configurado à localização do JDK suportada pela Plataforma do Aplicativo JBoss Enterprise 5. Você deve configurar o **JAVA_HOME** quando instalar a Plataforma do Aplicativo Enterprise 5 pela primeira vez. Refira-se ao *Installation Guide* para maiores informações.

NOTA

Você pode usar o comando **alternatives** para selecionar a partir dos JDKs instalados em seu sistema Linux, com o objetivo de garantir de que o JVM correto está instalado. Refira-se ao Apêndice A, *Configuração do JDK padrão com a Utilidade* ***/usr/sbin/alternatives*** para maiores informações.

Execute o seguinte comando num terminal:

Linux

```
[~]$ sudo $JBOSS_HOME/bin/keytool -import -alias jboss -file
JBossPublicKey.RSA \
-keystore $JAVA_HOME/lib/security/cacerts
```

Windows

```
C:> $JBOSS_HOME\bin\keytool -import -alias jboss -file
JBossPublicKey.RSA -keystore $JAVA_HOME\lib\security\cacerts
```

Os comandos acima devem ser inseridos em uma única linha num terminal, embora separados em linhas diferentes neste terminal.

NOTA

A senha padrão para o armazenamento da chave cacerts é **changeit**.

Resultado:

A chave usada para a assinatura do código da Plataforma do Aplicativo JBoss Enterprise está instalada.

14.2. DEPURAÇÃO DOS PROBLEMAS DA POLÍTICA DE SEGURANÇA

Você pode ativar a informação de depuração para auxílio na solução dos problemas relacionados à política de segurança. A opção **java.security.debug** configura o nível de informação relacionada à segurança relatada.

O comando **java -Djava.security.debug=help** produzirá um resultado de ajuda com todas as opções de depuração. A configuração à nível de depuração para **all** é útil quando a solucionando o problema de uma falha relacionada com a segurança, cuja causa é totalmente desconhecida. No entanto, isto produzirá muita informação para uso geral. O padrão geral de sensibilidade é **access:failure**.

Procedimento 14.2. Ativação da depuração geral

Este procedimento irá ativar o nível geral de sensibilidade da informação de depuração relacionada com a segurança.

- Adicione a seguinte linha ao arquivo **`run.conf`** (Linux) ou **`run.conf.bat`** (Windows):

  Linux

  ```
  JAVA_OPTS="$JAVA_OPTS -Djava.security.debug=access:failure"
  ```

  Windows

  ```
  JAVA_OPTS="%JAVA_OPTS% -Djava.security.debug=access:failure"
  ```

14.2.1. Depuração do Gerenciador de Segurança

NOTA

O Gerenciador de Segurança de Depuração foi introduzido com a Plataforma do Aplicativo do JBoss Enterprise 5.1.

O Gerenciador de Segurança da Depuração **`org.jboss.system.security.DebuggingJavaSecurityManager`** imprime o domínio de proteção correspondente a uma permissão com falha. Esta informação adicional é bastante útil quando depurando os problemas de permissão.

Procedimento 14.3. Ativação do Gerenciador de Segurança de Depuração

Este procedimento ativará o Gerenciador de Segurança de Depuração.

1. Adicione a opção ao **`$JBOSS_HOME/bin/run.conf`** (Linux) ou **`$JBOSS_HOME/bin/run.conf.bat`**. Consulte o Arquivo de Configuração para a localização deste arquivo.

   Linux

   ```
   JAVA_OPTS="$JAVA_OPTS -
   Djava.security.manager=org.jboss.system.security.DebuggingJavaSecuri
   tyManager"
   ```

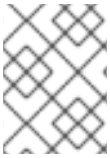
   Windows

   ```
   JAVA_OPTS="%JAVA_OPTS% -
   Djava.security.manager=org.jboss.system.security.DebuggingJavaSecuri
   tyManager"
   ```

2. Comente todas as referências **`java.security.manager`** no arquivo.

3. Certifique-se de que o arquivo ainda contém uma opção **`java.security.policy`** especificando o arquivo da política para uso.

4. Ative a depuração geral seguindo as instruções no Procedimento 14.2, “Ativação da depuração geral”.



NOTA

O Gerenciador de Segurança de Depuração possui um custo de execução significativo. Não recomenda-se o uso do mesmo na produção em geral.

14.3. GRAVAÇÃO DA POLÍTICA DE SEGURANÇA PARA A PLATAFORMA DO APLICATIVO DO JBOSS ENTERPRISE

O arquivo **`jboss-as/bin/server.policy.cert`** é um exemplo da política de segurança para a Plataforma do Aplicativo JBoss Enterprise. Você pode utilizá-lo como base para sua própria política de segurança.

O aplicativo **`policytool`**, incluído com o JDK, fornece uma ferramenta gráfica para edição e gravação da política de segurança.

IMPORTANTE

Considere cuidadosamente quais permissões você irá conceder. Tenha um cuidado especial ao conceder permissão ao **`java.security.AllPermission`**: você pode potencialmente permitir alterações ao binário do sistema, incluindo o ambiente do período de rodagem JVM.

Refira-se à documentação no http://download-llnw.oracle.com/javase/6/docs/technotes/guides/security/PolicyFiles.html para informações gerais sobre os arquivos da política de segurança e permissões Java. O **`java.lang.RuntimePermissions`** específico do JBoss está descrito abaixo:

Permissões do Período de Rodagem específico do JBoss

`org.jboss.security.SecurityAssociation.getPrincipalInfo`

Fornece acesso aos métodos **`org.jboss.security.SecurityAssociation getPrincipal()`** e **`getCredential()`**. O risco relacionado com o uso desta permissão do período de rodagem é a habilidade de verificar o chamador do segmento atual e credenciais.

`org.jboss.security.SecurityAssociation.getSubject`

Fornece acesso ao método **`org.jboss.security.SecurityAssociation getSubject()`**.

`org.jboss.security.SecurityAssociation.setPrincipalInfo`

Fornece acesso aos métodos **`org.jboss.security.SecurityAssociation setPrincipal()`**, **`setCredential()`**, **`setSubject()`**, **`pushSubjectContext()`** e **`popSubjectContext()`**. O risco de uso desta permissão do período de rodagem é a habilidade de configurar o chamador do segmento atual e credenciais.

`org.jboss.security.SecurityAssociation.setServer`

Fornece acesso ao método **`org.jboss.security.SecurityAssociation setServer`** . O risco de uso desta permissão do período de rodagem é a habilidade de ativar ou desativar o armazenamento múltiplo de segmentação do chamador da entidade de segurança do chamador e

credenciais.

org.jboss.security.SecurityAssociation.setRunAsRole

Fornece acesso aos métodos **org.jboss.security.SecurityAssociation pushRunAsRole** e **popRunAsRole**, **pushRunAsIdentity** e **popRunAsIdentity**. O risco de uso desta permissão do período de rodagem é a habilidade de alterar o chamador atual rodando como entidade de segurança da função.

org.jboss.security.SecurityAssociation.accessContextInfo

Fornece acesso aos métodos **org.jboss.security.SecurityAssociation accessContextInfo, "Get"** e **accessContextInfo, "Set"**, permitindo que você determine e obtenha a informação do contexto de segurança atual.

org.jboss.naming.JndiPermission

Fornece permissões especiais aos arquivos e diretórios num caminho da árvore JNDI ou recursivamente a todos os arquivos e subdiretórios. Um JndiPermission consiste de um pathname e de um conjunto de permissões válidas relacionadas ao arquivo ou diretório.

As permissões disponíveis incluem: **bind** , **rebind** , **unbind** , **lookup** , **list** , **listBindings** , **createSubcontext** e **all**.

Os pathnames terminando com **/*** indicam que as permissões especificadas são aplicadas a todos os arquivos e diretórios do pathname. Os pathnames terminando em **/-** indicam permissões a todos os arquivos e diretórios do pathname. Os pathnames que contém um token **<<ALL BINDINGS>>** especial combinando com qualquer arquivo em qualquer diretório.

org.jboss.security.srp.SRPPermission

A classe de permissão personalizada para proteção de acesso à informação SRP confidencial como a chave de sessão privada e chave privada. Esta permissão não possui quaisquer ações definidas. O getSessionKey também fornece acesso à chave de sessão privada resultada a partir da negociação SRP. O acesso a esta chave permitirá que você criptografe e descriptografe as mensagens que foram criptografadas com a chave de sessão.

org.hibernate.secure.HibernatePermission

Esta classe de permissão fornece permissões básicas para proteção das sessões do Hibernate. O destino desta propriedade é o nome da entidade. As informações disponíveis incluem: insert , delete , update , read , * (all).

org.jboss.metadata.spi.stack.MetaDataStackPermission

Fornece uma classe de permissão personalizada para controle de como os chamadores interagem com a pilha dos metadados. As permissões disponíveis são: **modify** (push/pop na pilha), **peek** (peek an pilha) e ***** (todos).

org.jboss.config.spi.ConfigurationPermission

Protege a determinação das propriedades de configuração. Define apenas os nomes de destino da permissão e não as ações. Os destinos para esta propriedade incluem: <property name> - propriedade pela qual o código possui permissão para configuração; * - todas as propriedades.

org.jboss.kernel.KernelPermission

Protege o acesso à configuração do kernel. Isto define apenas os nomes do destino de permissão e nenhuma das ações. Os destinos para esta propriedade incluem: access - acesso à configuração do kernel, configure - configuração do kernel (o acesso é imperativo) e * - todos acima.

`org.jboss.kernel.plugins.util.KernelLocatorPermission`

Protege acesso ao kernel. Define apenas os nomes do destino de permissão e nenhuma ação. Os destinos para esta propriedade incluem: kernel - acesso ao kernel e * - acesso a todas as áreas.

CAPÍTULO 15. PROTEÇÃO DA CAMADA DE TRANSPORTE EJB RMI

O Servidor do Aplicativo JBoss usa uma camada do invocador baseado no soquete para a Invocação do Método Remoto - Remote Method Invocation (RMI) do EJB2 e EJB3 Beans. Este tráfego de rede não é criptografado por padrão. Siga as instruções deste capítulo para uso da Camada de Soquetes de Segurança - Secure Sockets Layer (SSL)

Opções de transporte para o EJB3 através do SSL

Este capítulo descreve a configuração de dois planos diferentes para o RMI dos EJB3s sobre um transporte criptografado: RMI + SSL e RMI através dos HTTPS. O HTTPS é uma opção de transporte para o RMI quando a configuração do firewall previne o uso das portas RMI.

Maiores informações referentes a geração de um par chave para o SSL podem ser encontradas na Seção 15.2, “Geração de chaves de criptografia e certificado”.

Informações sobre a configuração do RMI + SSL podem ser encontradas na Seção 15.3, “Configuração EJB3 RMI + SSL”.

Informações referentes ao RMI através do HTTPS para o EJB3 estão descritas na Seção 15.4, “EJB3 RMI através da Configuração HTTPS”.

Informações sobre o RMI + SLL para o EJB2 podem ser encontradas na Seção 15.5, “Configuração EJB2 RMI + SSL”.

15.1. VISÃO GERAL DE CRIPTOGRAFIA DO SSL

15.1.1. Pares Chaves e Certificados

O (SSL) encripta o tráfego de rede entre os dois sistemas. O tráfego entre dois sistemas é encriptado usando uma chave bidirecional, gerada durante a fase *handshake* da conexão e conhecida por estes dois sistemas.

Para a troca de segurança de uma chave de criptografia bidirecional, o SSL usa uma Infra-estrutura de Chave Pública - Public Key Infrastructure (PKI), um método de criptografia que usa um *key pair*. Um par chave consiste de duas chaves criptografadas separadas, porém coincidentes: a chave pública e a chave privada. A chave pública é compartilhada com outros e é usada para criptografar dados, enquanto que a chave privada é secreta e usada para descriptografar dados que já foram criptografados usando a chave pública.

Quando o cliente solicita a conexão de segurança, uma fase handshake entra em ação antes da comunicação de segurança possa começar. Durante o SSL handshake, o servidor passa a chave pública ao cliente na forma de um certificado. O certificado contém a identidade de um servidor (seu URL), a chave pública e uma assinatura digital que valida o certificado. Em seguida, o cliente valida o certificado e decide se o certificado é confiável ou não. Caso o certificado seja confiável, o cliente gera uma chave de criptografia bidirecional para a conexão SSL, encripta-a usando a chave pública e a envia de volta ao servidor. O servidor decripta a chave de criptografia bidirecional, usando sua própria chave privada e comunicação futura entre as duas máquinas sobre esta conexão é encriptada usando a chave de criptografia bidirecional.

No lado do servidor, os pares de chave privada/pública são armazenados num *key store*, um arquivo encriptado que armazena os pares de chave e os certificados confiáveis. Cada par de chave com o armazenamento de chave é identificado por um *alias* - um nome único que é usado no armazenamento ou solicitação de um par chave do armazenamento chave. A chave pública está distribuída aos clientes

na forma de um *certificate*, uma assinatura digital que aplica o bind juntamente com uma chave pública e uma identidade. No lado do cliente, os certificados da validação conhecida são mantidos no armazenamento de chave padrão conhecido como um *trust store*.

Certificados de auto-assinatura e assinatura CA

A Infra-estrutura da Chave Pública depende de uma corrente de confiança para estabelecer as credenciais de máquinas desconhecidas. O uso de chaves públicas não encripta apenas o tráfego entre máquinas, mas também trabalha para estabelecer a identidade da máquina de outra parte da conexão de rede. Uma "Confiança da Web" é usada para verificar a identidade dos servidores. Você pode desconhecer um servidor, mas caso sua chave pública é assinada por alguém de sua confiança, você estende aquela confiança ao servidor. As Autoridades de Certificado são entidades comerciais que verificam a identidade de clientes e emitem isto como certificados assinados. O JDK inclui um arquivo **`cacerts`** com os certificados de diversas Autoridades de Certificado - Certificate Authorities (CAs). Quaisquer chaves assinadas por estes CAs são automaticamente confiáveis. Neste caso, o certificado assinado da Autoridade de Certificado interna é normalmente instalada em clientes como parte da Construção Padrão Empresarial e todos os certificados assinados com aquele certificado serão confiáveis. Os certificados assinados CA são a melhor prática para cenários de produção.

Durante o desenvolvimento e teste, ou para cenários de produção interna apenas ou de escala menor, você pode usar um *self-signed certificate*. Este é um certificado que não é assinado pela Autoridade de Certificado, mas sim um certificado gerado localmente. Uma vez que o certificado gerado localmente não está no arquivo **`cacerts`** dos clientes, você precisa expor um certificado para aquela chave no servidor e importar aquele certificado em qualquer cliente que conecta-se através do SSL.

O JDK inclui **`keytool`**, uma ferramenta de linha de comando para geração de pares chaves e certificados. Os certificados gerados pelo **`keytool`** podem ser enviados para assinatura do CA ou podem ser distribuídos aos clientes como um certificado de auto-assinatura.

- Maiores informações sobre a geração de um certificado de auto-assinatura para uso de desenvolvimento e importação do mesmo certificado podem ser encontradas na Seção 15.2.1, “Geração de um certificado auto-assinado com o keytool”.

- A geração de um certificado e possuir o mesmo assinado por um CA para uso de produção vai além do descrito nesta edição. Por favor refira-se ao manpage em keytool para maiores informações sobre a execução desta tarefa.

15.2. GERAÇÃO DE CHAVES DE CRIPTOGRAFIA E CERTIFICADO

15.2.1. Geração de um certificado auto-assinado com o keytool

15.2.1.1. Geração de um par chave

O comando **keytool**, parte do JDK, é usado para gerar um novo par chave. O keytool pode tanto adicionar um novo par a um novo armazenamento existente ou criar um novo armazenamento ao mesmo tempo que o par chave.

Este par chave será usado para negociar uma criptografia SSL entre o servidor e clientes remotos. O seguinte procedimento gera um par chave e armazena-o num armazenamento chave chamado **`localhost.keystore`**. Você precisará disponibilizar este armazenamento ao invocador EJB3 no servidor. O par chave na nossa amostra será salvo no armazenamento chave sob o alias 'ejb-ssl'. Nós precisaremos deste alias chave, e a senha do par chave que você fornecerá (se alguma), quando configurando o conector Remoto EJB3 no Crie um conector remoto de segurança para o RMI.

Procedimento 15.1. Gere o novo par chave e adicione-o ao "localhost.keystore" do armazenamento chave no diretório config do servidor do JBoss.

Este procedimento gera um novo par chave para a criptografia SSL.

- O seguinte comando criará um par chave de uso com a criptografia SSL:

```
keytool -genkey -alias ejb-ssl -keystore localhost.keystore -
storepass KEYSTORE_PASSWORD
  -keypass EJB-SSL_KEYPAIR_PASSWORD
  -dname "CN=SERVER_NAME,OU=QE,O=example.com,L=Brno,C=CZ"
```

Resultado:

Um novo par chave será adicionado ao **`localhost.keystore`** de armazenamento chave sob o alias **`ejb-ssl`**.

Os parâmetros para este comando podem ser encontrados no Parâmetros **`keytool`**

Parâmetros `keytool`

alias

Um token alfanumérico usado para identificar o par chave com o armazenamento chave. O armazenamento chave pode conter as chaves múltiplas. O alias fornece um significado para exclusivamente identificar um par chave com um armazenamento chave. O alias para um par chave deve ser único com o armazenamento chave.

keystore

O par chave que será usado para armazenar o par chave. Isto pode ser um caminho de arquivo absoluto ou relativo.

storepass

A senha para um armazenamento chave. Caso o armazenamento chave existir, deverá existir também uma senha para o armazenamento chave. Caso a senha do armazenamento chave especificada não existir ainda, ela poderá ser criada e esta será a nova senha. Esta senha é necessária para acessar o armazenamento chave para recuperar ou armazenar chaves e certificados.

keypass

A senha para o novo par chave. Esta senha deve ser fornecida para uso de um par chave no futuro.

dname

Os detalhes de identificação do certificado.

CN

Nome Comum: o nome do servidor. Ele deve coincidir com o nome do servidor conforme retornado aos clientes numa pesquisa JNDI. Caso um cliente tente realizar uma conexão SSL no servidor usando um nome do JNDI e receber um certificado com um nome diferente, a conexão falhará.

OU

Unidade Organizacional: o nome da unidade organizacional que é responsável pelo servidor.

O

Organização: O nome da organização, às vezes expressado como um URL.

L

Localização: a localização do servidor.

C

País: duas letras de código do país

NOTA

Armazene os arquivos de armazenamento chave num sistema de arquivo de segurança, de leitura apenas pelo proprietário do processo do Servidor do Aplicativo JBoss, para uma melhor prática de proteção.

Perceba que caso nenhum armazenamento for especificado na linha de comando, o **keytool** adiciona o novo armazenamento chave chamado **keystore**, no diretório principal do usuário atual. Este arquivo de armazenamento chave é um arquivo oculto.

15.2.1.2. Exportação de um certificado auto-assinado

Uma vez que um par chave é gerado ao servidor para uso, um certificado deve ser criado. O Procedimento 15.2, “Exportação de um certificado” descreve os passos de exportação da chave **ejb-ssl** a partir do armazenamento chave nomeado **localhost.keystore**.

Procedimento 15.2. Exportação de um certificado

Este procedimento exporta um certificado a partir de um armazenamento chave num arquivo.

1. Emita o seguinte comando:

```
keytool -export -alias ejb-ssl -file mycert.cer -keystore
localhost.keystore
```

2. Insira a senha do armazenamento chave

Resultado:

O certificado é exportado ao **mycert.cer** do arquivo.

15.2.2. Configure um cliente para aceitar um certificado do servidor auto-assinado

O cliente precisa confiar o certificado do servidor para realizar invocações de método remoto sobre o SSL. O certificado que geramos é auto-assinado e não possui uma corrente de confiabilidade para uma autoridade de certificado conhecido. O cliente deve estar claramente configurado para confiar o certificado, do contrário a conexão falhará. Importe o certificado do servidor auto-assinado a um *trust store* no cliente com o objetivo de configurar um cliente para confiar um certificado auto-assinado.

Um armazenamento de confiança é um armazenamento chave que contém o certificados confiáveis. Os certificados que localizam-se no armazenamento de confiança local são aceitos como válidos. Caso seu servidor usar um certificado auto-assinado, qualquer cliente que realizar chamadas do método remoto

no SSL solicitará o certificado em seu próprio armazenamento de segurança. Exporte sua chave pública como um certificado e importe este certificado ao armazenamento de confiança naqueles clientes.

O certificado criado na Seção 15.2.1.2, “Exportação de um certificado auto-assinado” deve ser copiado ao cliente com o objetivo de executar os passos detalhados no Procedimento 15.3, “Importe o certificado ao "localhost.truststore" do armazenamento de confiança”.

Procedimento 15.3. Importe o certificado ao "localhost.truststore" do armazenamento de confiança

Este procedimento importa um certificado que foi exportado anteriormente ao armazenamento de confiança num cliente.

1. Emita o seguinte comando no cliente:

```
keytool -import -alias ejb-ssl -file mycert.cer -keystore
localhost.truststore
```

2. Entre a senha para este armazenamento de confiança caso ele já exista. Do contrário, insira e re-insira a senha para um novo armazenamento de confiança.

3. Verifique os detalhes do certificado. Caso seja o correto, digite "sim" para importá-lo ao armazenamento de confiança.

   Resultado:

   O certificado é importado ao armazenamento de confiança e uma conexão de segurança pode ser estabelecida com o servidor que usa este certificado.

Assim como o armazenamento chave, caso o armazenamento de confiança especificado ainda não existir, ele será criado. No entanto, diferente do armazenamento chave, não há armazenamento de confiança padrão e o comando falha caso isto não seja especificado.

Configure o cliente para uso do localhost.truststore

Após importar o certificado do servidor de auto-assinatura a um armazenamento de confiança no cliente, você deverá instruir o cliente a usar este armazenamento de confiança. Para isto, passe a localização **`localhost.truststore`** ao aplicativo usando a propriedade **`javax.net.ssl.trustStore`** e a senha de armazenamento de confiança usando a propriedade **`javax.net.ssl.trustStorePassword`**. O Exemplo 15.1, “Invocação do aplicativo com.acme.Runclient com um armazenamento de confiança específico” é uma amostra de comando que invoca o **com.acme.RunClient** do aplicativo, um aplicativo hipotético que realiza chamadas de métodos remotas a um EJB no Servidor do Aplicativo JBoss. Este comando é executado a partir do root do diretório do pacote do aplicativo (o diretório contendo o **`com`** no **`com/acme/RunClient.class`** do caminho do arquivo).

Exemplo 15.1. Invocação do aplicativo com.acme.Runclient com um armazenamento de confiança específico

```
java -cp $JBOSS_HOME/client/jbossall-client.jar:. -
Djavax.net.ssl.trustStore=${resources}/localhost.truststore \
    -Djavax.net.ssl.trustStorePassword=TRUSTSTORE_PASSWORD
com.acme.RunClient
```

15.3. CONFIGURAÇÃO EJB3 RMI + SSL

Procedimento 15.4. Configure o RMI + SSL para uma Visão Geral do EJB3

Este procedimento configura a criptografia SSL do tráfego de Invocação do Método Remoto entre os EJB3 beans no servidor e um cliente rodando em outra máquina da rede.

1. Geração de chaves de criptografia e certificado
2. Configure um conector remoto de segurança para o RMI
3. Anote os EJB3 beans para uso do conector RMI de segurança

Maiores informações sobre os certificados e chaves de criptografia podem ser encontradas na Seção 15.2, “Geração de chaves de criptografia e certificado”.

Crie um conector remoto de segurança para o RMI

O **`ejb3-connectors-jboss-beans.xml`** do arquivo num diretório **`deploy`** do perfil do Servidor do Aplicativo JBoss contém as definições do conector do JBoss Remoting para a invocação de método remoto EJB3.

Exemplo 15.2. Amostra do Conector EJB3 de Segurança

Os beans descritos na amostra do código são anexados ao arquivo **`ejb3-connectors-jboss-beans.xml`**. Ambos os beans são solicitados para configurar um conector de segurança para o EJB3 usando o par chave criado no Procedimento 15.1, “Gere o novo par chave e adicione-o ao "localhost.keystore" do armazenamento chave no diretório config do servidor do JBoss.”.

A propriedade **`keyPassword`** na configuração de amostra é a senha especificada do par chave quando o par chave foi criado.

A configuração de amostra cria um conector que escuta as conexões SSL na porta 3843. Esta porta precisa ser aberta no firewall do servidor para acesso aos clientes.

```
<bean name="EJB3SSLRemotingConnector"
class="org.jboss.remoting.transport.Connector">
    <property
name="invokerLocator">sslsocket://${jboss.bind.address}:3843</property>
    <property name="serverConfiguration">
      <inject bean="ServerConfiguration" />
    </property>
    <property name="serverSocketFactory">
      <inject bean="sslServerSocketFactory" />
    </property>
  </bean>

  <bean name="sslServerSocketFactory"
class="org.jboss.security.ssl.DomainServerSocketFactory">
     <constructor>
        <parameter><inject bean="EJB3SSLDomain"/></parameter>
     </constructor>
  </bean>
  <bean name="EJB3SSLDomain"
class="org.jboss.security.plugins.JaasSecurityDomain">
     <constructor>
```

```
        <parameter>EJB3SSLDomain</parameter>
    </constructor>
    <property name="keyStoreURL">resource:localhost.keystore</property>
    <property name="keyStorePass">KEYSTORE_PASSWORD</property>
    <property name="keyAlias">ejb-ssl</property>
    <property name="keyPassword">EJB-SSL_KEYPAIR_PASSWORD</property>
</bean>
```

Configuração dos Beans EJB3 para o Transporte SSL

Todos os EJB3 beans usam o conector RMI por padrão. Anote o bean com o **`@org.jboss.annotation.ejb.RemoteBinding`** para ativar a invocação remota de um bean através do SSL.

Exemplo 15.3. A anotação EJB3 bean para ativar a invocação remota de segurança

A anotação aplica o bind num EJB3 bean para o **`StatefulSSL`** do nome JNDI. A implementação proxy da interface remota, retornada a um cliente quando o bean é solicitado a partir do JNDI, comunica-se com o servidor através do SSL.

```
@RemoteBinding(clientBindUrl="sslsocket://0.0.0.0:3843",
jndiBinding="StatefulSSL")
  @Remote(BusinessInterface.class)
  public class StatefulBean implements BusinessInterface
  {
     ...
  }
```

NOTA

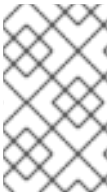
O endereço IP é especificado como 0.0.0.0, significando "todas as interfaces" no Exemplo 15.3, “A anotação EJB3 bean para ativar a invocação remota de segurança”. Altere isto para o valor da propriedade do sistema ${jboss.bind.address}.

Ativação de ambas invocações de segurança e insegurança de um EJB3 bean

Você pode ativar ambas invocações do método remoto de segurança e insegurança. O Exemplo 15.4, “Anotação do EJB3 para invocação de segurança e sem segurança” demonstra as anotações para isto.

Exemplo 15.4. Anotação do EJB3 para invocação de segurança e sem segurança

```
 @RemoteBindings({
     @RemoteBinding(clientBindUrl="sslsocket://0.0.0.0:3843",
jndiBinding="StatefulSSL")
     @RemoteBinding(jndiBinding="StatefulNormal")
  })
  @Remote(BusinessInterface.class)
  public class StatefulBean implements BusinessInterface
  {
     ...
  }
```



NOTA

O endereço Ip é especificado como **`0.0.0.0`**, significando "todas as interfaces" no Exemplo 15.4, “Anotação do EJB3 para invocação de segurança e sem segurança”. Altere isto para o valor da propriedade de sistema ${jboss.bind.address}.

Caso os clientes solicitarem o **`StatefulNormal`** a partir do JNDI, o roxy retornado implementando a interface remota comunica-se com o servidor através do protocolo do soquete descriptografado. Caso o **`StatefulSSL`** for solicitado, o proxy retornado implementando a interface remota comunica-se com o servidor através do SSL.

15.4. EJB3 RMI ATRAVÉS DA CONFIGURAÇÃO HTTPS

Procedimento 15.5. Configure o EJB3 RMI através da Visão Geral HTTPS

Este procedimento configura o encapsulamento do tráfego de Invocação do Método Remoto sobre o HTTP criptografado do SSL. Isto possui efeito duplo da criptografia do tráfego, além de permitir a transversão dos firewalls que bloqueiam a porta RMI.

1. Gera as chaves de criptografia e certificados.
2. Configura o RMI através do conector da web HTTPS.
3. Configura Servlets.
4. Configura o conector remoto de segurança para o RMI através do HTTPS.
5. Configura os EJB3 beans para o transporte HTTPS.
6. Configura clientes para o RMI através do HTTPS.

Maiores informações sobre os certificados e chaves de criptografia podem ser encontradas na Seção 15.2, “Geração de chaves de criptografia e certificado”.

Procedimento 15.6. Configuração do RMI através do conector da web HTTPS.

Este procedimento cria um conector da web que escuta na porta 8443 e aceita conexões SSL dos clientes.

- Edite o **`jboss-as/server/$PROFILE/deploy/jbossweb.sar/server.xml`** do arquivo e descomente o conector HTTPS.

```
<!-- SSL/TLS Connector configuration using the admin devl guide
keystore -->
<Connector protocol="HTTP/1.1" SSLEnabled="true"
   port="8443" address="${jboss.bind.address}"
   scheme="https" secure="true" clientAuth="false"
   keystoreFile="${jboss.server.home.dir}/conf/localhost.keystore"
   keystorePass="KEYSTORE_PASSWORD" sslProtocol = "TLS" />
```

Resultado:

Você criou um conector da web para aceitar as conexões SSL.

Procedimento 15.7. Configuração de Servlets

Este procedimento configura um servlet que passa solicitações a partir de um conector da web para o **`ServletServerInvoker`**.

1. Crie um diretório nomeado **`servlet-invoker.war`** no **`jboss-as/server/$PROFILE/deploy/`**.

2. Cire um diretório **`WEB-INF`** no diretório **`servlet-invoker.war`**.

3. Crie um arquivo nomeado **`web.xml`** naquele diretório **`WEB-INF`** com o seguinte conteúdo:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">


<web-app>
<servlet>
<servlet-name>ServerInvokerServlet</servlet-name>
<description>The ServerInvokerServlet receives requests via HTTP
protocol from within a web container and passes it onto the
ServletServerInvoker for processing.
</description>
<servlet-
class>org.jboss.remoting.transport.servlet.web.ServerInvokerServlet<
/servlet-class>

<init-param>
<param-name>locatorUrl</param-name>
<param-value>servlet://${jboss.bind.address}:8080/servlet-
invoker/ServerInvokerServlet</param-value>
<description>The servlet server invoker</description>
</init-param>

<load-on-startup>1</load-on-startup>
</servlet>


<servlet>
<servlet-name>SSLServerInvokerServlet</servlet-name>
<description>The ServerInvokerServlet receives requests via HTTPS
protocol from within a web container and passes it onto the
ServletServerInvoker for processing.
</description>
<servlet-
class>org.jboss.remoting.transport.servlet.web.ServerInvokerServlet<
/servlet-class>

<init-param>
<param-name>locatorUrl</param-name>
<param-value>sslservlet://${jboss.bind.address}:8443/servlet-
invoker/SSLServerInvokerServlet</param-value>
<description>The servlet server invoker</description>
</init-param>
```

```
<load-on-startup>2</load-on-startup>
</servlet>

<servlet-mapping>
<servlet-name>ServerInvokerServlet</servlet-name>
<url-pattern>/ServerInvokerServlet/*</url-pattern>
</servlet-mapping>

<servlet-mapping>
<servlet-name>SSLServerInvokerServlet</servlet-name>
<url-pattern>/SSLServerInvokerServlet/*</url-pattern>
</servlet-mapping>

</web-app>
```

Resultado:

Você criou um servlet para envio de solicitações SSL do recipiente da web a um chamador do servidor.

O **`locatorUrl`** é usado para conectar o servlet ao conector remoto através do atributo **`InvokerLocator`** do conector remoto definido no Procedimento 15.8, “Configuração do conector remoto de segurança para RMI através do HTTPS” .

Procedimento 15.8. Configuração do conector remoto de segurança para RMI através do HTTPS

Este procedimento cria um Chamador do Servidor que implementa o RMI.

- Crie um arquivo nomeado **`servlet-invoker-service.xml`** no **`jboss-as/server/$PROFILE/deploy/`**, com o seguinte conteúdo:

```
<?xml version="1.0" encoding="UTF-8"?>

<server>
   <mbean code="org.jboss.remoting.transport.Connector"
name="jboss.remoting:service=connector,transport=servlet"
    display-name="Servlet transport Connector">
      <attribute
name="InvokerLocator">servlet://${jboss.bind.address}:8080/servlet-
invoker/ServerInvokerServlet</attribute>
      <attribute name="Configuration">
         <handlers>
            <handler
subsystem="AOP">org.jboss.aspects.remoting.AOPRemotingInvocationHand
ler</handler>
         </handlers>
      </attribute>
   </mbean>

   <mbean code="org.jboss.remoting.transport.Connector"
name="jboss.remoting:service=connector,transport=sslservlet"
    display-name="Servlet transport Connector">
      <attribute
name="InvokerLocator">sslservlet://${jboss.bind.address}:8443/servle
t-invoker/SSLServerInvokerServlet</attribute>
```

```
      <attribute name="Configuration">
         <handlers>
            <handler
subsystem="AOP">org.jboss.aspects.remoting.AOPRemotingInvocationHand
ler</handler>
         </handlers>
      </attribute>
   </mbean>
</server>
```

Resultado:

Você criou um conector remoto que pode receber solicitações a partir do servlet e chamar métodos de um EJB3.

Procedimento 15.9. Configuração dos EJB3 beans para transporte HTTPS

Este procedimento configura o EJB3 para aplicar o bind ao transporte HTTPS

- Anote o bean para o RMI através do HTTPS:

  Exemplo 15.5. Anotação de um EJB3 através do HTTPS

  ```
  // RMI tunneled over HTTPS
  @Stateless
  @RemoteBinding(clientBindUrl = "https://0.0.0.0:8443/servlet-
  invoker/SSLServerInvokerServlet")
  @Remote(Calculator.class)
  @SecurityDomain("other")
  public class CalculatorHttpsBean implements Calculator
  {
  ....
  ```

  Resultado:

  O EJB3 está disponível para invocação remota através do HTTPS.

Anotação de um bean para RMI através do HTTP

Você pode anotar opcionalmente o bean de invocação através do RMI através do HTTP. Este procedimento pode ser útil para testes, uma vez que isto permite que você passe chamadas RMI através de firewalls que bloqueiam portas RMI, mas que removem a camada extra da configuração de segurança.

Exemplo 15.6. Anotação de um bean para RMI através do HTTP

```
// RMI tunneled over HTTP
@Stateless
@RemoteBinding(clientBindUrl = "http://0.0.0.0:8080/servlet-
invoker/ServerInvokerServlet")
@Remote(Calculator.class)
@SecurityDomain("other")
public class CalculatorHttpBean extends CalculatorImpl
{
....
```

Configure clientes para o RMI através do HTTPS

O cliente EJB deve usar as seguintes propriedades para a busca JNDI quando pesquisando por beans:

O acesso do cliente ao RMI através do HTTP(S)

HTTPS

```
Properties props = new Properties();
props.put("java.naming.factory.initial",
"org.jboss.naming.HttpNamingContextFactory");
props.put("java.naming.provider.url",
"https://localhost:8443/invoker/JNDIFactory");
props.put("java.naming.factory.url.pkgs", "org.jboss.naming");
Context ctx = new InitialContext(props);
props.put(Context.SECURITY_PRINCIPAL, username);
props.put(Context.SECURITY_CREDENTIALS, password);
Calculator calculator = (Calculator) ctx.lookup(jndiName);
// use the bean to do any operations
```

HTTP

```
Properties props = new Properties();
props.put("java.naming.factory.initial",
"org.jboss.naming.HttpNamingContextFactory");
props.put("java.naming.provider.url",
"http://localhost:8080/invoker/JNDIFactory");
props.put("java.naming.factory.url.pkgs", "org.jboss.naming");
Context ctx = new InitialContext(props);
props.put(Context.SECURITY_PRINCIPAL, username);
props.put(Context.SECURITY_CREDENTIALS, password);
Calculator calculator = (Calculator) ctx.lookup(jndiName);
// use the bean to do any operations
```

Os valores *user name* e *password* correspondem a um nome do usuário e senha para o domínio de segurança que é usado para protegero chamador do http, no O acesso do cliente ao RMI através do HTTP(S). O domínio de segurança está configurado no **`jboss-as/$PROFILE/deploy/http-invoker.sar/invoker.war/WEB-INF/jboss-web.xml`** .

15.5. CONFIGURAÇÃO EJB2 RMI + SSL

Procedimento 15.10. Configuração SSL para Visão Geral do EJB2

1. Geração de chaves de criptografia e certificado
2. Configuração do Chamador Unificado para o SSL

Maiores informações sobre os certificados e chaves de criptografia podem ser encontradas na Seção 15.2, “Geração de chaves de criptografia e certificado”.

Configuração do Chamador Unificado para o SSL

A invocação remota do EJB2 usa um chamador unificado único, que roda por padrão na porta 4446. A configuração do chamador unificado usado para a invocação remota do EJB2 está definida no arquivo **$JBOSS_HOME/server/deploy/remoting-jboss-beans.xml** de um perfil do Servidor do Aplicativo JBoss. Adicione o seguinte SSL Socket Factory bean e um SSL Domain bean neste arquivo.

Exemplo 15.7. Fábrica do Servidor SSL para o EJB2

```
<bean name="sslServerSocketFactoryEJB2"
class="org.jboss.security.ssl.DomainServerSocketFactory">
  <constructor>
    <parameter><inject bean="EJB2SSLDomain"/></parameter>
  </constructor>
</bean>

<bean name="EJB2SSLDomain"
class="org.jboss.security.plugins.JaasSecurityDomain">
  <constructor>
    <parameter>EJB2SSLDomain</parameter>
  </constructor>
  <property name="keyStoreURL">resource:localhost.keystore</property>
  <property name="keyStorePass">changeit</property>
  <property name="keyAlias">ejb-ssl</property>
  <property name="keyPassword">EJB-SSL_KEYPAIR_PASSWORD</property>
</bean>
```

O próximo passo é personalizar o SSLSocketBuilder, pela adição do seguinte ao arquivo **$JBOSS_HOME/server/*$PROFILE*/conf/jboss-service.xml** de um perfil do Servidor do Aplicativo JBoss:

Exemplo 15.8. Configuração SSLSocketBuilder

```
<!-- This section is for custom (SSL) server socket factory  -->
   <mbean code="org.jboss.remoting.security.SSLSocketBuilder"
      name="jboss.remoting:service=SocketBuilder,type=SSL"
      display-name="SSL Server Socket Factory Builder">
      <!-- IMPORTANT - If making ANY customizations, this MUST be set
to false. -->
      <!-- Otherwise, will used default settings and the following
attributes will be ignored. -->
      <attribute name="UseSSLServerSocketFactory">false</attribute>
      <!-- This is the url string to the key store to use -->
      <attribute name="KeyStoreURL">localhost.keystore</attribute>
      <!-- The password for the key store -->
      <attribute name="KeyStorePassword">sslsocket</attribute>
      <!-- The password for the keys (will use KeystorePassword if this
is not set explicitly. -->
      <attribute name="KeyPassword">sslsocket</attribute>
      <!-- The protocol for the SSLContext.  Default is TLS. -->
      <attribute name="SecureSocketProtocol">TLS</attribute>
      <!-- The algorithm for the key manager factory.  Default is
SunX509. -->
      <attribute name="KeyManagementAlgorithm">SunX509</attribute>
      <!-- The type to be used for the key store. -->
      <!-- Defaults to JKS.  Some acceptable values are JKS (Java
```

```
Keystore - Sun's keystore format), -->
      <!-- JCEKS (Java Cryptography Extension keystore - More secure
version of JKS), and -->
      <!-- PKCS12 (Public-Key Cryptography Standards #12
                  keystore - RSA's Personal Information Exchange Syntax
Standard). -->
      <!-- These are not case sensitive. -->
      <attribute name="KeyStoreType">JKS</attribute>
   </mbean>

  <mbean
code="org.jboss.remoting.security.SSLServerSocketFactoryService"
     name="jboss.remoting:service=ServerSocketFactory,type=SSL"
     display-name="SSL Server Socket Factory">
     <depends optional-attribute-name="SSLSocketBuilder"
        proxy-
type="attribute">jboss.remoting:service=SocketBuilder,type=SSL</depends>
  </mbean>
```

Configuração do Transporte SSL para Beans

Atualize o código para refletir na informação abaixo, no arquivo **`deploy/remoting-jboss-beans.xml`** do perfil do Servidor do Aplicativo JBoss:

Exemplo 15.9. Transporte SSL para Beans

```
...
<bean name="UnifiedInvokerConnector"
class="org.jboss.remoting.transport.Connector">
<annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX(name="jboss.re
moting:service=Connector,transport=socket",
exposedInterface=org.jboss.remoting.transport.ConnectorMBean.class,regis
terDirectly=true)
</annotation>
<property name="serverConfiguration"><inject
bean="UnifiedInvokerConfiguration"/></property>
<property name="serverSocketFactory"><inject
bean="sslServerSocketFactoryEJB2"/></property>
<!-- add this to configure the SSL socket for the UnifiedInvoker -->
</bean>
...
<bean name="UnifiedInvokerConfiguration"
class="org.jboss.remoting.ServerConfiguration">
<constructor>
<!-- transport: Others include sslsocket, bisocket, sslbisocket, http,
https, rmi, sslrmi, servlet, sslservlet. -->
<parameter>sslsocket</parameter><!-- changed from socket to sslsocket --
>
</constructor>
...
</bean>
...
```

CAPÍTULO 16. MASCARANDO SENHAS NA CONFIGURAÇÃO XML

Siga as instruções abaixo neste capítulo para aumentar a segurança de sua Instalação do Aplicativo JBoss Enterprise mascarando senhas que seriam armazenadas no sistema de arquivo como textos simples.

16.1. VISÃO GERAL DE MASCARAMENTO DA SENHA

As senhas são tokens de autenticação secretos que são usados para limitar o acesso aos recursos para apenas as partes autorizadas. A senha deve ser disponibilizada para o serviço do JBoss, com o objetivo dos serviços do JBoss acessarem os recursos protegidos por senha. Isto pode ser realizado através de argumentos na linha de comando passados ao Servidor do Aplicativo do JBoss no período de inicialização, no entanto isto não é prático num ambiente de produção. Normalmente, as senhas são disponibilizadas em ambientes de produção aos serviços do JBoss através da inclusão nos arquivos de configuração.

Todos os arquivos de configuração da Plataforma do JBoss Enterprise devem ser armazenados nos sistemas do arquivo de segurança e devem possuir autorização de leitura pelo processo proprietário do Servidor do Aplicativo JBoss. Além disso, você pode mascarar a senha no arquivo de configuração para um nível adicional de segurança. Siga as instruções abaixo neste capítulo para substituir uma senha de texto simples numa configuração de bean Microcontainer por uma máscara de senha. Refira-se ao Capítulo 17, *Criptografia das Senhas de Fonte de Dados* para instruções sobre as senhas de Fonte de Dados criptografadas, Capítulo 18, *Criptografia da Senha Keystore num Conector Tomcat* para instruções sobre a criptografia da senha de armazenamento de chave no Tomcat e Capítulo 19, *Uso do LdapExtLoginModule com o JaasSecurityDomain* para instruções sobre a criptografia de senha para o LdapExtLoginModule.

NOTA

A segurança impenetrável não existe na realidade. Todas as medidas de segurança aumentam muito pouco o custo relacionado ao acesso não autorizado a um sistema. Não há exceções para o mascaramento de senhas - isto não é impenetrável, mas vence a inspeção casual de arquivos de configuração e aumenta o esforço necessário para extração de uma senha em texto simples.

Procedimento 16.1. Visão geral de uma senha de texto simples

1. Gera um par de chaves para uso da criptografia de senhas.
2. Encripta a senha do armazenamento chave.
3. Cria máscaras de senha.
4. Substitui as senhas de texto simples pelas máscaras de senha.

16.2. GERA UM ARMAZENAMENTO DE CHAVE E UMA SENHA MASCARADA.

O mascaramento de senha usa um par de chave público/privado para as senhas criptografadas. Você precisa gerar um par de chave para uso com o mascaramento de senha. Por padrão, a Plataforma do Aplicativo JBoss Enterprise 5 espera por um par de chaves com o alias **`jboss`** num armazenamento de chave no **`jboss-as/bin/password/password.keystore`**.

Os seguintes procedimentos seguem a configuração padrão. Caso você deseje alterar a localização do armazenamento da chave ou o alias da chave, você precisará alterar a configuração padrão e deverá referir-se à Seção 16.6, “Alteração dos padrões da mascaramento da senha” para maiores informações.

Procedimento 16.2. Criação de um par de chaves e armazenamento de chave para mascaramento de senha.

1. Altere o diretório para o diretório **`jboss-as/bin/password`** a partir da linha de comando.
2. Use o **`keytool`** para gerar um par de chave com o seguinte comando:

   ```
   keytool -genkey -alias jboss -keyalg RSA -keysize 1024 -keystore
   password.keystore
   ```

   Importante:

   Você deverá especificar a mesma senha para o armazenamento da chave e o par de chave.
3. **Opcional:**

   Disponibilize o password.keystore resultante para leitura pelo processo do Servidor do Aplicativo JBoss para o proprietário apenas.

   Isto é realizado nos sistemas baseados no Unix pelo uso do comando para alterar a propriedade do processo do Servidor do Aplicativo JBoss do proprietário e o **`chmod 600 password.keystore`** para disponibilizar o arquivo para leitura apenas para o proprietário.

   Esta etapa é recomendada para aumentar a segurança do seu servidor.

   Perceba: a propriedade do processo do Servidor do Aplicativo JBoss não deve ser um acesso de logon de console interativo. Neste caso, você estará executando estas operações como outro usuário. A criação de senhas mascaradas requer acesso de leitura ao armazenamento da chave, de forma que você desejará completar a configuração das senhas mascaradas antes de restringir as permissões do arquivo de armazenamento de chave.

Refira-se à Seção 15.1, “Visão Geral de Criptografia do SSL”, para maiores informações sobre os armazenamentos de chave e o comando **`keytool`**.

16.3. CRIPTOGRAFIA DA SENHA DO ARMAZENAMENTO DA CHAVE

A senhas necessárias pelos serviços do JBoss não precisam ser armazenadas em texto simples nos arquivos de configuração. Ao invés disto, elas são armazenadas num arquivo que é criptografado usando um par de chave que você fornecer.

O Servidor do Aplicativo JBoss precisa estar apto a usar um par de chave que você criou, com o objetivo de criptografar este arquivo e acesso às senhas mascaradas no período de rodagem. Você fornece a senha do armazenamento da chave ao Servidor do Aplicativo JBoss através da Ferramenta de Senha do JBoss, **`password_tool`**. Esta ferramenta irá criptografar e armazenar sua senha de armazenamento da chave. A sua senha de armazenamento da chave estará, então, disponível à Ferramenta de Senha do JBoss para mascaramento das senhas e ao Servidor do Aplicativo JBoss para criptografia das mesmas no período de rodagem.

Procedimento 16.3. Criptografia da senha do armazenamento da chave

1. Altere o diretório **`jboss-as/bin`** a partir da linha de comando.

2. Rode a ferramenta da senha usando o comando **`./password_tool.sh`** para sistemas baseados no Unix ou **`password_tool.bat`** para sistemas baseados no Windows.

   Resultado:

   A Ferramenta de Senha do JBoss será inicializada e relatará '**`Keystore is null. Please specify keystore below:`**'.

3. Selecione o '**`0: Encrypt Keystore Password`**' pressionando 0 e Enter.

   Resultado:

   A ferramenta de senha responde: '**`Enter keystore password`**'.

4. Insira a senha de armazenamento de chave especificada no Procedimento 16.2, “Criação de um par de chaves e armazenamento de chave para mascaramento de senha.”.

   Resultado:

   A ferramenta de senha responde: '**`Enter Salt (String should be at least 8 characters)`**'.

5. Insira uma sequência aleatória de caracteres para fortalecer a criptografia.

   Resultado:

   A ferramenta de senha responde: '**`Enter Iterator Count (integer value)`**'.

6. Insira um número inteiro para fortalecer a criptografia.

   Resultado:

   A ferramenta da senha responde: '**`Keystore Password encrypted into password/jboss_keystore_pass.dat`**'.

7. Selecione '**`5:Exit`**' para sair.

   Resultado:

   A ferramenta da senha finalizará com a seguinte mensagem: '**`Keystore is null. Cannot store.`**'. Isto é normal.

8. **Opcional:**

   Disponibilize para leitura o arquivo **`password/jboss_keystore_pass.dat`** resultante pelo processo Servidor do Aplicativo do JBoss do proprietário apenas.

   Nos sistemas baseados no Unix isto é concluído usando o comando **`chown`** para alterar a propriedade do processo do Servidor do Aplicativo JBoss e o **`chmod 600 jboss-keystore_pass.dat`** para disponibilizar o arquivo para leitura pelo proprietário.

   Esta etapa é recomendada para aumentar a segurança de seu servidor. Perceba que se esta chave criptografada estiver comprometida, a segurança oferecida pela senha mascarada será significativamente reduzida. Este arquivo deve ser armazenado num sistema de arquivo de segurança.

   Nota: o processo proprietário do Servidor do Aplicativo JBoss não deve possuir acesso de logon de console interativo. Neste caso, você executará estas operações como outro usuário. A criação de senhas mascaradas requer acesso de leitura ao armazenamento da chave, portanto

você deverá completar a configuração de senhas mascaradas antes de restringir as permissões do arquivo de armazenamento de chave.

Nota:

Você deve apenas executar este procedimento de criptografia de senha do armazenamento de chave uma vez. Caso você insira a senha errada do keystore ou altere o armazenamento da chave um dia mais tarde, você deverá deletar o arquivo **`jboss-keystore_pass.dat`** e repetir este procedimento. Perceba que se alterar o armazenamento da chave, quaisquer senhas mascaradas que foram anteriormente geradas não funcionarão.

16.4. CRIAÇÃO DE MÁSCARAS DE SENHA

A Ferramenta da Senha do JBoss mantém um arquivo de senha criptografado **`jboss-as/bin/password/jboss_password_enc.dat`**. Este arquivo é criptografado usando o par de chaves que você forneceu à ferramenta da senha, além de conter as senhas que você irá mascarar nos arquivos de configuração. As senhas são armazenadas e restauradas a partir deste arquivo pelo "domínio", um identificador único arbitrário que você especifica à Ferramenta da Senha quando armazenando a senha, e que você especifica como parte da anotação que substitui a senha de texto simples nos arquivos de configuração. Isto permite que o Servidor do Aplicativo JBoss recupere a senha correta a partir do arquivo no período de rodagem.

NOTA

Caso você tenha criado e criptografado o armazenamento de chave anteriormente, o arquivo da senha para leitura apenas pelo processo do Servidor do Aplicativo JBoss do proprietário, você precisará executar o seguinte procedimento como o proprietário do processo do Servidor do Aplicativo JBoss. Do contrário, disponibilize o keystore (**`jboss-as/bin/password/password.keystore`**) e o arquivo de senha do armazenamento da chave criptografada para leitura para seu usuário e criptografe o arquivo de senhas **`jboss-as/bin/password/jboss_password_enc.dat`** (caso já exista) para leitura e gravação, enquanto executando esta operação.

Procedimento 16.4. Criação de máscaras de senha

Pré-requisitos:

- Procedimento 16.2, “Criação de um par de chaves e armazenamento de chave para mascaramento de senha.”.
- Procedimento 16.3, “Criptografia da senha do armazenamento da chave”.

1. Altere o diretório **`jboss-as/bin`** a partir da linha de comando.
2. Rode a ferramenta da senha usando o comando **`./password_tool.sh`** para sistemas baseados no Unix ou **`password_tool.bat`** para sistemas baseados no Windows.

   Resultado:

   A Ferramenta de Senha do JBoss será inicializada e relatará '**`Keystore is null. Please specify keystore below:`**'.

3. Selecione o '**`1:Specify KeyStore`**' pressionando 1 e Enter.

   Resultado:

A ferramenta de senha responde com o '**`Enter Keystore location including the file name`**'.

4. Insira o caminho para o armazenamento de chave criado no Procedimento 16.2, “Criação de um par de chaves e armazenamento de chave para mascaramento de senha.”. Você pode especificar o caminho absoluto, ou o caminho relativo para o **`jboss-as/bin`**. Isto deve ser **`password/password.keystore`**, a não ser que você tenha executado uma instalação avançada e alterado os padrões conforme a Seção 16.6, “Alteração dos padrões da mascaramento da senha”.

   Resultado:

   A ferramenta de senha responde com '**`Enter Keystore alias`**'.

5. Insira o alias da chave. Ele deve ser **`jboss`**, a não ser que você tenha executado uma instalação e alterado os padrões conforme a Seção 16.6, “Alteração dos padrões da mascaramento da senha”.

   Resultado:

   Caso um armazenamento e um alias de chave sejam acessíveis, a ferramenta da senha responderá com algumas mensagens de AVISO log4j, uma linha '**`Loading domains [`**', seguidos de máscaras de senha existentes e do menu principal.

6. Selecione '**`2:Create Password`**' pressionando 2 e Enter.

   Resultado:

   A ferramenta da senha responde com: '**`Enter security domain:`**'.

7. Insira o nome da máscara da senha. Ela é um nome único arbitrário que você usará para identificar a máscara da senha nos arquivos de configuração.

   Resultado:

   A ferramenta da senha responde com: '**`Enter passwd:`**'.

8. Insira a senha que você deseja aplicar a máscara.

   Resultado:

   A ferramenta de senha responde com: '**`Password created for domain:`*`mask name`***'.

9. Repita o processo de criação da máscara para criar máscaras para todas as senhas pelas quais você desejar aplicar a máscara.

10. Enecerre o programa selecionando '**`5:Exit`**'.

16.5. SUBSTITUA AS SENHAS DE TEXTO SIMPLES POR SUAS MÁSCARAS DE SENHA

As senhas de texto simples nos arquivos de configuração podem ser substituídas pelas máscaras de senha, pela alteração da atribuição da propriedade por uma anotação. Gere máscaras de senha para todas as senhas de texto simples que você deseja aplicar a máscara nos arquivos de configuração do bean do Microcontainer conforme no Procedimento 16.4, “Criação de máscaras de senha”. Em seguida, substitua a ocorrência de configuração de cada senha de texto simples por uma anotação referenciando a própria máscara.

O modelo geral de anotação é o seguinte:

Exemplo 16.1. Modelo geral de anotação da máscara de senha

```
<annotation>@org.jboss.security.integration.password.Password(securityDo
main=MASK_NAME, methodName=setPROPERTY_NAME)</annotation>
```

A senha do JBoss Messaging está armazenada no perfil do servidor no **deploy/messaging/messaging-jboss-beans.xml**, como uma amostra concreta. Caso você tenha criado uma máscara de senha nomeada "messaging", o trecho do código do arquivo de configuração parecerá com o seguinte (antes e depois):

Exemplo 16.2. Configuração do JBoss Messaging Microcontainer Bean Antes

```
<property name="suckerPassword">CHANGE ME!!</property>
```

Exemplo 16.3. Configuração do JBoss Messaging Microcontainer Bean Depois

```
<annotation>@org.jboss.security.integration.password.Password(securityDo
main=messaging,
methodName=setSuckerPassword)</annotation>
```

16.6. ALTERAÇÃO DOS PADRÕES DA MASCARAMENTO DA SENHA

A Plataforma do Aplicativo JBoss Enterprise 5 lança os perfis do servidor configurados pela mascaramento da senha. Os perfis do servidor são configurados por padrão para uso do keystore **jboss-as/bin/password/password.keystore** e do key alias **jboss**. Caso você armazene um par de chaves usados para mascaramento da senha em algum outro local ou sob um alias diferente, você precisará atualizar os perfis do servidor por uma nova localização ou alias de chave.

A localização do armazenamento da chave de mascaramento e o alias de chave é especificado no **deploy/security/security-jboss-beans.xml** do arquivo, sob cada um dos perfis do Servidor do Aplicativo JBoss inclusos.

Exemplo 16.4. Padrões de Mascaramento da Senha no security-jboss-beans.xml

```
<!-- Password Mask Management Bean-->
   <bean name="JBossSecurityPasswordMaskManagement"

class="org.jboss.security.integration.password.PasswordMaskManagement" >
         <property
name="keyStoreLocation">password/password.keystore</property>
         <property name="keyStoreAlias">jboss</property>
         <property
name="passwordEncryptedFileName">password/jboss_password_enc.dat</proper
ty>
         <property
```

```
name="keyStorePasswordEncryptedFileName">password/jboss_keystore_pass.da
t</property>
   </bean>
```

CAPÍTULO 17. CRIPTOGRAFIA DAS SENHAS DE FONTE DE DADOS

As conexões do banco de dados para a Plataforma do Aplicativo do JBoss Enterprise estão definidas nos arquivos de fonte de dados **`*-ds.xml`**. Estes detalhes de conexão do banco de dados inclui senhas de texto limpo. Você pode aumentar a segurança de seu servidor pela substituição de senhas de texto limpo nos arquivos de fonte de dados com as senhas criptografadas.

Este capítulo apresenta dois métodos diferentes para senhas de fonte de dados criptografados.

A *Identidade Protegida* usando o módulo **`SecureIdentityLoginModule`** está descrita na Seção 17.1, “Identidade com Proteção”.

A *Identidade Configurada com a Senha baseada na Criptografia* usando o módulo **`JaasSecurityDomainIdentityLoginModule`**, está descrita na Seção 17.1, “Identidade com Proteção”.

17.1. IDENTIDADE COM PROTEÇÃO

A classe **`org.jboss.resource.security.SecureIdentityLoginModule`** pode ser usada para ambas as senhas do banco de dados criptografadas e para fornecer uma versão descriptografada da senha quando a configuração da fonte de dados for solicitada pelo servidor. O **`SecureIdentityLoginModule`** usa uma senha de código rígido paras criptografar/descriptografar a senha de fonte de dados.

Procedimento 17.1. Visão Geral: Uso do SecureIdentityLoginModule para criptografar uma senha de fonte de dados

1. Encripte a senha de fonte de dados.
2. Crie uma política de autenticação com a senha criptografada.
3. Configure a fonte de dados para uso da política de autenticação do aplicativo.

17.1.1. Criptografia da senha de fonte de dados

A senha de fonte de dados é criptografada usando o método principal **`SecureIdentityLoginModule`** passando uma mensagem de texto não-criptografado. O SecureIdentityLoginModule é fornecido pelo **`jbosssx.jar`**.

Procedimento 17.2. Criptografia de uma senha de fonte de dados - versões de Plataforma 5.0 e 5.0.1

Este procedimento criptografa uma senha de fonte de dados nas versões da Plataforma do Aplicativo JBoss Enterprise 5.0 e 5.0.1.

1. Altere o diretório **`jboss-as`**.
2. Invoque o SecureIdentityLoginModule pelo seguinte comando, fornecendo uma senha de texto não-criptografada como *PASSWORD*.

   Comando no Linux:

```
java -cp client/jboss-logging-spi.jar:common/lib/jbosssx.jar \
org.jboss.resource.security.SecureIdentityLoginModule PASSWORD
```

Comando no Windows:

```
java -cp client\jboss-logging-spi.jar;common\lib\jbosssx.jar \
 org.jboss.resource.security.SecureIdentityLoginModule PASSWORD
```

Resultado:

O comando retornará uma senha criptografada.

Procedimento 17.3. Criptografia de uma senha de fonte de dados - versão da Plataforma 5.1 e mais recentes

Este procedimento criptografa a senha de fonte de dados para a Plataforma do Aplicativo JBoss Enterprise versão 5.1 e mais recentes.

1. Altere o diretório **`jboss-as`**.
2. **Comando no Linux:**

```
java -cp client/jboss-logging-spi.jar:lib/jbosssx.jar \
 org.jboss.resource.security.SecureIdentityLoginModule PASSWORD
```

Comando no Windows:

```
java -cp client\jboss-logging-spi.jar;lib\jbosssx.jar \
 org.jboss.resource.security.SecureIdentityLoginModule PASSWORD
```

Resultado:

O comando retornará uma senha criptografada.

17.1.2. Criação de uma política de autenticação do aplicativo com a senha criptografada

Cada perfil do Servidor do Aplicativo possui um arquivo **`conf/login-config.xml`**, onde as políticas de autenticação do aplicativo são definidas para aquele perfil. Adicione um novo elemento <application-policy> ao elemento <policy>, com o objetivo de criar uma política de autenticação para sua senha criptografada.

O Exemplo 17.1, “Amostra da política de autenticação do aplicativo com uma senha de fonte de dados criptografada” é um fragmento de um arquivo **`login-config.xml`** apresentado na política de autenticação do nome "EncryptDBPassword".

Exemplo 17.1. Amostra da política de autenticação do aplicativo com uma senha de fonte de dados criptografada

```
<policy>
...
    <!-- Example usage of the SecureIdentityLoginModule -->
    <application-policy name="EncryptDBPassword">
```

```
          <authentication>
              <login-module
code="org.jboss.resource.security.SecureIdentityLoginModule"
flag="required">
                  <module-option name="username">admin</module-option>
                  <module-option
name="password">5dfc52b51bd35553df8592078de921bc</module-option>
                  <module-option
name="managedConnectionFactoryName">jboss.jca:name=PostgresDS,service=Lo
calTxCM</module-option>
              </login-module>
          </authentication>
      </application-policy>
  </policy>
```

Opções do módulo SecureIdentityLoginModule

nome do usuário

Especifica o nome do usuário para uso quando estabelecendo uma conexão ao banco de dados.

senha

Fornece uma senha criptografada gerada na Seção 17.1.1, “Criptografia da senha de fonte de dados”.

managedConnectionFactoryName

jboss.jca:name

Nomeia uma Interface do Diretório e Nomeação do Java - Java Naming and Directory Interface (JNDI) para esta fonte de dados.

jboss.jca:service

Especifica o tipo de transação.

Tipos de Transação

NoTxCM

Nenhum suporte de transação

LocalTxCM

Suporte de transação de recurso único

TxCM

Suporte de transação distribuída ou recurso único

XATxCM

Suporte de Transação Distribuída

17.1.3. Configura a fonte de dados para uso da política de autenticação do aplicativo.

A política do aplicativo é vinculada ao JNDI sob o nome da política do aplicativo, no período de rodagem, e é disponibilizada como um domínio de segurança.

A fonte de dados é configurada no arquivo **`*-ds.xml`**. Remova os elementos <user-name> e <password> deste arquivo e substitua-os por um elemento <security-domain>. Este elemento conterá o nome da política de autenticação do aplicativo segundo a Seção 17.1.2, “Criação de uma política de autenticação do aplicativo com a senha criptografada”.

O uso do nome da amostra a partir da Seção 17.1.2, “Criação de uma política de autenticação do aplicativo com a senha criptografada”, "EncryptDBPassword", resultará num arquivo de fonte de dados que parece-se com o seguinte Exemplo 17.2, “Amostra do arquivo da fonte de dados usando a identidade protegida”.

Exemplo 17.2. Amostra do arquivo da fonte de dados usando a identidade protegida

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources>
<local-tx-datasource>
<jndi-name>PostgresDS</jndi-name>
<connection-url>jdbc:postgresql://127.0.0.1:5432/test?
protocolVersion=2</connection-url>
<driver-class>org.postgresql.Driver</driver-class>
<min-pool-size>1</min-pool-size>
<max-pool-size>20</max-pool-size>

<!-- REPLACED WITH security-domain BELOW
<user-name>admin</user-name>
<password>password</password>
-->

<security-domain>EncryptDBPassword</security-domain>

<metadata>
<type-mapping>PostgreSQL 8.0</type-mapping>
</metadata>
</local-tx-datasource>
</datasources>
```

17.2. IDENTIDADE CONFIGURADA COM A SENHA BASEADA NA CRIPTOGRAFIA - PASSWORD BASED ENCRYPTION (PBE)

O **`org.jboss.resource.security.JaasSecurityDomainIdentityLoginModule`** é um módulo de logon para definição de forma estatística de uma fonte de dados usando uma senha que foi criptografada pelo JaasSecurityDomain. Segue abaixo o formato base64 da senha de fonte de dados gerado usando o PBEUtils:

Procedimento 17.4. Criptografia da senha com os PBEUtils - Plataformas de versão 5.0 e 5.0.1

Este procedimento criptografa uma senha na Plataforma do Aplicativo JBoss Enterprise de versões 5.0 e 5.0.1.

- Execute o comando:

```
java -cp jboss-as/common/lib/jbosssx.jar
org.jboss.security.plugins.PBEUtils \
  salt count domain-password data-source-password
```

Resultado:

A senha criptografada é exibida.

Procedimento 17.5. Criptografia da senha com PBEUtils - Platforma da versão 5.1

Este procedimento criptografa uma senha da Plataforma do Aplicativo JBoss Enterprise de versões 5.1 e versões superiores.

- Execute o comando:

```
java -cp jboss-as/lib/jbosssx.jar
org.jboss.security.plugins.PBEUtils \
 salt count domain-password data-source-password
```

Resultado:

A senha criptografada é exibida.

Os parâmetros para os **`PBEUtils`** são os seguintes:

salt

O atributo Salt a partir do JaasSecurityDomain (Ele deve conter oito caracteres).

contagem

O atributo IterationCount a partir do domínio JaasSecurity.

senha-domínio

A senha de texto plano mapeada ao atributo KeyStorePass a partir do JaasSecurityDomain.

senha-fonte-de-dados

A senha de texto plano para a fonte de dados que deve ser criptografado com a senha JaasSecurityDomain.

O Exemplo 17.3, “Amostra do comando PBEUtils ” fornece uma amostra do comando com seu resultado.

Exemplo 17.3. Amostra do comando PBEUtils

```
java -cp jbosssx.jar org.jboss.security.plugins.PBEUtils abcdefgh 13
master password
Encoded password: 3zbEkBDfpQAASa3H39pIyP
```

Adiciona a seguinte política do aplicativo ao arquivo **`$JBOSS_HOME/server/$PROFILE/conf/login-config.xml`**.

```
<application-policy name="EncryptedHsqlDbRealm">
<authentication>
<login-module code=
"org.jboss.resource.security.JaasSecurityDomainIdentityLoginModule"
flag = "required">
<module-option name="username">sa</module-option>
<module-option name="password">E5gtGMKcXPP</module-option>
<module-option name="managedConnectionFactoryName">
jboss.jca:service=LocalTxCM,name=DefaultDS
</module-option>
<module-option name="jaasSecurityDomain">
jboss.security:service=JaasSecurityDomain,domain=ServerMasterPassword
</module-option>
</login-module>
</authentication>
</application-policy>
```

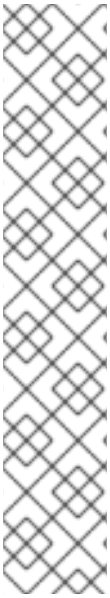
O **$JBOSS_HOME/docs/examples/jca/hsqldb-encrypted-ds.xml** ilustra a configuração da fonte de dados juntamente com a configuração JaasSecurityDomain para o keystore:

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- The Hypersonic embedded database JCA connection factory config
that illustrates the use of the JaasSecurityDomainIdentityLoginModule
to use encrypted password in the data source configuration.

$Id: hsqldb-encrypted-ds.xml,v 1.1.2.1 2004/06/04 02:20:52 starksm Exp $ -
->


<datasources>
...


<application-policy name="EncryptedHsqlDbRealm">
<authentication>
<login-module
code="org.jboss.resource.security.JaasSecurityDomainIdentityLoginModule"
flag = "required">
<module-option name="username">sa</module-option>
<module-option name="password">E5gtGMKcXPP</module-option>
<module-option name="managedConnectionFactoryName">
jboss.jca:service=LocalTxCM,name=DefaultDS
</module-option>
<module-option name="jaasSecurityDomain">
jboss.security:service=JaasSecurityDomain,domain=ServerMasterPassword
</module-option>
</login-module>
</authentication>
</application-policy>

<mbean code="org.jboss.security.plugins.JaasSecurityDomain"
name="jboss.security:service=JaasSecurityDomain,
domain=ServerMasterPassword">
<constructor>
```

```
<arg type="java.lang.String" value="ServerMasterPassword"></arg>
</constructor>
<!-- The opaque master password file used to decrypt the encrypted
database password key -->
<attribute
name="KeyStorePass">{CLASS}org.jboss.security.plugins.FilePassword:${jboss
.server.home.dir}/conf/server.password</attribute>
<attribute name="Salt">abcdefgh</attribute>
<attribute name="IterationCount">13</attribute>
</mbean>

<!-- This mbean can be used when using in process persistent db -->
<mbean code="org.jboss.jdbc.HypersonicDatabase"
name="jboss:service=Hypersonic,database=localDB">
<attribute name="Database">localDB</attribute>
<attribute name="InProcessMode">true</attribute>
</mbean>

...

</datasources>
```

ATENÇÃO

Lembre-se de usar o mesmo Salt e IterationCount no MBean que foi usado durante a etapa de geração da senha.



NOTA

Quando inicializando um serviço que depende da fonte de dados criptografada, o erro **`java.security.InvalidAlgorithmParameterException: Parameters missing`** é exibido quando o seguinte MBean não for inicializado como serviço:

```
(jboss.security:service=JaasSecurityDomain,domain=ServerMasterP
assword)
```

Adicione o seguinte elemento de forma que o MBean inicie antes do banco de dados, conforme a amostra do código **`hsqldb-encrypted-ds.xml`** apresentado anteriormente.

```
<depends>jboss.security:service=JaasSecurityDomain,domain=Serve
rMasterPassword</depends>
```

CAPÍTULO 18. CRIPTOGRAFIA DA SENHA KEYSTORE NUM CONECTOR TOMCAT

O JBoss Web está baseado no Apache Tomcat.

O SSL com o Tomcat requer um conector de segurança. Isto significa que a senha keystore/truststore não pode ser passada como um atributo no elemento conector do arquivo **server.xml** do Tomcat.

Recomenda-se um trabalho de entendimento do JaasSecurityDomain que suporta os keystores, truststores e senha baseados na criptografia.

Refira-se ao Capítulo 13, *Protocolo de Senha Remota de Segurança* e Capítulo 17, *Criptografia das Senhas de Fonte de Dados* para informações de suporte e procedimentos relacionados.

Procedimento 18.1. Criptografia da Senha Keystore do Recipiente Tomcat

1. **Anexe o elemento conector**
   Adicione o elemento conector ao **server.xml** no **$JBOSS_HOME/server/*$PROFILE*/deploy/jbossweb.sar**

   ```
   <!-- SSL/TLS Connector with encrypted keystore password
   configuration -->
   <Connector port="8443" address="${jboss.bind.address}"
   maxThreads="100" minSpareThreads="5" maxSpareThreads="15"
   scheme="https" secure="true" clientAuth="true"
   sslProtocol="TLS"
   securityDomain="java:/jaas/encrypt-keystore-password"
   SSLImplementation="org.jboss.net.ssl.JBossImplementation" >
   </Connector>
   ```

   .

2. **Configuração do JaasSecurityDomain MBean**
   Consulte o JaasSecurityDomain MBean no arquivo **$JBOSS_HOME/server/*$PROFILE*/deploy/security-service.xml**.

   Crie o arquivo caso ele ainda não exista. A amostra do código descreve o conteúdo solicitado quando o arquivo não existir. Caso você já tenha um **security-service.xml**, anexe o bloqueio do elemento <mbean> ao arquivo.

   ```
   <server>
   <mbean code="org.jboss.security.plugins.JaasSecurityDomain"
   name="jboss.security:service=PBESecurityDomain">
   <constructor>
   <arg type="java.lang.String" value="encrypt-keystore-
   password"></arg>
   </constructor>
   <attribute
   name="KeyStoreURL">resource:localhost.keystore</attribute>
   <attribute
   name="KeyStorePass">{CLASS}org.jboss.security.plugins.FilePassword:$
   {jboss.server.home.dir}/conf/keystore.password</attribute>
   <attribute name="Salt">welcometojboss</attribute>
   ```

```
<attribute name="IterationCount">13</attribute>
</mbean>
</server>
```

O Salt e IterationCount são variáveis que definem a intensidade de sua senha criptografada, de forma que você pode variar isto da maneira que é apresentada. Certifique-se de gravar novos valores e usá-los quando gerando a senha criptografada.

NOTA

O Salt deve possuir pelo menos oito caracteres.

3. **Geração de uma senha criptografada**
   A configuração <mbean> especifica que o keystore está armazenado no arquivo **`jboss-as/server/$PROFILE/conf/localhost.keystore`**. O <mbean> também especifica que o arquivo de senha criptografado está armazenado no arquivo **`jboss-as/server/$PROFILE/conf/keystore.password`**.

   Você deve criar um arquivo **`localhost.keystore`**.

   Execute o seguinte comando no diretório **`jboss-as/server/$PROFILE/conf`**.

   ```
   [conf]$ java -cp $JBOSS_HOME/lib/jbosssx.jar
   \org.jboss.security.plugins.FilePassword welcometojboss 13 unit-
   tests-server keystore.password
   ```

   Este comando usa um jbosssx.jar como classpath (**`-cp`**) e o puglin de segurança FilePassword para criação de um arquivo **`keystore.password`** com a senha configurada para **`unit-tests-server`**. Você deverá fornecer os parâmetros salt e interation configurados nos elementos <mbean> <attribute> do JaasSecurityDomain, com o objetivo de certificar-se de que tem permissão para criar um arquivo **`keystore.password`**.

   Você pode executar este comando no diretório **`/conf`**, de forma que o arquivo **`keystore.password`** é salvo a este diretório.

4. **Atualização do MBean do serviço Tomcat**
   Navegue ao **`$JBOSS_HOME/server/$PROFILE/deploy/jbossweb.sar/META-INF`** .

   Abra um **`jboss-service.xml`** e anexe a seguinte tag <depends> no final do arquivo. A adição da tag <depends> especifica que o Tomcat deve iniciar após o **`jboss.security:service=PBESecurityDomain`** .

   ```
   <depends>jboss.security:service=PBESecurityDomain</depends>
   </mbean>
   </server>
   ```

Exemplo 18.1. Definição do JaasSecurityDomain para pkcs12 keystores

Segundo o Procedimento 18.1, “Criptografia da Senha Keystore do Recipiente Tomcat”, os recipientes pkcs12 keystore referenciados pelo Tomcat Connector parecem-se com o seguinte:

```
<mbean code="org.jboss.security.plugins.JaasSecurityDomain"
name="jboss.security:service=PBESecurityDomain">
```

```
<constructor>
<arg type="java.lang.String" value="encrypt-keystore-password"></arg>
</constructor>
<attribute name="KeyStoreType">pkcs12</attribute>
<attribute name="KeyStoreURL">resource:localhost.keystore</attribute>
<attribute
name="KeyStorePass">{CLASS}org.jboss.security.plugins.FilePassword:${jbo
ss.server.home.dir}/conf/keystore.password</attribute>
<attribute name="Salt">welcometojboss</attribute>
<attribute name="IterationCount">13</attribute>
</mbean>
```

18.1. CASO DE USO DE SEGURANÇA MÉDIA

Um usuário não deseja criptografar a senha keystore, mas deseja externalizá-la (fora do **server.xml**) ou deseja fazer uso de um JaasSecurityDomain pré-definido.

Procedimento 18.2. JaasSecurityDomain Pré-definido

1. **Atualize o jboss-service.xml para adicionar um conector**
   Navegue ao **$JBOSS_HOME/server/ *$PROFILE* /deploy/jbossweb.sar/META-INF** e adicione o bloqueio de código ao arquivo **jboss-service.xml**.

   ```
   <mbean code="org.jboss.security.plugins.JaasSecurityDomain"
   name="jboss.security:service=SecurityDomain">
   <constructor>
   <arg type="java.lang.String" value="jbosstest-ssl"></arg>
   </constructor>
   <attribute
   name="KeyStoreURL">resource:localhost.keystore</attribute>
   <attribute name="KeyStorePass">unit-tests-server</attribute>
   </mbean>
   ```

2. **Adicione uma tag <depends> ao serviço Tomcat**
   Navegue ao **$JBOSS_HOME/server/*$PROFILE*/deploy/jbossweb.sar** .

   Abra o **server.xml** e anexe o seguinte elemento <depends> no final do arquivo:

   ```
   <depends>jboss.security:service=SecurityDomain</depends>
   </mbean>
   </server>
   ```

3. **Defina o JaasSecurityDomain MBean num arquivo *-service.xml**
   Por exemplo, o **security-service.xml** no diretório implantar:

   ```
   <mbean code="org.jboss.security.plugins.JaasSecurityDomain"
   name="jboss.security:service=SecurityDomain">
   <constructor>
   <arg type="java.lang.String" value="jbosstest-ssl"></arg>
   </constructor>
   <attribute
   ```

```
name="KeyStoreURL">resource:localhost.keystore</attribute>
<attribute name="KeyStorePass">unit-tests-server</attribute>
</mbean>
```

NOTA

Caso você encontre este erro, lembre-se que a id do usuário deve possuir permissão de gravação do arquivo keystore que está rodando a Plataforma do Aplicativo JBoss Enterprise.

CAPÍTULO 19. USO DO LDAPEXTLOGINMODULE COM O JAASSECURITYDOMAIN

Este capítulo fornece orientação de como o LdapExtLoginModule pode ser usado com uma senha criptografada a ser descriptografada pelo JaasSecurityDomain. Este capítulo assume que o LdapExtLoginModule já está rodando corretamente com uma senha não-criptografada.

Procedimento 19.1.

1. **Define o JaasSecurityDomain MBean**
   Define o JaasSecurityDomain MBean usado para descriptografar a versão criptografada da senha. Você pode adicionar o MBean ao **`$JBOSS_HOME/server/$PROFILE/conf/jboss-service.xml`** ou ao descritor de implantação ***`-service.xml`** na pasta **`$JBOSS_HOME/server/$PROFILE/deploy`**.

   ```
     <mbean code="org.jboss.security.plugins.JaasSecurityDomain"
         name="jboss.security:service=JaasSecurityDomain,domain=jmx-
   console">
         <constructor>
            <arg type="java.lang.String" value="jmx-console"></arg>
         </constructor>
         <attribute name="KeyStorePass">some_password</attribute>
         <attribute name="Salt">abcdefgh</attribute>
         <attribute name="IterationCount">66</attribute>
      </mbean>
   ```

   NOTA

   O algoritmo criptografado padrão usado pela implementação JaasSecurityDomain é **`PBEwithMD5andDES`**. Outros algoritmos criptografados incluem **`DES`**, **`TripleDES`**, **`Blowfish`** e **`PBEWithMD5AndTripleDES`**. Todos os algoritmos são simétricos. Você pode especificar um algoritmo criptografado anexando um elemento <attribute> com o atributo ***`CypherElement`*** configurado para um destes valores.

2. **Ajuste a senha, salt e contagem de interação**
   O Primeiro Passo contém uma configuração simples onde a senha solicitada, Salt e Contagem de Interação usados para criptografia ou descriptografia estão contidos na definição do MBean.

   Certifique-se de alterar os valores KeyStorePass, Salt e IterationCount adequados a sua implantação.

3.

Após este MBean ser definido, inicie a Plataforma do Aplicativo JBoss Enterprise. Navegue ao JMX Console (http://localhost:8080/jmx-console/ por padrão) e selecione o **`org.jboss.security.plugins.JaasSecurityDomain`** MBean.

Busque pelo método **`encode64(String password)`** na página **`org.jboss.security.plugins.JaasSecurityDomain`**. Passe a versão de texto simples do **`password`** usado pelo LdapExtLoginModule para este método e o invoque. O valor de retorno deve ser a versão criptografada da senha codificada conforme Base64.

As seguintes opções de módulo devem ser configuradas com a configuração do módulo de logon:

```
 <module-option
name="jaasSecurityDomain">jboss.security:service=JaasSecurityDomain,domain
=jmx-console</module-option>
  <module-option name="bindCredential">2gx7gcAxcDuaHaJMgO5AVo</module-
option>
```

A primeira opção é a nova opção para especificar que o \nJaasSecurityDomain usado anteriormente deve ser usado para descriptografar a senha.

O bindCredential é então substituído pela forma criptografada como Base64.

CAPÍTULO 20. FIREWALLS

A Plataforma do Aplicativo JBoss Enterprise lança diversos serviços baseados em soquete que requerem portas de firewall aberta. A Tabela 20.1, “As portas encontradas na configuração padrão” lista serviços que escutam portas que devem ser ativadas quando acessando o JBoss atrás da porta. A Tabela 20.2, “Portas Adicionais de todas as configurações” lista portas adicionais existentes em todos os perfis.

Tabela 20.1. As portas encontradas na configuração padrão

Porta	Tipo	Serviço
1098	TCP	**`org.jboss.naming.NamingService`**
1099	TCP	**`org.jboss.naming.NamingService`**
4444	TCP	**`org.jboss.invocation.jrmp.server.JRMPInvoker`**
4445	TCP	**`org.jboss.invocation.pooled.server.PooledInvoker`**
4446	TCP	**`org.jboss.invocation.unified.server.UnifiedInvoker`**
4457	TCP	Soquete do JBoss Messaging 1.x
4712	TCP	Soquete Gerenciador de Recuperação do JBossTS
4713	TCP	Gerenciador do Status de Transação do JBossTS
8009	TCP	**`org.jboss.web.tomcat.tc4.EmbeddedTomcatService`**
8080	TCP	**`org.jboss.web.tomcat.tc4.EmbeddedTomcatService`**
8083	TCP	**`org.jboss.web.WebService`**
8093	TCP	**`org.jboss.mq.il.uil2.UILServerILService`**

Tabela 20.2. Portas Adicionais de todas as configurações

Porta	Tipo	Serviço
1100	TCP	**`org.jboss.ha.jndi.HANamingService`**
1101	TCP	**`org.jboss.ha.jndi.HANamingService`**
1102	UDP	**`org.jboss.ha.jndi.HANamingService`**
1161	UDP	**`org.jboss.jmx.adaptor.snmp.agent.SnmpAgentService`**

Porta	Tipo	Serviço
1162	UDP	**org.jboss.jmx.adaptor.snmp.trapd.TrapdService**
1389	TCP	**ldaphost.jboss.org.LdapLoginModule**
3843[a].	TCP	**org.jboss.ejb3.SSLRemotingConnector**
3528	TCP	**org.jboss.invocation.iiop.IIOPInvoker**
3873	TCP	**org.jboss.ejb3.RemotingConnectors**
4447	TCP	**org.jboss.invocation.jrmp.server.JRMPInvokerHA**
4448	TCP	**org.jboss.invocation.pooled.server.PooledInvokerHA**\n\t\n
4448	TCP	**org.jboss.invocation.pooled.server.PooledInvokerHA**\n\t\n
7900	TCP	
45566[b]	UDP	**org.jboss.ha.framework.server.ClusterPartition**\n\t\n

[a] Apenas necessário caso o transporte SSL seja configurado para EJB3

[b] Mais duas portas UDP adicionais anônimas, uma pode ser configurada usando o **rcv_port** e a outra não pode ser configurada.

CAPÍTULO 21. CONSOLES E INVOCADORES

A Plataforma do Aplicativo do JBossEnterprise lança diversos pontos de acesso administrativos que devem ser protegidos ou removidos para prevenir o acesso não autorizado às funções administrativas numa implementação. Este capítulo discute os diversos serviços administrativos e como protegê-los.

21.1. CONSOLE JMX

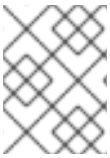
O **`jmx-console.war`** encontrado no diretório **`deploy`** fornece uma visualização HTML no JMX Microkernel. Desta maneira, ele fornece acesso às ações administrativas tais como encerramento do servidor, interrupção dos serviços, implantação de novos serviços, etc. Ele deve ser protegido como outro qualquer aplicativo da web ou removido.

21.2. CONSOLE ADMIN

O Console Admin substitui o Console da Web e usa os elementos de segurança da Rede de Operações do JBoss para proteger o console. Por favor refira-se ao *JBoss Admin Console Quick Start User Guide* para maiores informações sobre este respeito.

21.3. INVOCADORES HTTP

O **`http-invoker.sar`** encontrado no diretório **`deploy`** é um serviço que fornece o acesso RMI/HTTP para EJBs e serviço **`Naming`** do JNDI. Isto inclui um servlet que processa postagens dos objetos **`org.jboss.invocation.Invocation`** com marshall, dos quais representam invocações que devem ser despachadas no **`MBeanServer`**. Isto permite acesso através dos MBeans que suportam a operação do invocador desanexado através das solicitações HTTP POST. A proteção deste ponto de acesso envolve a proteção do **`JMXInvokerServlet`** servlet encontrado no descritor **`http-invoker.sar/invoker.war/WEB-INF/web.xml`**. Por padrão, existe um mapeamento de segurança definido para cada caminho **`/restricted/JMXInvokerServlet`**. Remova os outros caminhos e configure o domínio de segurança **`http-invoker`** no descritor de implantação **`http-invoker.sar/invoker.war/WEB-INF/jboss-web.xml`**.



NOTA

Consulte o *Admin Console Quick Start Guide* para uma informação mais detalhada sobre a proteção do invocador HTTP.

21.4. INVOCADOR JMX

O **`jmx-invoker-service.xml`** é um arquivo de configuração que expõe a interface JMX MBeanServer através de uma interface compatível usando o serviço invocador desanexado RMI/JRMP.

21.5. ACESSO REMOTO A SERVIÇOS, INVOCADORES DESANEXADOS

Adicionado à noção dos serviços MBean que permitem a habilidade de integrar a funcionalidade arbitrária, o JBoss também possui um conceito invocador desanexado que permite que os serviços MBean exponham interfaces funcionais através de protocolos arbitrários para o acesso remoto por clientes. A noção de um invocador desanexado é que o remoting e o protocolo pelo qual um serviço é acessado é um aspecto funcional ou serviço independente do componente. Portanto, você pode tornar disponível um serviço de nomeação para uso através do RMI/JRMP, RMI/HTTP, RMI/SOAP, ou qualquer outro transporte personalizado arbitrário.

A discussão da arquitetura do invocador desanexado começará por uma visão geral dos componentes invocados. Os componentes principais da arquitetura desanexada estão apresentados abaixo na Figura 21.1, “Os componentes principais da arquitetura do invocador desanexada”.

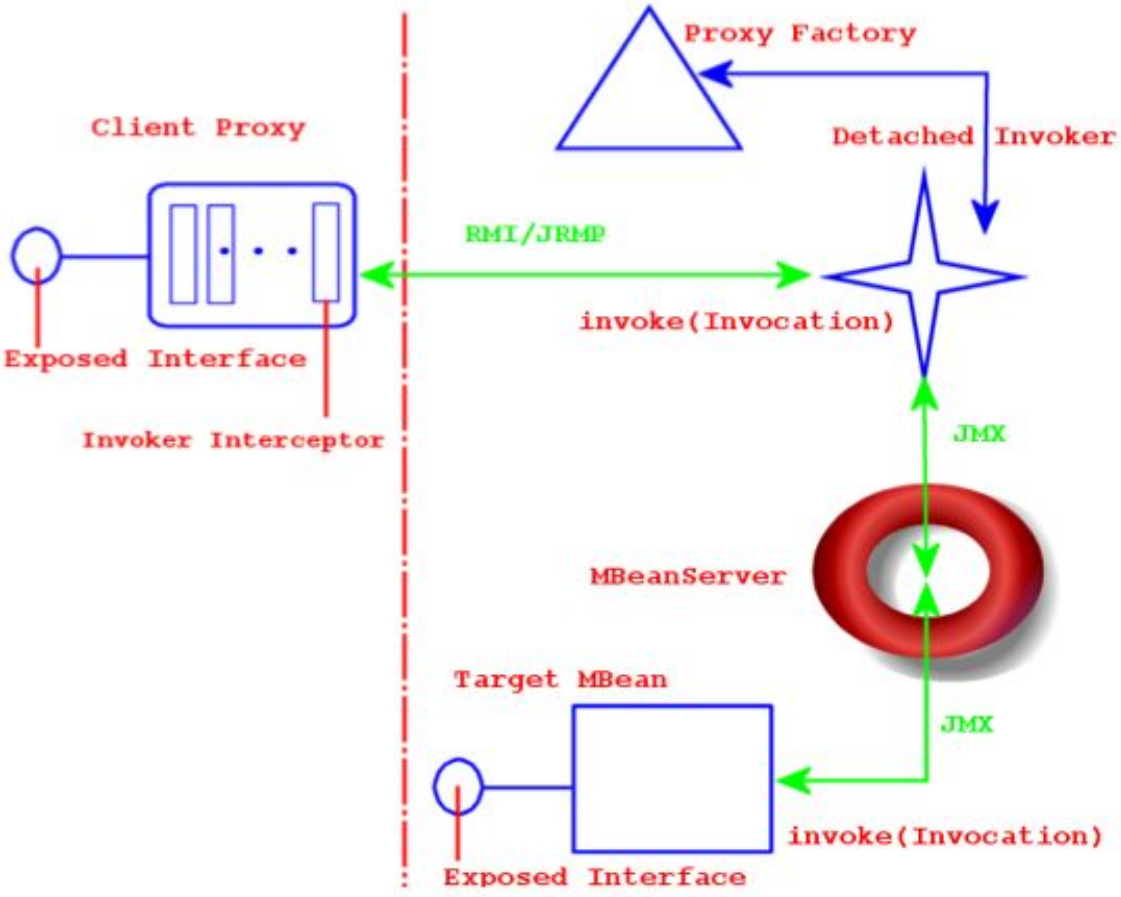


Figura 21.1. Os componentes principais da arquitetura do invocador desanexada

Ao lado do cliente, existe um cliente proxy que expõe a(s) interface(s) do serviço MBean. Este é o mesmo proxy dinâmico de menor compilação que é usado para o EJB principal e interfaces remotas. A única diferença entre um proxy para o serviço arbritário e o EJB é o conjunto de interfaces expostas como também os interceptores encontrados dentro do proxy. Os interceptores do cliente são representados pelos retângulos encontrados dentro do proxy do cliente. Um interceptor é um tipo de linha de montagem do padrão que permite a transformação de uma invocação de método e/ou valores de retorno. Um cliente obtém um proxy através de alguns mecanismos de busca, tipicamente o JNDI. Embora o RMI esteja indicado na Figura 21.1, “Os componentes principais da arquitetura do invocador desanexada”, a única solicitação real da interface exposta é que eles sejam serializados entre o servidor do cliente sobre o JNDI assim como a camada de transporte.

A escolha da camada de transporte é determinada pelo último interceptor no proxy do cliente, que é referenciado como um *Interceptor do Invocador* na Figura 21.1, “Os componentes principais da arquitetura do invocador desanexada”. O interceptor do invocador contém uma referência ao stub específico de transporte ao lado do servidor do serviço MBean do *Invocador Desanexado*. O interceptor do invocador também manuseia a otimização das chamadas que ocorrem com o mesmo VM como um MBean de destino. Quando o interceptor do invocador detecta que é esta a situação, a chamada é passada a um invocador de referência por chamada, que simplesmente passa a invocação através do MBean de destino.

O serviço do invocador desanexado é responsável por disponibilizar a operação de invocação genérica através do transporte que o invocador desanexado manuseia. A interface **`Invoker`** ilustra a operação de invocação genérica.

```
package org.jboss.invocation;

import java.rmi.Remote;
import org.jboss.proxy.Interceptor;
import org.jboss.util.id.GUID;


public interface Invoker
extends Remote
{
GUID ID = new GUID();

String getServerHostName() throws Exception;

Object invoke(Invocation invocation) throws Exception;
}
```

A interface do Invocador estende o **`Remote`** para ser compatível com o RMI, mas não significa que um invocador deve expor um stub de serviço RMI. O serviço invocador desanexado age como um transporte de fuga que aceita invocações representadas como um objeto **`org.jboss.invocation.Invocation`** sobre o transporte específico. O serviço invocador desempacota a invocação e a envia à destinação do serviço MBean representado pelo *MBean de Destino* na Figura 21.1, “Os componentes principais da arquitetura do invocador desanexada”. Além disso, ele empacota o valor de retorno ou exceção resultante do retorno de chamada enviado ao cliente.

O objeto **`Invocation`** é apenas uma representação do contexto da invocação do método. Ele inclui o nome do MBean de destino, o método, os argumentos do método, um contexto de informação associada com o proxy pela fábrica do proxy e um mapa arbitrário de dados associados com a invocação pelos interceptores do proxy do cliente.

A configuração do proxy do cliente é feita pela fábrica do proxy ao lado do servidor do serviço MBean, indicada pelo componente *Fábrica do Proxy* na Figura 21.1, “Os componentes principais da arquitetura do invocador desanexada”. A fábrica do proxy executa as seguintes tarefas:

- Criação do proxy dinâmico que implementa a interface do MBean de destino pelo qual deseja expor.
- Associa os interceptores do proxy do cliente com o manuseador do proxy dinâmico.
- Associa o contexto de invocação com o proxy dinâmico. Isto inclui o MBean de destino, o stub do invocador desanexado e o nome JNDI do proxy.
- Disponibiliza o proxy aos clientes pelo binding do proxy no JNDI.

O último componente na Figura 21.1, “Os componentes principais da arquitetura do invocador desanexada” é o serviço do *MBean de Destino* que deseja expor a interface às invocações para os clientes remotos. Segue abaixo os passos solicitados a um serviço MBean ser acessível através de uma interface gerada:

- Definição de uma operação JMX coincidente com a assinatura: **`public Object invoke(org.jboss.invocation.Invocation) throws Exception`**.

- Criação de um mapeamento **`HashMap<Long, Method>`** a partir dos **`java.lang.reflect.Method`**s de interface exibidos à representação hash longa usando o método **`org.jboss.invocation.MarshalledInvocation.calculateHash`**.

- Implementa a operação JMX **`invoke(Invocation)`** e usa o mapeamento hash da interface para transformar a longa representação hash do método invocado para o **`java.lang.reflect.Method`** da interface exposta. A reflexão é usada para executar a invocação atual no objeto associado com o serviço MBean que atualiza a interface exposta.

21.5.1. A Amostra do Invocador Desanexada, o Serviço do Invocador MBeanServer

Esta seção apresenta o **`org.jboss.jmx.connector.invoker.InvokerAdaptorService`** e sua configuração de acesso através do RMI/JRMP como uma amostra dos passos solicitados para fornecimento de acesso remoto a um serviço MBean.

Exemplo 21.1. O InvokerAdaptorService MBean

O **`InvokerAdaptorService`** é um serviço MBean simples que existe para preenchimento da função do MBean de destino no invocador padrão desanexado.

```
package org.jboss.jmx.connector.invoker;
public interface InvokerAdaptorServiceMBean
    extends org.jboss.system.ServiceMBean
{
    Class getExportedInterface();
    void setExportedInterface(Class exportedInterface);

    Object invoke(org.jboss.invocation.Invocation invocation)
        throws Exception;
}

package org.jboss.jmx.connector.invoker;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.UndeclaredThrowableException;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

import javax.management.MBeanServer;
import javax.management.ObjectName;

import org.jboss.invocation.Invocation;
import org.jboss.invocation.MarshalledInvocation;
import org.jboss.mx.server.ServerConstants;
import org.jboss.system.ServiceMBeanSupport;
import org.jboss.system.Registry;

public class InvokerAdaptorService
    extends ServiceMBeanSupport
    implements InvokerAdaptorServiceMBean, ServerConstants
{
    private static ObjectName mbeanRegistry;
```

```
    static {
        try {
            mbeanRegistry = new ObjectName(MBEAN_REGISTRY);
        } catch (Exception e) {
            throw new RuntimeException(e.toString());
        }
    }

    private Map marshalledInvocationMapping = new HashMap();
    private Class exportedInterface;

    public Class getExportedInterface()
    {
        return exportedInterface;
    }

    public void setExportedInterface(Class exportedInterface)
    {
        this.exportedInterface = exportedInterface;
    }

    protected void startService()
        throws Exception
    {
        // Build the interface method map
        Method[] methods = exportedInterface.getMethods();
        HashMap tmpMap = new HashMap(methods.length);
        for (int m = 0; m < methods.length; m ++) {
            Method method = methods[m];
            Long hash = new
Long(MarshalledInvocation.calculateHash(method));
            tmpMap.put(hash, method);
        }

        marshalledInvocationMapping =
Collections.unmodifiableMap(tmpMap);
        // Place our ObjectName hash into the Registry so invokers can
        // resolve it
        Registry.bind(new Integer(serviceName.hashCode()),
serviceName);
    }

    protected void stopService()
        throws Exception
    {
        Registry.unbind(new Integer(serviceName.hashCode()));
    }


    public Object invoke(Invocation invocation)
        throws Exception
    {
        // Make sure we have the correct classloader before
unmarshalling
        Thread thread = Thread.currentThread();
```

```
        ClassLoader oldCL = thread.getContextClassLoader();

        // Get the MBean this operation applies to
        ClassLoader newCL = null;
        ObjectName objectName = (ObjectName)
            invocation.getValue("JMX_OBJECT_NAME");
        if (objectName != null) {
            // Obtain the ClassLoader associated with the MBean
deployment
            newCL = (ClassLoader)
                server.invoke(mbeanRegistry, "getValue",
                              new Object[] { objectName, CLASSLOADER
},
                              new String[] {
ObjectName.class.getName(),
                                              "java.lang.String" });
        }

        if (newCL != null && newCL != oldCL) {
            thread.setContextClassLoader(newCL);
        }

        try {
            // Set the method hash to Method mapping
            if (invocation instanceof MarshalledInvocation) {
                MarshalledInvocation mi = (MarshalledInvocation)
invocation;
                mi.setMethodMap(marshalledInvocationMapping);
            }

            // Invoke the MBeanServer method via reflection
            Method method = invocation.getMethod();
            Object[] args = invocation.getArguments();
            Object value = null;
            try {
                String name = method.getName();
                Class[] sig = method.getParameterTypes();
                Method mbeanServerMethod =
                    MBeanServer.class.getMethod(name, sig);
                value = mbeanServerMethod.invoke(server, args);
            } catch(InvocationTargetException e) {
                Throwable t = e.getTargetException();
                if (t instanceof Exception) {
                    throw (Exception) t;
                } else {
                    throw new UndeclaredThrowableException(t,
method.toString());
                }
            }

            return value;
        } finally {
            if (newCL != null && newCL != oldCL) {
                thread.setContextClassLoader(oldCL);
            }
```

```
        }
    }
}
```

O código foi separado em blocos lógicos, com comentários explicativos de como cada bloco opera, com o objetivo de ajudar a entender o **`InvokerAdaptorServiceMBean`**.

Exemplo 21.2. Bloco Um

```
package org.jboss.jmx.connector.invoker;
public interface InvokerAdaptorServiceMBean
    extends org.jboss.system.ServiceMBean
{
    Class getExportedInterface();
    void setExportedInterface(Class exportedInterface);

    Object invoke(org.jboss.invocation.Invocation invocation)
        throws Exception;
}

package org.jboss.jmx.connector.invoker;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.UndeclaredThrowableException;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

import javax.management.MBeanServer;
import javax.management.ObjectName;

import org.jboss.invocation.Invocation;
import org.jboss.invocation.MarshalledInvocation;
import org.jboss.mx.server.ServerConstants;
import org.jboss.system.ServiceMBeanSupport;
import org.jboss.system.Registry;

public class InvokerAdaptorService
    extends ServiceMBeanSupport
    implements InvokerAdaptorServiceMBean, ServerConstants
{
    private static ObjectName mbeanRegistry;

    static {
        try {
            mbeanRegistry = new ObjectName(MBEAN_REGISTRY);
        } catch (Exception e) {
            throw new RuntimeException(e.toString());
        }
    }

    private Map marshalledInvocationMapping = new HashMap();
    private Class exportedInterface;
```

```
    public Class getExportedInterface()
    {
        return exportedInterface;
    }

    public void setExportedInterface(Class exportedInterface)
    {
        this.exportedInterface = exportedInterface;
    }
...
```

A interface do MBean Padrão **`InvokerAdaptorServiceMBean`** do **`InvokerAdaptorService`** possui um atributo **`ExportedInterface`** único e uma operação **`invoke(Invocation)`** única.

`ExportedInterface`

O atributo permite personalização do tipo de interface que o serviço expõe aos clientes. Ela deve ser compatível à classe **`MBeanServer`** no que se diz respeito ao nome e assinatura do método.

`invoke(Invocation)`

A operação é o ponto de entrada solicitado que os serviços MBean devem expor para participar do padrão do invocador desanexado. Esta operação é invocada pelos serviços do invocador desanexado que foram configurados para fornecer acesso ao **`InvokerAdaptorService`**.

Exemplo 21.3. Bloco Dois

```
    protected void startService()
        throws Exception
    {
        // Build the interface method map
        Method[] methods = exportedInterface.getMethods();
        HashMap tmpMap = new HashMap(methods.length);
        for (int m = 0; m &lt; methods.length; m ++) {
            Method method = methods[m];
            Long hash = new
Long(MarshalledInvocation.calculateHash(method));
            tmpMap.put(hash, method);
        }

        marshalledInvocationMapping =
Collections.unmodifiableMap(tmpMap);
        // Place our ObjectName hash into the Registry so invokers can
        // resolve it
        Registry.bind(new Integer(serviceName.hashCode()),
serviceName);
    }
    protected void stopService()
        throws Exception
    {
        Registry.unbind(new Integer(serviceName.hashCode()));
    }
```

Este bloco de código possui o HashMap<Long, Method> da Classe **`exportedInterface`** usando o método de utilidade **`org.jboss.invocation.MarshalledInvocation.calculateHash(Method)`**.

Devido às instâncias **`java.lang.reflect.Method`** não serem serializadas, uma versão **`MarshalledInvocation`** da classe **`Invocation`** é usada para empacotar a invocação entre o cliente e o servidor. O **`MarshalledInvocation`** substitui as instâncias de Método pela sua representação hash correspondente. Ao lado do servidor, o **`MarshalledInvocation`** deve ser informado qual é o hash para o mapeamento do Método.

Este bloco de código cria um mapeamento entre o nome do serviço **`InvokerAdaptorService`** e sua representação de código hash. Isto é usado pelos invocadores desanexados para determinar qual é o **`ObjectName`** para o **`Invocation`**.

Quando o nome MBean é armazenado no **`Invocation`**, seu armazenamento assim como seu hashCode uma vez que os **`ObjectName`**s são objetos relativamente caros para criação. O **`org.jboss.system.Registry`** é um mapa global como a construção que os invocadores usam para armazenar o código hash para mapeamentos do **`ObjectName`**.

Exemplo 21.4. Bloco Três

```
    public Object invoke(Invocation invocation)
        throws Exception
    {
        // Make sure we have the correct classloader before
unmarshalling
        Thread thread = Thread.currentThread();
        ClassLoader oldCL = thread.getContextClassLoader();

        // Get the MBean this operation applies to
        ClassLoader newCL = null;
        ObjectName objectName = (ObjectName)
            invocation.getValue(&quot;JMX_OBJECT_NAME&quot;);
        if (objectName != null) {
            // Obtain the ClassLoader associated with the MBean
deployment
            newCL = (ClassLoader)
                server.invoke(mbeanRegistry, &quot;getValue&quot;,
                              new Object[] { objectName, CLASSLOADER
},
                              new String[] {
ObjectName.class.getName(),

&quot;java.lang.String&quot; });
        }

        if (newCL != null &amp;&amp; newCL != oldCL) {
            thread.setContextClassLoader(newCL);
        }
```

Este código de bloco obtém o nome do MBean que a operação MBeanServer executou e busca o carregador de classe associado com a implementação SAR do MBean. Esta informação encontra-se disponível através do **`org.jboss.mx.server.registry.BasicMBeanRegistry`**, uma classe de implementação específica do JBoss JMX.

Normalmente, é necessário que um MBean estabeleça o contexto de carregamento de classe uma vez que a camada do protocolo do invocador desanexado possa não ter acessado os carregadores de classe necessários para desempacotar os tipos associados com uma invocação.

Exemplo 21.5. Bloco Quatro

```
...
try {
// Set the method hash to Method mapping
if (invocation instanceof MarshalledInvocation) {
MarshalledInvocation mi = (MarshalledInvocation) invocation;
mi.setMethodMap(marshalledInvocationMapping);
}
...
```

Este bloco de código instala o hash do método de classe **`ExposedInterface`** para o mapeamento do método caso o argumento da invocação for do tipo **`MarshalledInvocation`**. O mapeamento do método calculado no Exemplo 21.3, “Bloco Dois” é utilizado aqui.

Um segundo mapeamento é executado a partir do método **`ExposedInterface`** para o método correspondente da classe do MBeanServer. O **`InvokerServiceAdaptor`** descompila o **`ExposedInterface`** a partir da classe **`MBeanServer`** pela qual ele permite uma interface arbitrária. Isto é solicitado uma vez que a classe **`java.lang.reflect.Proxy`** pode apenas aplicar proxy nas interfaces. Além disso, isto permite também que você apenas exponha um sub-conjunto dos métodos MBeanServer e adicione transporte às execuções específicas tais como **`java.rmi.RemoteException`** às assinaturas do método **`ExposedInterface`**.

Exemplo 21.6. Bloco Cinco

```
...
// Invoke the MBeanServer method via reflection
Method method = invocation.getMethod();
Object[] args = invocation.getArguments();
Object value = null;
try {
String name = method.getName();
Class[] sig = method.getParameterTypes();
Method mbeanServerMethod =
MBeanServer.class.getMethod(name, sig);
value = mbeanServerMethod.invoke(server, args);
} catch(InvocationTargetException e) {
Throwable t = e.getTargetException();
if (t instanceof Exception) {
throw (Exception) t;
} else {
throw new UndeclaredThrowableException(t, method.toString());
}
}

return value;
} finally {
if (newCL != null &amp;amp;&amp;amp; newCL != oldCL) {
```

```
thread.setContextClassLoader(oldCL);
}
}
}
}
```

O bloco do código expede a invocação de método da instância do **`InvokerAdaptorService`** MBeanServer pela qual ele foi implementado. A variável da instância do servidor é herdada a partir da super-classe **`ServiceMBeanSupport`**.

Quaisquer exceções que resultem de uma invocação refletiva são manuseadas, incluindo desembrulhar quaisquer exceções declaradas lançadas pela invocação. O código MBean encerra com o retorno do resultado com êxito da invocação do método MBeanServer.

NOTA

O **`InvokerAdaptorService`** MBean não lida diretamente com quaisquer detalhes específicos de transporte. Existe um cálculo do hash de método para mapeamento do Método, mas isto é um detalhe independente de transporte.

Vamos observar agora como o **`InvokerAdaptorService`** pode ser usado para expor a mesma interface **`org.jboss.jmx.adaptor.rmi.RMIAdaptor`** através do RMI/JRMP, conforme visto na Conexão para JMX Usando RMI.

Nós começamos pela apresentação da fábrica proxy e configurações **`InvokerAdaptorService`** encontradas na configuração padrão da implementação **`jmx-invoker-adaptor-service.sar`**. O Exemplo 21.7, “Descritor da implantação jmx-invoker-adaptor-server.sar padrão” apresenta o descritor **`jboss-service.xml`** para esta implantação.

Exemplo 21.7. Descritor da implantação jmx-invoker-adaptor-server.sar padrão

```
<server>
<!-- The JRMP invoker proxy configuration for the InvokerAdaptorService
-->
<mbean code="org.jboss.invocation.jrmp.server.JRMPProxyFactory"
name="jboss.jmx:type=adaptor,name=Invoker,protocol=jrmp,service=proxyFac
tory">
<!-- Use the standard JRMPInvoker from conf/jboss-service.xml -->
<attribute
name="InvokerName">jboss:service=invoker,type=jrmp</attribute>
<!-- The target MBean is the InvokerAdaptorService configured below -->
<attribute
name="TargetName">jboss.jmx:type=adaptor,name=Invoker</attribute>
<!-- Where to bind the RMIAdaptor proxy -->
<attribute name="JndiName">jmx/invoker/RMIAdaptor</attribute>
<!-- The RMI compatible MBeanServer interface -->
<attribute
name="ExportedInterface">org.jboss.jmx.adaptor.rmi.RMIAdaptor</attribute
>
<attribute name="ClientInterceptors">
<iterceptors>
<interceptor>org.jboss.proxy.ClientMethodInterceptor</interceptor>
<interceptor>
```

```
org.jboss.jmx.connector.invoker.client.InvokerAdaptorClientInterceptor
</interceptor>
<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
</iterceptors>
</attribute>
<depends>jboss:service=invoker,type=jrmp</depends>
</mbean>
<!-- This is the service that handles the RMIAdaptor invocations by
routing
them to the MBeanServer the service is deployed under. -->
<mbean code="org.jboss.jmx.connector.invoker.InvokerAdaptorService"
name="jboss.jmx:type=adaptor,name=Invoker">
<attribute
name="ExportedInterface">org.jboss.jmx.adaptor.rmi.RMIAdaptor</attribute
>
</mbean>
</server>
```

O primeiro MBean, **`org.jboss.invocation.jrmp.server.JRMPProxyFactory`**, é a fábrica proxy do serviço MBean que cria proxies para o protocolo RMI/JRMP. Conforme apresentado no Exemplo 21.7, “Descritor da implantação jmx-invoker-adaptor-server.sar padrão” a configuração deste serviço declara que o JRMPInvoker será usada como invocador desanexado, o **`InvokerAdaptorService`** é o mbean de destino pelo qual solicitações serão enviadas, que o proxy irá expor a interface **`RMIAdaptor`**. O proxy será vinculado ao JNDI sob o nome de **`jmx/invoker/RMIAdaptor`**, além de conter 3 interceptores: **`ClientMethodInterceptor`**, **`InvokerAdaptorClientInterceptor`** e **`InvokerInterceptor`**. A configuração do **`InvokerAdaptorService`** apenas determina a interface RMIAdaptor que o serviço está expondo.

O último trecho de configuração para exposição do **`InvokerAdaptorService`** através do RMI/JRMP é o invocador desanexado. O invocador desanexado que usaremos é o invocador RMI/JRMP usado pelos recipientes EJB para invocações remotas e principais, além de ser o serviço MBean **`org.jboss.invocation.jrmp.server.JRMPInvoker`** configurado no descritor **`conf/jboss-service.xml`**. Uma das naturezas desanexadas dos invocadores é que podemos usar a mesma instância de serviço. O JRMPInvoker age simplesmente como um ponto final do RMI/JRMP para todos os proxies independente da(s) interface(s) que os proxies expõem ou o serviço que os proxies utilizam.

APÊNDICE A. CONFIGURAÇÃO DO JDK PADRÃO COM A UTILIDADE `/USR/SBIN/ALTERNATIVES`

O **`/usr/sbin/alternatives`** é uma ferramenta para gerenciamento de pacotes de software diferentes que fornecem a mesma funcionalidade. O **Red Hat Enterprise Linux** usa o **`/usr/sbin/alternatives`** para garantir que apenas um Kit de Desenvolvimento do Java é configurado como o padrão do sistema de uma só vez.

IMPORTANTE

A instalação do Kit de Desenvolvimento Java a partir da Rede da Red Hat normalmente resultará num sistema automaticamente configurado. No entanto, caso múltiplos JDKs estiverem instalados, é possível que o **`/usr/sbin/alternatives`** contenha configurações em conflito. Refira-se ao Procedimento A.1, “Usando o **`/usr/sbin/alternatives`** para configurar o JDK padrão. ” para síntese do comando **`/usr/sbin/alternatives`**.

Procedimento A.1. Usando o `/usr/sbin/alternatives` para configurar o JDK padrão.

1. **Torne-se usuário root.**
   O **`/usr/sbin/alternatives`** precisa rodar com privilégios root. Use o comando **su** ou outro mecanismo para obter estes privilégios.

2. **Configure o `java`.**
   Insira o comando: **`/usr/sbin/alternatives --config java`**

   Em seguida, siga as instruções para garantir que a versão correta do **`java`** é instalada. A Tabela A.1, “comandos alternativos do **`java`**” apresenta as configurações de comando relevante para cada um dos JDKs diferentes.

   Tabela A.1. comandos alternativos do `java`

JDK	comando alternativo
OpenJDK 1.6	**`/usr/lib/jvm/jre-1.6.0-openjdk/bin/java`**
Sun Microsystems JDK 1.6	**`/usr/lib/jvm/jre-1.6.0-sun/bin/java`**

3. **Configure o `javac`.**
   Insira o comando: **`/usr/sbin/alternatives --config javac`**

   Siga as instruções da tela para garantir que a versão correta do **`javac`** é selecionada. A Tabela A.2, “comandos alternativos do **`javac`** ” apresenta as configurações de comando apropriadas para os diferentes JDKs.

   Tabela A.2. comandos alternativos do `javac`

JDK	comando alternativo
OpenJDK 1.6	**`/usr/lib/jvm/java-1.6.0-openjdk/bin/javac`**

JDK	comando alternativo
Sun Microsystems JDK 1.6	**`/usr/lib/jvm/java-1.6.0-sun/bin/javac`**

4. **Passo adicional: Configure o `java_sdk_1.6.0`.**
   O **Sun Microsystems JDK 1.6** um comando adicional a ser rodado:

   `/usr/sbin/alternatives --config java_sdk_1.6.0`

   Siga as instruções da tela para garantir que o **`java_sdk`** correto é selecionado. Ele é **`/usr/lib/jvm/java-1.6.0-sun`**.

APÊNDICE B. HISTÓRICO DE REVISÃO

Revisão 5.1-0.1.400 **2013-10-31** **Rüdiger Landmann**

Rebuild with publican 4.0.0

Revisão 5.1-0.1 **Fri Aug 31 2012** **Leticia de Lima**

Translation files synchronised with XML sources 5.1-0

Revisão 5.1-0 **Wed Sep 15 2010** **Jared Morgan**

Contém problemas e melhorias relacionadas com a Certificação de Critério Comum para a Plataforma do Aplicativo JBoss Enterprise v5.1 e outras questões levantadas por clientes e da comunidade.

Os capítulos sobre o uso do SSL forem re-escritos para proteger a Invocação do Método Remoto do EJBs e Mascaramento de Senhas.

JBOSSCC-47 - Problemas encontrados nos comentários do Critério Comum.

JBOSSCC-53 - Adicionada a Seção 1.7, “Ativação da Autenticação baseada no Formulário” .

JBOSSCC-54 - Adicionada a informação do mecanismo de segurança Tomcat para a Figura 4.2, “Etapas de Autenticação do Método da Página Principal do EJB e Invocação da Autorização.”

JBOSSCC-55 - Explicação da localização da informação de segurança do Console de Administração.

JBOSSCC-56 - Duplicata do JBOSSCC-55.

JBOSSCC-57 - Explicação do livro contendo configuração de segurança para invocadores de legacia.

JBOSSCC-58 - Adicionada a Seção 21.5, “Acesso Remoto a Serviços, Invocadores Desanexados” .

JBOSSCC-59 - Atualização das portas firewall que devem ser ativadas dependendo do perfil em uso (consulte o Capítulo 20, *Firewalls*.

JBOSSCC-62 - Atualização do Capítulo 3, *Modelo de Segurança do JBoss*.

JBOSSCC-63 - Adição do Capítulo 16, *Mascarando senhas na Configuração XML* .

JBOSSCC-65 - Adição da Informação da Porta Firewall EJB3 ao Capítulo 20, *Firewalls*.

JBPAPP-3298 - Atualização da declaração de concedimento no Capítulo 14, *Gerenciador de Segurança do Java*, para especificação do JBoss

JBPAPP-4942 - Adição do Capítulo 17, *Criptografia das Senhas de Fonte de Dados* .

JBPAPP-4973 - Comentários de revisão final do QE do Critério Comum.