



JBoss Enterprise Application Platform 5

セキュリティガイド

JBoss Enterprise Application Platform 5 のユーザー向け
エディション 5.1.2

JBoss Enterprise Application Platform 5 セキュリティガイド

JBoss Enterprise Application Platform 5 のユーザー向け
エディション 5.1.2

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

法律上の通知

Copyright © 2011 Red Hat, Inc.

This document is licensed by Red Hat under the [Creative Commons Attribution-ShareAlike 3.0 Unported License](https://creativecommons.org/licenses/by-sa/3.0/). If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

概要

本セキュリティガイドはシステム管理者と開発者向けであり、JBoss Enterprise Application Platform 5 とその修正リリースにおけるセキュリティ実装の方法について説明しています。本書では Java EE 宣言型セキュリティや、Java 認証承認サービス、セキュリティモデルおよび拡張アーキテクチャの概要説明、セキュリティドメインの管理と設定、設定ファイルでのクリアテキストのパスワードとマスクとの置換、SSL を使用した EJB のリモートメソッド呼び出しをセキュアにする方法について取り上げます。

目次

パート I. セキュリティの概要	5
第1章 JAVA EE の宣言型セキュリティの概要	6
1.1. セキュリティ参照	6
1.2. セキュリティアイデンティティ	7
1.3. セキュリティロール	9
1.4. EJB メソッドパーミッション	11
1.5. エンタープライズ BEAN セキュリティアノテーション	14
1.6. WEB コンテンツのセキュリティ制約	14
1.7. フォームベースの認証を有効にする	17
1.8. 宣言型セキュリティを有効にする	18
第2章 JAAS の概略	19
2.1. JAAS コアクラス	19
2.1.1. サブジェクトとプリンシパルクラス	19
2.1.2. サブジェクトの認証	20
第3章 JBOSS セキュリティモデル	24
3.1. 宣言型セキュリティの有効化の再確認	26
第4章 JBOSS セキュリティ拡張アーキテクチャ	31
4.1. JAASSECURITYMANAGER による JAAS の使用方法	33
4.2. JAASSECURITYMANAGERSERVICE MBEAN	36
4.3. JAASSECURITYDOMAIN MBEAN	39
パート II. アプリケーションセキュリティ	41
第5章 概要	42
第6章 セキュリティドメインスキーマ	44
6.1. セキュリティドメイン要素	44
6.1.1. <authentication>	44
6.1.2. <authorization>	45
6.1.3. <mapping>	46
第7章 認証	48
7.1. カスタムコールバックハンドラ	49
第8章 承認	54
8.1. 委譲されるモジュール	58
第9章 マッピング	60
第10章 監査	62
第11章 セキュリティドメインのデプロイ	69
第12章 ログインモジュール	71
12.1. モジュールの使用	71
12.1.1. LdapLoginModule	71
12.1.2. LdapExtLoginModule	76
12.1.3. パスワードスタック	86
12.1.4. パスワードのハッシュ化	87
12.1.5. 認証されていないアイデンティティ	89
12.1.6. UsersRolesLoginModule	89

12.1.7. DatabaseServerLoginModule	90
12.1.8. BaseCertLoginModule	92
12.1.9. IdentityLoginModule	95
12.1.10. RunAsLoginModule	95
12.1.11. RunAsIdentity の作成	96
12.1.12. ClientLoginModule	97
12.1.13. SPNEGOLoginModule	98
12.2. カスタムモジュール	98
12.2.1. Subject の使用パターンのサポート	99
12.2.2. カスタムの LoginModule の例	104
パート III. 暗号化とセキュリティ	108
第13章 セキュアリモートパスワードプロトコル	109
13.1. アルゴリズムの理解	113
13.2. セキュアリモートパスワード情報の設定	115
13.3. セキュアリモートパスワード (SRP) の例	117
第14章 JAVA セキュリティマネージャー	121
14.1. セキュリティマネージャーの使用	122
14.2. セキュリティポリシーの問題点のデバッグ	124
14.2.1. デバッグセキュリティマネージャー	125
14.3. JBOSS ENTERPRISE APPLICATION PLATFORM 向けのセキュリティポリシーを書く	126
第15章 EJB RMI トランスポート層をセキュアにする	129
15.1. SSL 暗号化の概要	129
15.1.1. キーペアと証明書	129
15.2. 暗号化キーと証明書の生成	130
15.2.1. keytool を使用した自己署名証明書の生成	130
15.2.1.1. キーペアの生成	130
15.2.1.2. 自己署名証明書のエクスポート	132
15.2.2. 自己署名サーバー証明書を受け入れるためのクライアント設定	132
15.3. EJB3 RMI + SSL 設定	133
15.4. HTTPS 設定による EJB3 RMI	135
15.5. EJB2 RMI + SSL 設定	140
第16章 XML 設定のパスワードマスク	143
16.1. パスワードマスクの概要	143
16.2. キーストアとマスクされたパスワードの生成	143
16.3. キーストアのパスワードの暗号化	144
16.4. パスワードマスクの作成	146
16.5. クリアテキストのパスワードとそれらのパスワードマスクとの置換	147
16.6. パスワードマスクのデフォルト変更	148
第17章 データソースのパスワードの暗号化	149
17.1. セキュアなアイデンティティ	149
17.1.1. データソースのパスワードの暗号化	149
17.1.2. 暗号化されたパスワードでのアプリケーションの認証ポリシーの作成	150
17.1.3. アプリケーションの認証ポリシーを使用するためのデータソースの設定	151
17.2. PASSWORD BASED ENCRYPTION (パスワードベースの暗号化 : PBE) で設定されたアイデンティティ	152
第18章 TOMCAT CONNECTOR のキーストアパスワードの暗号化	157
18.1. 中程度のセキュリティユースケース	159

第19章 JAASSECURITYDOMAIN での LDAPEXTLOGINMODULE の使用	161
第20章 ファイアウォール	163
第21章 コンソールとインボーカー	167
21.1. JMX コンソール	167
21.2. ADMIN コンソール	167
21.3. HTTP インボーカー	167
21.4. JMX インボーカー	167
21.5. サービス、分離インボーカーへのリモートアクセス	167
21.5.1. 分離インボーカーの例、MBeanServer インボーカーアダプターサービス	170
付録A /USR/SBIN/ALTERNATIVES ユーティリティによるデフォルト JDK の設定	179
付録B 改訂履歴	181

パート I. セキュリティの概要

セキュリティはすべてのエンタープライズアプリケーションの根本的な役割を果たします。セキュリティにより、ご使用のアプリケーションにアクセスできる人を制限し、アプリケーションユーザーが実行できる操作を制御することができます。

Java Enterprise Edition (Java EE) 仕様は *Enterprise Java Beans* (EJBs) と Web コンポーネントのシンプルなロールベースのセキュリティモデルを定義します。*JBoss セキュリティ拡張* (JBossSX) フレームワークはプラットフォームセキュリティを処理し、ロールベースの宣言型 Java EE セキュリティモデルとセキュリティプロキシレイヤを通じたカスタムセキュリティの統合の両方に対応します。

宣言型セキュリティモデルのデフォルト実装は *Java 認証承認サービス* (JAAS) のログインモジュールとサブジェクトに基づいています。セキュリティプロキシレイヤによって、宣言型モデルを使用して記述できないカスタムセキュリティを EJB ビジネスオブジェクトから独立した形で EJB に追加することができます。



注記

EJB とサーブレット仕様セキュリティモデル、さらに JAAS に関する詳細は [パート I「セキュリティの概要」](#) に記載されています。

第1章 JAVA EE の宣言型セキュリティの概要

Java EE セキュリティモデルは、ビジネスコンポーネントにセキュリティを組み込むのではなく、セキュリティロールとパーミッションを標準 XML 記述子に記述するため宣言的です。セキュリティは、コンポーネントのビジネスロジック固有の側面よりもコンポーネントがデプロイされる機能に依存しているため、セキュリティをビジネスレベルのコードから分離させます。

例えば、銀行口座にアクセスするために使用される現金自動預払機 (ATM) のコンポーネントを考えてみましょう。このコンポーネントのセキュリティ要件、ロール、パーミッションは銀行口座にアクセスする方法とは関係なく異なります。口座情報にアクセスする方法も、口座を管理している銀行、ATM の場所によって違います。

Java EE のアプリケーションをセキュアにすることは、標準 Java EE デプロイメント記述子によるアプリケーションセキュリティ要件の仕様にに基づきます。**ejb-jar.xml** と **web.xml** デプロイメント記述子を使用して、エンタープライズアプリケーションの EJB と Web コンポーネントへのアクセスをセキュアにします。次項では様々なセキュリティ要素の目的と使用方法について見ていきます。

1.1. セキュリティ参照

Enterprise Java Beans (EJB) とサーブレットのどちらも 1 つ以上の `<security-role-ref>` 要素を宣言できます。図1.1「`<security-role-ref>` 要素」は `<security-role-ref>` 要素、その子要素、属性について説明しています。

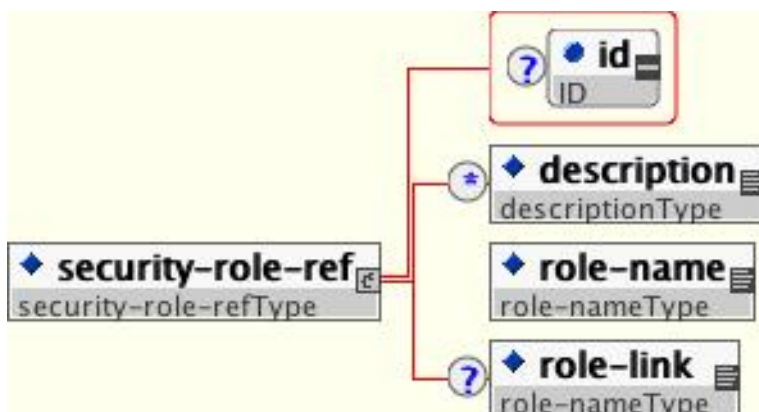


図1.1 `<security-role-ref>` 要素

この要素は、コンポーネントが `<role-name>` 要素の `role-nameType` 属性値を `isCallerInRole(String)` メソッドへの引数として使用していることを宣言します。`isCallerInRole` メソッドを使用することで、コンポーネントは呼び出し側が `<security-role-ref>` または `<role-name>` 要素で宣言されたロールにあるか検証できます。`<role-name>` 要素値は `<role-link>` 要素を通じて `<security-role>` 要素にリンクする必要があります。`isCallerInRole` の一般的な使用法は、ロールベースの `<method-permissions>` 要素を使用して定義できないセキュリティ確認を実行することです。

例1.1「`ejb-jar.xml` 記述子の一部」では `ejb-jar.xml` ファイルの `<security-role-ref>` の使用について説明しています。

例1.1 `ejb-jar.xml` 記述子の一部

```
<!-- 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>

```

例1.2 「web.xml 記述子の一部」 では **web.xml** ファイルの <security-role-ref> の使用について示しています。



注記

この記述子の一部はほんの一例です。デプロイメントでは、本項の要素に EJB デプロイメントに関連するロール名とリンクが含まれる必要があります。

例1.2 web.xml 記述子の一部

```

<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. セキュリティアイデンティティ

Enterprise Java Bean (EJB) は、<security-identity> 要素を使用して別の EJB がコンポーネントでメソッドを呼び出すときに、別の EJB が使用しなければならないアイデンティティを特定することができます。図1.2 「security-identity 要素」 は <security-identity> 要素、その子要素、属性について説明しています。

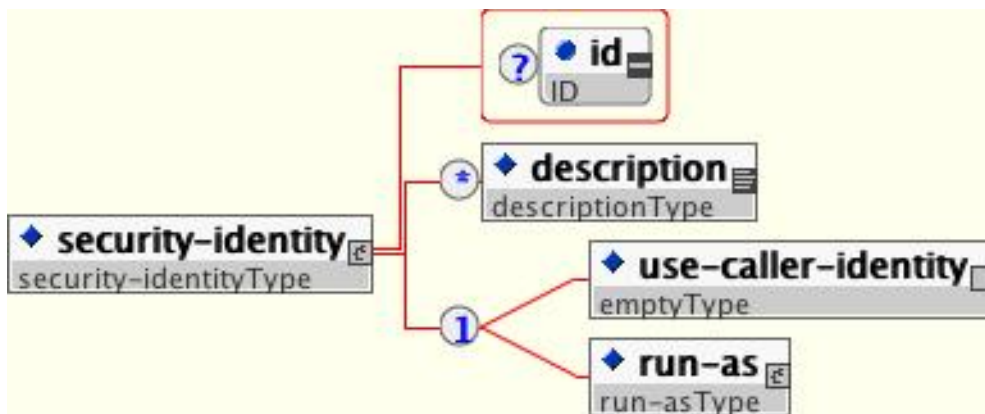


図1.2 security-identity 要素

呼び出しアイデンティティは現在の呼び出し側のアイデンティティ、または特定のロールとなることができます。アプリケーションアセンブラは `<use-caller-identity>` 子要素を持つ `<security-identity>` 要素を使用します。これは現在の呼び出し側のアイデンティティが EJB によるメソッド呼び出しのセキュリティアイデンティティとして伝播されることを示しています。呼び出し側のアイデンティティの伝播は、明示的な `<security-identity>` 要素宣言がないときに使用されるデフォルトです。

別の方法として、アプリケーションアセンブラは `<run-as>` または `<role-name>` 子要素を使用して、`<role-name>` 要素値によって提供される特定のセキュリティロールが EJB によるメソッド呼び出しのセキュリティアイデンティティとして使用されるよう指定できます。

これにより呼び出し側のアイデンティティを `EJBContext.getCallerPrincipal()` メソッドで見られるように変更することはありません。むしろ呼び出し側のセキュリティロールは `<run-as>` または `<role-name>` 要素値によって指定される単一のロールに設定されます。

`<run-as>` 要素を使用するユースケースとして、外部のクライアントが内部の EJB にアクセスできないようにします。この動作を設定するには、内部の EJB `<method-permission>` 要素を割り当てます。これにより外部クライアントに割り当てられたことがないロールへのアクセスを制限します。次に、内部の EJB を使用する必要がある EJB は制限されたロールに等しい `<run-as>` または `<role-name>` で設定されます。次の記述子の一部は `<security-identity>` 要素の使用例を説明しています。

```

<!-- 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>

```

```

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

```

`<run-as>` を使用して発信呼び出しに特定のロールを割り当てる場合、**anonymous** という名前のプリンシパルがすべての発信呼び出しに割り当てられます。別のプリンシパルにその呼び出しを関連付けたい場合は、**jboss.xml** ファイルの Bean と `<run-as-principal>` を関連付ける必要があります。次の例では **internal** という名前のプリンシパルを前述の例からの **RunAsBean** と関連付けます。

```

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

```

`<run-as>` 要素は **web.xml** ファイルのサーブレット定義でも使用できます。次の例では ロール **InternalRole** をサーブレットに割り当てる方法を示しています。

```

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

```

このサーブレットからの呼び出しは匿名 **principal** と関連付けられます。`<run-as-principal>` 要素は **run-as** ロールと合う特定のプリンシパルを割り当てるため **jboss-web.xml** ファイルで使用できます。次の例では **internal** という名前のプリンシパルを上記のサーブレットと関連付ける方法を示しています。

```

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

```

1.3. セキュリティロール

`<security-role-ref>` または `<security-identity>` 要素のどちらかで参照されるセキュリティロール名は、アプリケーションの宣言されたロールの 1 つにマップする必要があります。アプリケーションアセンブラは、`<security-role>` 要素を宣言することで論理セキュリティロールを定義します。role-name 属性値は Administrator、Architect、Sales_Manager のような論理アプリケーションロール名です。

Java EE 仕様で注意することは、デプロイメント記述子のセキュリティロールはアプリケーションの論理セキュリティビューを定義するために使用されることを覚えておくことが重要である点です。Java EE デプロイメント記述子で定義されるロールと、ユーザーグループ、ユーザー、プリンシパル、目的のエンタープライズの動作環境に存在する他のコンセプトとを混同しないでください。デプロイメント記述子のロールはアプリケーションドメイン固有の名前を持つアプリケーション構成です。例えば、銀行のアプリケーションでは Bank_Manager、Teller、Customer などのロール名を使用することがあります。

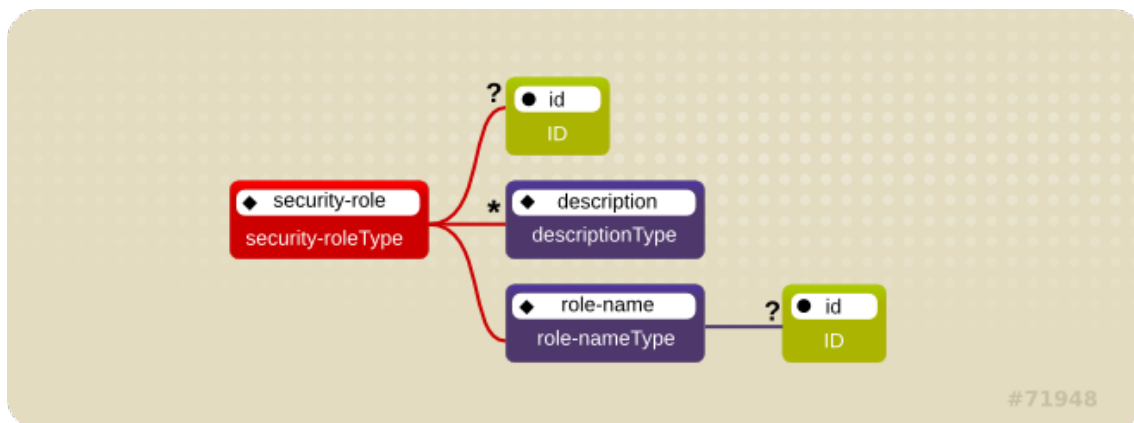


図1.3 <security-role> 要素

JBoss では <security-role> 要素が使用されるのは、<security-role-ref> 値または <role-name> 値をコンポーネントロールが参照する論理ロールにマップするときだけです。JBossSX 実装の詳細を説明する時に触れますが、ユーザーの割り当てられたロールは、アプリケーションのセキュリティマネージャの動的関数になります。

JBoss ではメソッドパーミッションを宣言するために <security-role> 要素の定義は必要ありません。ただし、<security-role> 要素の仕様は、アプリケーションサーバー全体で移植性を確保し、デプロイメント記述子を管理するために今でも推奨されています。例1.3「[ejb-jar.xml 記述子の一部](#)」では **ejb-jar.xml** ファイルの <security-role> の使用方法について説明しています。

例1.3 ejb-jar.xml 記述子の一部

```
<!-- 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>
```

例1.4「[web.xml 記述子の一例](#)」では **web.xml** ファイルの <security-role> の使用方法について示しています。

例1.4 web.xml 記述子の一例

```
<!-- 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. EJB メソッドパーミッション

アプリケーションアセンブラは <method-permission> 要素宣言を使用して、EJB のホームとリモートインターフェースのメソッドを呼び出すために可能なロールを設定できます。

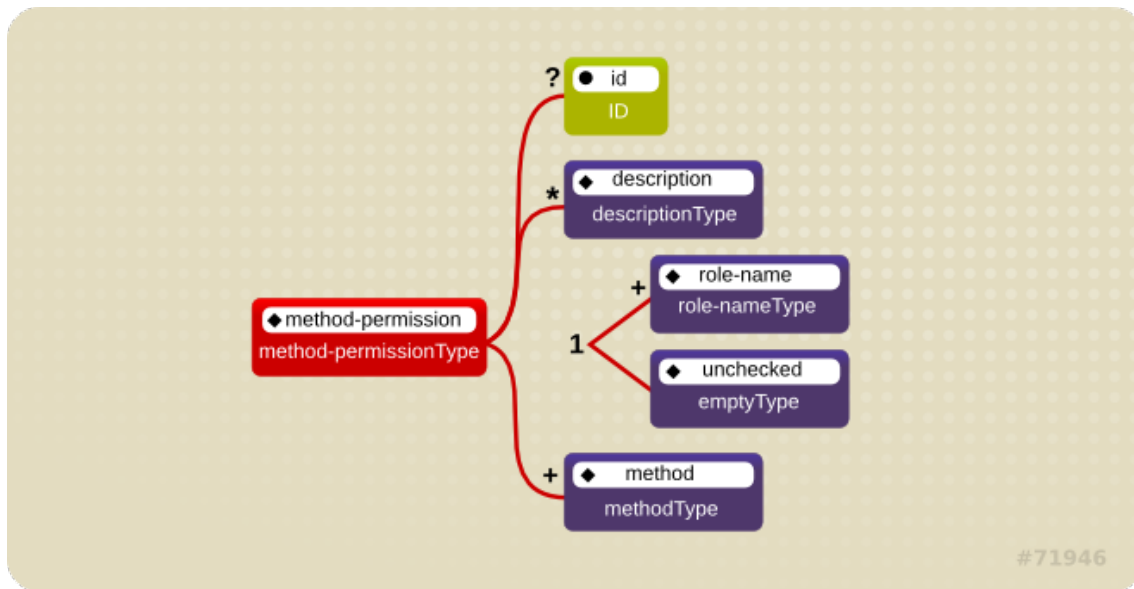


図1.4 <method-permission> 要素

各 <method-permission> 要素には <role-name> 子要素が 1 つ以上含まれています。<role-name> は <method> 子要素で識別されるとおり EJB メソッドにアクセスできる論理ロールを定義します。<role-name> 要素の代わりに <unchecked> 要素を指定して、認証ユーザーなら誰でもメソッド子要素で識別されたメソッドにアクセスできることを宣言できます。さらに、誰も exclude-list 要素を持つメソッドにアクセスできないことも宣言できます。EJB に <method-permission> 要素を使用してロールでアクセスできると宣言されていないメソッドがある場合、EJB メソッドは使用から除外されるようデフォルト設定します。これはメソッドを **exclude-list** にデフォルト設定するのと同様です。

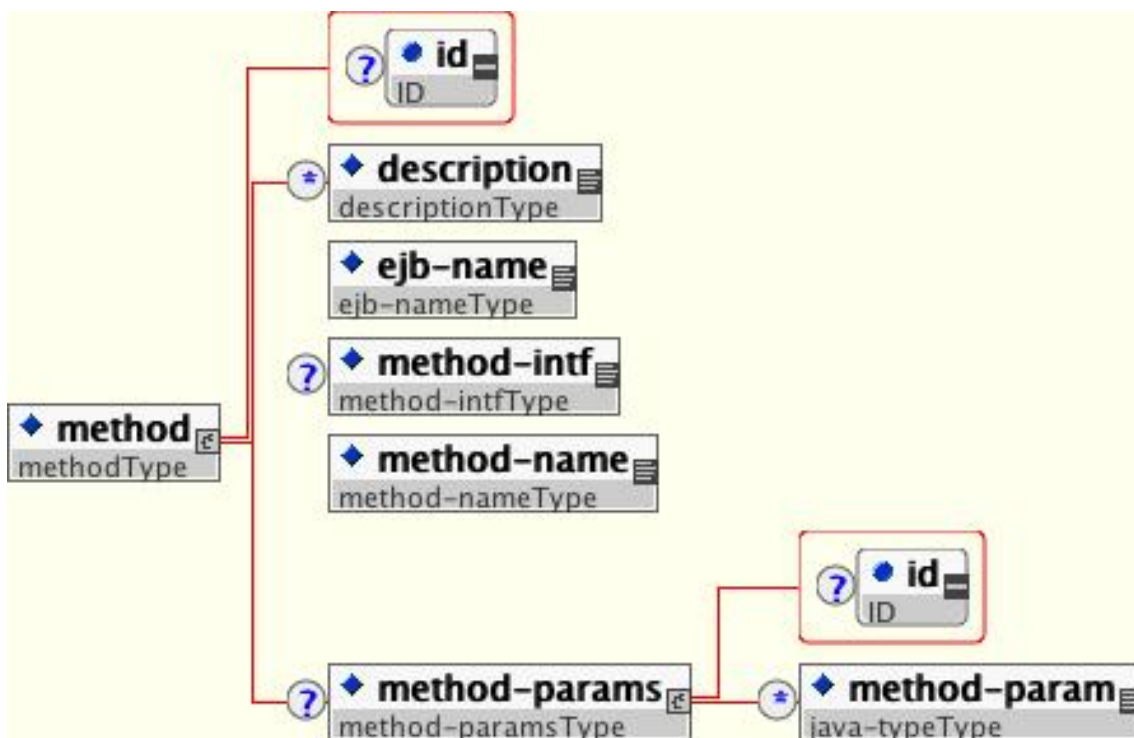


図1.5 <method> 要素

メソッド要素宣言にはサポートされているスタイルが3つあります。

1つ目は名前付きエンタープライズ Bean のホームとコンポーネントインターフェースメソッドすべてを参照するために使用します。

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

2つ目のスタイルは名前付きエンタープライズ Bean のホームまたはコンポーネントインターフェースの指定されたメソッドを参照するために使用されます。

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

同じオーバーロードした名前を持つメソッドが複数ある場合は、このスタイルはオーバーロードしたメソッドすべてを参照します。

3つ目のスタイルはオーバーロードした名前を持つメソッドのセット内で指定されたメソッドを参照するために使用されます。

```
<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>
```

メソッドは特定のエンタープライズ Bean のホームまたはリモートインターフェースで定義される必要があります。<method-param> 要素値は、対応するメソッドパラメータタイプの完全修飾名です。同じオーバーロードされたシグネチャを持つメソッドが複数ある場合は、パーミッションは一致するオーバーロードしたメソッドすべてに当てはまります。

オプションの <method-intf> 要素は、エンタープライズ Bean のホームとリモートインターフェースで定義された同じ名前とシグネチャを持つメソッドを区別するために使用できます。

例1.5「<method-permission> 要素の使用方法」には <method-permission> 要素の使用方法の完全な例が示されています。

例1.5 <method-permission> 要素の使用方法

```
<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,
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>

```

```
</exclude-list>
</assembly-descriptor>
</ejb-jar>
```

1.5. エンタープライズ BEAN セキュリティアノテーション

エンタープライズ Bean はアノテーションを使用して、アプリケーションのセキュリティや他の側面についての情報をデプロイに渡します。アノテーションまたはデプロイメント記述子で指定されている場合、デプロイはアプリケーションに適切なエンタープライズ Bean セキュリティポリシーを設定することができます。

デプロイメント記述子で明示的に指定されるすべてのメソッド値はアノテーション値を上書きします。メソッド値がデプロイメント記述子で指定されていない場合は、アノテーションを使用して設定された値が使用されます。上書きする粒度はメソッドごとです。

セキュリティに対処し、エンタープライズ Bean で使用できるアノテーションには次が含まれます。

@DeclareRoles

コードで宣言される各セキュリティロールを宣言します。ロールの設定に関する詳細は『Java EE 5 Tutorial』 [Declaring Security Roles Using Annotations](#) を参照してください。

@RolesAllowed, @PermitAll, and @DenyAll

アノテーションに対してメソッドパーミッションを指定します。アノテーションメソッドパーミッションの設定に関する詳細は『Java EE 5 Tutorial』 [Specifying Method Permissions Using Annotations](#) を参照してください。

@RunAs

コンポーネントの伝播したセキュリティアイデンティティを設定します。アノテーションを使用した伝播したセキュリティアイデンティティの設定についての詳細は『Java EE 5 Tutorial』 [Configuring a Component's Propagated Security Identity](#) を参照してください。

1.6. WEB コンテンツのセキュリティ制約

Web アプリケーションでは、保護されたコンテンツを識別する URL パターンによってコンテンツへのアクセスが許可されるロールがセキュリティを定義します。この一連の情報は、**web.xml** security-constraint 要素を使用して宣言されます。

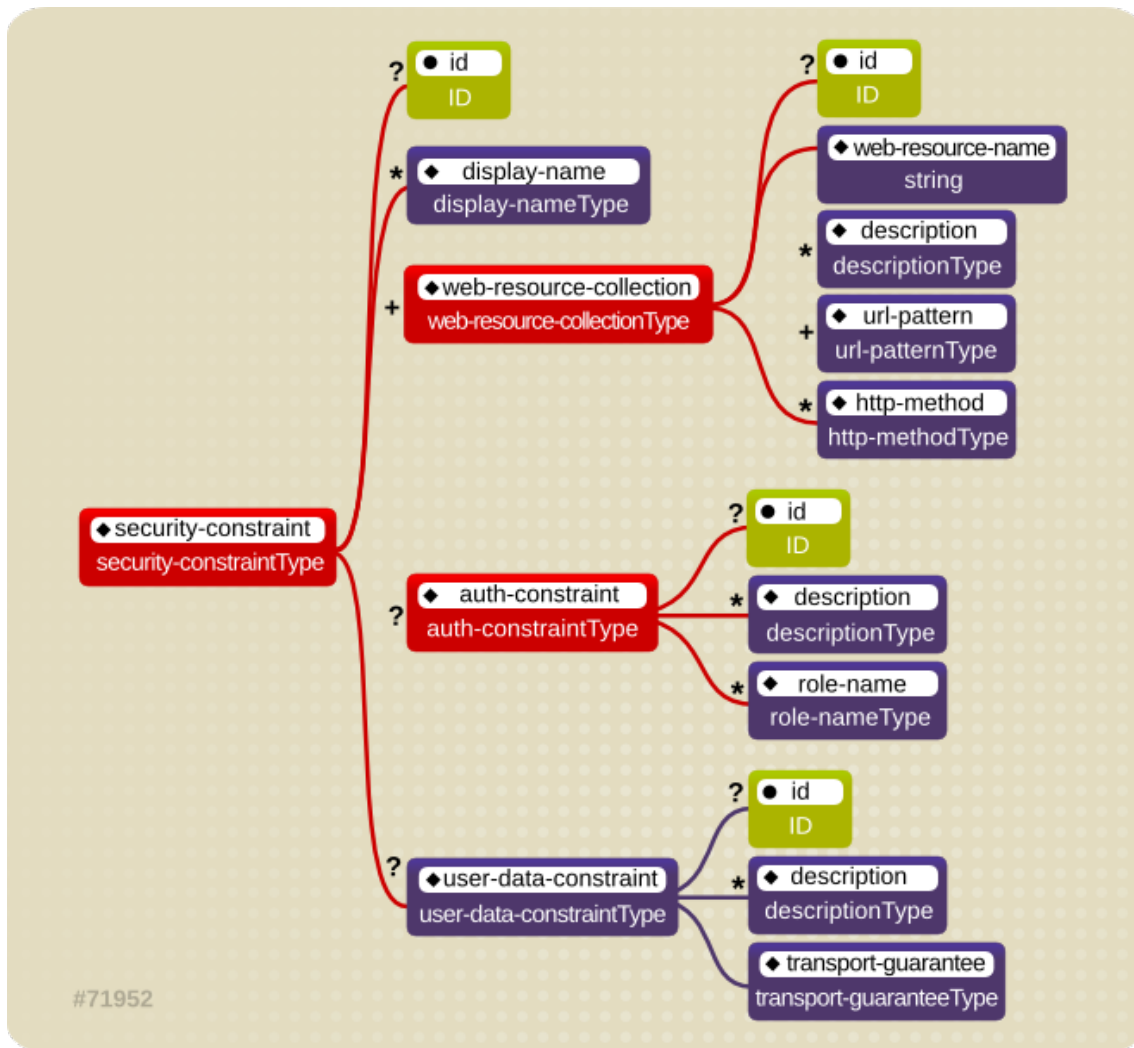


図1.6 <security-constraint> 要素

セキュリティ保護されたコンテンツは 1 つ以上の <web-resource-collection> 要素を使用して宣言されます。各 <web-resource-collection> 要素にはオプションの <url-pattern> 要素があり、その後にオプションの <http-method> 要素が含まれます。<url-pattern> 要素値は、要求 URL が要求に対して一致しなければならない URL パターンを指定し、安全なコンテンツへのアクセスに対応します。<http-method> 要素値は許可する HTTP 要求のタイプを指定します。

オプションの <user-data-constraint> 要素はサーバー接続のクライアントのトランスポートレイヤに対する要件を指定します。要件はコンテンツの整合性 (通信プロセスでのデータの改ざんを防ぐ) または機密性 (伝送中の読み取りを防ぐ) を目的とすることがあります。<transport-guarantee> 要素値はクライアントとサーバー間でどの通信がどの程度保護されるかを指定します。その値は

NONE、**INTEGRAL**、**CONFIDENTIAL** になります。**NONE** の値は、アプリケーションはどのトランスポートの保証も必要としないことを意味します。**INTEGRAL** の値は、アプリケーションでは伝送中に変化不可能な方法でクライアントとサーバー間で送信されたデータが送信される必要があることを意味します。**CONFIDENTIAL** の値は、アプリケーションでは他のエンティティが送信のコンテンツを監視しない方法でデータが送信される必要があることを意味します。ほとんどの場合、**INTEGRAL** か **CONFIDENTIAL** のフラグがあると SSL の使用が必要になります。

オプションの <login-config> 要素は使用されるべき認証メソッドやアプリケーションに使用されるべきレルム名、フォームログインメカニズムで必要な属性を設定するために使用します。

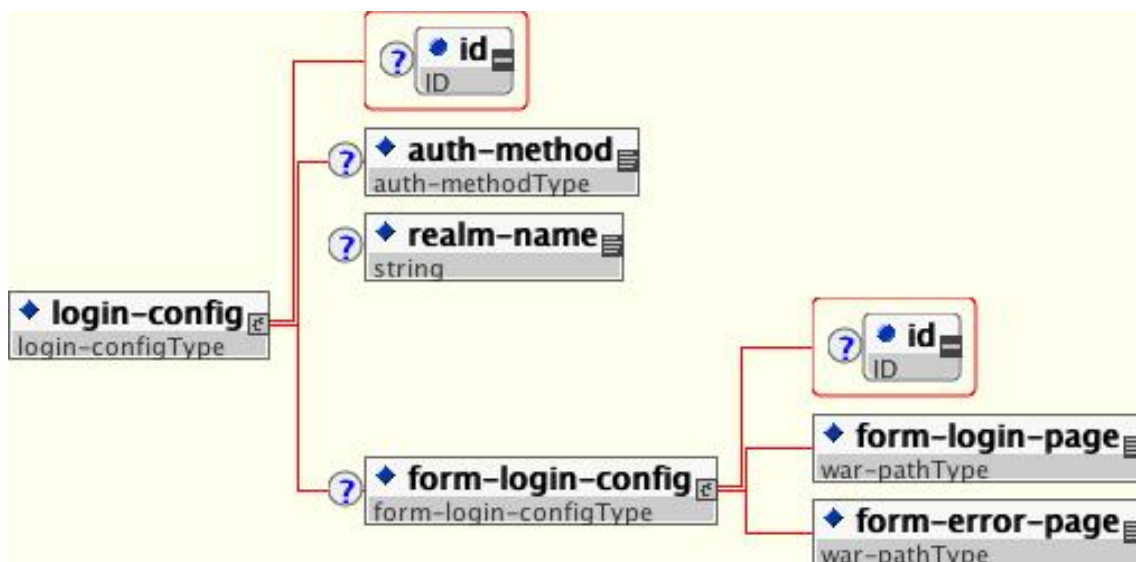


図1.7 <login-config> 要素

<auth-method> 子要素は Web アプリケーションに対し認証メカニズムを指定します。承認制約によって保護されている Web リソースへのアクセスを取得する前提条件として、ユーザーは設定されたメカニズムを使用して認証されている必要があります。有効な <auth-method> 値は **BASIC**、**DIGEST**、**FORM**、**CLIENT-CERT** です。<realm-name> 子要素は HTTP ベーシックおよびダイジェスト認証で使用するレルム名を指定します。<form-login-config> 子要素はフォームベースのログインで使用するエラーページだけでなくログインも指定します。<auth-method> 値が **FORM** でない場合は、**form-login-config** とその子要素は無視されます。

例1.6 「web.xml 記述子の一部」は Web アプリケーションの **/restricted** パスにあるすべての URL が **AuthorizedUser** ロールを必要とすることを示しています。必要となるトランスポートの保証はなく、ユーザーアイデンティティの取得に使用される認証メソッドは BASIC HTTP 認証になります。

例1.6 web.xml 記述子の一部

```

<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. フォームベースの認証を有効にする

フォームベースの認証により、ログインのカスタム JSP/HTML ページ、ログイン時にエラーが発生した場合にユーザーが移動される別のページを柔軟に定義することができます。

フォームベースの認証はデプロイメント記述子 **web.xml** の `<login-config>` 要素の `<auth-method>FORM</auth-method>` を含むることにより定義されます。ログインページとエラーページは以下のとおり `<login-config>` でも定義されます。

```

<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>

```

フォームベースの認証を持つ Web アプリケーションがデプロイされる場合、Web コンテナは **FormAuthenticator** を使用して、ユーザーを適切なページへ移動します。JBoss Enterprise Application Platform はセッションプールを管理することで、認証情報が各要求に対して存在する必要はありません。**FormAuthenticator** が要求を受け取る場合、既存のセッションに対し **org.apache.catalina.session.Manager** をクエリします。セッションが存在しない場合は、新しいセッションが作成されます。その後、**FormAuthenticator** がセッションの資格情報を検証します。



注記

各セッションは無作為の値から生成された 16 バイトの文字列であるセッションアイデンティティにより識別されます。こうした値はデフォルトで **/dev/urandom** (Linux) から取得され、MD5 でハッシュされます。セッションアイデンティティの作成時にそれが一意であることを確認します。

検証された時点で、セッションアイデンティティはクッキーの一部として割り当てられ、クライアントに返されます。このクッキーは後続のクライアント要求で必要になり、ユーザーセッションを識別するために使用されます。

クライアントに渡されたクッキーはオプションの属性を複数持つ名前と値のペアです。識別子の属性は **JSESSIONID** と呼ばれます。その値はセッションアイデンティティの HEX 文字列です。このクッキーは非永続と設定されます。これはクライアント側ではブラウザが存在する場合は削除されることを意味します。サーバー側ではセッションは非アクティブの時間が 60 秒を越えると期限切れになり、その時にセッションオブジェクトと資格情報が削除されます。

例えば、ユーザーがフォームベースの認証で保護されている Web アプリケーションへのアクセスを試みているとしましょう。**FormAuthenticator** は要求をキャッシュし、必要であれば新しいセッションを作成し、ユーザーを **login-config** で定義されたログインページに再移動させます (前回の例のコードでは、ログインページは **login.html** です)。次に、ユーザーは提供されている HTML フォームにユーザー名とパスワードを入力します。ユーザー名とパスワードは **j_security_check** フォームアクションにより **FormAuthenticator** に渡されます。

次に、**FormAuthenticator** は Web アプリケーションコンテキストに接続されるレルムに対してユーザー名とパスワードを認証します。JBoss Enterprise Application Platform では、レルムは **JBossWebRealm** になります。認証が成功すると、**FormAuthenticator** はキャッシュから保存した要求を取得し、ユーザーを元の要求に再移動させます。



注記

サーバーがフォーム認証要求を認識するのは、URI の最後が **/j_security_check** で終わり、少なくとも **j_username** と **j_password** パラメータが存在しているときだけです。

1.8. 宣言型セキュリティを有効にする

これまで取り上げた Java EE セキュリティの要素は、アプリケーションの観点からのみのセッション要件について説明しています。Java EE セキュリティの要素では論理ロールを宣言するため、アプリケーションデプロイヤーはアプリケーションドメインからデプロイメント環境にロールをマップします。Java EE 仕様ではこうしたアプリケーションのサーバー固有の詳細を省略します。

デプロイメント環境にアプリケーションロールをマップするためには、JBoss サーバー固有のデプロイメント記述子を使用して Java EE セキュリティモデルを実装するセキュリティマネージャを指定する必要があります。セキュリティ設定に関する詳細は [例12.16「JndiUserAndPass カスタムのログインモジュール」](#) に記載されています。

第2章 JAAS の概略

JBossSX フレームワークは JAAS API を基にしています。JBossSX の実装について詳しく理解するためには、JAAS API の基本要素を理解する必要があります。JBossSX アーキテクチャについては本書の後半で説明するため、次項では JAAS について少し説明しておきます。

JAAS 1.0 API はユーザー認証と承認を目的としてつくられた Java パッケージのセットで構成されています。API は標準の PAM (Pluggable Authentication Modules : プラグ可能な認証モジュール) フレームワークの Java バージョンを実装し、ユーザーベースの承認に対応するよう Java 2 Platform のアクセス制御アーキテクチャを拡張します。

JAAS は最初 JDK 1.3 の拡張パッケージとしてリリースされ、JDK 1.5 に同梱されています。JBossSX フレームワークは JAAS の認証機能のみを使用して、宣言型ロールベースの J2EE セキュリティモデルを実装するため、本項ではこの部分にのみ焦点を置いて説明します。

JAAS 認証はプラグ可能な方法で実行されます。これにより Java アプリケーションが基礎となる認証技術に依存することなく、JBossSX セキュリティマネージャが異なるセキュリティインフラストラクチャで動作することが可能になります。セキュリティインフラストラクチャとの統合は JBossSX セキュリティマネージャの実装を変更することなく実現できます。変更が必要なのは、JAAS が使用する認証スタックの設定のみです。

2.1. JAAS コアクラス

JAAS コアクラスは共通、認証、承認の 3 つのカテゴリに区分的ことができます。本項で取り上げる JBossSX の機能性を実装するために使用されるクラスは共通と認証のクラスであるため、以下の一覧ではそれら 2 つのみ記載します。

共通のクラスは次の通りです。

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

認証のクラスは次の通りです。

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

2.1.1. サブジェクトとプリンシパルクラス

リソースへのアクセスを承認するには、アプリケーションは最初に要求元を認証する必要があります。JAAS フレームワークは要求元を表すための用語サブジェクトを定義します。**Subject** のクラスは JAAS の中心クラスです。**Subject** は人やサービスなど単一のエンティティの情報を表します。情報はエンティティのプリンシパル、パブリックの資格情報、プライベートの資格情報などに渡ります。JAAS API は既存の Java 2 `java.security.Principal` インターフェースを使用して、プリンシパルを表します。プリンシパルは基本的に入力された名前になります。

認証プロセスでは、サブジェクトには関連付けられたアイデンティティまたはプリンシパルが含まれて

います。サブジェクトは多くのプリンシパルを持つことができます。例えば、一個人は名前のプリンシパル (John Doe)、ソーシャルセキュリティ番号のプリンシパル (123-45-6789)、ユーザー名のプリンシパル (johnd) を持つことができ、これらすべては他のサブジェクトから区別するのに役立ちます。1 つのサブジェクトに関連付けられたプリンシパルを取得する方法は 2 つあります。

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

getPrincipals() はサブジェクトに含まれるすべてのプリンシパルを返します。**getPrincipals(Class c)** はクラス **c** のインスタンスまたはそのサブクラスのひとつであるプリンシパルのみを返します。サブジェクトに一致するプリンシパルがない場合は空のセットが返されます。

java.security.acl.Group インターフェースは **java.security.Principal** のサブインターフェースであるため、プリンシパルのセットのインスタンスは他のプリンシパルやプリンシパルのグループの論理グループを表します。

2.1.2. サブジェクトの認証

サブジェクトの認証には JAAS ログインが必要です。ログイン手順は次のようになります。

1. アプリケーションは **LoginContext** のインスタンスを作成し、ログイン設定の名前と **CallbackHandler** を渡して、設定 **LoginModule** で必要とされるとおり **Callback** オブジェクトを追加します。
2. **LoginContext** は、名前付きログイン設定に含まれているすべての **LoginModules** をロードするよう **Configuration** と確認します。そうした名前付き設定が存在しない場合は、**other** 設定がデフォルトとして使用されます。
3. アプリケーションが **LoginContext.login** メソッドを呼び出します。
4. ログインメソッドはロードされたすべての **LoginModule** を呼び出します。各 **LoginModule** はサブジェクトの認証を試行するため、関連付けられた **CallbackHandler** でハンドルメソッドを呼び出し、認証プロセスに必要な情報を取得します。必要な情報は **Callback** オブジェクトの配列の形式でハンドルメソッドに渡されます。成功すると、**LoginModule** は関連のプリンシパルと資格情報をサブジェクトに関連付けします。
5. **LoginContext** はアプリケーションに認証状態を返します。ログインメソッドからの返されると成功となります。ログインメソッドによって **LoginException** がスローされると失敗となります。
6. 認証が成功したら、アプリケーションは **LoginContext.getSubject** メソッドを使用して認証されたサブジェクトを取得します。
7. サブジェクトの認証が完了した後に **LoginContext.logout** メソッドを呼び出すと、ログインメソッドによりサブジェクトに関連付けられたすべてのプリンシパルと関連情報を削除することができます。

LoginContext クラスは認証しているサブジェクトに基本メソッドを提供し、基礎となる認証技術に依存しないアプリケーションを開発する方法を提供します。**LoginContext** は特定のアプリケーション向けに設定された認証サービスを決定するよう **Configuration** と確認します。**LoginModule** クラスは認証サービスを表します。そのため、アプリケーション自体を変更することなくログインモジュールをアプリケーションにプラグインすることが可能です。次のコードは、サブジェクトを認証するためアプリケーションに必要なステップを示しています。

■


```

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");
            }
        }
    }
}

```

開発者は **LoginModule** インターフェースの実装を作成することで、認証技術を統合します。これにより管理者は異なる認証技術を 1 つのアプリケーションにプラグインできます。複数の **LoginModule** をチェーン化し複数の認証技術を認証プロセスに加えることが可能です。例えば、ある **LoginModule** がユーザー名 / パスワードベースの認証を行い、別の **LoginModule** はスマートカードリーダーや生体認証などのハードウェアデバイスに接続することができます。

LoginModule のライフサイクルは、クライアントがログインメソッドを作成し公開する **LoginContext** オブジェクトによって決定されます。このプロセスには 2 つのフェーズがあり、プロセスの手順は次のようになります。

- **LoginContext** は引数のないパブリックコンストラクタを使用して、設定された **LoginModule** を作成します。
- 各 **LoginModule** は **initialize** メソッドへの呼び出しによって初期化されます。**Subject** 引数は **null** 以外になることが保証されます。**initialize** メソッドのシグネチャは **public void initialize(Subject subject, CallbackHandler callbackHandler, Map sharedState, Map options)** です。
- **login** メソッドは認証プロセスを開始するために呼び出されます。例えば、あるメソッド実装はユーザーにユーザー名とパスワードの入力を求め、NIS または LDAP などのネーミングサービスで保存されているデータに対してこの情報を確認することがあります。別の実装ではスマートカードや生体認証デバイスに接続されるか、単に基礎となるオペレーティングシステムからユーザー情報を抽出することがあります。各 **LoginModule** によるユーザーアイデンティティの検証は JAAS 認証のフェーズ 1 とみなされます。**login** メソッドのシグネチャは **boolean login() throws LoginException** です。**LoginException** は失敗を意味します。**true** の戻り値はメソッドが成功したことを示し、**false** の戻り値はログインモジュールが無視されることを示しています。
- **LoginContext** の全体的な認証が成功すると、各 **LoginModule** で **commit** が呼び出されます。フェーズ 1 が **LoginModule** に対し成功すると、コミットメソッドではフェーズ 2 が続き、関連するプリンシパル、パブリックの資格情報、プライベートの資格情報をサブジェクトに関連付けます。フェーズ 1 が **LoginModule** に対し失敗すると、**commit** はユーザー名やパスワードなど以前に保存していた認証状態をすべて削除します。**commit** メソッドのシグネチャは **boolean commit() throws LoginException** です。**LoginException** がスローされると、コミットフェーズの完了が失敗したことを示します。**true** が返されるとメソッドが成功したことを示し、**false** が返されるとログインモジュールが無視されることを示します。
- **LoginContext** の全体的な認証が失敗すると、各 **LoginModule** で **abort** メソッドが呼び出されます。**abort** メソッドはログインまたは **initialize** メソッドによって作成されたすべての認証状態を削除または破棄します。**abort** メソッドのシグネチャは **boolean abort() throws LoginException** です。**LoginException** がスローされると **abort** フェーズの完了が失敗したことを示します。**true** が返されるとメソッドが成功したことを示し、**false** が返されるとログインモジュールが無視されることを示します。
- ログイン成功後に認証状態を削除するには、アプリケーションは **LoginContext** で **logout** を呼び出します。これにより、各 **LoginModule** で **logout** メソッド呼び出しが発生します。**logout** メソッドは **commit** 動作時に当初サブジェクトに関連付けられていたプリンシパルと資格情報を削除します。資格情報は削除時に破棄されるべきです。**logout** メソッドのシグネチャは **boolean logout() throws LoginException** です。**LoginException** がスローされるとログアウトプロセスの完了が失敗したことを示します。**true** が返されるとメソッドが成功したことを示し、**false** が返されるとログインモジュールが無視されることを示します。

LoginModule が認証情報を取得するためユーザーと通信する必要がある場合、**CallbackHandler** オブジェクトを使用します。アプリケーションは **CallbackHandler** インターフェースを実装してそれを **LoginContext** に渡し、直接基礎となるログインモジュールに認証情報を送ります。

ログインモジュールは、パスワードやスマートカード PIN などのユーザーからの入力を収集し、状態情報などをユーザーに提供するために **CallbackHandler** を使用します。アプリケーションに **CallbackHandler** を指定できるようにすることで、基礎となる **LoginModule** はアプリケーションがユーザーと対話する様々な方法に依存しない状態を維持します。例えば GUI アプリケーションの **CallbackHandler** の実装は、ウィンドウを表示してユーザーの入力を求めることがあります。一方でアプリケーションサーバーなど GUI でない環境の **CallbackHandler** 実装は、アプリケーションサーバー API を使用して単に資格情報を取得することがあります。**CallbackHandler** インターフェースには実装するメソッドが 1 つあります。

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

最後に説明する認証クラスは **Callback** インターフェースです。これは複数のデフォルト実装が提供されているタグ付けインターフェースで、前述の例で使用した **NameCallback** と **PasswordCallback** が含まれます。**LoginModule** は **Callback** を使用し、認証メカニズムで必要となる情報を要求します。**LoginModule** は認証のログインフェーズの間に **Callback** のアレイを直接 **CallbackHandler.handle** メソッドに渡します。**callbackhandler** がハンドルのメソッドに渡された **Callback** オブジェクトの使用方法が分からない場合は、**UnsupportedCallbackException** をスローしてログイン呼び出しを中止します。

第3章 JBOSS セキュリティモデル

他の JBoss アーキテクチャと同様に、最低限レベルのセキュリティは代替の実装が提供されるインターフェースのセットとして定義されます。次のインターフェースは JBoss サーバーセキュリティレイヤを定義します。

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

図3.1「JBoss サーバー EJB コンテナ要素とのセキュリティモデルのインターフェース関係」はセキュリティインターフェースと EJB コンテナアーキテクチャとの関係のクラスダイアグラムを示しています。

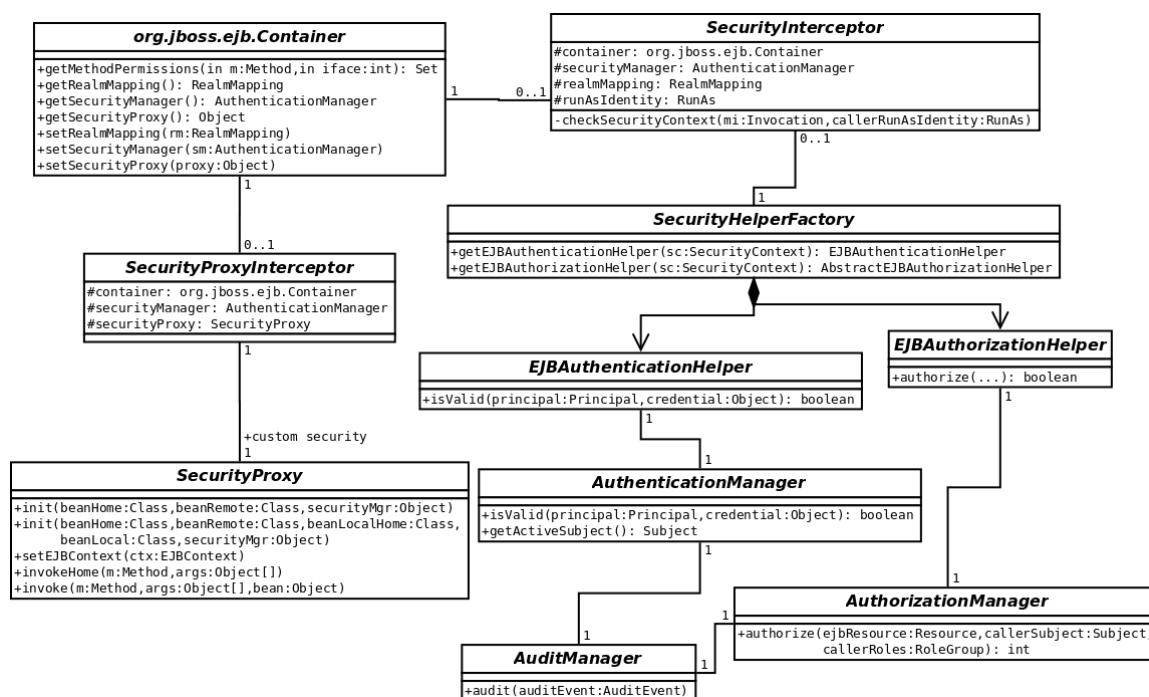


図3.1 JBoss サーバー EJB コンテナ要素とのセキュリティモデルのインターフェース関係

EJB コンテナレイヤはクラス

`org.jboss.ejb.Container`、`org.jboss.SecurityInterceptor`、`org.jboss.SecurityProxyInterceptor` で表されます。その他のクラスは JBoss セキュリティサブシステムにより提供されるインターフェースとクラスです。

J2EE セキュリティモデルの実装に必要な 2 つのインターフェースは以下のとおりです。

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

図3.1「JBoss サーバー EJB コンテナ要素とのセキュリティモデルのインターフェース関係」で示されたセキュリティインターフェースのロールは以下のとおり要約されます。

セキュリティインターフェースのロール

AuthenticationManager

このインターフェースの役割は プリンシパル と関連付けられる資格情報を検証することです。Principal とはユーザー名、雇用者番号、ソーシャルセキュリティ番号などのアイデンティティです。資格情報はパスワード、セッションキー、デジタル署名などアイデンティティの証明です。isValid メソッドは呼び出されて、ユーザーアイデンティティと動作環境で既知の関連付けられた資格情報がユーザーアイデンティティの有効な証明となるかを決定します。

AuthorizationManager

このインターフェースの役割は Java EE 仕様により指示されたアクセス制御です。このインターフェースの実装により、プラグ可能な承認に役立つ Policy Provider のセットをスタックできます。

SecurityProxy

このインターフェースはカスタムの SecurityProxyInterceptor プラグインの要件を記述します。SecurityProxy により EJB ホームとリモートインターフェースメソッドの両方に対してメソッドベースでカスタムセキュリティチェックの外部化が可能になります。

AuditManager

このインターフェースの役割はセキュリティイベントの監査証拠を提供することです。

MappingManager

このインターフェースの役割は Principal、Role、Attribute のマッピングを提供することです。AuthorizationManager の実装は内部的にマッピングマネージャを呼び出し、アクセス制御を実行する前にロールをマップすることができます。

SecurityDomain

これは AuthenticationManager、RealmMapping、SubjectSecurityManager インターフェースの拡張です。SecurityDomain がコンポーネントでセキュリティを実装するための推奨される方法です。その理由として、JAAS Subject が持つ利点、ASP スタイルアプリケーションとリソースデプロイメントに提供される強化したサポートがあげられます。java.security.KeyStore、Java Secure Socket Extension (JSSE) com.sun.net.ssl.KeyManagerFactory、com.sun.net.ssl.TrustManagerFactory インターフェースはクラスに含まれます。

RealmMapping

このインターフェースの役割はプリンシパルマッピングとロールマッピングです。getPrincipal メソッドは動作環境でユーザーアイデンティティを既知として取り、アプリケーションドメインアイデンティティを返します。doesUserHaveRole メソッドは動作環境のユーザーアイデンティティがアプリケーションドメインから指定されたロールを割り当てられていることを検証します。

AuthenticationManager、RealmMapping、SecurityProxy インターフェースは JAAS 関連クラスと関連がないことに注意してください。JBossSX フレームワークは JAAS に大きく依存していますが、Java EE セキュリティモデルの実装に必要な基本的なセキュリティインターフェースは依存していません。JBossSX フレームワークは JAAS に基づく基本的なセキュリティプラグインインターフェースの単なる実装です。

これについては [図3.2「JBossSX フレームワーク実装クラスと JBoss サーバー EJB コンテナレイヤ」](#) のコンポーネント図で示されています。このプラグインアーキテクチャでは、JAAS ベースの JBossSX 実装クラスを自由に非 JAAS カスタムセキュリティマネージャ実装に置き換えることができることを表しています。この方法については [図3.2「JBossSX フレームワーク実装クラスと JBoss サーバー EJB コンテナレイヤ」](#) の JBossSX 設定に使用できる JBossSX MBean を参照してください。

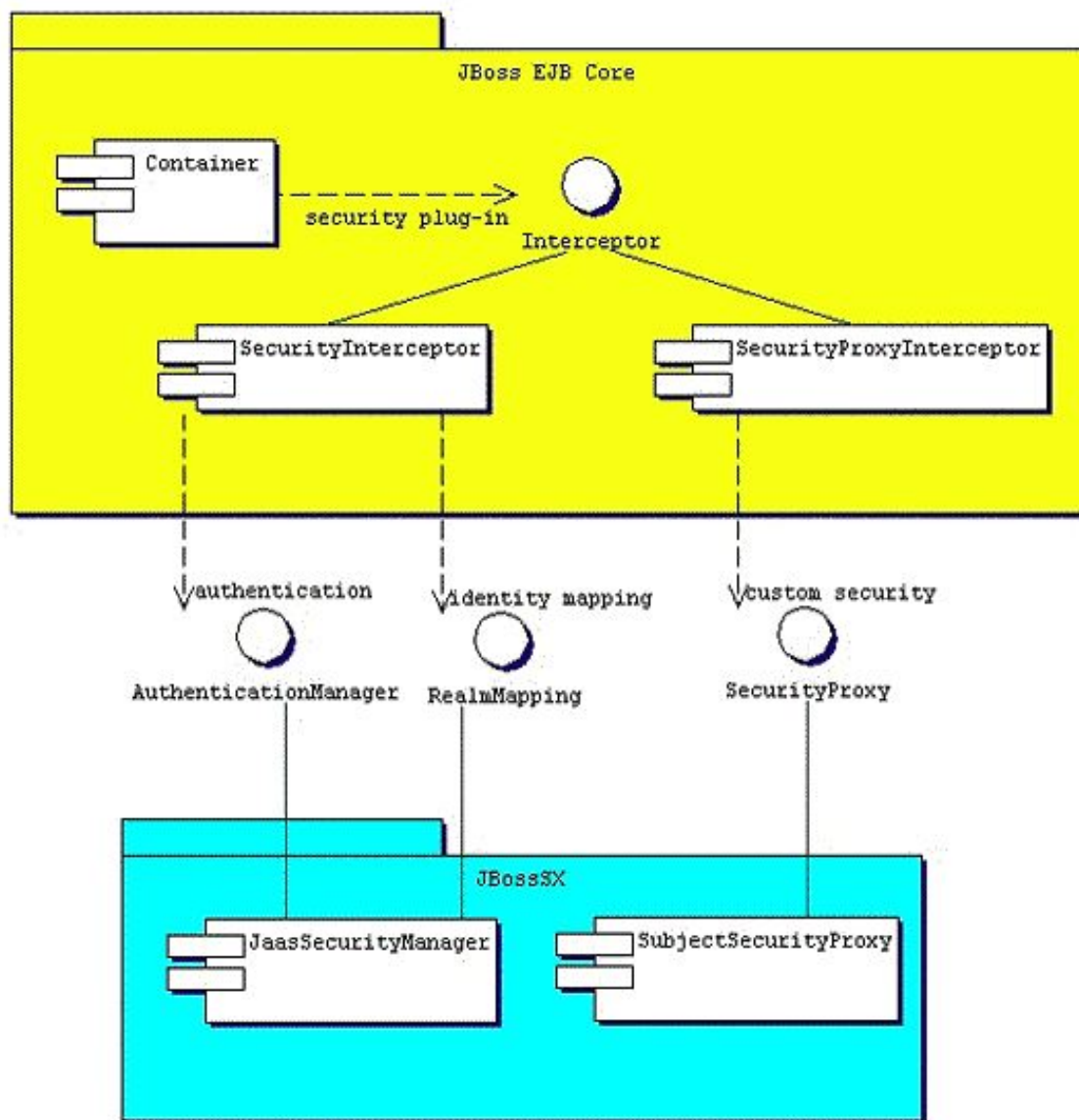


図3.2 JBossSX フレームワーク実装クラスと JBoss サーバー EJB コンテナレイヤ

3.1. 宣言型セキュリティの有効化の再確認

本項の冒頭で Java EE 標準セキュリティモデルに関する説明は、セキュリティを有効にする JBoss サーバー固有のデプロイメント記述子を使用する要件を説明して終わりました。この設定に関する詳細は [図3.3「jboss.xml と jboss-web.xml セキュリティ要素のサブセット」](#) に示されています。ここでは、JBoss 固有の EJB と Web アプリケーションのデプロイメント記述子のセキュリティ関連要素を示しています。



図3.3 jboss.xml と jboss-web.xml セキュリティ要素のサブセット

<security-domain> 要素の値は、JBoss が EJB と Web コンテナに使用するセキュリティマネージャインターフェース実装の JNDI 名を指定します。これは **AuthenticationManager** と **RealmMapping** インターフェースの両方を実装するオブジェクトです。トップレベルの要素として指定すると、デプロイメントユニットのすべての EJB に対しどのセキュリティドメインを指定するかを定義します。デプロイメントユニット内で複数のセキュリティマネージャを混在させると、内部コンポーネントの動作や管理が複雑になるため、これが一般的な使用方法です。

個別の EJB にセキュリティドメインを指定するには、コンテナの設定レベルで <security-domain> を指定します。これによりすべてのトップレベルの <security-domain> 要素を上書きします。

<unauthenticated-principal> 要素は、認証されていないユーザーが EJB を呼び出す場合に **EJBContext.getUserPrincipal** メソッドにより返される **Principal** オブジェクトに使用する名前を指定します。これにより認証されていない呼び出し側に特別なパーミッションを伝えることはありません。主要な目的は、セキュアでないサーブレットと JSP ページがセキュアでない EJB を呼び出すことができ、目的の EJB が **getUserPrincipal** メソッドを使用して呼び出し側に対して null でない **Principal** を取得できるようにすることです。これは J2EE 仕様要件です。

<security-proxy> 要素はカスタムのセキュリティプロキシ実装を特定します。これにより、セキュリティロジックを EJB 実装に組み込むことなく EJB 宣言型セキュリティモデルのスコープ外で要求ごとのセキュリティチェックが可能になります。**org.jboss.security.SecurityProxy** インターフェースの実装を使用できます。別の方法として、EJB のホーム、リモート、ローカルホーム、またはローカルインターフェースでメソッドを実装するオブジェクトを使用する共通のインターフェースを使用することもできます。特定のクラスが **SecurityProxy** インターフェースを実装しない場合は、イ

ンスタンスはメソッド呼び出しをオブジェクトに委譲する **SecurityProxy** 実装でラップされる必要があります。**org.jboss.security.SubjectSecurityProxy** はデフォルトの JBossSX インストールで使用される **SecurityProxy** 実装の一例です。

簡易ステートレスセッション Bean のコンテキストのカスタム **SecurityProxy** の簡単な例を見てみましょう。カスタムの **SecurityProxy** は、Bean の **echo** メソッドを 4 文字の単語を使用してその引数として呼び出す者がいないことを検証します。これはロールベースのセキュリティでは不可能な確認です。セキュリティコンテキストはメソッド引数であり、呼び出し側のプロパティではないため **FourLetterEchoInvoker** ロールを定義することはできません。カスタムの **SecurityProxy** のコードは [例3.1「カスタムの EchoSecurityProxy 実装」](#) を参照してください。

例3.1 カスタムの EchoSecurityProxy 実装

```
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
    }
}

```

EchoSecurityProxy は、Bean インスタンスで呼び出されるメソッドが **init** メソッドをロードした **echo(String)** メソッドと一致することを確認します。一致があると、メソッド引数は取得され、その長さが 4 または null に対して比較されます。どちらの場合でも **SecurityException** がスローされます。

これは確かに不自然な例ですが、不自然なのはそのアプリケーション内のみです。アプリケーションがメソッド引数の値に基づいてセキュリティチェックを実行する必要があることは一般的な要件です。この例のポイントは、標準宣言型セキュリティモデルの範囲を越えたカスタムのセキュリティが Bean の実装に依存せずに導入できる方法を示していることです。これによりセキュリティ要件の仕様とコーディングをセキュリティの専門家に委譲することが可能です。セキュリティプロキシレイヤは Bean 実装に依存せずに行うことが可能なため、セキュリティはデプロイメント環境の要件と一致するよう変更できます。

EchoBean に対するカスタムのプロキシとして **EchoSecurityProxy** をインストールする関連付けられた **jboss.xml** 記述子は [例3.2「jboss.xml 記述子」](#) で示されています。

例3.2 jboss.xml 記述子

```

<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>

```

さて、この一部で図解されたように引数 **Hello** と **Four** を付けて **EchoBean.echo** メソッドを呼び出そうとするクライアントを実行して、カスタムのプロキシをテストしてみましょう。

```
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"));
    }
}
```

最初の呼び出しは成功するはずですが、2 番目の呼び出しは、**Four** が 4 文字の単語であるため失敗するはず。examples ディレクトリから Ant を使用して次のようにクライアントを実行します。

```
[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
...
```

予想通り **echo('Hello')** メソッド呼び出しは成功し、**echo('Four')** メソッド呼び出しは乱雑な例外という結果となりました。上記の出力は本書では紙面の都合上省略されています。例外で重要な箇所は、**EchoSecurityProxy** により生成された **SecurityException("No 4 letter words")** が希望通り試行されたメソッド呼び出しを中止するために投げられたということです。

第4章 JBOSS セキュリティ拡張アーキテクチャ

一般的な JBoss セキュリティレイヤに関する前述の説明では、*JBoss セキュリティ拡張フレームワーク* (JBossSX) がセキュリティレイヤインターフェースの実装であると記載しました。これが JBossSX フレームワークの第一の目的です。フレームワークにより、既存のセキュリティインフラストラクチャと統合するカスタマイズの可能性が高くなります。セキュリティインフラストラクチャはデータベースまたは LDAP サーバーから高度なセキュリティソフトウェアスイートまで何でも構いません。JAAS フレームワークで利用できる *プラグ可能な認証モデル (PAM)* を使用し、柔軟性のある統合が実現します。

JBossSX フレームワークの中核は `org.jboss.security.plugins.JaasSecurityManager` です。これは `AuthenticationManager` と `RealmMapping` インターフェースのデフォルト実装です。[図4.1「<security-domain> デプロイメント記述子の値、コンポーネントコンテナ、JaasSecurityManager 間の関係」](#) は、対応するコンポーネントのデプロイメント記述子の `<security-domain>` 要素を基にした EJB と Web コンテナレイヤへの `JaasSecurityManager` の統合方法について示しています。

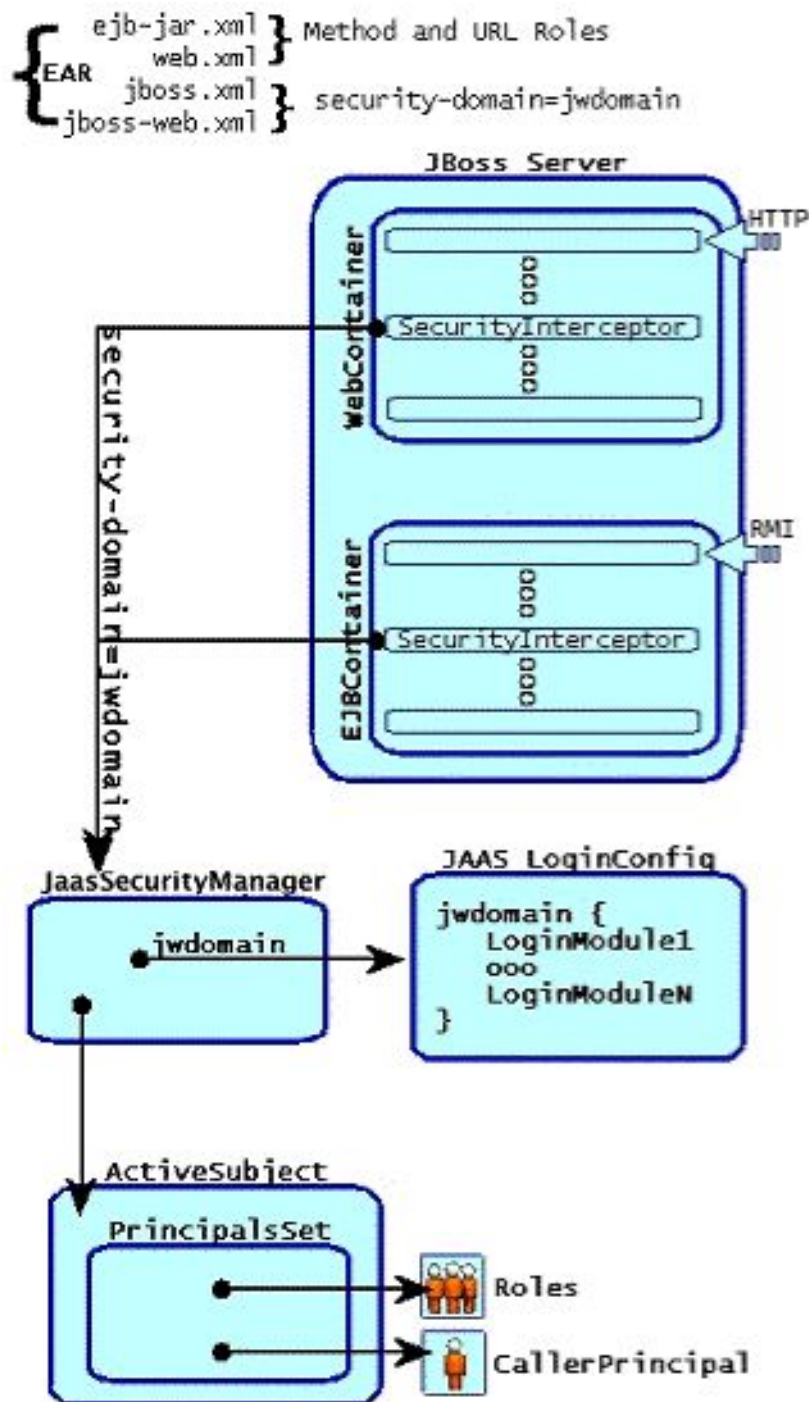


図4.1 <security-domain> デプロイメント記述子の値、コンポーネントコンテナ、JaasSecurityManager 間の関係

図4.1「<security-domain> デプロイメント記述子の値、コンポーネントコンテナ、JaasSecurityManager 間の関係」では、セキュリティドメイン `jwdomain` でセキュアな EJB と Web コンテンツを含むエンタープライズアプリケーションを示しています。EJB と Web コンテナにはセキュリティインターセプタを含む要求インターセプタアーキテクチャがあり、これによりコンテナセキュリティモデルを強制実施します。デプロイメント時に、`jboss.xml` と `jboss-web.xml` 記述子の `<security-domain>` 要素値を使用して、コンテナに関連付けられたセキュリティマネージャインスタンスを取得します。次にセキュリティインターセプタはセキュリティマネージャを使用して、そのロールを実行します。セキュアなコンポーネントが要求されると、セキュリティインターセプタはセキュリティチェックをコンテナに関連付けられたセキュリティマネージャインスタンスに委譲します。

JBossSX `JaasSecurityManager` 実装は、`<security-domain>` 要素値と一致する名前のもとで設定される JAAS ログインモジュールを実行することで得られる `Subject` インスタンスに関連付けられた情

報を基にセキュリティチェックを行います。次項では **JaasSecurityManager** 実装とその JAAS の使用について詳しく見ていきます。

4.1. JAASSECURITYMANAGER による JAAS の使用方法

JaasSecurityManager は JAAS パッケージを使用して、**AuthenticationManager** と **RealmMapping** インターフェース動作を実装します。特に、その動作は **JaasSecurityManager** が割り当てられたセキュリティドメインと一致する名前のもとで設定されたログインモジュールインスタンスを実行することで発生します。ログインモジュールはセキュリティドメインのプリンシパル認証とロールマッピング動作を実装します。こうして、ドメインに対して異なるログインモジュール設定をプラグインするだけで異なるセキュリティドメインに渡し **JaasSecurityManager** を使用することができます。

JAAS 認証プロセスにおける **JaasSecurityManager** の使用方法の詳細を図解するため、EJB ホームメソッド呼び出しのクライアント呼び出しを見ていきます。前提条件となる設定は、EJB が JBoss サーバーでデプロイされ、そのホームインターフェースメソッドが **ejb-jar.xml** 記述子の `<method-permission>` 要素を使用してセキュアであること、JBoss サーバーには **jboss.xml** 記述子 `<security-domain>` 要素を使用して **jwdomain** という名前のセキュリティドメインが割り当てられることです。

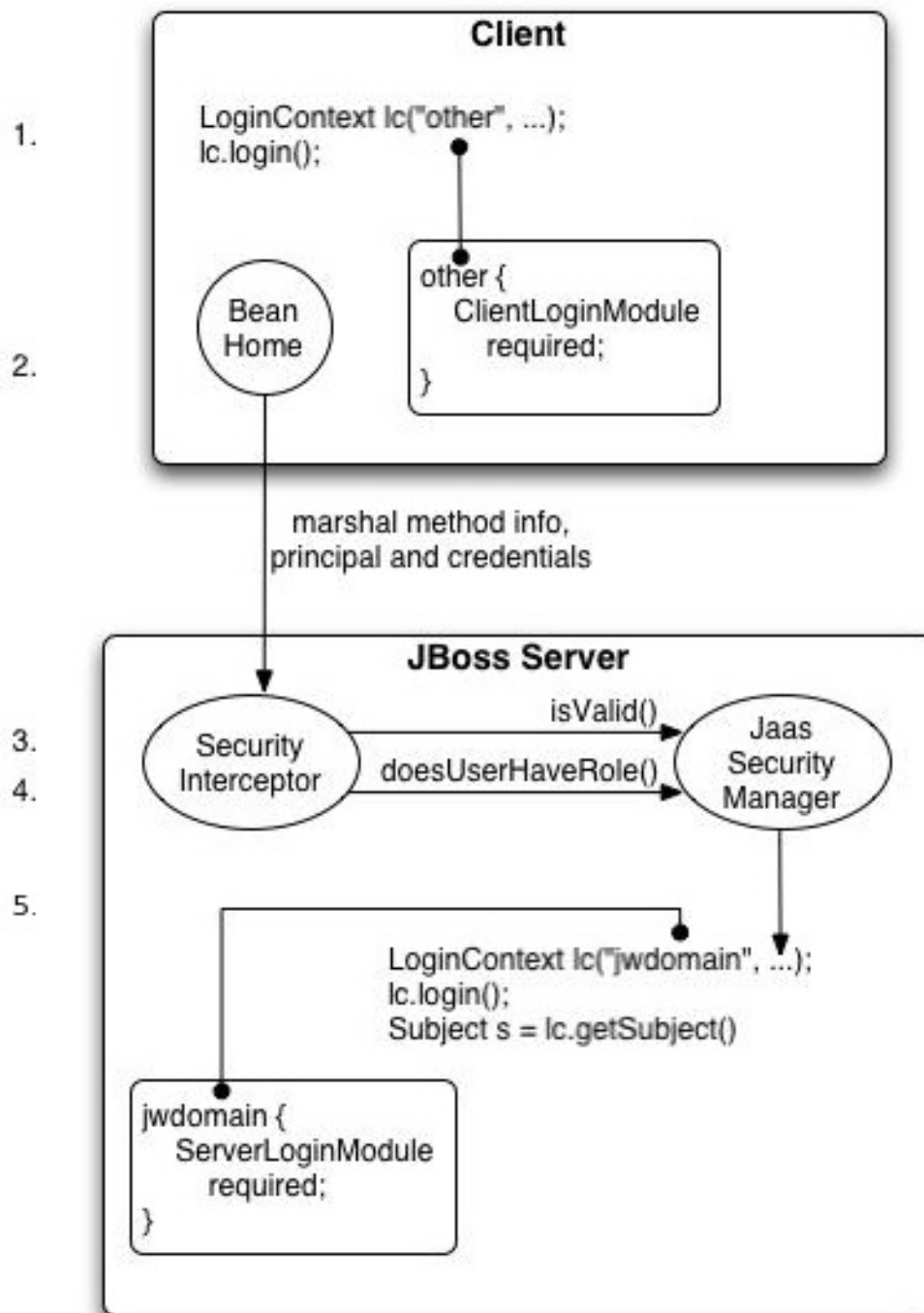


図4.2 セキュアな EJB ホームメソッド呼び出しの認証と承認のステップ

図4.2「セキュアな EJB ホームメソッド呼び出しの認証と承認のステップ」はサーバー通信にクライアントのビューを提供します。実行手順は以下のとおりです。

1. クライアントは JAAS ログインを実行して、認証に対してプリンシパルと資格情報を確立する必要があります。図では **Client Side Login (クライアント側のログイン)** と記されています。これがクライアントが JBoss でログインアイデンティティを確立する方法です。JNDI **InitialContext** プロパティによりログイン情報を提示するサポートは別の設定で行われます。

JAAS ログインでは、**LoginContext** インスタンスを作成し、使用する設定名を渡すことが必要です。設定名は **other** です。この 1 度きりのログインによってログインプリンシパルと資格情報をすべての後続 EJB メソッド呼び出しに関連付けます。このプロセスではユーザーを認証しない場合があることに注意してください。クライアント側ログインの性質は、クライアント

が使用するログインモジュール設定に依存しています。この例では **other** クライアント側のログイン設定エントリは **ClientLoginModule** モジュール

(**org.jboss.security.ClientLoginModule**) を使用するよう設定されています。これはサーバー上で後ほど行われる認証に対して単にユーザー名とパスワードを JBoss EJB 呼び出しレイヤにバインドするデフォルトのクライアント側のモジュールです。クライアントのアイデンティティはクライアント上では認証されません。

2. クライアントは EJB ホームインターフェースを取得し、Bean の作成を試みます。このイベントは **Home Method Invocation (ホームメソッド呼び出し)** と記されます。これにより、ホームインターフェースメソッド呼び出しが JBoss サーバーに送られます。呼び出しには、ステップ 1 で行われたクライアント側の JAAS ログインからのユーザーアイデンティティと資格情報とともに、クライアントから渡されるメソッド引数が含まれます。
3. サーバー側ではセキュリティインターセプタは最初にコールを呼び出すユーザーの認証が必要となり、クライアント側では JAAS ログインが必要です。
4. EJB がセキュアであるセキュリティドメインがログインモジュールの選択を行います。セキュリティドメイン名は **LoginContext** コンストラクタに渡されるログイン設定エントリ名として使用されます。EJB セキュリティドメインは **jwdomain** です。JAAS ログインがユーザーを認証すると、**PrincipalsSet** に以下を含む JAAS **Subject** が作成されます。
 - **java.security.Principal** は、デプロイメントセキュリティ環境で既知のクライアントアイデンティティに該当します。
 - **Roles** という名前の **java.security.acl.Group** には、ユーザーが割り当てられるアプリケーションドメインからのロール名が含まれています。**org.jboss.security.SimplePrincipal** オブジェクトを使用してロール名を表します。**SimplePrincipal** はシンプルな文字列をベースにした **Principal** の実装です。こうしたロールを使用して、**ejb-jar.xml** にあるメソッドに割り当てられたロールと **EJBContext.isCallerInRole(String)** メソッド実装を検証します。
 - **CallerPrincipal** という名前のオプションの **java.security.acl.Group** には、アプリケーションドメインの呼び出し側のアイデンティティに該当する単一の **org.jboss.security.SimplePrincipal** が含まれます。**CallerPrincipal** の唯一のグループメンバーは **EJBContext.getCallerPrincipal()** メソッドにより返される値となります。このマッピングの目的は、セキュリティ動作環境で既知の **Principal** がアプリケーションに既知の名前を持つ **Principal** にマップできるようにすることです。**CallerPrincipal** がないと、デプロイメントセキュリティ環境のプリンシパルのマッピングが **getCallerPrincipal** メソッド値として使用されます。つまり、動作しているプリンシパルはアプリケーションドメインのプリンシパルと同じです。
5. セキュリティインターセプタチェックの最後のステップは、認証されたユーザーが要求されたメソッドを呼び出すパーミッションを持っていることの確認です。これは [図4.2「セキュアな EJB ホームメソッド呼び出しの認証と承認のステップ」](#) で **Server Side Authorization (サーバー側の承認)** と記されています。承認を実行することで、次のステップを実行する必要があります。
 - EJB コンテナから EJB メソッドへのアクセスを許可したロール名を取得します。ロール名は呼び出されたメソッドを含むすべての `<method-permission>` 要素の **ejb-jar.xml** 記述子 `<role-name>` 要素により決定されます。
 - 割り当てられたロールがない場合や、`exclude-list` 要素でメソッドが指定されていない場合は、メソッドへのアクセスは拒否されます。そうでない場合は、**doesUserHaveRole** メソッドはセキュリティインターセプタによりセキュリティマネージャで呼び出され、呼び出し側が割り当てられたロール名を持っているか確認します。このメソッドはロール名を

使って反復し、認証されたユーザーの Subject **Roles** グループが割り当てられたロール名を持つ **SimplePrincipal** を含んでいるか確認します。ロール名が **Roles** グループのメンバーである場合はアクセスは許可されます。いずれのロール名もメンバーでない場合はアクセスは拒否されます。

- EJB がカスタムのセキュリティプロキシで設定された場合、メソッド呼び出しはそれに委譲されます。セキュリティプロキシが呼び出し側へのアクセスを拒否したい場合は、**java.lang.SecurityException** を投げます。**SecurityException** が投げられなかった場合は、EJB メソッドへのアクセスは許可され、メソッド呼び出しは次のコンテナインターセプタに渡されます。**SecurityProxyInterceptor** がこのチェックを処理するため、このインターセプタは表示されないことに注意してください。
- JBoss Web 接続では、Tomcat がセキュリティ制約の要素とロール検証を管理します。

Web 接続に対して要求が受け取られると、Tomcat は要求されたリソースとアクセスされた HTTP メソッドに一致する **web.xml** で定義されたセキュリティ制約を確認します。

ある制限が要求に対して存在する場合、Tomcat は **JaasSecurityManager** を呼び出してプリンシパルの認証を実行します。これにより確実にユーザーロールがそのプリンシパルのオブジェクトと関連付けられているようにします。

唯一 Tomcat がロールの検証を実行します。例えば **allRoles** などのパラメータや **STRICT_MODE** が使用されたかどうかを確認します。

すべてのセキュアな EJB メソッド呼び出し、またはセキュアな Web コンテンツアクセスには呼び出し側の認証と承認が必要です。その理由は、セキュリティ情報は各要求で表示、検証される必要がある要求のステートレス属性として処理されるためです。JAAS ログインにクライアントとサーバー間の通信が含まれる場合、これはコストの高い操作となる場合があります。このため、**JaasSecurityManager** は前の成功したログインからのプリンシパルと資格情報を保存するために使用する認証キャッシュの概念をサポートします。次項の関連する MBean サービスに関する説明のとおり、**JaasSecurityManager** 設定の一部として使用する認証キャッシュのインスタンスを指定できます。ユーザー定義のキャッシュがない場合は、設定可能な期間中に資格情報を管理するデフォルトキャッシュが使用されます。

4.2. JAASSECURITYMANAGERSERVICE MBEAN

JaasSecurityManagerService MBean サービスはセキュリティマネージャを管理します。名前は **Jaas** で始まりますが、処理するセキュリティマネージャはその実装で JAAS を使用する必要はありません。名前は、デフォルトのセキュリティマネージャの実装が **JaasSecurityManager** であることに由来しています。**JaasSecurityManagerService** の主要な役割はセキュリティマネージャ実装を外在させることです。セキュリティマネージャの実装を変更するには、**AuthenticationManager** と **RealmMapping** インターフェースの別の実装を提供します。

JaasSecurityManagerService の基本的な第 2 の役割は、JNDI **javax.naming.spi.ObjectFactory** 実装を提供し、JNDI 名からセキュリティマネージャへの実装マッピングにシンプルなコード不要の管理を実現することです。セキュリティを有効にするには、<security-domain> デプロイメント記述子の要素によるセキュリティマネージャの実装の JNDI 名を指定します。

JNDI 名を指定する場合、使用するオブジェクトバインディングがなければなりません。セキュリティマネージャのバインディングに対する JNDI 名の設定を簡略化するには、次のネーミングシステム参照を **java:/jaas** という名前でも JNDI **ObjectFactory** としてバインドすることで、**JaasSecurityManagerService** が名前へのセキュリティマネージャのインスタンスの関連付け

を管理します。これにより `<security-domain>` 要素に対する値として `java:/jaas/XYZ` という形式の命名規則が許可されます。`XYZ` セキュリティドメインのセキュリティマネージャインスタンスは必要に応じて作成されます。

ドメイン `XYZ` のセキュリティマネージャは、セキュリティドメインの名前を取るコンストラクタを使用して、`SecurityManagerClassName` 属性により指定されるクラスのインスタンスを作成することによって `java:/jaas/XYZ` バインディングに対する最初のルックアップで作成されます。

重要

Enterprise Application Platform 5.0 以前のバージョンでは 各 `<security-domain>` デプロイメント記述子の要素のプレフィックス「`java:/jaas`」は、セキュリティドメインの JNDI 名をセキュリティマネージャのバインディングに適切にバインドするために必要でした。

JBoss Enterprise Application Platform 5 では、`java:/jaas` プレフィックスはセキュリティドメイン宣言には必要ありません。`java:/jaas` プレフィックスはなおサポートされており、後方互換性を保っています。

例えば、次のコンテナセキュリティの設定スニペットを考えてみましょう。

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

名前 `customer` を検索すると、`customer` という名前のセキュリティドメインに関連したセキュリティマネージャのインスタンスが返されます。このセキュリティマネージャは `AuthenticationManager` と `RealmMapping` セキュリティインターフェースを実装し、`JaasSecurityManagerServiceSecurityManagerClassName` 属性により指定されるタイプとなります。

`JaasSecurityManagerService` MBean はデフォルトでは標準 JBoss ディストリビューションで使われるよう設定されており、多くの場合はデフォルト設定としてそのまま使用できます。`JaasSecurityManagerService` の設定可能な属性として以下が挙げられます。

SecurityManagerClassName

セキュリティマネージャ実装を提供するクラス名です。実装は `org.jboss.security.AuthenticationManager` と `org.jboss.security.RealmMapping` インターフェース両方に対応している必要があります。指定がない場合は、JAAS ベースの `org.jboss.security.plugins.JaasSecurityManager` にデフォルト設定されます。

CallbackHandlerClassName

`JaasSecurityManager` により使用される `javax.security.auth.callback.CallbackHandler` 実装を提供するクラス名です。



注記

デフォルト実装

(**org.jboss.security.auth.callback.SecurityAssociationHandler**) がニーズに合わない場合は、**JaasSecurityManager** により使用されるハンドラを一時的に変更できます。ほとんどの実装ではデフォルトのハンドラで十分です。

SecurityProxyFactoryClassName

org.jboss.security.SecurityProxyFactory 実装を提供するクラス名です。指定がない場合は、**org.jboss.security.SubjectSecurityProxyFactory** にデフォルト設定されます。

AuthenticationCacheJndiName

セキュリティ資格情報のキャッシュポリシーの場所を特定します。最初は <security-domain> ごとに **CachePolicy** インスタンスを返すことができる **ObjectFactory** 場所として扱われます。ドメインの **CachePolicy** を検索するときに、セキュリティドメインの名前をこの名前に追加します。これが失敗した場合は、場所はすべてのセキュリティドメインに対して単一の **CachePolicy** として扱われます。デフォルトとして、時間制限を設けたキャッシュポリシーが使用されます。

DefaultCacheTimeout

デフォルトの時間制限が設けられたキャッシュポリシーのタイムアウトを秒単位で指定します。デフォルト値は 1800 秒 (30 分) です。タイムアウトに使用する値は、頻繁な認証動作と資格情報がセキュリティ情報ストアに対して同期していない可能性がある期間とのトレードオフです。セキュリティ資格情報のキャッシュを無効にしたい場合は、毎回認証が強制的に実行されるようこの値を 0 に設定します。**AuthenticationCacheJndiName** がデフォルト値から変更されている場合は効果がありません。

DefaultCacheResolution

デフォルトの時間制限が設けられたキャッシュポリシー解決を秒単位で指定します。キャッシュの現在のタイムスタンプが更新される間隔を制御し、タイムアウトを有効にするため **DefaultCacheTimeout** 未満になるようにします。デフォルトの解決は 60 秒 (1 分) です。これは **AuthenticationCacheJndiName** がデフォルト値から変更されている場合は効果がありません。

DefaultUnauthenticatedPrincipal

認証されていないユーザーが使用するプリンシパルを指定します。この設定により、認証されていないユーザーがデフォルトのパーミッションを設定することが可能になります。

JaasSecurityManagerService は数多くの役立つ動作をサポートします。これには、ランタイム時のセキュリティドメイン認証キャッシュのフラッシュ、セキュリティドメイン認証キャッシュのアクティブなユーザーの一覧取得、セキュリティマネージャインターフェースメソッドなどが含まれます。

セキュリティドメイン認証キャッシュをフラッシュする機能を使用すると、基礎となるストアが更新され、ストアの状態を直ちに使用されるようにしたいときに、すべてのキャッシュされた資格情報をドロップすることができます。MBean 動作のシグネチャは **public void flushAuthenticationCache(String securityDomain)** です。

これは次のコードスニペットを使用してプログラムによって呼び出すことが可能です。

```
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);
```

アクティブなユーザーの一覧を取得すると、失効していないセキュリティドメイン認証キャッシュの **Principals** キーのスナップショットを得ることができます。MBean 動作のシグネチャは **public List getAuthenticationCachePrincipals(String securityDomain)** です。

これは次のコードスニペットを使用してプログラムによって呼び出すことが可能です。

```
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);
```

セキュリティマネージャには追加のアクセスメソッドがいくつかあります。

```
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);
```

securityDomain 引数により名前が付いた関連するセキュリティドメインの対応する **AuthenticationManager** と **RealmMapping** インターフェースメソッドへのアクセスを提供します。

4.3. JAASSECURITYDOMAIN MBEAN

org.jboss.security.plugins.JaasSecurityDomain は **JaasSecurityManager** の拡張で、**KeyStore**、**JSSE KeyManagerFactory**、**TrustManagerFactory** の概念を追加し、SSL と他の暗号の使用事例をサポートします。追加で設定可能な **JaasSecurityDomain** の属性として以下が挙げられます。

KeyStoreType

KeyStore 実装のタイプです。これは **java.security.KeyStore.getInstance(String type)** ファクトリメソッドに渡されるタイプ引数です。デフォルトは **JKS** です。

KeyStoreURL

KeyStore データベースがある場所への URL です。これを使用して、**InputStream** を取得し **KeyStore** を初期化します。文字列に URL の名前 / 値が含まれていない場合は、値はファイルとして扱われます。

KeyStorePass

KeyStore データベースコンテンツに関連付けられたパスワードです。**KeyStorePass** は、エンコード / デコード動作で使用される PBE 秘密キーを作成する **Salt** と **IterationCount** 属性と組み合わせても使用されます。**KeyStorePass** 属性値の形式は次のうちのいずれかです。

- **KeyStore** のプレーンテキストのパスワードです。文字列の **toCharArray()** 値は変更せずに使用します。
- プレーンテキストのパスワードを取得するために実行するコマンドです。形式は **{EXT}...** であり、... はプラットフォーム固有のコマンドを実行するために **Runtime.exec(String)** メソッドに渡される正しいコマンドラインです。コマンド出力の 1 行目はパスワードとして使用されます。
- プレーンテキストのパスワードを取得するために作成するクラスです。形式は **{CLASS}classname[:ctorarg]** であり、**[:ctorarg]** は **classname** のインスタンスを作成するときにコンストラクタに渡されるオプションの文字列です。パスワードは **toCharArray()** メソッドがあればそれ呼び出してクラス名から取得されます。ない場合は、**toString()** メソッドが使用されます。

Salt

PBEParameterSpec salt の値です。

IterationCount

PBEParameterSpec 反復カウント値です。

TrustStoreType

TrustStore 実装のタイプです。これは **java.security.KeyStore.getInstance(String type)** ファクトリメソッドに渡されるタイプ引数です。デフォルトは **JKS** です。

TrustStoreURL

TrustStore データベースがある場所への URL です。これを使用して、**InputStream** を取得し **KeyStore** を初期化します。文字列が URL 値でない場合は、ファイルとして扱われます。

TrustStorePass

信頼できるストアデータベースのコンテンツに関連付けられたパスワードです。**TrustStorePass** は簡易パスワードで、**KeyStorePass** と同じ設定オプションはありません。

ManagerServiceName

セキュリティマネージャサービス MBean の JMX オブジェクト名の文字列を設定します。これを使用してデフォルトを登録し、**java:/jaas/<domain>** のもとでセキュリティマネージャとして **JaasSecurityDomain** を登録します。**<domain>** は MBean コンストラクタに渡される名前です。この名前は **jboss.security.service=JaasSecurityManager** にデフォルト設定されません。

パート II. アプリケーションセキュリティ

第5章 概要

アプリケーションセキュリティでは、認証、承認、マッピング、監査について取り上げます。

認証

サーバーがユーザーがシステムまたはオペレーションにアクセスできるかを決定するプロセスです。

承認

認証されたユーザーがシステムまたはオペレーションの特定の権限またはリソースにアクセスする権限を持っているかをサーバーが決定するプロセスです。

マッピング

認証されたユーザーと事前定義した承認プロファイルをサーバーが関連付けるプロセスです。

監査

サーバーが認証と承認のセキュリティイベントを監視するプロセスです。

JBoss Enterprise Application Platform 5 では、認証、承認、マッピングポリシーはセキュリティドメインの概念を使用してアプリケーションレベルの粒度で設定されます。

セキュリティドメイン

XML で定義され、Java Naming and Directory Interface (JNDI) を使用してランタイム時にアプリケーションが使用できる認証、承認、マッピングポリシーのセットです。

セキュリティドメインは、サーバープロファイルまたはアプリケーションデプロイメント記述子内で定義することができます。

EJB3 の監査と Web コンテナの監査は互いに依存することなく設定され、サーバーレベルで動作します。

JBoss Enterprise Application Platform 5 はプラグ可能なモジュールフレームワークを使用してセキュリティを実装し、セキュリティ実装とアプリケーションデザインを分離させます。プラグ可能なモジュールフレームワークにより、アプリケーションのデプロイメント、設定、今後の開発を柔軟に行うことができます。セキュリティの機能性を実装するモジュールはフレームワークにプラグインされ、アプリケーションにセキュリティ機能を提供します。アプリケーションはセキュリティドメインを通じてセキュリティフレームワークにプラグインします。

アプリケーションは、異なるセキュリティドメインとそれを関連付けることにより、変更したセキュリティ設定で新しいシナリオ内でデプロイ可能です。新しいセキュリティメソッドは、JBoss、カスタムビルト、またはサードパーティのモジュールをフレームワークにプラグインすることで実装できます。アプリケーションはセキュリティドメインのシンプルな再構成により、アプリケーションの再コーディングがないそのモジュールのセキュリティ機能を取得します。

[6章 セキュリティドメインスキーマ](#) ではセキュリティドメインの XML 設定について説明しています。

[7章 認証](#) にはセキュリティドメイン認証ポリシーに関する詳細情報が記載されています。

[8章 承認](#) にはセキュリティドメインの承認ポリシーに関する詳細情報が記載されています。

9章 [マッピング](#)にはセキュリティドメインのマッピングポリシーに関する詳細情報が記載されています。

第6章 セキュリティドメインスキーマ

セキュリティドメインスキーマは XML を使用して構築されます。

セキュリティドメインの構造を定義する XML Schema Definition (XSD) は security-beans_1_0.xsd ファイルで宣言されます。ファイルの場所をご使用の JBoss Enterprise Application Platform のバージョンにより異なります。

5.1

`jboss-as/lib/jbosssx.jar` / 中の `/schema/security-beans_1_0.xsd`

5.0、5.0.1

`jboss-as/common/lib/jbosssx.jar` 中の `/schema/security-beans_1_0.xsd`

スキーマは使用する JBoss Enterprise Application Platform サーバーに関わらず同じになります。

```
<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>
```

図6.1 セキュリティドメイン定義のスキーマ

6.1. セキュリティドメイン要素

<application-policy>

セキュリティドメインの要素は、システム内にどのようにデプロイされているかに関わらず、<application-policy> 要素内に含まれます。この要素は **xmlns** 属性で宣言されているように XML 名前空間を使用します。

name 属性はアプリケーションによって参照されたセキュリティドメインの名前を設定します。セキュリティドメイン名は **java:jaas** コンテキスト下の JNDI でバインドされ、デプロイメント記述子の参照を通じてアプリケーションによってアクセスされます。

<application-policy> 要素には、セキュリティドメインの動作とそれを使用するすべてのアプリケーションを設定する多くの子要素を含むことができます。こうした要素のさらなる詳細は「<authentication>」、「<authorization>」、「<mapping>」に記載されています。

6.1.1. <authentication>

<authentication> 要素には次の子要素が含まれます。

<authentication>

この要素には <login-module> 要素が含まれ、アプリケーションより接続するユーザーの認証時にどの認証モジュールが使用されるかを制御します。

<login-module> 要素が複数存在している場合は、認証が検証される前に満たすべき集団グループの要件を形成します。この集団グループは **スタック** と呼ばれます。

<login-module>

この要素はアプリケーションが使用できるログインモジュール実装を指定する **code** 属性を使用し、スタックに存在している各ログインモジュールを解析する方法をアプリケーションに伝える **flag** 属性を使用します。**flag** 属性は次の値に対応します。

required

認証が成功するためにはログインモジュールが成功する必要があります。必須の <login-module> が失敗した場合は、認証は失敗します。スタックの残りのログインモジュールは認証の結果に関わらず呼び出されます。

requisite

モジュールは成功する必要があります。成功すれば、認証はスタックまで続きます。モジュールが失敗すれば、直ちに制御はアプリケーションに返します。

sufficient

ログインモジュールは成功する必要はありません。成功すれば、直ちに制御はアプリケーションに返します。モジュールが失敗すると、認証はスタックまで続きます。

optional

ログインモジュールは成功する必要はありません。ログインモジュールが成功するか失敗するかに関わらず、認証はなおスタックまで続きます。

各 <login-module> にはログインモジュール実装に必要な設定を更に定義する <module-option> 要素のセットが含まれます。

<module-option>

各ログインモジュールには独自の設定オプションのセットがあります。名前の属性はログインモジュールが必要なプロパティを指定し、値は <module-option> 要素の CDATA で宣言されます。モジュールオプションは選択するログインモジュールにより異なります。「[モジュールの使用](#)」にはモジュールオプションに関して詳しく記載されています。

6.1.2. <authorization>

<authorization>

この要素にはアプリケーションユーザーを承認するために使用されるポリシーモジュールとそのモジュールが必要かどうかを定義する <policy-module> 要素が含まれます。

<policy-module> 要素が複数存在している場合は、承認が検証される前に満たすべき集団グループの要件を形成します。この集団グループは **スタック** と呼ばれます。

<policy-module>

この要素はアプリケーションが使用できるポリシーモジュール実装を指定する **code** 属性を使用し、ポリシースタックに存在している各ポリシーモジュールを解析する方法をアプリケーションに伝える **flag** 属性を使用します。**flag** 属性は次の値に対応します。

required

承認が成功するためにはモジュールが成功する必要があります。必須の <policy-module> が失敗した場合は、承認の試行は失敗します。スタックの残りのモジュールはモジュールの結果に関わらず呼び出されます。

requisite

モジュールは成功する必要があります。成功すれば、承認はスタックまで続きます。失敗すれば、直ちに制御はアプリケーションに返します。

sufficient

ログインモジュールは成功する必要はありません。成功すれば、直ちに制御はアプリケーションに返します。失敗すると、承認はスタックまで続きます。

optional

ログインモジュールは成功する必要はありません。モジュールが成功するか失敗するかに関わらず、承認はなおスタックまで続きます。

6.1.3. <mapping>

<mapping>

この要素には mapping-module 要素のパラメータを定義するために使用される <mapping-module> 要素が含まれます。

<mapping-module> 要素が複数存在している場合は、マッピングが成功する前に満たすべき集団グループの要件を形成します。この集団グループは **スタック** と呼ばれます。

<mapping-module>

この要素はアプリケーションが使用できるマッピングモジュール実装を指定する **code** 属性を使用し、ポリシースタックに存在している各マッピングモジュールを解析する方法をアプリケーションに伝える **flag** 属性を使用します。**flag** 属性は次の値に対応します。

required

マッピングが成功するためにはモジュールが成功する必要があります。必須の <mapping-module> が失敗した場合は、認証は失敗します。スタックの残りのモジュールは認証の結果に関わらず呼び出されます。

requisite

モジュールは成功する必要があります。成功すれば、マッピングはスタックまで続きます。モジュールが失敗すれば、直ちに制御はアプリケーションに返します。

sufficient

モジュールは成功する必要はありません。成功すれば、直ちに制御はアプリケーションに返します。モジュールが失敗すると、マッピングはスタックまで続きます。

optional

モジュールは成功する必要はありません。モジュールが成功するか失敗するかに関わらず、マッピングはなおスタックまで続きます。

第7章 認証

次の例では、セキュリティドメインでアプリケーションポリシーを使用できる方法を説明します。

明確にするために、例では認証ポリシーだけが宣言されています。ただし、同じ `<application-policy>` に `<authorization>` と `<mapping>` 要素を含めることができます。`<authentication>` 要素に関する詳細は「[<authentication>](#)」を参照してください。

例7.1 シングルログインスタックの認証ポリシー

この例では、シングルログインモジュール `UsersRolesLoginModule` (「[UsersRolesLoginModule](#)」を参照) を使用する `jmx-console` という名前のシンプルなセキュリティドメイン設定について説明します。

ログインモジュールは `jboss-as/server/$PROFILE/conf/props` ディレクトリのファイルからのユーザーとロールのプロパティにより提供されます。

この例では、`<login-module>` は成功する必要があります。そうでない場合は、認証は失敗します。

```
<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>
```

例7.2 複数のログインスタックの認証ポリシー

この例では、認証ログインモジュールスタックで2つのログインモジュールを使用する `web-console` という名前のセキュリティドメイン設定について説明します。

ある `<login-module>` は `LdapLoginModule` (「[LdapLoginModule](#)」を参照) を使用してログイン資格情報を取得します。一方で、別の `<login-module>` は `BaseCertLoginModule` (「[BaseCertLoginModule](#)」を参照) を使用して認証資格情報を取得します。

この例では、両方のモジュールとも `sufficient` であるとマークされているため、認証が成功するためにはどちらか一方のみ成功する必要があります。

```
<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. カスタムコールバックハンドラ

認証手順にコールバックハンドラを実装することにより、ログインモジュールがクライアントアプリケーションの認証メソッドに依存することなくユーザーを認証できます。

次のメソッドを使用してコールバックハンドラを実装できます。

- **conf/jboss-service.xml** JaasSecurityManagerService MBean 定義で CallbackHandlerClassName 属性を指定します。
- **deploy/security/security-jboss-beans.xml** JNDISecurityManagement Bean にコールバックハンドラのインスタンスをインジェクトします。

手順7.1 属性を使用したコールバックハンドラの設定

これは **jboss-service.xml** 設定ファイルでコールバックハンドラを指定する手順を示しています。

1. 設定ファイルを開きます

\$JBOSS_HOME/server/\$PROFILE/conf/ に移動します。

jboss-service.xml ファイルを開きます。

デフォルトで **jboss-service.xml** ファイルには [例7.3「jboss-service デフォルト設定」](#) の設定が含まれています。

例7.3 jboss-service デフォルト設定

```

    <?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. 属性を追加します

カスタムコールバックハンドラを設定するには、`<attribute>` 要素を `<mbean>` 要素の子として追加し、コールバックハンドラの完全修飾名を指定します。コールバックハンドラが指定された例の `<attribute>` 要素については、[例7.4「コールバックハンドラを追加した jboss-service」](#)を参照してください。

例7.4 コールバックハンドラを追加した 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. サーバーを再起動します

これで **jboss-service.xml** ファイルを設定し、カスタムコールバックハンドラを使用できるようになりました。

サーバーを再起動して、新しいセキュリティポリシーが有効であることを確認してください。

手順7.2 インジェクションを使用したセキュリティコールバックハンドラの設定

これは JNDISecurityManagement Bean にセキュリティコールバックハンドラのインスタンスをインジェクトする方法を示した手順です。

1. カスタムコールバックのインスタンスを作成します
カスタムコールバックハンドラのインスタンスを作成し、登録する必要があります。
2. 設定ファイルを開きます
`$JBOSS_HOME/server/$PROFILE/deploy/security/` に移動します。

`security-jboss-beans.xml` ファイルを開きます。

デフォルトで `security-jboss-beans.xml` ファイルには [例7.5「security-jboss-beans デフォルト設定」](#) の JNDIBasedSecurityManagement Bean 設定が含まれています。

例7.5 security-jboss-beans デフォルト設定

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

3. インジェクションのプロパティを追加します
コールバックハンドラをインジェクトするには、`<property>` 要素を JNDIBasedSecurityManagement `<mbean>` 要素の子として追加します。 [例7.4「コールバックハンドラを追加した jboss-service」](#) に説明された `<property>` と `<inject>` 要素を使用してコールバックハンドラを指定します。

例7.6 security-jboss-beans コールバックハンドラ

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

4. サーバーを再起動します
これで `security-jboss-beans.xml` ファイルを設定し、カスタムコールバックハンドラをインジェクトできるようになりました。

サーバーを再起動して、新しいセキュリティポリシーが有効であることを確認してください。

第8章 承認

承認は、保護したいコンポーネントがあるレイヤではなく、そのタイプと関連しています。

セキュリティドメインは明示的には承認ポリシーは必要ありません。承認ポリシーが指定されていない場合は、**jboss-as/server/\$PROFILE/deploy/security/security-policies-jboss-beans.xml** で設定されたデフォルトの **jboss-web-policy** と **jboss-ejb-policy** 承認が使用されます。

承認ポリシーを指定すること、または有効な承認ポリシーでカスタムのデプロイメント記述子ファイルを作成することを選択した場合は、これらの設定は **security-policies-jboss-beans.xml** のデフォルト設定を無効にします。

ユーザーはカスタム動作を実装する承認ポリシーを提供することができます。カスタム動作を設定することで、承認コントロールスタックが特定のコンポーネントに対してプラグ可能になり、(EJB では) **jboss.xml** および (WAR では) **jboss-web.xml** に含まれているデフォルトの承認を上書きします。

仕様動作を実装するデフォルトモジュールの他に、EJB または Web コンポーネントに対するデフォルトの承認の上書きが Java Authorization Contract for Containers (JACC) と Extensible Access Control Markup Language (XACML) に対して行われます。

<authorization> 要素スキーマに関する情報は、[「<authorization>」](#) を参照してください。

手順8.1 すべての EJB および WAR コンポーネントに承認ポリシーを設定

すべての EJB および Web コンポーネント、または特定のコンポーネントに対し承認を上書きすることができます。

この手順ではすべての EJB および WAR コンポーネントに対する JACC 承認制御の定義方法を説明しています。例では、Web および EJB アプリケーションに対しアプリケーションポリシーのモジュール **jboss-web-policy** と **jboss-ejb-policy** を定義します。

1. セキュリティポリシー Bean を開きます

\$JBOSS_HOME/server/\$PROFILE/deploy/security に移動します。

security-policies-jboss-beans.xml ファイルを開きます。

デフォルトで **security-policies-jboss-beans.xml** ファイルには [例8.1「security-policies-jboss-beans.xml」](#) の設定が含まれています。

例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.DelegatingAuthorizationModule" flag="required"/>
        </authorization>
    </application-policy>

</deployment>
```

```

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

</deployment>

```

2. application-policy 定義を変更します

JACC を使用して、各コンポーネントの単一の承認ポリシーを設定するには、JACC 承認モジュールの名前が付いた各 **<policy-module>** **code** 属性を修正します。

```

<?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. サーバーを再起動します

これで、各アプリケーションポリシーに対して JACC 承認を有効にした **security-policy-jboss-beans.xml** ファイルを設定できました。

サーバーを再起動して、新しいセキュリティポリシーが有効であることを確認してください。

特定の EJB および WEB コンポーネントに対する承認の設定

アプリケーションにさらに粒度の細かいセキュリティポリシーが必要な場合は、各アプリケーションポリシーに対して複数の承認セキュリティポリシーを宣言することができます。新しいセキュリティドメインは、別のセキュリティドメインからの基本設定を引き継ぎ、承認ポリシーモジュールなど特定の設定を上書きすることができます。

手順8.2 特定のセキュリティドメインに対する承認ポリシーの設定

特定のコンポーネントに承認を上書きすることができます。

この手順では、他のセキュリティドメインの定義から設定を引き継ぐ方法、セキュリティドメインごとに異なる承認ポリシーを指定する方法を説明しています。

この手順では、セキュリティドメインは2つ定義されています。**test-domain** セキュリティドメインは **UsersRolesLoginModule** ログインモジュールを使用し、JACC 承認を使用します。**test-domain-inherited** セキュリティドメインは **test-domain** からのログインモジュール情報を引き継ぎ、XACML 承認を使用する必要があると指定しています。

1. セキュリティポリシーを開きます

jboss-as/server/\$PROFILE/conf/login-config.xml ファイルでセキュリティドメインの設定を指定するか、または設定を含むデプロイメント記述子ファイルを作成することができます。ご使用のアプリケーションでセキュリティドメインの設定をパッケージ化したい場合は、デプロイメント記述子を選択します。

o login-config.xml を検索し、開きます

使用しているサーバープロファイルの **login-config.xml** ファイルに移動し、編集するファイルを開きます。

```
$JBASS_HOME/jboss-as/server/$PROFILE/conf/login-config.xml
```

o jboss-beans.xml 記述子を作成します

[prefix]-jboss-beans.xml 記述子を作成し、**[prefix]** を有効な名前と置換します (**test-war-jboss-beans.xml** など)。

設定しているサーバープロファイルの **/deploy** ディレクトリにこのファイルを保存します。

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

2. test-domain セキュリティドメインを指定します

ステップ1で選択した目的のファイルで、**test-domain** セキュリティドメインを指定します。このドメインには、<login-module> 定義や JACC 承認ポリシーモジュールの定義などの認証情報が含まれています。

```
<?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. **test-domain-inherited** セキュリティドメインを追加します

test-domain アプリケーションポリシーの後に、**test-domain-inherited** アプリケーションポリシーの定義を追加します。

extends 属性を **other** に設定します。これでログインモジュール情報が引き継がれます。

<policy-module> 要素の XACML 承認モジュールを指定します。

```

<?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>
            </authorization>
        </application-policy>
    </application-policy>
</deployment>

```

```

    </application-policy>
  </deployment>

```

4. サーバーを再起動します

これで、異なる承認メソッドを使用する 2 つのセキュリティドメインを持つ目的のファイルを設定できました。

サーバーを再起動して、新しいセキュリティポリシーが有効であることを確認してください。

8.1. 委譲されるモジュール

手順8.1「すべての EJB および WAR コンポーネントに承認ポリシーを設定」と手順8.2「特定のセキュリティドメインに対する承認ポリシーの設定」では、セキュリティドメインでの基本認証の設定方法について示した簡略化した例を説明します。

承認は保護したいコンポーネントの (レイヤではなく) タイプに関係しているため、デプロイメント記述子 (*-jboss-beans.xml) 内で承認モジュールを委譲することができ、実装内の標準認証から異なる承認ポリシーを指定することができます。

`org.jboss.security.authorization.modules.AuthorizationModuleDelegate` クラスは、委譲されるモジュールの実装を可能にする多くのサブクラスを提供します。

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

モジュールが

`org.jboss.security.authorization.modules.AuthorizationModuleDelegate` クラスを拡張する場合は、委譲された承認モジュールを作成できます。

委譲されたモジュールを実装するには、<authorization> ポリシーの <module-option> 要素内でモジュールの委譲を宣言します。各モジュールにはプレフィックスとして 例8.2「モジュールの委譲の宣言」のとおりに関係するコンポーネントが付いています。

例8.2 モジュールの委譲の宣言

```

<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>
```

第9章 マッピング

JBoss Enterprise Application Platform 5 では、セキュリティドメインレベル (EAR レベルなど) で発生したロールからデプロイメントレベルで追加のロールをマップすることが可能です。

これは、`org.jboss.security.mapping.providers.DeploymentRolesMappingProvider` クラスを `<mapping-module>` 要素の `code` 属性に対する値として宣言することで実行できます。加えて、`type` 属性は `role` に設定する必要があります。`<mapping>` 要素スキーマに関する情報は「[<mapping>](#)」を参照してください。

ロールベースのパラメータ内でマッピング設定要素を設定することにより、特定のデプロイメント (war、ear、ejb-jar など) に対し指定された宣言されたプリンシパルに追加のロールの解釈を強制できます。



重要

JBoss Enterprise Application Platform 5 以前のバージョンでは、`<rolemapping>` 要素には `<mapping-module>` 要素とクラス宣言が含まれていました。現在では `<rolemapping>` は廃止され、`<mapping>` 要素に置き換えられました。

例9.1 `<mapping-module>` の宣言

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

セキュリティドメインが正しく設定されると、`<security-role>` 要素グループを `<assembly-descriptor>` の子要素として **WEB-INF/jboss-web.xml** (.war または .sar) ファイルに追加することができます。

例9.2 `<security-role>` の宣言

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

Support プリンシパルに関連するセキュリティロールは、**WEB-INF/jboss-web.xml** に含まれるベースのセキュリティロール情報に加えて実装されます。

I

第10章 監査

政府機関の中には、実装のソフトウェアコンポーネントを確実に追跡できるようにエンタープライズアプリケーションにおける監査や、設計パラメータ内での動作を義務付けているところがあります。さらに、標準のアプリケーション監査に加え、政府規制および規格には監査制御が必要です。

システム管理者は、セキュリティイベント監査が常にセキュリティドメインの動作とデプロイされた Web および EJB アプリケーションを監視できるようにします。



重要

セキュリティイベント監査は、高いイベントボリュームを管理するサーバーのパフォーマンスに影響することがあります。監査はデフォルトでは無効になっており、要求に応じて使用できるよう設定した方がよいでしょう。

セキュリティイベント監査の有効化は Web コンポーネントと EJB コンポーネントでは異なります。[手順10.1「セキュリティ監査機能の有効化」](#)では実装における EJB の監査サービスを有効にする最小限のステップを説明しています。[手順10.2「Web コンテナにセキュリティ監査を有効化」](#)では Web コンテナに対するセキュリティイベント監査を有効にする方法を説明しています。



重要

Web コンテナイベント監査によりユーザーの機密情報が露呈される恐れがあります。管理者は、Web コンテナイベントにセキュリティ監査を設定する場合はパスワードハッシュなどの適切なデータ保護の手順を確実に実行する必要があります。

手順10.1 セキュリティ監査機能の有効化

1. log4j 設定ファイルを開きます

`$JBOSS_HOME/server/$PROFILE/conf/` に移動します。

テキストエディタを使用して `jboss-log4j.xml` ファイルを開きます。

2. セキュリティ監査カテゴリを非コメント化します

デフォルトでは、`jboss-log4j.xml` ファイルの Security Audit Provider カテゴリの定義はコメントアウトされています。[例10.1「log4j Security Audit Provider のカテゴリ」](#)で示されているカテゴリ定義を非コメント化します。

例10.1 log4j Security Audit Provider のカテゴリ

```
<?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. Audit appender を非コメント化します

デフォルトでは、**jboss-log4j.xml** ファイルの AUDIT appender の定義はコメントアウトされています。例10.1「log4j Security Audit Provider のカテゴリ」で示されている appender の定義を非コメント化します。

例10.2 log4j Security Audit Provider のカテゴリ

```

...
<!-- 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. 保存して、サーバーを再起動します

これで、**jboss-log4j.xml** ファイルで設定したとおり実装に対し監査サービスをアクティブにすることができました。

サーバーを再起動して、新しいセキュリティポリシーが有効であることを確認してください。

5. セキュリティ監査が正しく機能していることを確認します

監査サービスが設定、デプロイされた時点で、監査ログエントリは監査サービスと EJB 呼び出しが成功したかを確認します。

audit.log ファイルは **jboss-as/server/\$PROFILE/log/** ディレクトリにあります。

成功する EJB 呼び出しは、次の **audit.log** 出力に似ています。

例10.3 成功する EJB 呼び出しのログエントリ

```

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
object:
    Principal: [roles=[identitySubstitutionCaller,
extraRunAsRole],principal=runAsUser]
    Principal:
Roles(members:extraRunAsRole,identitySubstitutionCaller)
:callerRunAs=[roles=[identitySubstitutionCaller,
extraRunAsRole],principal=runAsUser]:callerRunAs=[roles=
[identitySubstitutionCaller,

```

```
extraRunAsRole],principal=runAsUser]:ejbRestrictionEnforcement=false;ejbVersion=null];Source=org.jboss.security.plugins.javaee.EJBAuthorizationHelper;Exception:=;
```

失敗する EJB 呼び出しは、次の **audit.log** 出力に似ています。

例10.4 失敗する EJB 呼び出しのログエントリ

```
[Error]policyRegistration=org.jboss.security.plugins.JBossPolicyRegistration@76ed4518;Resource:=
[org.jboss.security.authorization.resources.EJBResource:contextMap=
{policyRegistration=org.jboss.security.plugins.JBossPolicyRegistration@76ed4518}:method=public java.security.Principal
org.jboss.test.security.ejb3.SimpleStatelessSessionBean.invokeUnavailableMethod():ejbMethodInterface=Remote:ejbName=SimpleStatelessSessionBean:ejbPrincipal=UserA:MethodRoles=Roles(<NOBODY>,):securityRoleReferences=null:callerSubject=Subject:
Principal: UserA
Principal: Roles(members:RegularUser,Administrator)
:callerRunAs=null:callerRunAs=null:ejbRestrictionEnforcement=false
:ejbVersion=null];Source=org.jboss.security.plugins.javaee.EJBAuthorizationHelper;Exception:=Authorization Failed: ;
```

手順10.2 Web コンテナにセキュリティ監査を有効化

1. EJB セキュリティ監査を有効にします

手順10.1「セキュリティ監査機能の有効化」の説明通りセキュリティを有効にする必要があります。

2. サーバーレلمで監査を有効にします

Web コンテナ監査は **server.xml** ファイルのサーバーレلمで最初にアクティブにする必要があります。

server.xml ファイルは **jboss-as/server/\$PROFILE/deploy/jbossweb.sar/** ディレクトリにあります。

<Realm> 要素には 例10.5「server.xml 監査のアクティブ化」ごとに **enableAudit="true"** 属性セットがなければなりません。

例10.5 server.xml 監査のアクティブ化

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

3. 監査レベルのシステムプロパティを指定します

Web アプリケーションの監査レベルは **run.sh** (Linux) または **run.bat** (Microsoft Windows) スクリプトの `org.jboss.security.web.audit` システムプロパティを使用して指定する必要があります。

別の方法として、**jboss-as/server/\$PROFILE/deploy/properties-service.xml** ファイルのシステムプロパティを指定することもできます。

- Linux

jboss-as/bin/run.sh ファイルにシステムプロパティを追加します。

```
## 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

jboss-as/bin/run.bat ファイルにシステムプロパティを追加します。

```
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

jboss-as/server/\$PROFILE/deploy/properties-service.xml ファイルの **SystemPropertiesService** クラス MBeanを更新し、`<attribute>` として java プロパティを宣言します。以下のコードサンプルで関連するオペレーティングシステムブロックを非コメント化できます。

```
...

<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. セキュリティ監査が正しく機能していることを確認します

システムプロパティがファイルで指定された時点で、監査ログエントリは Web 呼び出しが成功するか確認します。

audit.log ファイルは **jboss-as/server/\$PROFILE/log/** ディレクトリにあります。

成功する Web 呼び出しは、次の **audit.log** 出力に似ています。

例10.6 成功する Web 呼び出しのログエントリ

```

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:=;

```


失敗する EJB 呼び出しは、次の **audit.log** 出力に似ています。

例10.7 失敗する Web 呼び出しのログエントリ

```

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]:)
```


第11章 セキュリティドメインのデプロイ

モジュールのセキュリティドメイン

セキュリティドメインのデプロイメントメソッドで、セキュリティドメイン宣言は **デプロイメント記述子**に含まれます。モジュールのセキュリティドメインは ***-jboss-beans.xml** の形式を取ります。EJB Jar の **META-INF** ディレクトリ、または Web アプリケーション (WAR) の **WEB-INF** ディレクトリに含まれています。

デプロイメント記述子

アプリケーションのデプロイメント設定を記述する宣言型の XML 設定ファイルです。アプリケーションがデプロイされる方法は、このファイル内で変更可能なため、アプリケーションの基礎となるコードに変更を加える必要がなくなります。

JBoss Enterprise Application Platform でセキュリティドメインをデプロイする方法は 2 つあります。

1. **jboss-as/server/\$PROFILE/conf/login-config.xml** ファイルで、セキュリティドメインを宣言します。
2. モジュールのセキュリティドメインを作成、デプロイします。

手順11.1 モジュールのセキュリティドメイン設定

この手順に従って、EJB と Web アプリケーションに対し 2 つのドメインを持つ基本的なモジュールのセキュリティドメインのデプロイメント記述子を設定します。

各ドメインは承認ポリシーに対して **UsersRolesLoginModule** を使用しますが、モジュールのセキュリティドメインを作成する場合はこのログインモジュールに制限されることはありません。JBoss Enterprise Application Platform に同梱されている追加のログインモジュールに関しては、「[モジュールの使用](#)」を参照してください。

1. デプロイメント記述子を作成します

デプロイメント記述子のファイルを作成し、セキュリティドメイン設定を含める必要があります。

ご使用のアプリケーションにデプロイメント記述子をすでに作成している場合は、このステップを飛ばしてステップ 2 に進んでください。

ファイル名は **[domain_name]-jboss-beans.xml** の形式を取ります。*domain_name* が任意なら、デプロイメント記述子の名前がサーバープロファイル全体に一意であるようにアプリケーションに有効な名前を選択する必要があります。

ファイルには標準 XML 宣言と正しく設定した **<deployment>** 要素を含む必要があります。

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

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

</deployment>
```

2. アプリケーションポリシーを定義します

個別のセキュリティドメインは **<deployment>** 要素内で定義されます。

以下の例では、2つのセキュリティドメインが指定されています。各認証ポリシーは同じログインモジュールとモジュールパラメータを使用します。



注記

他のログインモジュールは Enterprise Application Platform の向けに使用できません。使用できるログインモジュールに関する詳細は「[モジュールの使用](#)」を参照してください。

```
<?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. デプロイメント記述子をデプロイ、またはパッケージ化します

デプロイメント記述子ファイルをインストール環境に必要なサーバープロファイルの **jboss-as/server/\$PROFILE/deploy** ディレクトリに移動します。

ご使用のアプリケーションを広く流通させる場合は、EJB Jar の **META-INF** ディレクトリ、または Web アプリケーション (WAR) の **WEB-INF** ディレクトリでデプロイメント記述子をパッケージ化します。

第12章 ログインモジュール

12.1. モジュールの使用

JBoss Enterprise Application Platform には大部分の管理ニーズに合うバンドルされたログインモジュールが含まれています。JBoss Enterprise Application Platform はリレーショナルデータベース、Lightweight Directory Access Protocol (ライトウェイトディレクトリアクセスプロトコル: LDAP) サーバーまたはフラットファイルからのユーザー情報を読み取ることができます。こうしたコアのログインモジュールに加えて、JBoss はユーザー情報を JBoss の非常にカスタマイズされたニーズに提供する他のログインモジュールも備えています。

12.1.1. LdapLoginModule

LdapLoginModule は Lightweight Directory Access Protocol (LDAP) サーバーに対して認証する **LoginModule** 実装です。Java Naming and Directory Interface (JNDI) LDAP プロバイダを使用してアクセス可能な LDAP サーバーにユーザー名と資格情報が保存されている場合は、**LdapLoginModule** を使用します。

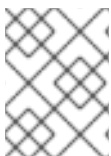


注記

LDAP を SPNEGO 認証と使用したり、LDAP サーバーの使用中に認証フェーズの一部をスキップしたい場合は、SPNEGOLoginModule とチェーンされた AdvancedLDAPLoginModule の使用や、AdvancedLDAPLoginModule のみの使用を検討してください (Negotiation ユーザーガイドを参照してください)。

識別名 (DN)

Lightweight Directory Access Protocol (LDAP) では、識別名は一意的にディレクトリのオブジェクトを特定します。各識別名にはすべての他のオブジェクトからの一意の名前と場所がある必要があり、これは多くの属性値ペア (AVP) を使用することで可能になります。AVP は共通名、組織ユニットなどの情報を定義します。結果としてこうした値の組み合わせは、LDAP が必要とする一意の文字列となります。



注記

このログインモジュールは認証されていないアイデンティティとパスワードスタックにも対応します。

LDAP 接続情報は JNDI 初期コンテキストを作成するために使用する環境オブジェクトに渡される設定オプションとして提供されます。使用される標準の LDAP JNDI プロパティとして以下が挙げられます。

java.naming.factory.initial

InitialContextFactory 実装クラス名です。これは Sun LDAP プロバイダ実装 **com.sun.jndi.ldap.LdapCtxFactory** にデフォルト設定されています。

java.naming.provider.url

LDAP サーバーの LDAP URL です。

java.naming.security.authentication

使用するセキュリティプロトコルのレベルです。使用可能な値は **none**、**simple**、**strong** などです。プロパティが定義されていない場合は、動作はサービスプロバイダより決定されます。

java.naming.security.protocol

セキュアなアクセスのために使用されるトランスポートプロトコルです。この設定オプションをサービスプロバイダ (例えば SSL) のタイプに設定します。プロパティが定義されていない場合は、動作はサービスプロバイダより決定されます。

java.naming.security.principal

サービスに呼び出し側を認証するプリンシパルのアイデンティティを指定します。これは以下のような他のプロパティから構築されます。

java.naming.security.credentials

サービスに呼び出し側を認証するプリンシパルの資格情報を指定します。資格情報はハッシュされたパスワード、クリアテキストのパスワード、キー、または証明書の形式を取ることができます。プロパティが定義されていない場合は、動作はサービスプロバイダより決定されます。

対応するログインモジュールの設定オプションとして以下が挙げられます。

principalDNPrefix

ユーザー *識別名* を形成するためにユーザー名に追加されたプレフィックスです。詳細は **principalDNSuffix** を参照してください。

principalDNSuffix

ユーザーの識別名を形成するときにユーザー名に追加されるサフィックスです。これはユーザーにユーザー名の入力を促し、ユーザーが完全な識別名を入力する必要がある場合に役立ちます。このプロパティと **principalDNSuffix** を使用することで、**userDN** は **principalDNPrefix** + **username** + **principalDNSuffix** として形成されます。

useObjectCredential

資格情報を示す値は、JAAS **PasswordCallback** を使用した **char[]** パスワードとしてではなく、**Callback** の **org.jboss.security.auth.callback.ObjectCallback** タイプを使用して、不透明な **Object** として取得されるべきです。これにより、LDAP サーバーに **char[]** でない資格情報を渡すことができます。使用可能な値は **true** と **false** です。

rolesCtxDN

ユーザーロールを検索するためのコンテキストへの固定した識別名です。

userRolesCtxDNAttributeName

ユーザーロールを検索するためのコンテキストへの識別名を含むユーザーオブジェクトの属性名です。これはユーザーのロールを検索するコンテキストは各ユーザーに固有にできる点で、**rolesCtxDN** とは異なります。

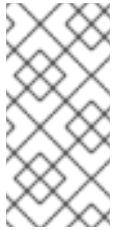
roleAttributeID

ユーザーロールを含む属性名です。指定されていない場合は、**roles** にデフォルト設定されます。

roleAttributesDN

roleAttributeID にロールオブジェクトの完全な識別名、またはロール名が含まれるかを示すフラグです。ロール名は識別名によるコンテキスト名の **roleNameAttributeID** 属性の値から付けられます。

true の場合、ロール属性はロールオブジェクトの識別名を表します。false の場合は、ロール名は **roleAttributeID** の値から付けられます。デフォルトは **false** です。



注記

特定のディレクトリスキーマ (例えば MS アクティブディレクトリ) では、ユーザーオブジェクトのロール属性は簡易な名前の代わりに、ロールオブジェクトに Distinguished Names (識別名 : DN) として保存されます。このスキーマタイプを使用する実装の場合、roleAttributeIsDN を **true** に設定する必要があります。

roleNameAttributeID

ロール名を含む roleCtxDN 識別名の値により指し示されたコンテキストの属性名です。roleAttributeIsDN プロパティが **true** に設定されている場合は、このプロパティを使用してロールオブジェクトの **name** 属性を見つけます。デフォルトは **group** です。

uidAttributeID

ユーザー ID に対応するユーザーロールを含むオブジェクトの属性名です。これを使用してユーザーロールを検索します。指定されていない場合は **uid** にデフォルト設定されます。

matchOnUserDN

ユーザーロールの検索がユーザーの完全な識別名と一致すべきかを指定するフラグです。**true** に設定されている場合は、完全な **userDN** は一致する値として使用されます。**false** に設定されている場合は、ユーザー名のみが uidAttributeName 属性に対し一致する値として使用されます。デフォルト値は **false** です。

unauthenticatedIdentity

認証情報を含まない要求に割り当てるプリンシパルの名前です。この動作は **UsernamePasswordLoginModule** スーパークラスから引き継がれます。

allowEmptyPasswords

空の (長さ 0) パスワードが LDAP サーバーに渡されるべきかを示すフラグです。一部の LDAP サーバーは空のパスワードを匿名ログインとして扱いますが、このような動作は望ましくない場合があります。空のパスワードを拒否するには、**false** に設定します。**true** に設定すると、LDAP サーバーは空のパスワードを検証します。デフォルトは **true** です。

ユーザー認証を実行するには、ログインモジュールの設定オプションに基づき LDAP サーバーに接続します。LDAP サーバーに接続するためには、本項で前述したように LDAP JNDI プロパティから構成される環境を持つ **InitialLdapContext** を作成します。

Context.SECURITY_PRINCIPAL は principalDNPrefix と principalDNSuffix オプション値と組み合わせでコールバックハンドラから取得されたユーザーの識別名に設定されており、Context.SECURITY_CREDENTIALS プロパティは useObjectCredential オプションに応じて **String** パスワード、または **Object** 資格情報のどちらかに設定されます。

認証が成功した (**InitialLdapContext** インスタンスが作成された) 時点で、ユーザーのロールは roleAttributeName と uidAttributeName オプション値に設定された検索属性を持つ **rolesCtxDN** ロケーションに関する検索を実行することでクエリされます。ロール名は検索結果セットのロール属性の **toString** メソッドを呼び出すことで取得されます。

例12.1 LDAP ログインモジュール認証ポリシー

この認証ポリシーはセキュリティドメイン認証ポリシーのパラメータの使用方法について記述しています。

```
<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>
```

testLDAP <login-module> 設定の

java.naming.factory.initial、***java.naming.factory.url***、***java.naming.security*** オプションは次の条件を示しています。

- Sun LDAP JNDI プロバイダ実装が使用されます。
- LDAP サーバーはポート 1389 のホスト **ldaphost.jboss.org** にあります。
- LDAP 簡易認証メソッドを使用して、LDAP サーバーと接続します。

ログインモジュールは、認証を試行しているユーザーを表す識別名 (DN) を使用して LDAP サーバーへの接続を試行します。この DN は前述のとおり、渡された **principalDNPrefix**、ユーザー名、**principalDNSuffix** から構成されています。[例12.2「LDIF ファイルの例」](#)では、ユーザー名 **jsmith** は **uid=jsmith,ou=People,dc=jboss,dc=org** にマップされます。



注記

例では、LDAP サーバーがユーザーのエントリ (この例では **theduke**) の **userPassword** 属性を使用してユーザーを認証すると仮定しています。ほとんどの LDAP サーバーはこの方法で動作します。ただし、LDAP サーバーが違う方法で認証を行う場合は、ご使用の実稼働環境の要件に応じて LDAP を設定する必要があります。

認証が成功した時点で、uidAttributeID がユーザーと一致するエントリに対し **rolesCtxDN** のサブツリー検索を行うことにより、承認がベースとなるロールが取得されます。matchOnUserDN が true の場合、検索はユーザーの完全な DN に基づきます。そうでない場合は、検索は入力した実際のユーザー名に基づきます。この例では、検索は **uid=jduke,ou=People,dc=jboss,dc=org** と等しい **member** 属性を持つすべてのエントリの **ou=Roles,dc=jboss,dc=org** で行います。検索はロールエントリで **cn=JBossAdmin** を検索します。

検索は roleAttributeID オプションで指定された属性を返します。この例では、属性は **cn** です。返された属性は **JBossAdmin** であるため、**jsmith** ユーザーは **JBossAdmin** ロールに割り当てられます。

ローカルの LDAP サーバーは多くの場合アイデンティティと認証サービスを備えていますが、承認サービスを使用することはできません。これはアプリケーションロールが常に LDAP グループに適切にマップされているとは限らず、LDAP 管理者は中心の LDAP サーバーで外部のアプリケーション固有のデータを許可したくない場合が多いからです。一般的に LDAP 認証モジュールはデータベースのログインモジュールなど開発中のアプリケーションにより適したロールを提供できる別のログインモジュールとペアとなっています。

このデータが動作する対象のディレクトリの構造を表す LDAP Data Interchange Format (LDAP データ交換形式: LDIF) ファイルは [例12.2「LDIF ファイルの例」](#) に記載されています。

LDAP データ交換形式 (LDIF)

プレーンテキストデータは、LDAP ディレクトリのコンテンツと更新要求を表すために使用される形式を交換します。ディレクトリのコンテンツは各オブジェクトまたは各更新要求に対して 1 つの記録として表されます。コンテンツは追加、修正、削除、名前変更の要求で構成されています。

例12.2 LDIF ファイルの例

```
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. LdapExtLoginModule

識別名 (DN)

Lightweight Directory Access Protocol (LDAP) では、識別名は一意的にディレクトリのオブジェクトを特定します。各識別名には他のオブジェクトとは異なる一意の名前と場所が必要なため、複数の属性値ペア (AVP) を使用して対応します。AVP は共通名や組織単位などの情報を定義します。結果としてこうした値の組み合わせは、LDAP が必要とする一意の文字列となります。

org.jboss.security.auth.spi.LdapExtLoginModule は、認証のためにバインドするユーザーと関連するロールを検索します。ロールクエリーは、DN を再帰的に従いロール階層構造をナビゲートします。

LoginModule オプションには、選択された LDAP JNDI プロバイダーによりサポートされるすべてのオプションが含まれます。標準的なプロパティ名の例は以下のとおりです。

- Context.INITIAL_CONTEXT_FACTORY = "java.naming.factory.initial"
- Context.SECURITY_PROTOCOL = "java.naming.security.protocol"
- Context.PROVIDER_URL = "java.naming.provider.url"
- Context.SECURITY_AUTHENTICATION = "java.naming.security.authentication"
- Context.REFERRAL = "java.naming.referral"

ログインモジュール実装論理は次の順番に従います。

1. 最初の LDAP サーバーバインドは bindDN プロパティと bindCredential プロパティを使用して認証されます。bindDN はユーザーとロールのために baseCtxDN ツリーと rolesCtxDN ツリーの両方を検索するパーミッションを持つユーザーです。認証に使用する user DN は、baseFilter プロパティで指定されたフィルターを使用してクエリされます。
2. 結果として得られた userDN は userDN を InitialLdapContext 環境 Context.SECURITY_PRINCIPAL として使用し LDAP サーバーにバインドすることにより認証されます。Context.SECURITY_CREDENTIALS プロパティはコールバックハンドラーにより取得される String パスワードに設定されます。
3. これに成功すると、rolesCtxDN、roleAttributeID、roleAttributeIsDN、roleNameAttributeID、roleFilter のオプションを使用して関連するユーザーロールがクエリされます。

LdapExtLoginModule プロパティ

baseCtxDN

ユーザー検索を開始するコンテキストの固定 DN を指定します。

bindDN

ユーザークエリやロールクエリの LDAP サーバーに対してバインドするために使用する DN を指定します。baseCtxDN プロパティと rolesCtxDn プロパティで読み取り/検索パーミッションを持つ DN に bindDN を設定します。

bindCredential

bindDN のパスワード。bindCredential は、jaasSecurityDomain が指定されている場合に暗号化できます。このプロパティによって外部コマンドはパスワードを読み取ることができます。たとえば、**{EXT}cat file_with_password** のようになります。

jaasSecurityDomain

java.naming.security.principal を復号化するために使用される JaasSecurityDomain の JMX ObjectName。暗号化された形式のパスワードは **JaasSecurityDomainencrypt64(byte[])** メソッドによって返されます。暗号化形式のパスワードの生成には **org.jboss.security.plugins.PBEUtils** を使用することもできます。

baseFilter

認証するユーザーのコンテキストを特定するために使用される検索フィルター。ログインモジュールコールバックから取得される入力ユーザー名/userDN は、**{0}** 表現が存在するフィルターに代入されます。この代入の動作は標準的な **DirContext.search(Name, String, Object[], SearchControls cons)** メソッドから生じます。一般的なサンプル検索フィルターは **(uid={0})** です。

rolesCtxDN

ユーザーロールを検索するコンテキストの固定 DN。これは実際のロールが存在する DN ではなく、ユーザーロールを含むオブジェクトが存在する DN になります。たとえば、アクティブディレクトリではユーザーカウントが存在する DN になります。

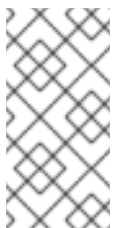
roleFilter

認証されたユーザーに関連するロールを特定するために使用される検索フィルター。ログインモジュールコールバックから取得される入力ユーザー名/userDN は、**{0}** 表現が存在するフィルターに代入されます。認証された userDN は **{1}** が表示されたフィルターに代入されます。入力ユーザー名に一致する検索フィルターの例は **(member={0})** です。認証された userDN に一致するのは **(member={1})** です。

roleAttributeIsDN

roleAttributeID にロールオブジェクトの完全な DN またはロール名が含まれるかを示すフラグです。ロール名は識別名によるコンテキスト名の roleNameAttributeId 属性の値から派生します。

true に設定された場合、ロール属性はロールオブジェクトの識別名を表します。**false** に設定された場合は、ロール名は **roleAttributeID** の値から付けられます。デフォルト値は **false** です。



注記

特定のディレクトリスキーマ (例えば MS アクティブディレクトリ) では、ユーザーオブジェクトのロール属性は簡易な名前の代わりに、ロールオブジェクトに Distinguished Names (識別名 : DN) として保存されます。このスキーマタイプを使用する実装の場合、roleAttributeIsDN を **true** に設定する必要があります。

roleAttributeID

ユーザーロールを含む属性の名前。roleAttributelsDN が **true** に設定されている場合、このプロパティは roleNameAttributeID 属性をクエリするコンテキストの DN になります。roleAttributelsDN プロパティが **false** に設定されている場合、このプロパティはロール名の属性名になります。

roleNameAttributeID

ロール名を含む roleCtxDN 識別名の値により指し示されたコンテキストの属性名です。roleAttributelsDN プロパティが **true** に設定されている場合は、このプロパティを使用してロールオブジェクトの **name** 属性を見つけます。デフォルトは **group** です。

roleRecursion

ロール検索が該当する一致コンテキストをトラバースするレベルを指定します。デフォルト値は **0** (無効) です。

searchTimeLimit

ユーザー/ロール検索のタイムアウト (ミリ秒単位)。デフォルト値は 10000 (10 秒) です。

searchScope

検索スコープをいずれかの文字列に設定します。デフォルト値は SUBTREE_SCOPE です。他のサポートされる値は以下のとおりです。

- OBJECT_SCOPE : 名前付ロールコンテキストのみを検索します。
- ONELEVEL_SCOPE : 名前付きロールコンテキスト以下で直接検索します。
- SUBTREE_SCOPE : ロールコンテキストが DirContext でない場合は、オブジェクトのみを検索します。ロールコンテキストが DirContext の場合、名前付きオブジェクト自体を含む名前付きオブジェクトをルートとするサブツリーを検索します。

allowEmptyPasswords

empty (length==0) パスワードを LDAP サーバーに渡すかどうかを示すフラグ。

空のパスワードは、一部の LDAP サーバーで匿名ログインとして扱われます。**false** に設定された場合、空のパスワードは拒否されます。**true** に設定された場合、LDAP サーバーは空のパスワードを検証します。デフォルト値は **true** です。

defaultRole

認証されたすべてのユーザーに含まれるロール。

parseRoleNameFromDN

DN がクエリによって返されたかを示すフラグに roleNameAttributeID が含まれます。**true** に設定すると、DN は roleNameAttributeID に対してチェックされます。**false** に設定すると、DN は roleNameAttributeID に対してチェックされません。このフラグは LDAP クエリのパフォーマンスを向上することができます。

parseUsername

DN が username によって解析されるかを示すフラグです。**true** に設定すると、DN は username に対して解析されます。**false** に設定すると、DN は username に対して解析されません。このオプションは usernameBeginString と usernameEndString と共に使用されます。

usernameBeginString

username を公開するため、DN の最初から削除する文字列を定義します。このオプションは usernameEndString と共に使用されます。

usernameEndString

username を公開するため、DN の最後から削除する文字列を定義します。このオプションは usernameBeginString と共に使用されます。

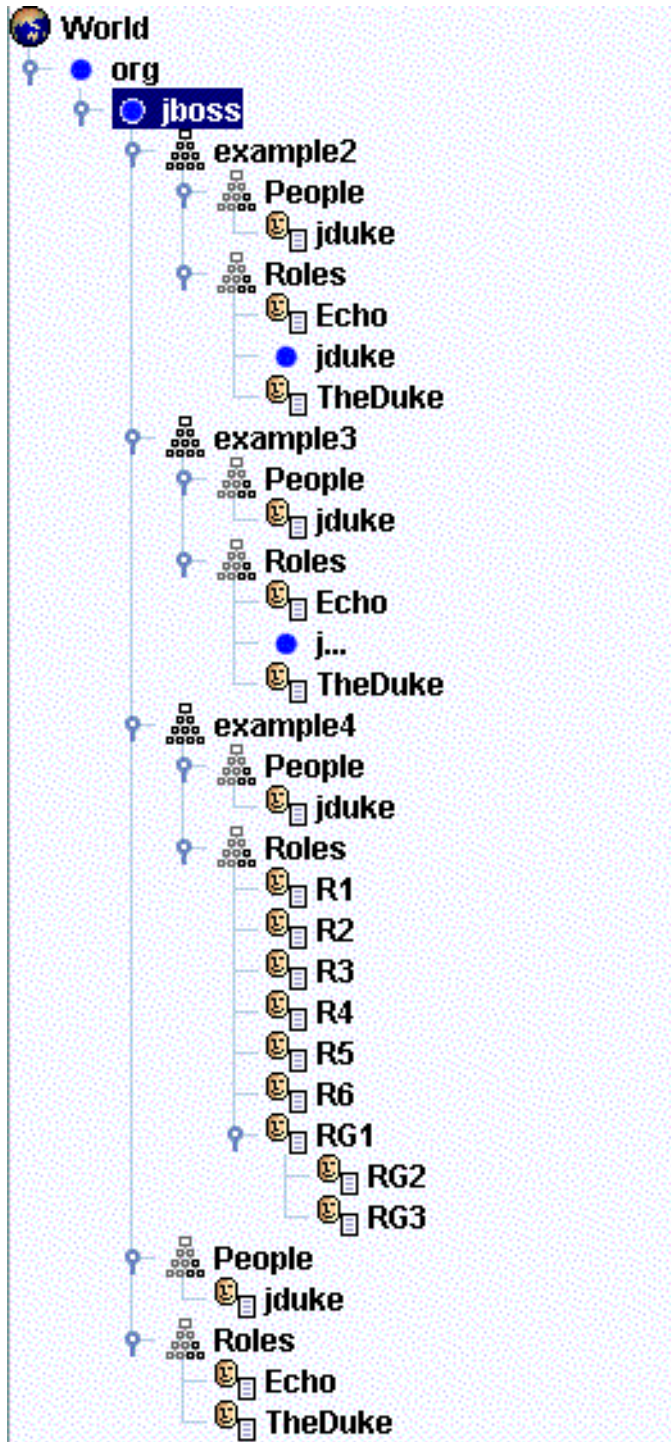


図12.1 LDAP 構造の例

例12.3 2 LDAP 設定の例

```
version: 1
dn: o=example2,dc=jboss,dc=org
```

```
objectClass: top
objectClass: dcObject
objectClass: organization
dc: jboss
o: JBoss

dn: ou=People,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: organizationalUnit
ou: People

dn: uid=jduke,ou=People,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: uidObject
objectClass: person
objectClass: inetOrgPerson
cn: Java Duke
employeeNumber: judke-123
sn: Duke
uid: jduke
userPassword:: dGhlZHVRZQ==

dn: uid=jduke2,ou=People,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: uidObject
objectClass: person
objectClass: inetOrgPerson
cn: Java Duke2
employeeNumber: judke2-123
sn: Duke2
uid: jduke2
userPassword:: dGhlZHVRZTI=

dn: ou=Roles,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: organizationalUnit
ou: Roles

dn: uid=jduke,ou=Roles,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: groupUserEx
memberOf: cn=Echo,ou=Roles,o=example2,dc=jboss,dc=org
memberOf: cn=TheDuke,ou=Roles,o=example2,dc=jboss,dc=org
uid: jduke

dn: uid=jduke2,ou=Roles,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: groupUserEx
memberOf: cn=Echo2,ou=Roles,o=example2,dc=jboss,dc=org
memberOf: cn=TheDuke2,ou=Roles,o=example2,dc=jboss,dc=org
uid: jduke2

dn: cn=Echo,ou=Roles,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: groupOfNames
cn: Echo
```

```

description: the echo role
member: uid=jduke,ou=People,dc=jboss,dc=org

dn: cn=TheDuke,ou=Roles,o=example2,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: TheDuke
description: the duke role
member: uid=jduke,ou=People,o=example2,dc=jboss,dc=org

dn: cn=Echo2,ou=Roles,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: groupOfNames
cn: Echo2
description: the Echo2 role
member: uid=jduke2,ou=People,dc=jboss,dc=org

dn: cn=TheDuke2,ou=Roles,o=example2,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: TheDuke2
description: the duke2 role
member: uid=jduke2,ou=People,o=example2,dc=jboss,dc=org

dn: cn=JBossAdmin,ou=Roles,o=example2,dc=jboss,dc=org
objectClass: top
objectClass: groupOfNames
cn: JBossAdmin
description: the JBossAdmin group
member: uid=jduke,ou=People,dc=jboss,dc=org

```

この LDAP 構造例のモジュール設定の概要はコード例に示されています。

```

testLdapExample2 {
    org.jboss.security.auth.spi.LdapExtLoginModule
        java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
        java.naming.provider.url="ldap://lamia/"
        java.naming.security.authentication=simple
        bindDN="cn=Root,dc=jboss,dc=org"
        bindCredential=secret1
        baseCtxDN="ou=People,o=example2,dc=jboss,dc=org"
        baseFilter="(uid={0})"
        rolesCtxDN="ou=Roles,o=example2,dc=jboss,dc=org";
        roleFilter="(uid={0})"
        roleAttributeIsDN="true"
        roleAttributeID="memberOf"
        roleNameAttributeID="cn"
};

```

例12.4 3 LDAP 設定の例

```

dn: o=example3,dc=jboss,dc=org
objectclass: top
objectclass: dcObject

```

```
objectclass: organization
dc: jboss
o: JBoss

dn: ou=People,o=example3,dc=jboss,dc=org
objectclass: top
objectclass: organizationalUnit
ou: People

dn: uid=jduke,ou=People,o=example3,dc=jboss,dc=org
objectclass: top
objectclass: uidObject
objectclass: person
objectClass: inetOrgPerson
uid: jduke
employeeNumber: judke-123
cn: Java Duke
sn: Duke
userPassword: theduke

dn: ou=Roles,o=example3,dc=jboss,dc=org
objectClass: top
objectClass: organizationalUnit
ou: Roles

dn: uid=jduke,ou=Roles,o=example3,dc=jboss,dc=org
objectClass: top
objectClass: groupUserEx
memberOf: cn=Echo,ou=Roles,o=example3,dc=jboss,dc=org
memberOf: cn=TheDuke,ou=Roles,o=example3,dc=jboss,dc=org
uid: jduke

dn: cn=Echo,ou=Roles,o=example3,dc=jboss,dc=org
objectClass: top
objectClass: groupOfNames
cn: Echo
description: the JBossAdmin group
member: uid=jduke,ou=People,o=example3,dc=jboss,dc=org

dn: cn=TheDuke,ou=Roles,o=example3,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: TheDuke
member: uid=jduke,ou=People,o=example3,dc=jboss,dc=org
```

この LDAP 構造例のモジュール設定の概要はコード例に示されています。

```
testLdapExample3 {
    org.jboss.security.auth.spi.LdapExtLoginModule
    java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
    java.naming.provider.url="ldap://lamia/"
    java.naming.security.authentication=simple
    bindDN="cn=Root,dc=jboss,dc=org"
    bindCredential=secret1
    baseCtxDN="ou=People,o=example3,dc=jboss,dc=org"
    baseFilter="(cn={0})"
```

```

        rolesCtxDN="ou=Roles,o=example3,dc=jboss,dc=org";
        roleFilter="(member={1})"
        roleAttributeID="cn"
    };

```

例12.5 4 LDAP 設定の例

```

dn: o=example4,dc=jboss,dc=org
objectclass: top
objectclass: dcObject
objectclass: organization
dc: jboss
o: JBoss

dn: ou=People,o=example4,dc=jboss,dc=org
objectclass: top
objectclass: organizationalUnit
ou: People

dn: uid=jduke,ou=People,o=example4,dc=jboss,dc=org
objectClass: top
objectClass: uidObject
objectClass: person
objectClass: inetOrgPerson
cn: Java Duke
employeeNumber: jduke-123
sn: Duke
uid: jduke
userPassword:: dGhlZHVrZQ==

dn: ou=Roles,o=example4,dc=jboss,dc=org
objectClass: top
objectClass: organizationalUnit
ou: Roles

dn: cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: RG1
member: cn=empty

dn: cn=RG2,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: RG2
member: cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org
member: uid=jduke,ou=People,o=example4,dc=jboss,dc=org

dn: cn=RG3,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: RG3
member: cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org

```

```

dn: cn=R1,ou=Roles,o=example4,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: R1
member: cn=RG2,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org

dn: cn=R2,ou=Roles,o=example4,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: R2
member: cn=RG2,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org

dn: cn=R3,ou=Roles,o=example4,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: R3
member: cn=RG2,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org
member: cn=RG3,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org

dn: cn=R4,ou=Roles,o=example4,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: R4
member: cn=RG3,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org

dn: cn=R5,ou=Roles,o=example4,dc=jboss,dc=org
objectClass: groupOfNames
objectClass: top
cn: R5
member: cn=RG3,cn=RG1,ou=Roles,o=example4,dc=jboss,dc=org
member: uid=jduke,ou=People,o=example4,dc=jboss,dc=org

```

この LDAP 構造例のモジュール設定の概要はコード例に示されています。

```

testLdapExample4 {
    org.jboss.security.auth.spi.LdapExtLoginModule
    java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
    java.naming.provider.url="ldap://lamia/"
    java.naming.security.authentication=simple
    bindDN="cn=Root,dc=jboss,dc=org"
    bindCredential=secret1
    baseCtxDN="ou=People,o=example4,dc=jboss,dc=org"
    baseFilter="(cn={0})"
    rolesCtxDN="ou=Roles,o=example4,dc=jboss,dc=org";
    roleFilter="(member={1})"
    roleAttributeID="memberOf"
};

```

例12.6 デフォルトの ActiveDirectory 設定

以下の例はデフォルトの Active Directory 設定を表します。

```

<?xml version="1.0" encoding="UTF-8"?>
<application-policy name="AD_Default">

```



```

<authentication>
  <login-module
code="org.jboss.security.auth.spi.LdapExtLoginModule" flag="required" >
  <!--
    Some AD configurations may require searching against
    the Global Catalog on port 3268 instead of the usual
    port 389. This is most likely when the AD forest
    includes multiple domains.
  -->
    <module-option
name="java.naming.provider.url">ldap://ldap.jboss.org:389</module-
option>
      <module-option name="bindDN">JBOSS\someadmin</module-option>
      <module-option name="bindCredential">password</module-option>
      <module-option
name="baseCtxDN">cn=Users,dc=jboss,dc=org</module-option>
      <module-option name="baseFilter">(sAMAccountName={0})</module-
option>
      <module-option
name="rolesCtxDN">cn=Users,dc=jboss,dc=org</module-option>
      <module-option name="roleFilter">(sAMAccountName={0})</module-
option>
      <module-option name="roleAttributeID">memberOf</module-option>
      <module-option name="roleAttributeIsDN">true</module-option>
      <module-option name="roleNameAttributeID">cn</module-option>
      <module-option name="searchScope">ONELEVEL_SCOPE</module-
option>
      <module-option name="allowEmptyPasswords">>false</module-
option>
    </login-module>
  </authentication>
</application-policy>

```

例12.7 再帰的ロール ActiveDirectory 設定

以下の例では、ActiveDirectory 内で再帰的ロール検索を実装します。[例12.6「デフォルトの ActiveDirectory 設定」](#)との主な違いは、ロール検索がユーザーの DN を使用してメンバー属性を検索するために置換されることです。ログインモジュールはロールの DN を使用してグループがメンバーであるグループを検出します。

```

<?xml version="1.0" encoding="UTF-8"?>
<application-policy name="AD_Recursive">
  <authentication>
    <login-module
code="org.jboss.security.auth.spi.LdapExtLoginModule" flag="required" >
      <module-option
name="java.naming.provider.url">ldap://ad.jboss.org:389</module-option>
      <module-option name="bindDN">JBOSS\searchuser</module-option>

```

```

        <module-option name="bindCredential">password</module-option>
        <module-option
name="baseCtxDN">CN=Users,DC=jboss,DC=org</module-option>
        <module-option name="baseFilter">(sAMAccountName={0})</module-
option>
        <module-option
name="rolesCtxDN">CN=Users,DC=jboss,DC=org</module-option>
        <module-option name="roleFilter">(member={1})</module-option>
        <module-option name="roleAttributeID">cn</module-option>
        <module-option name="roleAttributeIsDN">>false</module-option>
        <module-option name="roleRecursion">2</module-option>
        <module-option name="searchScope">ONELEVEL_SCOPE</module-
option>
        <module-option name="allowEmptyPasswords">>false</module-
option>
        <module-option name="java.naming.referral">follow</module-
option>
    </login-module>
</authentication>
</application-policy>

```

12.1.3. パスワードスタック

複数のログインモジュールはスタックでチェーン化でき、認証コンポーネントおよび承認コンポーネントを提供する各ログインモジュールがあります。これは多くの使用事例で動作しますが、認証と承認が複数のユーザー管理ストア全体で分かれる場合があります。

「[LdapLoginModule](#)」では、LDAP とリレーショナルデータベースを組み合わせ、どちらかのシステムでユーザーが認証されるようにする方法を説明しています。ここで、ユーザーは中心の LDAP サーバーで管理され、アプリケーション固有のロールはアプリケーションのリレーショナルデータベースに保存されているケースを考えてみましょう。パスワードスタックのモジュールオプションはこの関係を確立します。

パスワードスタックを使用するには、各ログインモジュールは `<module-option> password-stacking` 属性を **useFirstPass** に設定しなければなりません。パスワードスタックに設定された以前のモジュールがユーザーを認証している場合、その他すべてのスタックモジュールはユーザーが認証されたと見なし、承認ステップへのロールセットの提供のみ実行しようとしています。

password-stacking オプションが **useFirstPass** に設定されている場合、このモジュールは最初にログインモジュールの共有状態マップにあるプロパティ名 `javax.security.auth.login.name` で共有ユーザー名を探し、プロパティ名 `javax.security.auth.login.password` でパスワードを探します。

見つかった場合は、これらのプロパティはプリンシパル名とパスワードとして使用されます。見つからない場合は、プリンシパル名とパスワードはこのログインモジュールで設定され、プリンシパル名はプロパティ名 `javax.security.auth.login.name`、パスワードはプロパティ名 `javax.security.auth.login.password` で保存されます。



注記

パスワードスタックを使用する場合、必要になるようすべてのモジュールを設定します。これにより全モジュールが考慮され、ロールを承認プロセスに提供できるようになります。

例12.8 パスワードスタックのサンプル

この例はパスワードスタックの使用方法を示しています。

```
<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>
```

12.1.4. パスワードのハッシュ化

ほとんどのログインモジュールはクライアント提供のパスワードとユーザー管理システムに保存されているパスワードとを比較する必要があります。こうしたモジュールは通常プレーンテキストのパスワードと動作しますが、プレーンテキストのパスワードがサーバー側で保存されないようにハッシュされたパスワードに対応するよう設定することができます。

例12.9 パスワードのハッシュ化

以下は、プリンシパル名 **nobody** を認証されていないユーザーに割り当て、**usersb64.properties** ファイルにパスワードの Base 64 エンコード化と MD5 ハッシュが含まれるログインモジュール設定です。**usersb64.properties** ファイルはデプロイメントクラスパスの一部となり、**/conf** ディレクトリに保存することができます。

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

```

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

パスワードをハッシュするために使用する **java.security.MessageDigest** アルゴリズムの名前です。デフォルトはないため、ハッシュを有効にするにはこのオプションを指定します。一般的な値は **MD5** と **SHA** です。

hashEncoding

base64、**hex** または **rfc2617** の 3 つのエンコーディングタイプのうち 1 つを指定する文字列です。デフォルトは **base64** です。

hashCharset

クリアテキストのパスワードをバイト配列に変換するために使用するエンコーディング文字セットです。プラットフォームのエンコーディングがデフォルトです。

hashUserPassword

ユーザーが提出するパスワードにハッシュアルゴリズムが適用される必要があることを指定します。ハッシュされたユーザーパスワードはログインモジュールの値と比較され、パスワードのハッシュとして要求されます。デフォルトは **true** です。

hashStorePassword

サーバー側に保存されているパスワードにハッシュアルゴリズムが適用される必要があることを指定します。これはダイジェスト認証に使用され、ユーザーは比較するサーバーからの要求固有のトークンとともにユーザーパスワードのハッシュを提出します。ハッシュアルゴリズム (ダイジェストでは **rfc2617**) は、サーバー側のハッシュを計算するために使用され、クライアントから送られるハッシュ値と一致しなければなりません。

コードでパスワードを生成する必要がある場合、**org.jboss.security.Util** クラスは、指定されたエンコーディングを使用してパスワードをハッシュ化する静的ヘルパーメソッドを提供します。次の例は base64 でエンコードされ、MD5 にてハッシュ化されたパスワードになります。

```

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

```

OpenSSL はコマンドラインでハッシュ化されたパスワードを即座に生成できる別の方法を提供します。次の例も base64 でエンコードされ、MD5 でハッシュ化されたパスワードを作成します。この例では、base64 でエンコードされた形式に変換するため、プレーンテキストのパスワード **password** が OpenSSL のダイジェスト関数へパイプされた後、別の OpenSSL 関数にパイプされています。

```

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

```

両方の場合とも、ハッシュ化されたパスワードは **X03M01qnZdYdgyfeuILPmQ==** になります。前述の例のアプリケーションポリシーに指定されたユーザープロパティファイル **usersb64.properties** にこの値を保存する必要があります。

12.1.5. 認証されていないアイデンティティ

すべての要求が認証された形式で受け取られるとは限りません。**unauthenticated identity** は関連付けられた認証情報を持たない要求に特定のアイデンティティ (ゲストなど) を割り当てるログインモジュール設定オプションです。これを使用して、保護されていないサーブレットが特定のロールを必要としない EJB でメソッドを呼び出すことができます。そのようなプリンシパルには関連付けられたロールがないため、セキュアでない EJB または確認されていないパーミッション制約に関連付けられた EJB メソッドにのみアクセスできます。

- **unauthenticatedIdentity**: これは認証情報を含まない要求に割り当てられる必要があるプリンシパル名を定義します。

12.1.6. UsersRolesLoginModule

UsersRolesLoginModule は Java プロパティファイルからロードされた複数のユーザーとユーザーロールに対応する簡易ログインモジュールです。ユーザー名からパスワードへのマッピングファイルは **users.properties** と呼ばれ、ユーザー名からロールへのマッピングファイルは **roles.properties** と呼ばれます。

対応するログインモジュールの設定オプションとして以下が挙げられます。

usersProperties

ユーザー名からパスワードへのマッピング含むプロパティリソース (ファイル) の名前です。これはデフォルトで **<filename_prefix>-users.properties** に設定されます。

rolesProperties

ユーザー名からロールへのマッピング含むプロパティリソース (ファイル) の名前です。これはデフォルトで **<filename_prefix>-roles.properties** に設定されます。

このログインモジュールは、パスワードスタック、パスワードハッシュ、認証されていないアイデンティティに対応します。

プロパティファイルは、initialize メソッドのスレッドコンテキストクラスローダーを使用して初期化中にロードされます。そのため、こうしたファイルは Java EE deployment JAR、JBoss の設定ディレクトリ、JBoss サーバーまたはシステムクラスパスのどのディレクトリにも置くことができます。このログインモジュールの第一の目的は、アプリケーションでデプロイされたプロパティファイルを使用して複数のユーザーとロールのセキュリティ設定を簡単にテストすることです。

例12.10 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>

```

例12.10「UserRolesLoginModule」では、**ejb3-sampleapp-users.properties** ファイルは各ユーザーエントリが1行ずつになっている **username=password** 形式を使用します。

```

username1=password1
username2=password2
...

```

例12.10「UserRolesLoginModule」で参照された **ejb3-sampleapp-roles.properties** ファイルは、オプションのグループ名の値が付いたパターン **username=role1,role2,** を使用します。例は次の通りです。

```

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

```

ejb3-sampleapp-roles.properties に存在している **user name.XXX** プロパティ名のパターンを使用して、ユーザー名のロールを特定の名前付きロールグループに割り当てます。プロパティ名の **XXX** の部分はグループ名を表します。**user name=...** の形式は **user name.Roles=...** の省略形で、**Roles** グループ名は **JaasSecurityManager** がユーザーのパーミッションを定義するロールを含むために必要な標準名です。

次は **jduke** ユーザー名の定義と同等です。

```

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

```

12.1.7. DatabaseServerLoginModule

DatabaseServerLoginModule は認証とロールマッピングに対応する Java Database Connectivity-based (JDBC) ログインモジュールです。ユーザー名、パスワード、ロール情報がリレーショナルデータベースに保存されている場合は、このログインモジュールを使用します。



注記

このモジュールは、パスワードスタック、パスワードハッシュ、認証されていないアイデンティティに対応します。

DatabaseServerLoginModule は2つの論理テーブルに基づいています。

```

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

```

Principals テーブルはユーザー **PrincipalID** を有効なパスワードに関連付け、**Roles** テーブルはユーザー **PrincipalID** をそのロールのセットに関連付けます。ユーザーパーミッションに使用されるロールは **Roles** の **RoleGroup** 列の値がある行に含まれている必要があります。

ログインモジュールが使用する SQL クエリを指定できるという点でこれらのテーブルは論理的です。唯一の要件は、**java.sql.ResultSet** が前述した **Principals** および **Roles** テーブルと同じ論理構造を持つことです。その結果は列インデックスに基づいてアクセスされるため、テーブルと列の実際の名前は関連していません。

この概念を明確にするために、すでに宣言されているとおりの **Principals** および **Roles** の 2 つのテーブルを持つデータベースを考えてみましょう。次のステートメントで以下のデータを持つテーブルを生成します。

- **Principals** テーブルの **echoman** の **Password** を持つ **PrincipalIDjava**
- **Roles** テーブルの **RolesRoleGroup** の **Echo** という名前のロールを持つ **PrincipalIDjava**
- **Roles** テーブルの **CallerPrincipalRoleGroup** の **caller_java** という名前のロールを持つ **PrincipalIDjava**

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

対応するログインモジュールの設定オプションとして以下が挙げられます。

dsJndiName

論理テーブル **Principals** および **Roles** を含むデータベースの **DataSource** の JNDI 名です。指定されていない場合は、**java:/DefaultDS** にデフォルト設定されます。

principalsQuery

準備されたステートメントのクエリで、**select Password from Principals where PrincipalID=?** と同等です。指定されていない場合は、これが使用される正確な準備されたステートメントとなります。

rolesQuery

準備されたステートメントのクエリで **select Role, RoleGroup from Roles where PrincipalID=?** と同等です。指定されていない場合は、これが使用される正確な準備されたステートメントとなります。

ignorePasswordCase

パスワード比較で大文字と小文字の区別を無視するべきかを示すブール型のフラグです。これは、ハッシュされたパスワードの大文字と小文字の区別が重要でないハッシュされたパスワードのエンコードに役立つことがあります。

principalClass

Principal 実装クラスを指定するオプションです。これはプリンシパル名に文字列引数を取るコンストラクタに対応する必要があります。

DatabaseServerLoginModule 設定の例として、以下のように作成することができます。


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

対応する **login-config.xml** エントリは以下のようになります。

```
<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>
```

12.1.8. BaseCertLoginModule

BaseCertLoginModule は X509 証明書に基づいてユーザーを認証します。このログインモジュールの一般的な使用事例は Web 層の **CLIENT-CERT** 認証です。

このログインモジュールは認証のみ行います。セキュアな Web または EJB コンポーネントへのアクセスを完全に定義するためには、承認ロールの取得が可能な別のログインモジュールと組み合わせる必要があります。このログインモジュールの 2 つのサブクラスである **CertRolesLoginModule** と **DatabaseCertLoginModule** は、動作を拡張しプロパティファイルまたはデータベースのいずれかから承認ロールを取得します。

BaseCertLoginModule ではユーザー検証を実行するために **KeyStore** が必要です。これは **org.jboss.security.SecurityDomain** 実装から取得します。一般的には **SecurityDomain** 実装はこの **jboss-service.xml** 設定の一部で示されているように **org.jboss.security.plugins.JaasSecurityDomain** MBean を使用して設定します。

```
<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>
```

この設定では **jmx-console** という名前を持つセキュリティドメインと **java:/jaas/jmx-console** という名前のもとで JNDI を通じて使用可能な **SecurityDomain** 実装を作成します。セキュリティドメインは JBossSX セキュリティドメインの名前付けパターンに従います。

手順12.1 Web アプリケーションを証明書とロールベースの承認でセキュアにする

この手順ではクライアント証明書とロールベースの承認を使用して、**jmx-console.war** などの Web アプリケーションをセキュアにする方法について説明します。

1. リソースとロールを宣言します

web.xml を修正し、認証と承認に使用される許可されたロールとセキュリティドメインとともにリソースがセキュアされることを宣言します。

```
<?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. JBoss セキュリティドメインを指定します

jboss-web.xml ファイルで、必須のセキュリティドメインを指定します。

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

3. ログインモジュール設定を指定します

今指定した **jmx-console** セキュリティドメインに対しログインモジュール設定を定義します。**conf/login-config.xml** ファイルで行います。

```
<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>

```

手順12.1「Web アプリケーションを証明書とロールベースの承認でセキュアにする」では **BaseCertLoginModule** がクライアント証明書の認証に使用され、**UsersRolesLoginModule** が **password-stacking=useFirstPass** オプションにより承認にのみ使用されることを示しています。**localhost.keystore** と **jmx-console-roles.properties** とともにクライアント証明書に関連付けられたプリンシパルにマップするエントリが必要です。

デフォルトでは、プリンシパルは 例12.11「証明書の例」で指定された DN などクライアント証明書の識別名を使用して作成されます

例12.11 証明書の例

```

[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

```

localhost.keystore は **CN=unit-tests-client, OU=JBoss Inc., O=JBoss Inc., ST=Washington, C=US** のエイリアスで保存された 例12.11「証明書の例」の証明書が必要となります。**jmx-console-roles.properties** も同じエントリに対しエントリが必要です。DN には通常区切り記号として扱われる文字列が含まれるため、以下のようにバックスラッシュ (\) を使用して許可されていない文字列をエスケープします。

```
# A sample roles.properties file for use with the UsersRolesLoginModule
CN\=unit-tests-client,\ OU\=JBoss\ Inc.,\ O\=JBoss\ Inc.,\
ST\=Washington,\ C\=US=JBossAdmin
admin=JBossAdmin
```

12.1.9. IdentityLoginModule

IdentityLoginModule はハードコードされたユーザー名をモジュールに対して認証されるすべてのサブジェクトに関連付ける簡易ログインモジュールです。**principal** オプションで指定された名前を使用して **SimplePrincipal** インスタンスを作成します。



注記

このモジュールはパスワードスタックに対応します。

このログインモジュールは、固定アイデンティティをサービスに提供する場合や、特定のプリンシパルと関連するロールに関連付けられたセキュリティをテストするときの開発環境に役立ちます。

対応するログインモジュールの設定オプションとして以下が挙げられます。

principal

SimplePrincipal に対して使用する名前で、すべてのユーザーはこの名前として認証されます。プリンシパルオプションが指定されていない場合、プリンシパル名は **guest** にデフォルト設定されます。

roles

ユーザーに割り当てられるロールをコンマで区切った一覧です。

XMLLoginConfig 設定エントリのサンプルは以下のとおりです。エントリはすべてのユーザーを **jduke** という名前のプリンシパルとして認証し、**TheDuke** および **AnimatedCharacter** のロール名を割り当てます。

```
<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>
```

12.1.10. RunAsLoginModule

RunAsLoginModule (**org.jboss.security.auth.spi.RunAsLoginModule**) はヘルパーモジュールで、run as role を認証のログインフェーズの期間にスタックにプッシュし、コミットまたは中止中のいずれかで run as role をポップします。

このログインモジュールの目的は、認証を実行するためにセキュアなリソースにアクセスする必要がある他のログインモジュールにロールを提供することです (例えば、セキュアな EJB にアクセスするログインモジュールなど)。**RunAsLoginModule** は run as role を確立する必要があるログインモジュールより先に設定する必要があります。

唯一のログインモジュールの設定オプションは以下のとおりです。

roleName

ログインフェーズの間に run as role として使用するロール名です。指定されていない場合は、デフォルトの **nobody** が使用されます。

12.1.11. RunAsIdentity の作成

JBoss Enterprise Application Platform が EJB メソッドへのアクセスをセキュアにするため、メソッド呼び出しが実行された時にユーザーアイデンティティは既知とならなければなりません。

サーバーのユーザーアイデンティティは **javax.security.auth.Subject** インスタンスまたは **org.jboss.security.RunAsIdentity** インスタンスのいずれかで表されます。これらのクラスは共に、アイデンティティとそのアイデンティティが所有するロールの一覧を表す 1 つ以上のプリンシパルを保存します。**javax.security.auth.Subject** の場合は、資格情報の一覧も保存されます。

ejb-jar.xml デプロイメント記述子の <assembly-descriptor> セクションでは、様々な EJB メソッドにアクセスするためにユーザーが必要な 1 つ以上のロールを指定します。これらの一覧を比較することで、EJB メソッドにアクセスするためにユーザーが必要なロールの 1 つを持っているかが分かります。

例12.12 org.jboss.security.RunAsIdentity の作成

ejb-jar.xml ファイルで、<session> 要素の子として定義された <run-as> ロールを持つ <security-identity> 要素を指定します。

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

この宣言では「Admin」RunAsIdentity ロールが作成される必要があることを意味します。

Admin ロールにプリンシパルの名前を付けるには、**jboss-web.xml** ファイルの <run-as-principal> 要素を定義します。

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

ejb-jar.xml および **jboss-web.xml** ファイルの **<security-identity>** 要素はデプロイメント時に解析されます。その後、**<run-as>** ロール名と **<run-as-principal>** の名前が **org.jboss.metadata.SecurityIdentityMetaData** クラスに保存されます。

例12.13 RunAsIdentity への複数ロールの割り当て

RunAsIdentity にさらにロールを割り当てるには、**jboss-web.xml** デプロイメント記述子 **<assembly-descriptor>** 要素グループのプリンシパルにロールをマップします。

```
<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>
```

例12.12「[org.jboss.security.RunAsIdentity の作成](#)」では「Mark」の **<run-as-principal>** が作成されました。この例の設定は「Support」ロールを追加することで「Admin」ロールを拡張します。新しいロールには当初定義されたプリンシパル「John」などの追加のプリンシパルが含まれています。

ejb-jar.xml および **jboss.xml** ファイルの **<security-role>** 要素はデプロイメント時に解析されます。**<role-name>** と **<principal-name>** データは **org.jboss.metadata.SecurityIdentityMetaData** クラスに保存されます。

12.1.12. ClientLoginModule

ClientLoginModule (**org.jboss.security.ClientLoginModule**) は呼び出し側アイデンティティと資格情報を確立するために JBoss クライアントにより使用されることを目的とした **LoginModule** の実装です。これは単に **org.jboss.security.SecurityAssociation.principal** を **callbackhandler** によって入力された **NameCallback** の値に設定し、また **org.jboss.security.SecurityAssociation.credential** を **callbackhandler** によって入力された **PasswordCallback** の値に設定します。

ClientLoginModule は、クライアントが現在のスレッドの呼び出し側を確立するために唯一サポートされているメカニズムです。スタンドアローンのクライアントアプリケーションとサーバー環境 (セキュリティ環境は透過的に JBossSX を使用するように設定されていない場面で JBoss EJB クライアントとして働きます) の両方とも **ClientLoginModule** を使用する必要があります。

このログインモジュールは認証は行わないことに注意してください。そのログインモジュールに提供されたログイン情報をサーバー上の後続の認証に対して JBoss サーバー EJB 呼び出しレイヤにコピーするだけです。ユーザーのクライアント側認証を実行する必要がある場合は、**ClientLoginModule** に加えて別のログインモジュールを設定する必要があるでしょう。

対応するログインモジュールの設定オプションとして以下が挙げられます。

multi-threaded

ログインスレッドがプリンシパルと資格情報ストレージソースに接続する方法を指定する値です。true に設定する場合、各ログインスレッドは独自のプリンシパルと資格情報ストレージを持ち、それぞれ別々のスレッドが独自のログインを実行します。これは複数のユーザーアイデンティティが別々のスレッドでアクティブであるクライアント環境で役立ちます。false に設定する場合、ログインアイデンティティと資格情報は VM のすべてのスレッドに適用するグローバル変数です。デフォルト設定は **false** です。

password-stacking

LdapLoginModule などの他のログインモジュールを使用してクライアントのクライアント側認証をアクティブにします。**password-stacking** オプションが **useFirstPass** に設定されている場合、モジュールは最初にログインモジュールの共有状態マップの **javax.security.auth.login.name** と **javax.security.auth.login.password** をそれぞれ使用して共有のユーザー名とパスワードを探します。その結果、これより前に設定されたモジュールは有効な JBoss ユーザー名とパスワードを確立することができます。

restore-login-identity

login() メソッドへのエントリで見える **SecurityAssociation** プリンシパルと資格情報が中止またはログアウトのいずれかで保存、回復されているかを指定する値です。これが必要なのは、アイデンティティを変更してももとの呼び出し側のアイデンティティを回復する場合です。true に設定されている場合は、プリンシパルと資格情報は中止またはログアウトで保存、回復されます。false に設定されている場合は、中止とログアウトは **SecurityAssociation** をクリアにします。デフォルト値は **false** です。

12.1.13. SPNEGOLoginModule

SPNEGOLoginModule (**org.jboss.security.negotiation.spnego.SPNEGOLoginModule**) は、KDC で呼び出し側のアイデンティティと資格情報を確立する **LoginModule** の実装です。モジュールは SPNEGO (Simple and Protected GSSAPI Negotiation メカニズム) を実装する JBoss Negotiation プロジェクトの一部です。AdvancedLDAPLoginModule とチェーンされた設定でこの認証を使用し、LDAP サーバーと連携できるようにすることができます。JBoss Negotiation の詳細については、JBoss Negotiation ユーザーガイドを参照してください。

12.2. カスタムモジュール

JBossSX フレームワークとバンドルされたログインモジュールがセキュリティ環境で動作しない場合は、独自のカスタムのログインモジュール実装を書くことができます。**JaasSecurityManager** には **Subject** プリンシパルのセットの特定の使用パターンが必要です。JAAS Subject クラスの情報ストレージの機能および **JaasSecurityManager** と動作するログインモジュールを書くためのこうした機能の必要な使用方法を理解しておく必要があります。

本項ではこの要件を検証し、カスタムのログインモジュールを実装するときに役立つ 2 つの抽象ベースの **LoginModule** 実装を紹介します。

次のメソッドを使用することで **Subject** と関連付けられたセキュリティ情報を取得できます。

```
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)
```


Subject アイデンティティとロールに関しては、JBossSX は `getPrincipals()` と `getPrincipals(java.lang.Class)` により取得されるプリンシパルのセットという最も論理的な選択をしました。使用パターンは以下のとおりです。

- ユーザーアイデンティティ (例えばユーザー名、ソーシャルセキュリティ番号、従業員 ID) は **SubjectPrincipals** セットの `java.security.Principal` オブジェクトとして保存されます。ユーザーアイデンティティを表す **Principal** 実装は、プリンシパルの名前に関する比較と等値で始める必要があります。適切な実装は `org.jboss.security.SimplePrincipal` クラスとして使用できます。他の **Principal** インスタンスは必要に応じて **SubjectPrincipals** セットに追加できます。
- 割り当てられたユーザーロールも **Principals** セットに保存され、`java.security.acl.Group` インスタンスを使用して名前付きロールセットでグループ化されます。**Group** インターフェースは **Principal / Group** の集まりを定義し、`java.security.Principal` のサブインターフェースとなります。
- **Subject** に割り当てるロールセットの数はいくつでも構いません。
- JBossSX フレームワークは **Roles** と **CallerPrincipal** という名前のよく知られた 2 つのロールセットを使用します。
 - **Roles** グループは **Subject** が認証されたアプリケーションドメインで知られた名前付きロールに対する **Principal** の集まりです。このロールセットは `EJBContext.isCallerInRole(String)` のようなメソッドで使用され、EJB は現在の呼び出し側が名前付きアプリケーションドメインロールに属しているか確認するためにこれを使用できます。メソッドパーミッションの確認を実行するセキュリティインターセプタのロジックもこのロールセットを使用します。
 - **CallerPrincipalGroup** はアプリケーションドメインのユーザーに割り当てられた単一の **Principal** アイデンティティで構成されます。`EJBContext.getCallerPrincipal()` メソッドは **CallerPrincipal** を使用して、アプリケーションドメインが動作環境アイデンティティからアプリケーションに適したユーザーアイデンティティにマップすることができます。**Subject** に **CallerPrincipalGroup** がない場合は、アプリケーションアイデンティティは動作環境アイデンティティと同じです。

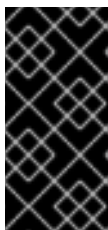
12.2.1. Subject の使用パターンのサポート

「[カスタムモジュール](#)」で説明した **Subject** 使用パターンの正しい実装を簡略化するために、JBossSX には **Subject** を強制的に正しく使用するテンプレートパターンを使用して認証された **Subject** を生成するログインモジュールが含まれています。

AbstractServerLoginModule

2 つのうち最も汎用的なクラスは `org.jboss.security.auth.spi.AbstractServerLoginModule` クラスです。

これは `javax.security.auth.spi.LoginModule` インターフェースの実装を提供し、動作環境セキュリティインフラストラクチャに固有の主要なタスクに対し抽象メソッドが可能になります。そのクラスに関する重要な詳細は [例12.14「AbstractServerLoginModule クラスの一部」](#) で強調表示されています。JavaDoc コメントがサブクラスの役割を詳しく説明します。



重要

loginOk インスタンス変数は極めて重要です。ログインが成功する場合は**true**に、そうでない場合はログインメソッドを上書きするサブクラスにより **false** に設定されます。この変数が正しく設定されていないと、コミットメソッドはサブジェクトを正しく更新しません。

ログインフェーズの結果を追跡することで、ログインモジュールは制御フラグとのチェーン化が可能になります。これらの制御フラグは、認証処理の一環としてログインモジュールが成功する必要はありません。

例12.14 AbstractServerLoginModule クラスの一部

```
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

カスタムのログインモジュールに適する2つ目の抽象ベースログインモジュールは **org.jboss.security.auth.spi.UsernamePasswordLoginModule** です。

このログインモジュールは、文字列ベースのユーザー名をユーザーアイデンティティとして強制し、**char[]** パスワードを認証資格情報として強制することで、カスタムのログインモジュール実装をさらに簡略化します。また、ロールを持たないプリンシパルへの匿名ユーザー (null ユーザー名とパスワードで表示) のマッピングにも対応しています。クラスに関する重要な詳細は次のクラスの一部で強調表示されています。JavaDoc コメントがサブクラスの役割を詳しく説明しています。

例12.15 UsernamePasswordLoginModule クラスの一部

```
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;
}

```

ログインモジュールのサブクラス化

作成中のログインモジュールの認証技術に対して文字列ベースのユーザー名と資格情報の使用が可能であるかによって **AbstractServerLoginModule** と **UsernamePasswordLoginModule** のどちらをサ

ブクラス化するか選択します。文字列ベースのセマンティックが有効な場合、サブクラスは **UsernamePasswordLoginModule** となり、有効でない場合は **AbstractServerLoginModule** となります。

サブクラス化の手順

カスタムのログインモジュールが実行しなければならない手順は、選択するベースのログインモジュールクラスにより異なります。セキュリティインフラストラクチャと統合するカスタムのログインモジュールを書く場合、**AbstractServerLoginModule** または **UsernamePasswordLoginModule** をサブクラス化することから始めて、ログインモジュールが認証された **Principal** 情報を JBossSX セキュリティマネージャーが要求する形式で提供するようにします。

AbstractServerLoginModule をサブクラス化する場合は、次を上書きする必要があります。

- **void initialize(Subject, CallbackHandler, Map, Map):** 解析するカスタムのオプションがある場合。
- **boolean login():** 認証アクティビティを実行するためです。ログインに成功する場合、**loginOk** インスタンス変数を **true** に設定するようにします。ログインに失敗する場合は **false** となるようにします。
- **Principal getIdentity():** **log()** のステップで認証されるユーザーの **Principal** オブジェクトを返すためです。
- **Group[] getRoleSets():** **login()** の間に認証される **Principal** に割り当てられるロールが含まれる **Roles** という名前の **Group** を最低でも 1 つ返すためです。2 番目の共通の **Group** は **CallerPrincipal** という名前で、セキュリティドメインアイデンティティではなく、ユーザーのアプリケーションアイデンティティを提供します。

UsernamePasswordLoginModule をサブクラス化する場合は、次を上書きする必要があります。

- **void initialize(Subject, CallbackHandler, Map, Map):** 解析するカスタムのオプションがある場合。
- **Group[] getRoleSets():** **login()** の間に認証される **Principal** に割り当てられるロールが含まれる **Roles** という名前の **Group** を最低でも 1 つ返すためです。2 番目の共通の **Group** は **CallerPrincipal** という名前で、セキュリティドメインアイデンティティではなく、ユーザーのアプリケーションアイデンティティを提供します。
- **String getUsersPassword():** **getUsername()** メソッドで取得できる現在のユーザー名に必要なパスワードを返すためです。**getUsersPassword()** メソッドは **callbackhandler** がユーザー名と候補パスワードを返した後に **login()** 内から呼び出されます。

12.2.2. カスタムの LoginModule の例

UsernamePasswordLoginModule を拡張し、JNDI ルックアップからユーザーのパスワードとロール名を取得するカスタムの Login Module の例を作成するときに、次の情報が役立ちます。

password/<username> (<username> が認証されている現在のユーザーである場合) の形式の名前を使用してコンテキストで検索を実行する場合に、ユーザーのパスワードを返すカスタムの JNDI コンテキストログインモジュールを作成することができるよう本項で説明していきます。同様に、**roles/<username>** の形式の検索は、要求されたユーザーのロールを返します。

例12.16 「JndiUserAndPass カスタムのログインモジュール」は **JndiUserAndPass** カスタムのログインモジュールのソースコードを示しています。

これは JBoss **UsernamePasswordLoginModule** を拡張するため、**JndiUserAndPass** が行うことは JNDI ストアからユーザーのパスワードとロールを取得することだけです。**JndiUserAndPass** は JAAS **LoginModule** 動作には関係しません。

例12.16 JndiUserAndPass カスタムのログインモジュール

```
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));
        }
    }
}

```

JNDI ストアの詳細は **org.jboss.book.security.ex2.service.JndiStore** MBean に記載されています。このサービスは **javax.naming.Context** プロキシを返す **ObjectFactory** を JNDI にバインドします。プロキシは、**password** と **roles** に対して検索名のプレフィックスを確認することで、検索動作を処理します。

名前が **password** で始まる場合は、ユーザーのパスワードが要求されています。名前が **roles** で始まる場合は、ユーザーのロールが要求されています。サンプルの実装はユーザー名に関係なく常に **theduke** のパスワードと **{ "TheDuke", "Echo" }** に等しいロール名の配列を返します。希望であれば他の実装で試すことも可能です。

例のコードにはカスタムのログインモジュールをテストする簡易セッション Bean が含まれています。この例を構築、デプロイ、実行するには、例のディレクトリで次のコマンドを実行します。

```
[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
```

ユーザーのサーバー側の認証に **JndiUserAndPass** カスタムのログインモジュールを使用するという選択は、例のセキュリティドメインのログイン設定により決まります。EJB JAR **META-INF/jboss.xml** 記述子がセキュリティドメインを設定します。

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

SAR **META-INF/login-config.xml** 記述子はログインモジュール設定を定義します。

```
<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>
```

パート III. 暗号化とセキュリティ

第13章 セキュアリモートパスワードプロトコル

Secure Remote Password (SRP) プロトコルは Internet Standards Working Group Request For Comments 2945 (RFC2945) で記述された公開鍵交換のハンドシェイクの実装です。RFC2945 の要約は次のように記載されています。

本書では Secure Remote Password (SRP) プロトコルとして知られる強固な暗号化方式のネットワーク認証メカニズムについて説明します。このメカニズムはユーザーが入力するパスワードを使用したセキュアな接続交渉に適している一方で、再使用可能なパスワードに従来関連付けられたセキュリティ問題を排除します。このシステムは認証プロセスでセキュアな鍵交換も実施し、セキュリティレイヤ (プライバシー / 整合性の保護) をセッション中に有効にします。信頼できる鍵サーバーや証明書インフラストラクチャは必要なく、クライアントは長期間鍵を保存、管理する必要はありません。SRP は既存のチャレンジレスポンス技術に対してセキュリティとデプロイメントの両方が持つ利点を提供し、セキュアなパスワード認証が必要な場合に理想的な完全互換品となります。

RFC2945 の完全な規則は <http://www.rfc-editor.org/rfc.html> を参照してください。SRP アルゴリズムとその歴史に関する追加情報は <http://www-cs-students.stanford.edu/~tjw/srp/> を参照してください。

Diffie-Hellman や *RSA* のようなアルゴリズムは公開鍵交換アルゴリズムとして知られています。公開鍵アルゴリズムの概念は、誰でも見ることができる公開キーと自分しか知らない秘密キーの2つのキーを持っているということです。誰かがあなたに暗号化した情報を送りたい場合、あなたの公開キーを使用してその情報を暗号化します。あなただけが秘密キーを使用してその情報を解読することができます。これを送信者と受信者が共有パスワードを知らなければならない従来の共通パスワードに基づく暗号化スキームと比べてみます。公開鍵アルゴリズムによりパスワードを共有する必要はなくなります。

JBossSX フレームワークには次の要素で構成される SRP 実装が含まれます。

- どのクライアント / サーバープロトコルにも依存しない SRP ハンドシェイクプロトコルの実装
- デフォルトのクライアント / サーバー SRP 実装としてのハンドシェイクプロトコルの RMI 実装
- セキュアな方法でのクライアント認証を目的とする RMI 実装を使用するクライアント側 JAAS **LoginModule** 実装
- RMI サーバー実装を管理する JMX MBean、この MBean により RMI サーバー実装が JMX フレームワークにプラグインでき、検証情報ストアの設定を外部化します。また、JBoss サーバー JNDI 名前空間にバインドされる認証キャッシュを確立します。
- SRP JMX MBean により管理される認証キャッシュを使用するサーバー側 JAAS **LoginModule** 実装

図13.1 「SRP クライアントサーバーフレームワークの JBossSX コンポーネント」では SRP クライアント / サーバーフレームワークの JBossSX 実装に関連する主要コンポーネントについて説明します。

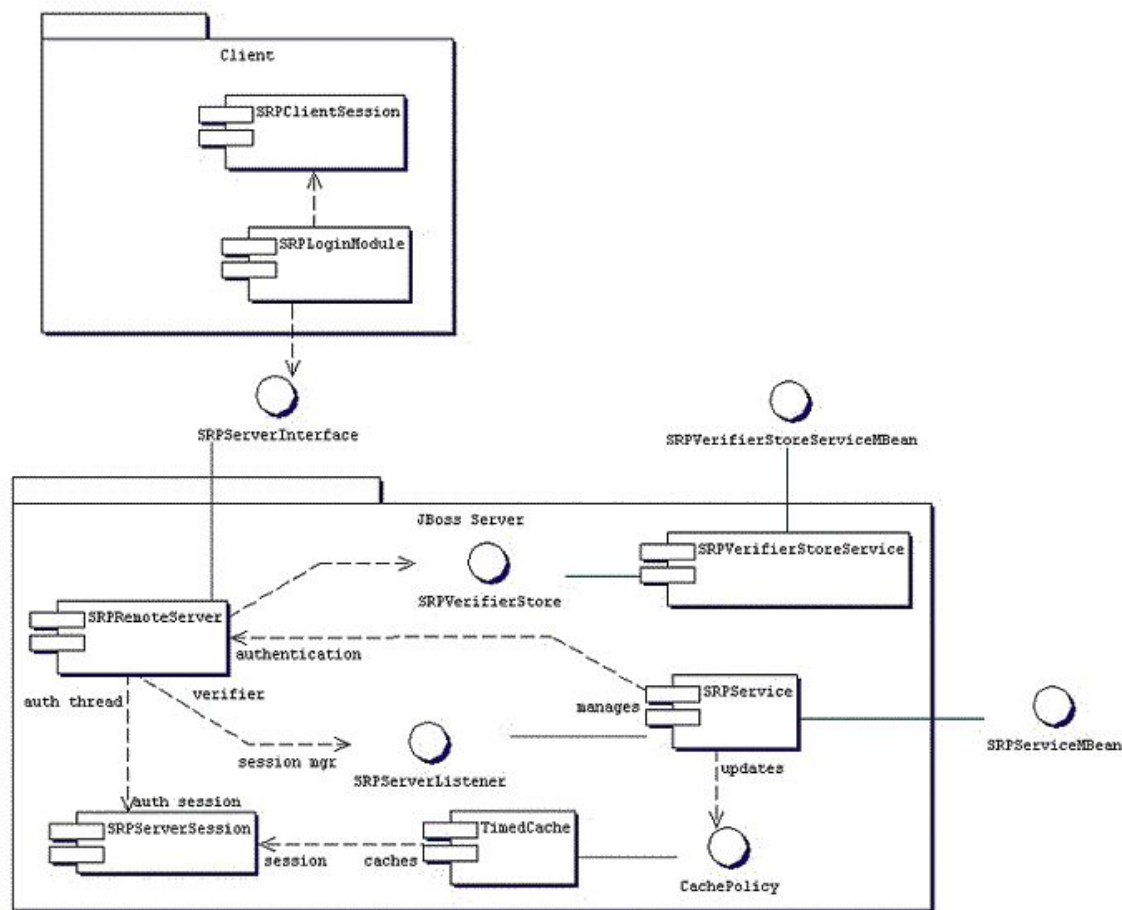


図13.1 SRP クライアントサーバーフレームワークの JBossSX コンポーネント

クライアント側では SRP は **org.jboss.security.srp.SRPServerInterface** プロキシを通じて認証サーバーと通信するカスタムの JAAS **LoginModule** 実装として表されます。クライアントは **org.jboss.security.srp.jaas.SRPLoginModule** を含むログイン設定エントリを作成することで SRP を使用した認証を有効にします。このモジュールは次の設定オプションに対応します。

principalClassName

定数値で、`org.jboss.security.srp.jaas.SRPPrincipal` に設定します。

srpServerIndiName

SRP 認証サーバーと通信するために使用する **SRPServerInterface** オブジェクトの JNDI 名です。**srpServerJndiName** と **srpServerRmiUrl** のオプションが両方とも指定されている場合は、**srpServerJndiName** が **srpServerRmiUrl** より優先されます。

srpServerRmiUrl

SRP 認証サーバーと通信するために使用する **SRPServerInterface** プロキシの場所の RMI プロトコル URL 文字列です。

externalRandomA

クライアントの公開キー「A」のランダムコンポーネントがユーザーのコールバックから来るべきかを指定するフラグです。これを使用してハードウェアトークンからの強固な暗号式の乱数を入力します。**true** に設定するとこの機能が有効になります。

hasAuxChallenge

サーバーが検証する追加チャレンジとして文字列がサーバーに送られるかを指定するフラグです。

クライアントセッションが暗号化方式に対応している場合は、セッション秘密キーと **javax.crypto.SealedObject** として送られたチャレンジオブジェクトを使用して一時的な暗号化方式が作成されます。**true** に設定するとこの機能は有効になります。

multipleSessions

特定のクライアントが複数の SRP ログインセッションをアクティブにするかを指定するフラグです。**true** に設定するとこの機能は有効になります。

上記記載のオプションのいずれにも一致しないその他の渡されたオプションは **InitialContext** コンストラクタに渡された環境が使用する JNDI プロパティとして処理されます。これは SRP サーバーインターフェースがデフォルトの **InitialContext** より使用できない場合に役立ちます。

SRP 認証資格情報をセキュリティ Java EE コンポーネントへのアクセス検証に使用できるように、**SRPLoginModule** と標準 **ClientLoginModule** を設定する必要があります。ログイン設定の例は [例13.1「ログイン設定エントリ」](#) に記載されています。

例13.1 ログイン設定エントリ

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

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

JBoss サーバー側には、SRP サーバーを集团的に構成するオブジェクトを管理する MBean が 2 つあります。主要なサービスは **org.jboss.security.srp.SRPService** MBean です。もう 1 つの MBean は **org.jboss.security.srp.SRPVerifierStoreService** です。

org.jboss.security.srp.SRPService の役割は、SRPServerInterface の RMI アクセス可能なバージョンを公開するだけでなく SRP 認証セッションキャッシュを更新することです。

設定可能な SRPService MBean 属性として以下が挙げられます。

JndiName

SRPServerInterface プロキシが使用可能であるべき場所の名前を指定します。これは **SRPService** がシリアル化可能な動的プロキシを **SRPServerInterface** にバインドする場所です。デフォルト値は **srp/SRPServerInterface** です。

VerifierSourceJndiName

SRPService が使用する必要がある **SRPVerifierSource** 実装の名前を指定します。ソースの JNDI 名は **srp/DefaultVerifierSource** にデフォルト設定されます。

AuthenticationCacheJndiName

認証情報をキャッシュするために使用する **org.jboss.util.CachePolicy** 認証実装がバインドされる名前を指定します。SRP セッションキャッシュはこのバインディングを通じて使用可能になります。認証 JNDI キャッシュは **srp/AuthenticationCache** にデフォルト設定されます。

ServerPort

SRPRemoteServerInterface の RMI ポートです。デフォルト値は 10099 です。

ClientSocketFactory

SRPServerInterface のエクスポート中に使用されるオプションのカスタム **java.rmi.server.RMIClientSocketFactory** 実装クラス名です。デフォルト値は **RMIClientSocketFactory** です。

ServerSocketFactory

SRPServerInterface のエクスポート中に使用されるオプションのカスタム **java.rmi.server.RMIServerSocketFactory** 実装クラス名です。デフォルト値は **RMIServerSocketFactory** です。

AuthenticationCacheTimeout

キャッシュポリシーのタイムアウト (秒単位) です。デフォルト値は 1800秒 (30 分) です。

AuthenticationCacheResolution

時間制限されたキャッシュポリシーのリゾリューションを指定します (秒単位)。これはタイムアウトのチェック間隔を制御します。デフォルト値は 60秒 (1 分) です。

RequireAuxChallenge

クライアントが検証フェーズの一部として補助チャレンジを与える必要があるか設定します。これでクライアントが使用する **SRPLoginModule** 設定によって **useAuxChallenge** オプションを有効にする必要があるか制御できます。

OverwriteSessions

既存セッションに対して成功するユーザー認証が現在のセッションを上書きすべきか指定します。クライアントが複数セッションをユーザーモードごとに有効にしていない場合は、サーバー SRP セッションキャッシュの動作を制御します。**false** に設定すると、2 番目のユーザー認証の試行は成功します。ただし、それによって生じる SRP セッションは前の SRP セッション状態を上書きしません。デフォルト値は **false** です。

VerifierStoreJndiName

JNDI を通じて提供され使用可能となる必要がある SRP パスワード情報ストア実装の場所を指定します。

org.jboss.security.srp.SRPVerifierStoreService は、固定ストアとしてシリアル化されたオブジェクトのファイルを使用する **SRPVerifierStore** インターフェースの実装をバインドする MBean サービスの一例です。実稼働環境には現実的ではありませんが、SRP プロトコルのテストが可能で、**SRPVerifierStore** サービスに要件の一例を提供します。

設定可能な **SRPVerifierStoreService** MBean 属性として以下が挙げられます。

JndiName

SRPVerifierStore 実装が使用可能であるべき場所の JNDI 名です。指定されていない場合は、**srp/DefaultVerifierSource** にデフォルト設定されます。

StoreFile

ユーザーパスワードベリファイアのシリアル化されたオブジェクトストアファイルの場所です。これはクラスパスにある URL またはリソース名のいずれかになります。指定されていない場合は、**SRPVerifierStore.ser** にデフォルト設定されます。

SRPVerifierStoreService MBean はユーザーの追加および削除に対する **addUser** と **delUser** 動作にも対応します。シグネチャは以下のとおりです。

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

こうしたサービスの設定例は 例13.2 「**SRPVerifierStore インターフェース**」 に記載されています。

13.1. アルゴリズムの理解

SRP アルゴリズムの利点は、セキュアな通信チャンネルがなくてもシンプルなテキストパスワードを使用してクライアントとサーバーの相互認証が可能であることです。



注記

SRP アルゴリズムとその歴史に関する追加情報は <http://srp.stanford.edu/> に記載されています。

認証を完了するためには 6 つのステップを実行します。

1. クライアント側 **SRPLoginModule** がネーミングサービスからリモート認証サーバーの **SRPServerInterface** インスタンスを取得します。
2. 次に、クライアント側 **SRPLoginModule** はログインを試行しているユーザー名に関連付けられた SRP パラメータを要求します。ユーザーパスワードが最初に SRP アルゴリズムを使用した検証形式に変換されるときに選択される必要がある SRP アルゴリズムに含まれるパラメータは多くあります。パラメータをハードコードするのではなく (最小限のセキュリティリスクで実行可能)、JBossSX 実装によりユーザーは交換プロトコルの一部としてこの情報を取得できます。**getSRPParameters(username)** 呼び出しは特定のユーザー名の SRP パラメータを取得します。
3. クライアント側 **SRPLoginModule** はログインユーザー名、クリアテキストのパスワード、ステップ 2 で取得した SRP パラメータを使用して **SRPClientSession** オブジェクトを作成することで SRP セッションを開始します。次に、クライアントは秘密 SRP セッションキーを構築するために使用される乱数を作成します。そして **SRPServerInterface.init** メソッドを呼び出すことで SRP セッションのサーバー側を初期化し、ユーザー名とクライアントが生成した乱数 **A** を渡します。サーバーは独自の乱数 **B** を返します。このステップは公開キーの交換に相当します。
4. クライアント側 **SRPLoginModule** は前のメッセージ交換の結果として生成された秘密 SRP セッションキーを取得します。これはログイン **Subject** で秘密の資格情報として保存されます。ステップ 4 でのサーバーのチャレンジレスポンス **M2** は **SRPClientSession.verify** メソッドを呼び出すことで検証されます。これが成功すると、クライアントからサーバー、およびサーバーからクライアントへの相互認証が完了します。次に、クライアント側 **SRPLoginModule** はサーバー乱数 **B** を引数として渡している **SRPClientSession.response** メソッドを呼び出すことでサーバーへのチャレンジ **M1** を作

成します。このチャレンジは **SRPServerInterface.verify** メソッドによりサーバーに送られ、サーバーのレスポンスは **M2** として保存されます。このステップはチャレンジの交換に相当します。この時点で、サーバーはユーザーがそのユーザー本人であることを確かめます。

5. クライアント側 **SRPLoginModule** はログインユーザー名と **M1** チャレンジを **LoginModule** `sharedState` マップに保存します。これが標準 JBoss **ClientLoginModule** により **Principal** の名前と資格情報として使用されます。**M1** チャレンジは、Java EE コンポーネントのすべてのメソッド呼び出しでアイデンティティの証明としてパスワードの代わりに使用されます。**M1** チャレンジは、SRP セッションと関連付けられた強固な暗号化方式のハッシュです。第3者による傍受はユーザーのパスワードを取得するためには使用できません。
6. この認証プロトコルの最後で、**SRPServiceSession** は **SRPCacheLoginModule** によって後続して使用できるよう **SRPService** 認証キャッシュに配置されました。

SRP には多くの興味深いプロパティがありますが、今もなお JBossSX フレームワークの発展中のコンポーネントであり、認識しておくべき制約が数点あります。注意すべき点は以下のとおりです。

- 認証が実行されたところで、JBoss がメソッドのトランスポートプロトコルをコンポーネントコンテナから分離する方法によって、ユーザーは SRP **M1** チャレンジをスヌープして、効果的にチャレンジを使用して関連付けられたユーザー名として要求を行うことが可能になります。SRP セッションキーを使用してチャレンジを暗号化することにより、カスタムのインターセプタを使用してこの問題を防止できます。
- **SRPService** は設定可能な期間が過ぎた後にタイムアウトする SRP セッションのキャッシュを管理します。タイムアウトした時点で、SRP 認証資格情報を透過的に再交渉するメカニズムは現在ないため、Java EE コンポーネントの後続アクセスはすべて失敗します。認証キャッシュのタイムアウトをかなり長く設定するか、失敗したらコードで再認証を行う必要があります。



注記

SRPService は最大で 2,147,483,647 秒、または約 68 年間のタイムアウトに対応します。

- デフォルトでは特定のユーザー名に対して SRP セッションは 1 つだけです。交渉される SRP セッションはクライアントとサーバー間の暗号化と復号化に使用できる秘密セッションキーを生成するため、セッションはステートフルとして扱われます。JBoss はユーザーごとに複数の SRP セッションをサポートしますが、1 つのセッションキーでデータを暗号化し、別のセッションキーで復号化することはできません。

Java EE コンポーネント呼び出しにエンドツーエンド SRP 認証を使用するには、コンポーネントが **org.jboss.security.srp.jaas.SRPCacheLoginModule** を使用するのにセキュアであるセキュリティドメインを設定する必要があります。**SRPCacheLoginModule** には SRP 認証 **CachePolicy** インスタンスの JNDI 場所を設定する **cacheJndiName** という名前の単一の設定オプションがあります。これは **SRPService** MBean の **AuthenticationCacheJndiName** 属性値と対応しなければなりません。

SRPCacheLoginModule は、認証キャッシュの **SRPServiceSession** オブジェクトからクライアントチャレンジを取得し、これをユーザーの資格情報として渡されるチャレンジと比較することでユーザーの資格情報を認証します。[図13.2「SRP セッションキャッシュを使用した SRPCacheLoginModule」](#)では **SRPCacheLoginModule.login** メソッド実装の動作を図解しています。

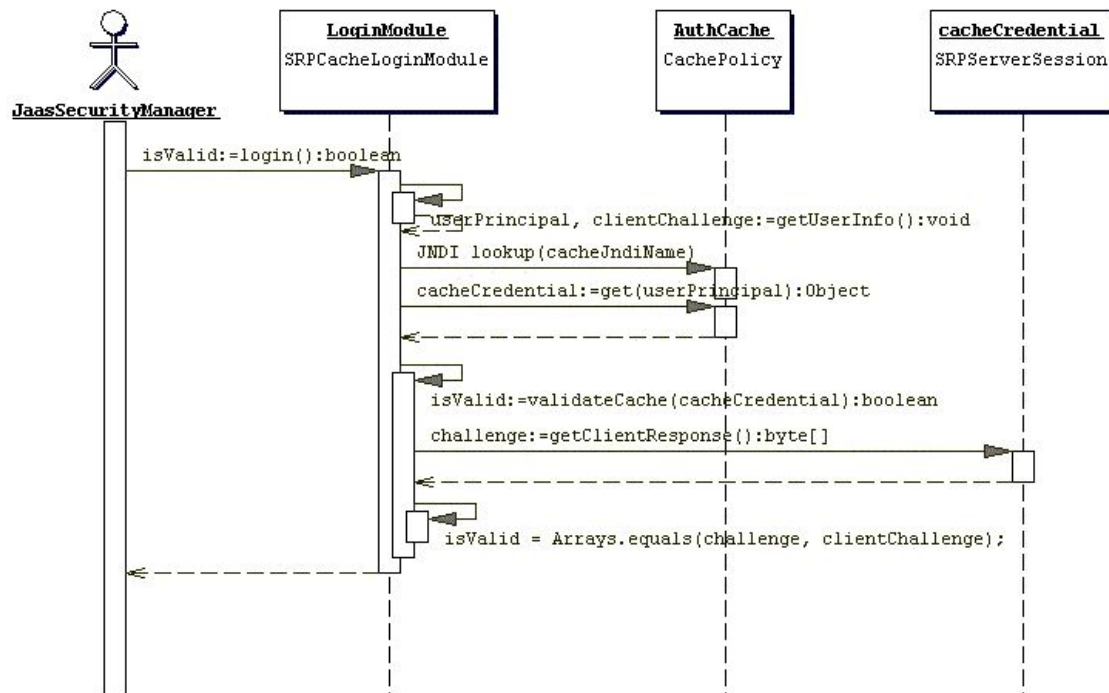


図13.2 SRP セッションキャッシュを使用した SRPCacheLoginModule

13.2. セキュアリモートパスワード情報の設定

既存のセキュリティ情報ストアと統合する **SRPVerifierStore** インターフェースの実装を提供する MBean サービスを作成する必要があります。**SRPVerifierStore** インターフェースは [例 13.2 「SRPVerifierStore インターフェース」](#) に表示されています。



注記

SRPVerifierStore インターフェースのデフォルト実装は、セキュリティの実稼働環境には推奨されません。すべてのパスワードハッシュ情報がシリアル化されたオブジェクトのファイルとして使用可能である必要があるためです。

例13.2 SRPVerifierStore インターフェース

```

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;
}

```

SRPVerifierStore 実装の主要機能は、**SRPVerifierStore.VerifierInfo** オブジェクトへのアクセスを特定のユーザー名に提供することです。**getUserVerifier(String)** メソッドは SRP アルゴリズムが必要なパラメータを取得するためユーザーの SRP セッションの開始時に **SRPService** より呼び出されます。**VerifierInfo** オブジェクトの要素は以下のとおりです。

user name

ログインに使用するユーザー名または ID です。

verifier

アイデンティティの証拠としてユーザーが入力するパスワードの一方方向ハッシュまたは PIN です。**org.jboss.security.Util** クラスにはそのパスワードハッシュアルゴリズムを実行する **calculateVerifier** メソッドがあります。出力パスワードは **H(salt | H(username | ':' | password))** の形式を取り、**H** は RFC2945 により定義されたとおりのセキュアハッシュ関数 SHA です。ユーザー名は UTF-8 エンコードを使用して文字列から **byte[]** に変換されます。

salt

データベースが危害を受けた場合に、データベースパスワードベリファイアにブルートフォース、辞書攻撃されにくくするために使用される乱数です。ユーザーの既存のクリアテキストのパスワードがハッシュされたときに、強固な暗号化形式の乱数アルゴリズムにより値が生成されます。

g

SRP アルゴリズムプリミティブジェネレータです。これはユーザーごとの設定ではなく、既知の固定パラメータとなります。**org.jboss.security.srp.SRPConf** ユーティリティクラスは、**SRPConf.getDefaultParams().g()** により取得された適切なデフォルトなど **g** に対して設定を複数提供します。

N

SRP アルゴリズムセーフプライムモジュールです。これはユーザーごとの設定ではなく、既知の固定パラメータとなります。**org.jboss.security.srp.SRPConf** ユーティリティクラスは **SRPConf.getDefaultParams().N()** により取得できる適切なデフォルトなどの **N** に対して設定を複数提供します。

手順13.1 既存のパスワードストアを統合する

既存のパスワードストアを統合するためのステップを理解するためにはこの手順をお読みください。

1. ハッシュされたパスワード情報ストアを作成します

パスワードがすでに復元できないハッシュされた形式で保存されている場合は、これはユーザーごとにしか行えません (例えば、アップグレード手順の一環として行います)。

noOp メソッドまたはストアが読み取り専用であることを示す例外を投げるメソッドとして **setUserVerifier(String, VerifierInfo)** を実装できます。

2. SRPVerifierStore インターフェースを作成します

作成したストアから **VerifierInfo** を取得する方法を理解するカスタムの **SRPVerifierStore** インターフェース実装を作成する必要があります。

verifyUserChallenge(String, Object) を使用して、SafeWord または Radius のような既存のハードウェアトークンに基づいたスキームを SRP アルゴリズムに統合できます。このインターフェースメソッドは、クライアント **SRPLoginModule** 設定が **hasAuxChallenge** オプションを指定する場合にのみ呼び出されます。

3. JNDI MBean を作成します

JNDI に使用可能となる **SRPVerifierStore** インターフェースを公開し、必要な設定可能なパラメータを公開する MBean を作成する必要があります。

デフォルトの **org.jboss.security.srp.SRPVerifierStoreService** によりこれを実装できますが、**SRPVerifierStore** ([「セキュアリモートパスワード \(SRP\) の例」](#) を参照) の Java プロパティファイル実装を使用して MBean を実装することもできます。

13.3. セキュアリモートパスワード (SRP) の例

本項で示す例では SRP によるユーザーのクライアント側認証だけでなく、ユーザーの資格情報として SRP セッションチャレンジを使用したシンプルな EJB へのセキュアな後続のアクセスについても説明しています。テストコードは、サーバー側のログインモジュール設定と SRP サービスの設定に対する SAR を含む EJB JAR をデプロイします。

サーバー側のログインモジュール設定は、**SecurityConfig** MBean を使用して動的にインストールされます。この例では **SRPVerifierStore** インターフェースのカスタム実装も使用されています。インターフェースは **SRPVerifierStoreService** により使用されるようなシリアル化されたオブジェクトストアではなく、Java プロパティファイルからシードされるインメモリストアを使用します。

このカスタムサービスは

org.jboss.book.security.ex3.service.PropertiesVerifierStore です。サンプルの EJB と SRP サービスを含む JAR のコンテンツを次に示します。

```
[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
```

この例の重要な SRP 関連項目は SRP MBean サービス設定と SRP ログインモジュール設定です。**security-ex3.sar** の **jboss-service.xml** 記述子は [例13.3「security-ex3.sar jboss-service.xml 記述子」](#) で説明します。

例のクライアント側およびサーバー側のログインモジュール設定は [例13.4「クライアント側標準 JAAS 設定」](#) と [例13.5「サーバー側 XMLLoginConfig 設定」](#) で説明します。

例13.3 security-ex3.sar jboss-service.xml 記述子

```
<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/SRPServiceInterface</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>
```

例のサービスは **ServiceConfig**、**PropertiesVerifierStore** MBean、**SRPService** MBean です。**PropertiesVerifierStore** の **JndiName** 属性は **SRPService** の **VerifierSourceJndiName** 属性と等しく、**SRPService** は **PropertiesVerifierStore** に依存していることに注意してください。これが必須な理由は、ユーザーのパスワード検証情報にアクセスするためには **SRPService** に **SRPVerifierStore** インターフェースの実装が必要なためです。

例13.4 クライアント側標準 JAAS 設定

```

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

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

```

クライアント側ログインモジュール設定では JBoss サーバーコンポーネントの **SRPService** JndiName 属性値 (**srp-test/SRPServiceInterface**) に対応する **srpServerJndiName** オプション値を持つ **SRPLoginModule** を使用します。また **SRPLoginModule** により生成されたユーザー認証資格情報を EJB 呼び出しレイヤに伝播するために、**ClientLoginModule** は **password-stacking="useFirstPass"** 値で設定する必要があります。

例13.5 サーバー側 XMLLoginConfig 設定

```

<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>

```

サーバー側のログインモジュール設定について注意すべき点が2つあります。

1. **cacheJndiName=srp-test/AuthenticationCache** 設定オプションは、**SRPService** に対して認証したユーザーの **SRPServiceSession** を含む **CachePolicy** の場所を **SRPCacheLoginModule** に伝えます。この値は **SRPServiceAuthenticationCacheJndiName** 属性値に相当します。
2. 設定には **password-stacking=useFirstPass** 設定オプションを持つ **UsersRolesLoginModule** が含まれます。SRP は認証技術でしかないので、**SRPCacheLoginModule** を持つ第2のログインモジュールを使用する必要があります。関連付けられたパーミッションを決定するプリンシパルのロールを設定するためには、第2のログインモジュールは **SRPCacheLoginModule** により検証された認証資格情報を受け入れるように設定する必要があります。

UsersRolesLoginModule はプロパティファイルベースの承認で SRP 認証を補強しています。ユーザーのロールは EJB JAR に含まれる **roles.properties** ファイルから取得されます。

ブックサンプルのディレクトリから次のコマンドを実行して例の 3 クライアントを実行します。

```
[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
```

examples/logs ディレクトリの **ex3-trace.log** ファイルには、SRP アルゴリズムのクライアント側の詳細トレースが含まれています。トレースは公開キー、チャレンジ、セッションキー、検証の構成を順を追って示しています。

他のシンプルな例と比べて、クライアントは実行するのに長時間かかります。その理由はクライアントの公開キーの構成にあります。これには強固な暗号化形式の乱数の作成が含まれ、このプロセスを初めて実行する場合はより長くかかります。同じ VM 内での後続の認証試行ははるかに速くなります。

Echo.echo()#2 は認証例外で失敗することに注意してください。クライアントコードは **SRPService** キャッシュの有効期限の動作を示すための最初の呼び出し後 15 秒間はスリープします。**SRPService** キャッシュポリシーのタイムアウトはこれを強制するため 10 秒に設定されています。「[セキュアリモートパスワード \(SRP\) の例](#)」で述べたとおり、キャッシュタイムアウトは正しく設定しなければなりません。そうしないと、失敗時に再認証を行う必要があります。

第14章 JAVA セキュリティマネージャー

Java セキュリティマネージャー

Java セキュリティマネージャー は Java Virtual Machine (JVM) サンドボックスの外部の境界を管理するクラスで、JVM 内で実行しているコードと JVM 外のリソースとの連携方法を制御します。Java セキュリティマネージャーがアクティブになると、Java API は潜在的に安全でない多様な動作を実行する前にセキュリティマネージャーと承認を確認します。

セキュリティマネージャーはセキュリティポリシーを使用して、特定のアクションが許可または拒否されるか決定します。

セキュリティポリシー

コードの様々なクラスに対して定義されたパーミッションのセットです。Java セキュリティマネージャーはセキュリティポリシーとアプリケーションから要求されたアクションを比較します。ポリシーがアクションを許可している場合は、セキュリティマネージャーはそのアクションが行われることを許可します。ポリシーがアクションを許可していない場合は、セキュリティマネージャーはそのアクションを拒否します。セキュリティポリシーはコードの場所またはコードのシグネチャに基づきパーミッションを定義することができます。

使用するセキュリティマネージャーとセキュリティポリシーは Java Virtual Machine オプション `java.security.manager` と `java.security.policy` を使用して設定されます。

セキュリティマネージャー関連のオプション

`java.security.manager`

セキュリティマネージャーを使用し、使用するセキュリティマネージャーをオプションで指定します。引数がこのオプションにない場合は、デフォルトの JDK セキュリティマネージャー `java.lang.SecurityManager` が使用されます。別のセキュリティマネージャー実装を使用するには、`java.lang.SecurityManager` のサブクラスの完全修飾クラス名にこのオプションを提供します。

`java.security.policy`

ポリシーファイルを指定して、VM に対するデフォルトのセキュリティポリシーを補強または置換します。このオプションは 2 つの形式を取ります。

`java.security.policy=policyFileURL`

`policyFileURL` により参照されたポリシーファイルは、VM により設定されたデフォルトのセキュリティポリシーを **補強** します。

`java.security.policy==policyFileURL`

`policyFileURL` により参照されたポリシーファイルは、VM により設定されたデフォルトのセキュリティポリシーを **置換** します。

`policyFileURL` 値は URL またはファイルパスとなります。

JBoss Enterprise Application Platform はデフォルトでは Java セキュリティマネージャーをアクティブにしません。セキュリティマネージャーを使用するために Platform を設定するには、[「セキュリティマネージャーの使用」](#) を参照してください。

14.1. セキュリティマネージャーの使用

JBoss Enterprise Application Platform は デフォルトの JDK セキュリティマネージャーまたはカスタムのセキュリティマネージャーを使用できます。カスタムのセキュリティマネージャーの選択に関する詳細は [セキュリティマネージャー関連のオプション](#) を参照してください。

セキュリティマネージャーを使用するように Platform を設定する場合は、セキュリティポリシーファイルを指定する必要があります。セキュリティポリシーファイル **jboss-as/bin/server.policy.cert** は開始点として含まれています。

セキュリティポリシーを書くにあたっての情報は「[JBoss Enterprise Application Platform 向けのセキュリティポリシーを書く](#)」を参照してください。

設定ファイル

ファイル **run.conf** (Linux) または **run.conf.bat** (Windows) を使用して、Security Manager とセキュリティポリシーを設定します。このファイルは **jboss-as/bin** ディレクトリにあります。

このファイルはサーバーレベルのオプションを設定するために使用され、すべてのサーバープロファイルに適用します。セキュリティマネージャーとセキュリティポリシーの設定には、プロファイル固有の設定が関係しています。グローバルな **run.conf** または **run.conf.bat** ファイルを **jboss-as/bin/** からサーバープロファイル (例えば **jboss-as/server/production/run.conf**) にコピーすることを選択でき、そこで設定変更が可能です。サーバープロファイルが開始された場合、サーバープロファイルの設定ファイルはグローバルな **run.conf** / **run.conf.bat** ファイルより優先されます。

手順14.1 セキュリティマネージャーをアクティブにする

これは JBoss Enterprise Application Platform を設定して、Java セキュリティマネージャーをアクティブにして開始する手順を示しています。

この手順のファイル編集のアクションは、あればサーバープロファイルディレクトリにあるファイル **run.conf** (Linux) または **run.conf.bat** (Windows)、なければ **jboss-as/bin** を参照します。このファイルの場所の詳細は [設定ファイル](#) を参照してください。

1. JBoss ホームディレクトリを指定します

ファイル **run.conf** (Linux) または **run.conf.bat** (Windows) を編集します。 **jboss.home.dir** オプションを追加し、インストール環境の **jboss-as** ディレクトリへのパスを指定します。

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. サーバーホームディレクトリを指定します

jboss.server.home.dir オプションを追加し、サーバープロファイルへのパスを指定します。

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. Protocol Handler を指定します

java.protocol.handler.pkgs オプションを追加し、JBoss スタブハンドラを指定します。

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. 使用するセキュリティポリシーを指定します

\$POLICY 変数を追加し、使用するセキュリティポリシーを指定します。セキュリティマネージャーをアクティブにする行の前に変数定義を追加します。

例14.1 Platform に含まれるセキュリティポリシーの使用

```
POLICY="server.policy.cert"
```

5. セキュリティマネージャーをアクティブにする

最初の **#** を削除して、次の行を非コメントします。

Linux

```
#JAVA_OPTS="$JAVA_OPTS -Djava.security.manager -Djava.security.policy=$POLICY"
```

Windows

```
#JAVA_OPTS="%JAVA_OPTS% -Djava.security.manager -Djava.security.policy=%POLICY%"
```

結果

これで JBoss Enterprise Application Platform はセキュリティマネージャーをアクティブにして開始できるよう設定できました。

6. オプション : Red Hat の JBoss 署名キーのインポート

含まれているセキュリティポリシーは JBoss 署名コードにパーミッションを付与します。含まれているポリシーを使用する場合は、JBoss 署名キーを JDK **cacerts** キーストアにインポートする必要があります。

次のコマンドでは、環境変数 **JAVA_HOME** が JBoss Enterprise Application Platform 5 でサポートされている JDK の場所に設定されていると仮定します。最初に JBoss Enterprise Application Platform 5 をインストールするときに **JAVA_HOME** を設定します。詳細は『Installation Guide』を参照してください。



注記

正しい JVM を選択するには、**alternatives** コマンドを使用して、ご使用の Linux システムにインストールされている JDK から選択することができます。[付録A /usr/sbin/alternatives ユーティリティによるデフォルト JDK の設定](#)を参照してください。

ターミナルで次のコマンドを実行し、**JAVA_HOME** を Java インストールのディレクトリがある場所に置き換えます。

Linux

```
[~]$ sudo JAVA_HOME/bin/keytool -import -alias jboss -file
JBossPublicKey.RSA \
-keystore JAVA_HOME/lib/security/cacerts
```

Windows

```
C:> JAVA_HOME\bin\keytool -import -alias jboss -file
JBossPublicKey.RSA -keystore JAVA_HOME\lib\security\cacerts
```

本書では上記のコマンドは 2 行に渡って書かれていますが、ターミナルでは 1 行で入力します。



注記

cacerts キーストアのデフォルトパスワードは **changeit** です。

結果

これで JBoss Enterprise Application Platform コードを署名するために使用するキーがインストールされました。

14.2. セキュリティポリシーの問題点のデバッグ

セキュリティポリシー関連の課題を解決するのに役立つデバッグ情報を有効にできます。 **java.security.debug** オプションは報告されるセキュリティ関連情報のレベルを設定します。

コマンド **java -Djava.security.debug=help** は一連のデバックオプションでヘルプ出力を生成します。原因が全く不明なセキュリティ関連のエラーを解決する場合にはデバッグレベルを **all** に設定すると便利ですが、一般使用では過度の情報を生成することになります。適切な一般的なデフォルトは **access:failure** です。

手順14.2 一般的なデバッグを有効にする

これはセキュリティ関連のデバッグ情報の適切な一般的なレベルを有効にする手順を示しています。

- 次の行をファイル **run.conf** (Linux) または **run.conf.bat** (Windows) に追加します。

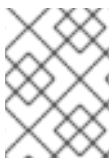
Linux

```
JAVA_OPTS="$JAVA_OPTS -Djava.security.debug=access:failure"
```

Windows

```
JAVA_OPTS="%JAVA_OPTS% -Djava.security.debug=access:failure"
```

14.2.1. デバッグセキュリティマネージャー



注記

デバッグセキュリティマネージャーが JBoss Enterprise Application Platform 5.1 に導入されました。

デバッグセキュリティマネージャー

org.jboss.system.security.DebuggingJavaSecurityManager が失敗しているパーミッションに対応する保護ドメインを表示します。この追加情報はパーミッションの問題をデバッグするときに非常に役立ちます。

手順14.3 デバッグセキュリティマネージャーを有効にする

これはデバッグセキュリティマネージャーを有効にする手順です。

1. 次のオプションを **\$JBOSS_HOME/bin/run.conf** (Linux) または **\$JBOSS_HOME/bin/run.conf.bat** に追加します。このファイルの場所は [設定ファイル](#) を参照してください。

Linux

```
JAVA_OPTS="$JAVA_OPTS -
Djava.security.manager=org.jboss.system.security.DebuggingJavaSecurityManager"
```

Windows

```
JAVA_OPTS="%JAVA_OPTS% -
Djava.security.manager=org.jboss.system.security.DebuggingJavaSecurityManager"
```

2. ファイル内の他のすべての **java.security.manager** 参照をコメントアウトします。
3. 使用するポリシーファイルを指定する **java.security.policy** オプションがなおファイルに含まれていることを確認します。

4. 手順14.2「一般的なデバッグを有効にする」の手順に従って一般的なデバッグを有効にします。



注記

デバッグセキュリティマネージャーは非常にパフォーマンスコストが高いため、一般的な実稼働環境では使用しないでください。

14.3. JBOSS ENTERPRISE APPLICATION PLATFORM 向けのセキュリティポリシーを書く

含まれているファイル `jboss-as/bin/server.policy.cert` は JBoss Enterprise Application Platform 向けのセキュリティポリシーの一例です。このファイルをご自身のセキュリティポリシーの基礎として使用できます。

`policytool` アプリケーションは JDK に含まれており、セキュリティポリシーを編集、書くためのグラフィカルツールを提供します。



重要

どのパーミッションを付与するか注意深く検討してください。`java.security.AllPermission` を付与するにあたっては特にご注意ください。JVM ランタイム環境などのシステムバイナリへの変更を許可してしまう可能性があります。

セキュリティポリシーファイルと Java パーミッションの一般的な取扱いについては <http://download-l1nw.oracle.com/javase/6/docs/technotes/guides/security/PolicyFiles.html> の公式 Java ドキュメントを参照してください。JBoss 固有の `java.lang.RuntimePermissions` は以下に記載されています。

JBoss 固有のランタイム権限

`org.jboss.security.SecurityAssociation.getPrincipalInfo`

`org.jboss.security.SecurityAssociation getPrincipal()` と `getCredential()` メソッドにアクセスを提供します。このランタイム権限を使用する場合のリスクは、現在のスレッド呼び出し側と資格情報を見ることができることです。

`org.jboss.security.SecurityAssociation.getSubject`

`org.jboss.security.SecurityAssociation getSubject()` メソッドにアクセスを提供します。

`org.jboss.security.SecurityAssociation.setPrincipalInfo`

`org.jboss.security.SecurityAssociation setPrincipal()`、`setCredential()`、`setSubject()`、`pushSubjectContext()` および `popSubjectContext()` メソッドにアクセスを提供します。このランタイム権限を使用する場合のリスクは、現在のスレッド呼び出し側と資格情報を設定できることです。

`org.jboss.security.SecurityAssociation.setServer`

`org.jboss.security.SecurityAssociation setServer` メソッドにアクセスを提供します。このランタイム権限を使用する場合のリスクは、呼び出し側のプリンシパルと資格情報の複数スレッドのストレージを有効または無効にできることです。

org.jboss.security.SecurityAssociation.setRunAsRole**org.jboss.security.SecurityAssociation**

pushRunAsRole、**popRunAsRole**、**pushRunAsIdentity** および **popRunAsIdentity** メソッドにアクセスを提供します。このランタイム権限を使用する場合のリスクは、現在の呼び出し側の run-as role プリンシパルを変更できることです。

org.jboss.security.SecurityAssociation.accessContextInfo

org.jboss.security.SecurityAssociation accessContextInfo, **"Get"** と **accessContextInfo**, **"Set"** メソッドにアクセスを提供し、これにより現在のセキュリティコンテキスト情報を設定、取得できます。

org.jboss.naming.JndiPermission

指定された JNDI ツリーパスのファイルとディレクトリ、または再帰的にすべてのファイルとサブディレクトリに特別なパーミッションを提供します。JndiPermission はファイルまたはディレクトリに関連するパス名と有効なパーミッションのセットで構成されます。

使用可能なパーミッションは

bind、**rebind**、**unbind**、**lookup**、**list**、**listBindings**、**createSubcontext** および **all** です。

/* で終わるパス名は、特定のパーミッションがパス名のすべてのファイルとディレクトリに適用することを意味しています。**/-** で終わるパス名は、パス名のすべてのファイルとサブディレクトリへの再帰的なパーミッションを意味しています。特別なトークン **<<ALL BINDINGS>>** で構成されるパス名は、すべてのディレクトリのすべてのファイルと一致します。

org.jboss.security.srp.SRPPermission

秘密セッションキーや秘密キーのような機密 SRP 情報へのアクセスを保護するカスタムのパーミッションクラスです。このパーミッションではアクションを定義しません。getSessionKey ターゲットは SRP 交渉の結果得られる秘密セッションキーへのアクセスを提供します。このキーへのアクセスにより、セッションキーで暗号化したメッセージを暗号化、復号化できます。

org.hibernate.secure.HibernatePermission

このパーミッションクラスは基本的なパーミッションを提供し、Hibernate セッションをセキュアにします。このプロパティのターゲットはエンティティ名です。使用可能なアクションには insert、delete、update、read および * (すべて) があります。

org.jboss.metadata.spi.stack.MetaDataStackPermission

カスタムのパーミッションクラスを提供し、呼び出し側がメタデータスタックと通信する方法を制御します。使用可能なパーミッションは **modify** (スタックにプッシュ / ポップする)、**peek** (スタックにピークする)、および * (すべて) があります。

org.jboss.config.spi.ConfigurationPermission

構成プロパティの設定をセキュアにします。アクションではなく、パーミッションのターゲット名だけを定義します。このプロパティのターゲットには、設定するパーミッションを持つコードのプロパティを示す <property name>、すべてのプロパティを示す * があります。

org.jboss.kernel.KernelPermission

カーネル設定へのアクセスをセキュアにします。アクションではなく、パーミッションのターゲット名だけを定義します。このプロパティのターゲットには、カーネル設定へのアクセスを示す access、カーネルの設定を示す (アクセスは暗黙的に行われます) configure、上記のすべてを示す *

があります。

org.jboss.kernel.plugins.util.KernelLocatorPermission

カーネルへのアクセスをセキュアにします。アクションではなく、パーミッションのターゲット名だけを定義します。このプロパティのターゲットには、カーネルへのアクセスを示す `kernel`、すべてのエリアへのアクセスを示す `*` があります。

第15章 EJB RMI トランスポート層をセキュアにする

JBoss Application Server は EJB2 および EJB3 Bean の Remote Method Invocation (リモートメソッド呼び出し : RMI) に対してソケットベースの invoker レイヤを使用します。このネットワークトラフィックはデフォルトでは暗号化されていません。本章の手順に従って、Secure Sockets Layer (セキュアソケットレイヤ : SSL) を使用してこのネットワークトラフィックを暗号化します。

SSL を使用した EJB のトランスポートオプション

本章では暗号化したトランスポートに対する EJB3 のリモートメソッド呼び出しの 2 つの異なる設定、RMI + SSL および HTTPS による RMI の設定について説明します。HTTPS はファイアウォール設定により RMI ポートを使用できない場合の RMI のトランスポートとしてのオプションです。

SSL のキーペアの生成については「[暗号化キーと証明書の生成](#)」で取り上げます。

EJB3 の RMI + SSL の設定については「[EJB3 RMI + SSL 設定](#)」で取り上げます。

EJB3 の HTTPS による RMI の設定については「[HTTPS 設定による EJB3 RMI](#)」で取り上げます。

EJB2 の RMI + SLL の設定については「[EJB2 RMI + SSL 設定](#)」で取り上げます。

15.1. SSL 暗号化の概要

15.1.1. キーペアと証明書

Secure Sockets Layer (SSL) は 2 つのシステム間のネットワークトラフィックを暗号化します。この 2 つのシステム間のトラフィックは、双方向キーを使用して暗号化され、接続の ハンドシェイクフェーズ中に生成され、これら 2 つのシステムにのみ知られます。

双方向の暗号化キーを安全に交換するために、SSL は Public Key Infrastructure (公開鍵基盤 : PKI) という キーペアを活用する暗号化メソッドを使用します。キーペアは公開キーと秘密キーという別々ですが対応する 2 つの暗号化キーで構成されています。公開キーは他人と共有し、データの暗号化に使用します。秘密キーは秘密にされ、公開キーを使用して暗号化されたデータの復号化に使用します。

クライアントがセキュアな接続を要求する場合は、セキュアな通信が始まる前にハンドシェイクフェーズが行われます。SSL ハンドシェイクの間、サーバーはその公開キーを証明書の形でクライアントに渡します。その証明書にはサーバーのアイデンティティ (URL)、サーバーの公開キー、証明書を検証するデジタル署名が含まれています。次にクライアントは証明書を検証し、証明書が信頼できるか決定します。証明書が信頼できる場合は、クライアントは SLL 接続に対し双方向の暗号化キーを生成し、サーバーの公開キーを使用してそれを暗号化し、サーバーに戻します。サーバーはその秘密キーを使用して双方向の暗号化キーを復号化し、双方向の暗号化キーを使用してこの接続に関して 2 つのマシン間での更なる通信が暗号化されます。

サーバーでは、公開 / 秘密キーのペアはキーペアと信頼できる証明書を保存する暗号化されたファイルである キーストアに保存されます。キーストア内の各キーペアは、キーストアからキーペアを保存または要求するときに使用される一意の名前、エイリアスとして識別されます。公開キーは公開キーとアイデンティティをバインドするデジタル署名である *証明書* の形でクライアントに配布されます。クライアントでは、既知の検証の証明書は *信頼ストア* として知られるデフォルトのキーストアで保管されます。

CA 署名証明書と自己署名証明書

公開鍵基盤は不明のマシンの資格情報を確立するための信頼チェーンに依存しています。公開キーの使用はマシン間のトラフィックを暗号化するだけでなく、ネットワーク接続のもう一方の最後でマシンのアイデンティティを確立するための機能を果たします。「Web of Trust (信用の輪)」を使用して、サーバーのアイデンティティを検証します。あなたにとって不明なサーバーがあるかもしれませんが、その

公開キーがあなたの信頼できる誰かにより署名された場合、サーバーにその信頼を広げます。Certificate Authority (認証局) はカスタムのアイデンティティを検証し、署名された証明書を発行する商用エンティティです。JDK には **cacerts** ファイルが含まれ、信頼できる Certificate Authority (CA) の証明書が複数付いています。これらの CA により署名されたすべてのキーは自動的に信頼されます。大規模な組織には Red Hat Certificate System を使用するなど、独自の内部の Certificate Authority があります。この場合、内部の Certificate Authority の証明書の署名は通常 Corporate Standard Build の一部としてクライアントにインストールされており、その証明書で署名されたすべての証明書は信頼できます。CA 署名証明書は稼働シナリオにとって最良事例です。

開発やテスト中、または小規模や内部のみの実稼働シナリオでは、*自己署名証明書* を使用できます。この証明書は Certificate Authority では署名されていませんが、ローカルで生成された証明書で署名されています。ローカルで生成された証明書はクライアントの **cacerts** ファイル内にはないため、サーバーからそのキーの証明書をエクスポートし、SSL により接続するすべてのクライアントからその証明書をインポートする必要があります。

JDK にはキーペアと証明書を生成するコマンドラインツール **keytool** が含まれています。**keytool** で生成された証明書は、CA による署名のために送られるか、自己署名証明書としてクライアントに配布されます。

- 開発使用のための自己署名証明書の生成、クライアントへのその証明書のインポートに関しては「[keytool を使用した自己署名証明書の生成](#)」で説明されています。
- 証明書を生成し、実稼働使用に対して CA によりそれを署名することは、本書の範囲外です。このタスクを実行する上での詳細情報は keytool の man ページを参照してください。

15.2. 暗号化キーと証明書の生成

15.2.1. keytool を使用した自己署名証明書の生成

15.2.1.1. キーペアの生成

JDK の一部である **keytool** コマンドを使用して、新しいキーペアを生成します。Keytool は既存のキーストアに新しいキーペアを追加、またはキーペアと同時に新しいキーストアを作成することができます。

このキーペアを使用して、サーバーとリモートクライアント間の SSL 暗号化を交渉します。次の手順ではキーペアを生成し、それを **localhost.keystore** と呼ばれるキーストアに保存します。サーバー上でこのキーストアを EJB3 invoker に使用可能にする必要がでできます。この例のキーペアはエイリアス「**ejb-ssl**」のもののキーストアに保存されます。[RMI のセキュアなリモートコネクタの作成](#) で EJB3 Remoting コネクタを設定するときには、このキーエイリアスとキーペアのパスワード (あれば) が必要となります。

手順15.1 新しいキーペアの生成、および JBoss server conf ディレクトリのキーストア「**localhost.keystore**」へのその追加

これは SSL 暗号化に対して新しいキーペアを生成する手順を示しています。

- 次のコマンドで SSL 暗号化で使用するキーペアを作成します。

```
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"
```

結果

キーペアがエイリアス **ejb-ssl** のキーストア **localhost.keystore** に追加されます。

このコマンドのパラメータは **keytool** [パラメータ](#) で説明します。

keytool パラメータ

alias

キーストア内のキーペアを特定するために使用される英数字のトークンです。キーストアは複数のキーを含むことができます。エイリアスはキーストアのキーペアを一意的に特定する方法を提供します。キーペアのエイリアスはキーストア内では一意である必要があります。

keystore

キーペアを保存するために使用されるキーストアです。これは相対または絶対ファイルパスになります。

storepass

キーストアのパスワードです。キーストアがすでに存在している場合は、これはキーストアの既存パスワードでなければなりません。指定されたキーストアが存在していない場合は、作成され、このパスワードが新しいパスワードとなります。このパスワードが必要となるのは、キーと証明書を取得または保存するためにキーストアにアクセスするためです。

keypass

新しいキーペアのパスワードです。このパスワードは今後キーペアを使用するために必要です。

dname

証明書を識別する詳細です。

CN

サーバー名である共通名です。これはサーバー名と一致する必要があり、JNDI ルックアップでクライアントに返されます。クライアントが JNDI からの 1 つの名前を使用してサーバーに SSL 接続を試行し、別の名前の証明書を受け取る場合、接続は失敗します。

OU

組織単位を意味します。サーバーの役割を持つ組織単位の名前です。

O

組織を意味します。組織の名前で、URL と表現されることがあります。

L

場所を意味します。サーバーの場所です。

C

国を意味します。2 文字の国のコードです。



注記

最適なセキュリティの実践を図るために、JBoss Application Server プロセスの所有者のみが読み取り可能なセキュアなファイルシステムにキーストアファイルを保存します。

コマンドラインで指定されたキーストアがない場合は、**keytool** はキーペアを現在のユーザーのホームディレクトリの **keystore** と呼ばれる新しいキーストアに追加します。このキーストアファイルは隠しファイルです。

15.2.1.2. 自己署名証明書のエクスポート

キーペアがサーバーが使用するために生成された時点で、証明書は作成されなければなりません。[手順 15.2 「証明書のエクスポート」](#) では、**localhost.keystore** と呼ばれるキーストアから **ejb-ssl** キーをエクスポートするステップを説明します。

手順15.2 証明書のエクスポート

これはキーストアからファイルに証明書をエクスポートする手順です。

1. 次のコマンドを発行します。

```
keytool -export -alias ejb-ssl -file mycert.cer -keystore  
localhost.keystore
```

2. キーストアパスワードを入力します。

結果

証明書はファイル **mycert.cer** にエクスポートされます。

15.2.2. 自己署名サーバー証明書を受け入れるためのクライアント設定

SSL でリモートメソッド呼び出しを行うには、クライアントがサーバーの証明書を信頼する必要があります。生成した証明書は自己署名で、既知の認証局への信頼チェーンがありません。自己署名証明書では、クライアントは明示的にその証明書を信頼するよう設定する必要があります。そうでないと、接続は失敗します。自己署名証明書を信頼するようクライアントを設定するには、クライアントの **信頼ストア** に自己署名サーバー証明書をインポートします。

信頼ストアは信頼できる証明書を含むキーストアです。ローカルの信頼ストアにある証明書は有効として受け入れられます。サーバーが自己署名証明書を使用する場合は、SSL でリモートメソッド呼び出しを行うどのクライアントにも信頼ストアの証明書が必要となります。証明書として公開キーをエクスポートし、そうしたクライアントの信頼ストアに証明書をインポートします。

「[自己署名証明書のエクスポート](#)」で作成された証明書は [手順15.3 「信頼ストア「localhost.truststore」への証明書のインポート」](#) で説明されたステップを行うためにはクライアントにコピーされなければなりません。

手順15.3 信頼ストア「localhost.truststore」への証明書のインポート

これはクライアントの信頼ストアにサーバー上で以前にエクスポートされた証明書をインポートする手順です。

1. クライアントで次のコマンドを発行します。


```
keytool -import -alias ejb-ssl -file mycert.cer -keystore
localhost.truststore
```

2. この信頼ストア用のパスワードが存在している場合はそれを入力します。なければ、新しい信頼ストア用のパスワードを入力し、再度それを入力します。
3. 証明書の詳細を確認します。正しければ「yes」と入力し、それを信頼ストアにインポートします。

結果

これで証明書が信頼ストアにインポートされ、この証明書を使用するサーバーでセキュアな接続が確立されました。

キーストアと同様に、指定された信頼ストアが存在しない場合はそれが作成されます。ただキーストアと大きく異なる点は、デフォルトの信頼ストアはなく、ひとつも指定されていない場合はコマンドは失敗するということです。

localhost.truststore を使用するためのクライアントの設定

これで自己署名サーバー証明書をクライアント上の信頼ストアにインポートしました。次は、この信頼ストアを使用するようクライアントに指示する必要があります。その方法

は、`javax.net.ssl.trustStore` プロパティを使用して `localhost.truststore` 場所をアプリケーションに渡し、`javax.net.ssl.trustStorePassword` プロパティを使用して信頼ストアパスワードを渡します。例15.1「特定の信頼ストアを使用した `com.acme.Runclient` アプリケーションの呼び出し」は JBoss Application Server 上で EJB にリモートメソッド呼び出しを行う仮定のアプリケーションである `com.acme.RunClient` アプリケーションを呼び出す例のコマンドです。アプリケーションのパッケージディレクトリのルート (ファイルパス `com/acme/RunClient.class` の `com` ディレクトリを含むディレクトリ) からこのコマンドを実行します。

例15.1 特定の信頼ストアを使用した `com.acme.Runclient` アプリケーションの呼び出し

```
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. EJB3 RMI + SSL 設定

手順15.4 EJB3 の RMI + SSL 設定の概要

この手順ではサーバーの EJB3 Bean とネットワークの別のマシンで実行しているファットクライアントとの間の Remote Method Invocation トラフィックの SSL 暗号化を設定します。

1. 暗号化キーと証明書の生成
2. RMI のセキュアなリモートコネクタの設定
3. セキュアな RMI コネクタを使用するための EJB3 Bean のアノテーション

暗号化キーと証明書の生成については「[暗号化キーと証明書の生成](#)」で取り上げています。

RMI のセキュアなリモートコネクタの作成

JBoss Application Server プロファイル **deploy** ディレクトリのファイル **ejb3-connectors-jboss-beans.xml** には、EJB3 リモートメソッド呼び出しの JBoss Remoting コネクタの定義が含まれています。

例15.2 セキュアな EJB3 コネクタのサンプル

コードサンプルで説明されている Bean は **ejb3-connectors-jboss-beans.xml** ファイルに追加されます。両方の Bean とも [手順15.1「新しいキーペアの生成、および JBoss server conf ディレクトリのキーストア「localhost.keystore」へのその追加」](#) で作成されたキーペアを使用して EJB3 に対しセキュアなコネクタを設定する必要があります。

サンプル設定の **keyPassword** プロパティは、キーペアが作成されたときに指定されるキーペアのパスワードです。

サンプル設定は 3843 ポートで SSL 接続をリッスンするコネクタを作成します。このポートはクライアントからのアクセスに対しサーバーファイアウォールで開く必要があります。

```
<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>
```

SSL トランスポートに対する EJB3 Bean の設定

すべての EJB3 Bean はデフォルトではセキュアでない RMI コネクタを使用します。SSL による Bean のリモート呼び出しを有効にするには、**@org.jboss.annotation.ejb.RemoteBinding** で Bean をアノテートします。

例15.3 セキュアなリモート呼び出しを有効にするための EJB3 Bean アノテーション

アノテーションは EJB3 Bean を JNDI 名 **StatefulSSL** にバインドします。リモートインターフェースを実装しているプロキシは、Bean が JNDI から要求されたときにクライアントに返され、SSL でサーバーと通信します。

```
@RemoteBinding(clientBindUrl="sslsocket://0.0.0.0:3843",
jndiBinding="StatefulSSL")
@Remote(BusinessInterface.class)
public class StatefulBean implements BusinessInterface
{
    ...
}
```



注記

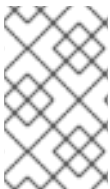
例15.3「セキュアなリモート呼び出しを有効にするための EJB3 Bean アノテーション」では、IP アドレスは 0.0.0.0 と指定されており「すべてのインターフェース」という意味です。これを `boss.bind.address` システムプロパティの値に変更します。

EJB3 Bean のセキュアおよびセキュアでない呼び出しを有効にする

同じ EJB3 Bean のセキュアおよびセキュアでないリモートメソッド呼び出し両方を有効にできます。例15.4「セキュアおよびセキュアでない呼び出しの EJB3 Bean アノテーション」ではこれを行うためのアノテーションを示しています。

例15.4 セキュアおよびセキュアでない呼び出しの EJB3 Bean アノテーション

```
@RemoteBindings({
    @RemoteBinding(clientBindUrl="sslsocket://0.0.0.0:3843",
jndiBinding="StatefulSSL")
    @RemoteBinding(jndiBinding="StatefulNormal")
})
@Remote(BusinessInterface.class)
public class StatefulBean implements BusinessInterface
{
    ...
}
```



注記

例15.4「セキュアおよびセキュアでない呼び出しの EJB3 Bean アノテーション」では、IP アドレスは **0.0.0.0** と指定されており「すべてのインターフェース」という意味です。これを `boss.bind.address` システムプロパティの値に変更します。

クライアントが JNDI から **StatefulNormal** を要求する場合、リモートインターフェースを実装している返されたプロキシは暗号化されていないソケットプロトコルによりサーバーと通信します。**StatefulSSL** が要求される場合、リモートインターフェースを実装している返されたプロキシは SSL によりサーバーと通信します。

15.4. HTTPS 設定による EJB3 RMI

手順15.5 HTTPS による EJB3 RMI 設定の概要

これは SSL で暗号化された HTTP に Remote Method Invocation トラフィックのトンネリングを設定する手順です。これにはトラフィックを暗号化し、RMI ポートをブロックするファイアウォールをトランスバースできるという 2 つの効果があります。

1. 暗号化キーと証明書の生成
2. HTTPS Web コネクタによる RMI の設定
3. サブレットの設定
4. HTTPS による RMI のセキュアなリモートコネクタの設定
5. HTTPS トラnsポートに対する EJB3 Bean の設定
6. HTTPS による RMI に対するクライアントの設定

暗号化キーと証明書の生成については「[暗号化キーと証明書の生成](#)」で取り上げています。

手順15.6 HTTPS Web コネクタによる RMI の設定

これは 8443 ポートでリッスンし、クライアントから SSL 接続を受け入れる Web コネクタを作成する手順です。

- ファイル `jboss-as/server/$PROFILE/deploy/jbossweb.sar/server.xml` を編集し、HTTPS コネクタを非コメントします。

```
<!-- SSL/TLS Connector configuration using the admin dev1 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" />
```

結果

Web コネクタを作成し、SSL 接続を受け入れます。

手順15.7 サブレットの設定

これは Web コネクタから `ServletServerInvoker` に要求を渡すサブレットを設定する手順です。

1. `jboss-as/server/$PROFILE/deploy/` に `servlet-invoker.war` という名前のディレクトリを作成します。
2. `servlet-invoker.war` ディレクトリ内に `WEB-INF` ディレクトリを作成します。
3. その `WEB-INF` ディレクトリに次の内容を含む `web.xml` という名前のファイルを作成します。

```
<?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>

```

結果

サーブレットを作成し、Web コンテナからサーバー invoker に SSL 要求を転送します。

locatorUrl を使用して、[手順15.8「HTTPS による RMI のセキュアなリモートコネクタの設定」](#) で定義するリモートコネクタの **InvokerLocator** 属性を通じてサーブレットをリモートコネクタに接続します。

手順15.8 HTTPS による RMI のセキュアなリモートコネクタの設定

これは RMI を実装する Server Invoker を作成する手順です。

- **jboss-as/server/\$PROFILE/deploy/** に次の内容を含む **servlet-invoker-service.xml** という名前のファイルを作成します。

```

<?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>

```

結果

サーブレットからの要求を受け入れ、EJB3 のメソッドを呼び出すことができるリモートコネクタを作成します。

手順15.9 HTTPS トランスポートに対する EJB3 Bean の設定

これは HTTPS トランスポートにバインドする EJB3 を設定する手順です。

- HTTPS により RMI を使用するために Bean をアノテートする

例15.5 HTTPS により RMI を使用するために EJB3 をアノテートする

```
// 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
{
    ....
}
```

結果

これで HTTPS によるリモート呼び出しに EJB3 が使用可能になりました。

HTTP により RMI を使用するために Bean をアノテートする

オプションとして、HTTP による RMI での呼び出しに Bean をアノテートすることができます。これは RMI ポートをブロックするファイアウォールを通じて RMI 呼び出しをトンネルできるためテストに役立ちますが、セキュリティ設定の別のレイヤを削除します。

例15.6 HTTP により RMI を使用するために Bean をアノテートする

```
// 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
{
    ....
}
```

HTTPS による RMI に対するクライアントの設定

Bean を検索する場合、EJB クライアントは JNDI ルックアップに次のプロパティを使用するべきです。

HTTP(S) による RMI へのクライアントアクセス

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

```

HTTP(S) による RMI へのクライアントアクセスでは、*user name* と *password* の値は `http-invoker` をセキュアにするために使用するセキュリティドメインの有効なユーザー名とパスワードに一致します。このセキュリティドメインは `jboss-as/$PROFILE/deploy/http-invoker.sar/invoker.war/WEB-INF/jboss-web.xml` で設定されます。

15.5. EJB2 RMI + SSL 設定

手順15.10 EJB2 の SSL の設定の概要

1. 暗号化キーと証明書の生成
2. SSL に対する統合 invoker の設定

暗号化キーと証明書の生成については「[暗号化キーと証明書の生成](#)」で取り上げています。

SSL に対する統合 invoker の設定

EJB2 リモート呼び出しは単一の統合 invoker を使用し、デフォルトではポート 4446 で実行します。EJB2 リモートメソッド呼び出しに使用される統合 invoker の設定は JBoss Application Server プロファイルの `$JBOSS_HOME/server/deploy/remoting-jboss-beans.xml` ファイルで定義されます。このファイルに次の SSL Socket Factory Bean と SSL Domain Bean を追加します。

例15.7 EJB2 の SSL Server Factory

```

<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>

```

ここで JBoss Application Server プロファイルの `$JBOSS_HOME/server/$PROFILE/conf/jboss-service.xml` ファイルに次を加えることによって、SSLSocketBuilder をカスタマイズします。

例15.8 SSLSocketBuilder 設定

```

<!-- 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>

```

Bean の SSL トランスポート設定

JBoss Application Server プロファイルの **deploy/remoting-jboss-beans.xml** ファイルで、コードを更新して以下の情報を反映させます。

例15.9 Beans の SSL トランスポート

```

...
<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>
...

```

第16章 XML 設定のパスワードマスク

本項の手順に従って、パスワードをマスクすることにより JBoss Enterprise Application Installation の安全性を高めます。そうでない場合はパスワードはクリアテキストとしてファイルシステムに保存されます。

16.1. パスワードマスクの概要

パスワードはリソースへのアクセスを承認された関係者にのみに制限するために使用される秘密の認証トークンです。JBoss サービスがパスワードで保護されたリソースにアクセスするためには、パスワードは JBoss サービスが使用できるものでなくてはなりません。これは起動時に JBoss Application Server に渡されるコマンドライン引数を用いて実行できますが、実稼働環境では実際的ではありません。実稼働環境では、通常設定ファイルにパスワードを含むことで JBoss サービスが使用可能になります。

すべての JBoss Enterprise Application Platform 設定ファイルはセキュアなファイルシステムに保存されるべきであり、JBoss Application Server プロセス所有者のみが読み取り可能であるべきです。さらに、セキュリティレベルを高めるために設定ファイルのパスワードをマスクすることができます。本項の手順に従って、Microcontainer Bean 設定のクリアテキストのパスワードをパスワードマスクと置換します。Data Source パスワードの暗号化手順は [17章 データソースのパスワードの暗号化](#) を、Tomcat のキーストアのパスワードの暗号化手順は [18章 Tomcat Connector のキーストアパスワードの暗号化](#) を、LdapExtLoginModule のパスワードの暗号化手順は [19章 JaasSecurityDomain での LdapExtLoginModule の使用](#) をそれぞれ参照してください。



注記

侵入できないセキュリティというものは存在しません。強固なセキュリティ手段は、システムの承認されていないアクセスを困難にするだけです。パスワードマスクも例外ではありません。侵入不可能ではありませんが、これにより設定ファイルを軽く閲覧できなくなり、クリアテキストのパスワードを抽出する必要がある手間が増えます。

手順16.1 クリアテキストのパスワードマスクの概要

1. パスワードを暗号化するために使用するキーペアの生成
2. キーストアパスワードの暗号化
3. パスワードマスクの作成
4. クリアテキストのパスワードとそれらのパスワードマスクとの置換

16.2. キーストアとマスクされたパスワードの生成

パスワードマスクでは公開 / 秘密キーペアを使用してパスワードを暗号化します。パスワードマスクで使用するにはキーペアを生成する必要があります。デフォルトでは、JBoss Enterprise Application Platform 5 には `jboss-as/bin/password/password.keystore` のキーストアのエイリアス `jboss` を持つキーペアが必要です。

次の手順はこのデフォルト設定に沿っています。キーストアの場所またはキーエイリアスを変更したい場合は、デフォルト設定を変更する必要があります。方法は [「パスワードマスクのデフォルト変更」](#) を参照してください。

手順16.2 パスワードマスクのためのキーペアとキーストアの生成

1. コマンドラインで、ディレクトリを **jboss-as/bin/password** ディレクトリに変更します。
2. **keytool** を使用して、次のコマンドでキーペアを生成します。

```
keytool -genkey -alias jboss -keyalg RSA -keysize 1024 -keystore password.keystore
```

重要

キーストアとキーペアには同じパスワードを指定する必要があります。

3. オプション

結果として得られた password.keystore を JBoss Application Server プロセス所有者のみが読み取り可能であるようにします。

Unix ベースのシステムでは、**chown** コマンドを使用してオーナーシップを JBoss Application Server プロセス所有者に変更し、**chmod 600 password.keystore** コマンドを使用してファイルを所有者のみが読み取り可能であるようにします。

このステップはご使用のサーバーのセキュリティを高めるために推奨されます。

注記：JBoss Application Server プロセス所有者は対話型コンソールのログインアクセスを持つべきではありません。その場合、これらの動作は別のユーザーとして実行します。マスクされたパスワードの作成には、キーストアへの読み取りアクセスが必要になるため、キーストアファイルのパーミッションを制限する前にマスクされたパスワードの設定を完了したいと思われるかもしれません。

キーストアと **keytool** コマンドに関する詳細は「[SSL 暗号化の概要](#)」を参照してください。

16.3. キーストアのパスワードの暗号化

パスワードをマスクした状態では、JBoss サービスに必要なパスワードは xml 設定ファイルのクリアテキストには保存されません。その代わりに、提供するキーペアを使用して暗号化されたファイルに保存されます。

このファイルを復号化し、ランタイム時にマスクされたパスワードにアクセスするには、JBoss Application Server は作成したキーペアを使用できる必要があります。JBoss Password Tool の **password_tool** を用いて、JBoss Application Server にキーストアのパスワードを提供します。このツールはキーストアのパスワードを暗号化し、保存します。そしてパスワードをマスクするためにキーストアのパスワードは JBoss Password Tool に使用可能となり、ランタイム時にそれらを復号化するために JBoss Application Server に使用可能となります。

手順16.3 キーストアのパスワードの暗号化

1. コマンドラインで、**jboss-as/bin** ディレクトリに変更します。
2. Unix ベースのシステムにはコマンド **./password_tool.sh** を、Windows ベースのシステムにはコマンド **password_tool.bat** を使用して、パスワードツールを実行します。

結果

JBoss Password Tool が開始され「**Keystore is null. Please specify keystore below:**」を報告します。

3. 0 を押して「**0: Encrypt Keystore Password**」を選択し、Enter を押します。

結果

パスワードツールは「**Enter keystore password**」で応答します。

4. [手順16.2「パスワードマスクのためのキーペアとキーストアの生成」](#) で指定したキーストアのパスワードを入力します。

結果

パスワードツールは「**Enter Salt (String should be at least 8 characters)**」で応答します。

5. 無作為の文字列を入力して、暗号化の強度を高めます。

結果

パスワードツールは「**Enter Iterator Count (integer value)**」で応答します。

6. 整数を入力して、暗号化の強度を高めます。

結果

パスワードツールは「**Keystore Password encrypted into password/jboss_keystore_pass.dat**」で応答します。

7. 「**5:Exit**」を選択して、終了します。

結果

パスワードツールはメッセージ「**Keystore is null. Cannot store.**」を表示し、終了します。これは通常です。

8. オプション

結果として得られたファイル **password/jboss_keystore_pass.dat** を JBoss Application Server プロセス所有者のみが読み取り可能であるようにします。

Unix ベースのシステムでは、**chown** コマンドを使用してオーナーシップを JBoss Application Server プロセス所有者に変更し、**chmod 600 jboss-keystore_pass.dat** コマンドを使用してファイルを所有者のみが読み取り可能であるようにします。

このステップはご使用のサーバーのセキュリティを高めるために推奨されます。この暗号化されたキーが危害を受けた場合は、パスワードマスクにより実現した安全性は極めて低下することを認識しておいてください。このファイルはセキュアなファイルシステムで保存されるべきです。

注記 : JBoss Application Server プロセス所有者は対話型コンソールのログインアクセスを持つべきではありません。この場合、これらの動作は別のユーザーとして実行します。マスクされたパスワードの作成には、キーストアへの読み取りアクセスが必要になるため、キーストアファイルのパーミッションを制限する前にマスクされたパスワードの設定を完了したいと思われるかもしれません。

注記

このキーストアのパスワード暗号化手順は一度だけ実行するべきです。キーストアのパスワード入力を間違えた場合や、後日キーストアを変更した場合は、**jboss-keystore_pass.dat** ファイルを削除して、この手順を繰り返すべきです。キーストアを変更すると、以前に生成されたすべてのマスクされたパスワードは機能しなくなることにご注意ください。

16.4. パスワードマスクの作成

JBoss Password Tool は暗号化されたパスワードファイル **jboss-as/bin/password/jboss_password_enc.dat** を管理します。このファイルはパスワードツールに提供するキーペアを使用して暗号化され、設定ファイルでマスクされるパスワードを含みます。パスワードを保存するときに Password Tool に指定する任意の一意識別子である「domain」によりパスワードは保存され、このファイルから取得されます。そして、設定ファイルのクリアテキストのパスワードを置換するアノテーションの一部として指定します。これにより JBoss Application Server がランタイム時にファイルから正しいパスワードを取得することが可能となります。



注記

以前にキーストアと暗号化されたキーストアのパスワードファイルを JBoss Application Server プロセス所有者のみが読み取りできるようにした場合は、JBoss Application Server プロセス所有者として次の手順を実行する必要があります。そうでない場合は、キーストア (**jboss-as/bin/password/password.keystore**) と暗号化されたキーストアのパスワードファイル (**jboss-as/bin/password/jboss_keystore_pass.dat**) をご使用のユーザーだけが読み取り可能であるようにし、この動作を実行している間に、(すでに存在する場合は) 暗号化されたパスワードファイル **jboss-as/bin/password/jboss_password_enc.dat** を読み取り、書き込み可能にします。

手順16.4 パスワードマスクの作成

前提条件

- [手順16.2「パスワードマスクのためのキーペアとキーストアの生成」](#)。
 - [手順16.3「キーストアのパスワードの暗号化」](#)。
1. コマンドラインで、**jboss-as/bin** ディレクトリに変更します。
 2. Unix ベースのシステムにはコマンド **./password_tool.sh** を、Windows ベースのシステムにはコマンド **password_tool.bat** を使用して、パスワードツールを実行します。

結果

JBoss Password Tool が開始され「**Keystore is null. Please specify keystore below:**」を報告します。

3. 1 を押して「**1:Specify KeyStore**」を選択し、Enter を押します。

結果

パスワードツールは「**Enter Keystore location including the file name**」で応答します。

4. [手順16.2「パスワードマスクのためのキーペアとキーストアの生成」](#) で作成したキーストアへのパスを入力します。絶対パス、または **jboss-as/bin** に相対するパスを指定できます。高度なインストールを実行し「[パスワードマスクのデフォルト変更](#)」のようにデフォルトを変更していない限りは、これは **password/password.keystore** のはずです。

結果

パスワードツールは「**Enter Keystore alias**」で応答します。

5. キーエイリアスを入力します。高度なインストールを実行し「[パスワードマスクのデフォルト変更](#)」のようにデフォルトを変更していない限りは、これは **jboss** のはずです。

結果

キーストアとキーエイリアスがアクセス可能な場合、パスワードツールは log4j WARNING メッセージで応答し、既存のパスワードマスク、メインメニューと続く行「**Loading domains** [」で応答します。

6. 2 を押して「**2:Create Password**」を選択し、Enter を押します。

結果

パスワードツールは「**Enter security domain:**」で応答します。

7. パスワードマスクの名前を入力します。これは設定ファイルのパスワードマスクを特定するために使用する任意の一意名です。

結果

パスワードツールは「**Enter passwd:**」で応答します。

8. マスクしたいパスワードを入力します。

結果

パスワードツールは「**Password created for domain:mask name**」で応答します。

9. パスワードマスクの作成プロセスを繰り返し、マスクしたいすべてのパスワードのマスクを作成します。
10. 「**5:Exit**」を選択して、プログラムを終了します。

16.5. クリアテキストのパスワードとそれらのパスワードマスクとの置換

XML 設定ファイルのクリアテキストのパスワードは、アノテーションのプロパティの割り当てを変更することで、パスワードマスクと置換可能です。[手順16.4「パスワードマスクの作成」](#)に従って、Microcontainer Bean 設定ファイルでマスクしたいクリアテキストのパスワードに対しパスワードマスクを生成します。次に、設定に各クリアテキストのパスワードがある時とそのマスクを参照するアノテーションを置換します。

アノテーションの一般的な形式は以下のとおりです。

例16.1 パスワードマスクのアノテーションの一般的な形式

```
<annotation>@org.jboss.security.integration.password.Password(securityDomain=MASK_NAME, methodName=setPROPERTY_NAME)</annotation>
```

具体的な例として、JBoss Messaging パスワードはファイル **deploy/messaging/messaging-jboss-beans.xml** のサーバープロファイルに保存されます。「messaging」という名前のパスワードマスクを作成する場合、設定ファイルのスニペットの前後はそれぞれ以下ようになります。

例16.2 JBoss Messaging Microcontainer Bean 設定前

```
<property name="suckerPassword">CHANGE ME!!</property>
```

例16.3 JBoss Messaging Microcontainer Bean 設定後

```
<annotation>@org.jboss.security.integration.password.Password(securityDomain=messaging,
methodName=setSuckerPassword)</annotation>
```

16.6. パスワードマスクのデフォルト変更

JBoss Enterprise Application Platform 5 にはパスワードマスクに設定されたサーバープロファイルが同梱しています。デフォルトでは、サーバープロファイルはキーストア **jboss-as/bin/password/password.keystore** とキーエイリアス **jboss** を使用するように設定されています。パスワードマスクに使用するキーペアを別の場所または別のエイリアスに保存する場合は、新しい場所またはキーエイリアスでサーバープロファイルを更新する必要があります。

パスワードマスクのキーストアの場所とキーエイリアスは、含まれている JBoss Application Server サーバープロファイルにあるファイル **deploy/security/security-jboss-beans.xml** で指定されます。

例16.4 security-jboss-beans.xml でのパスワードマスクのデフォルト

```
<!-- 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</property>
    <property
name="keyStorePasswordEncryptedFileName">password/jboss_keystore_pass.dat</property>
</bean>
```


第17章 データソースのパスワードの暗号化

JBoss Enterprise Application Platform のデータベース接続は ***-ds.xml** データソースファイルで定義されます。これらのデータベース接続の詳細はクリアテキストのパスワードを含みます。サーバーの安全性を高めるためには、データソースファイルのクリアテキストのパスワードを暗号化されたパスワードと置換します。

本項ではデータソースのパスワードを暗号化するためのメソッドを 2 つ示します。

モジュール **SecureIdentityLoginModule** を使用した セキュアなアイデンティティ は「**セキュアなアイデンティティ**」で説明します。

モジュール **JaasSecurityDomainIdentityLoginModule** を使用した パスワードベースの暗号化を使用した設定されたアイデンティティ は「**セキュアなアイデンティティ**」で説明します。

17.1. セキュアなアイデンティティ

データソースの設定がサーバーにより必要な場合に、クラス **org.jboss.resource.security.SecureIdentityLoginModule** を使用して、データベースのパスワードを暗号化しパスワードの復号化されたバージョンを提供します。**SecureIdentityLoginModule** はハードコードされたパスワードを使用して、データソースのパスワードを暗号化 / 復号化します。

手順17.1 概要 : SecureIdentityLoginModule を使用したデータソースのパスワードの暗号化

1. データソースのパスワードの暗号化
2. 暗号化されたパスワードでのアプリケーションの認証ポリシーの作成
3. アプリケーションの認証ポリシーを使用するためのデータソースの設定

17.1.1. データソースのパスワードの暗号化

データソースのパスワードは、クリアテキストのパスワードを渡すことによって **SecureIdentityLoginModule** メインメソッドを使用して暗号化されます。**SecureIdentityLoginModule** は **jbosssx.jar** により提供されます。

手順17.2 データソースのパスワードの暗号化 - Platform 5.0 および 5.0.1 バージョン

これは JBoss Enterprise Application Platform 5.0 および 5.0.1 バージョンでデータソースのパスワードを暗号化する手順です。

1. ディレクトリを **jboss-as** ディレクトリに変更します。
2. 次のコマンドで **SecureIdentityLoginModule** を呼び出し、**PASSWORD** としてクリアテキストのパスワードを提供します。

Linux でのコマンド

```
java -cp client/jboss-logging-spi.jar:common/lib/jbosssx.jar \
org.jboss.resource.security.SecureIdentityLoginModule PASSWORD
```

Windows でのコマンド

```
java -cp client\jboss-logging-spi.jar;common\lib\jbossx.jar \
org.jboss.resource.security.SecureIdentityLoginModule PASSWORD
```

結果

コマンドは暗号化されたパスワードを返します。

手順17.3 データソースのパスワードの暗号化 - Platform 5.1 およびそれ以降のバージョン

これは JBoss Enterprise Application Platform versions 5.1 およびそれ以降のバージョンのデータソースのパスワードを暗号化する手順です。

1. ディレクトリを **jboss-as** ディレクトリに変更します。

2. Linux でのコマンド

```
java -cp client/jboss-logging-spi.jar:lib/jbossx.jar \
org.jboss.resource.security.SecureIdentityLoginModule PASSWORD
```

Windows でのコマンド

```
java -cp client\jboss-logging-spi.jar;lib\jbossx.jar \
org.jboss.resource.security.SecureIdentityLoginModule PASSWORD
```

結果

コマンドは暗号化されたパスワードを返します。

17.1.2. 暗号化されたパスワードでのアプリケーションの認証ポリシーの作成

各 JBoss Application Server サーバプロファイルには **conf/login-config.xml** ファイルがあり、アプリケーションの認証ポリシーはそのプロファイルに対して定義されます。ご使用の暗号化されたパスワードに対するアプリケーションの認証ポリシーを作成するには、新しい `<application-policy>` 要素を `<policy>` 要素に追加します。

例17.1「暗号化されたデータソースのパスワードを使用したアプリケーションの認証ポリシーの例」は「EncryptDBPassword」という名前のアプリケーションの認証ポリシーを示す **login-config.xml** ファイルの一部です。

例17.1 暗号化されたデータソースのパスワードを使用したアプリケーションの認証ポリシーの例

```
<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>
```

SecureIdentityLoginModule モジュールのオプション

user name

データベースへの接続を確立するときに使用するユーザー名を指定します。

password

「[データソースのパスワードの暗号化](#)」で生成された暗号化されたパスワードを提供します。

managedConnectionFactoryName

jboss.jca:name

このデータソースの Java Naming and Directory Interface (JNDI) の名前を指定します。

jboss.jca:service

トランザクションのタイプを指定します。

トランザクションのタイプ

NoTxCM

トランザクションのサポートはありません。

LocalTxCM

単一のリソーストランザクションのサポートがあります。

TxCM

単一のリソースまたは分散トランザクションのサポートがあります。

XATxCM

分散トランザクションのサポートがあります。

17.1.3. アプリケーションの認証ポリシーを使用するためのデータソースの設定

ランタイム時には、アプリケーションポリシーはアプリケーションポリシーの名前で JNDI にバインドされ、セキュリティドメインとして使用可能です。

データソースは ***-ds.xml** ファイルで設定されます。 <user-name> および <password> 要素をこのファイルから削除し、<security-domain> 要素と置換します。この要素には次の「[暗号化されたパスワードでのアプリケーションの認証ポリシーの作成](#)」で指定されたアプリケーションの認証ポリシーの名前が含まれます。

「暗号化されたパスワードでのアプリケーションの認証ポリシーの作成」からサンプル名を使用すると、「EncryptDBPassword」は例17.2「セキュアなアイデンティティを使用したデータソースのファイルの例」のようなデータソースのファイルになります。

例17.2 セキュアなアイデンティティを使用したデータソースのファイルの例

```
<?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. PASSWORD BASED ENCRYPTION (パスワードベースの暗号化 : PBE) で設定されたアイデンティティ

`org.jboss.resource.security.JaasSecurityDomainIdentityLoginModule` は `JaasSecurityDomain` によって暗号化されたパスワードを使用してデータソースを静的に定義するためのログインモジュールです。データソースのパスワードの base64 形式は `PBEUtils` を使用して生成できます。

手順17.4 PBEUtils を使用したパスワードの暗号化 - Platforms 5.0 および 5.0.1 バージョン

これは JBoss Enterprise Application Platform 5.0 および 5.0.1 バージョンでのパスワードの暗号化手順です。

- 以下のコマンドを実行します。

```
java -cp jboss-as/common/lib/jbosssx.jar
org.jboss.security.plugins.PBEUtils \
  salt count domain-password data-source-password
```

結果

暗号化されたパスワードが表示されます。

手順17.5 PBEUtils を使用したパスワードの暗号化 - Platform 5.1 バージョン

これは JBoss Enterprise Application Platform 5.1 およびそれ以降のバージョンでのパスワードの暗号化手順です。

- 以下のコマンドを実行します。

```
java -cp jboss-as/lib/jbosssx.jar
org.jboss.security.plugins.PBEUtils \
    salt count domain-password data-source-password
```

結果

暗号化されたパスワードが表示されます。

PBEUtils のパラメータは以下のとおりです。

salt

JaasSecurityDomain からの Salt 属性です (8 文字ちょうどでなければなりません)。

count

JaasSecurity ドメインからの IterationCount 属性です。

domain-password

JaasSecurityDomain から KeyStorePass 属性にマップするプレーンテキストのパスワードです。

data-source-password

JaasSecurityDomain パスワードにより暗号化されるべきデータソースのプレーンテキストのパスワードです。

例17.3「PBEUtils コマンドの例」ではコマンドの例とその出力を示しています。

例17.3 PBEUtils コマンドの例

```
java -cp jbosssx.jar org.jboss.security.plugins.PBEUtils abcdefgh 13
master password
Encoded password: 3zbEkBDfpQAASa3H39pIyP
```

次のアプリケーションポリシーを **\$JBOSS_HOME/server/\$PROFILE/conf/login-config.xml** ファイルに追加します。

```
<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>

```

`$JBOSS_HOME/docs/examples/jca/hsqldb-encrypted-ds.xml` はキーストアの `JaasSecurityDomain` 設定と併せてデータソースの設定を図解します。

例17.4「FilePassword コマンドの例」はコマンドの例とその出力を提供します。

例17.4 FilePassword コマンドの例

次の例は、`.../conf/server.password` ファイルを参照します。最初に、暗号化の詳細を保存し、マスターパスワードを難読化するため、次のコマンドを実行する必要があります。

```

java -cp jboss-as/lib/jbosssx.jar
org.jboss.security.plugins.FilePassword salt count master_password
password_file

```

```

For example: java -cp jboss-as/lib/jbosssx.jar
org.jboss.security.plugins.FilePassword abcdefgh 13 master jboss-
as/server/$PROFILE/conf/server.password

```

```
<?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>

```

```

        </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>

...

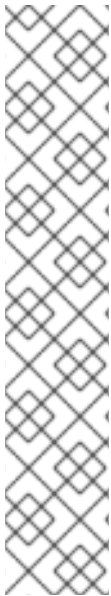
</datasources>

```



警告

パスワード生成ステップで使用された MBean の同じ Salt と IterationCount を使用するようにしてください。



注記

暗号化されたデータソースに依存するサービスを開始するとき、次の MBean がサービスとして開始されていない場合はエラー

`java.security.InvalidAlgorithmParameterException: Parameters missing` が発生します。

```
(jboss.security:service=JaasSecurityDomain, domain=ServerMasterP  
assword)
```

以前に示した例の **`hsqldb-encrypted-ds.xml`** コードのとおり次の要素を含めることで、データソースの前に MBean が開始します。

```
<depends>jboss.security:service=JaasSecurityDomain, domain=Serve  
rMasterPassword</depends>
```


第18章 TOMCAT CONNECTOR のキーストアパスワードの暗号化

JBoss Web は Apache Tomcat に基づいています。

Tomcat で SSL を使用するにはセキュアなコネクタが必要です。これはキーストア / 信頼ストアのパスワードは Tomcat の **server.xml** ファイルのコネクタ要素の属性として渡されないことを意味しています。

キーストア、信頼ストア、パスワードベースの暗号化に対応する JaasSecurityDomain に関する基礎を理解することが推奨されます。

サポート情報および関連手順については [13章 セキュアリモートパスワードプロトコル](#) と [17章 データソースのパスワードの暗号化](#) を参照してください。

手順18.1 Tomcat Container のキーストアのパスワードの暗号化

1. コネクタ要素を追加します

\$JBOSS_HOME/server/\$PROFILE/deploy/jbossweb.sar の **server.xml** にコネクタ要素を追加します。

```
<!-- 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. JaasSecurityDomain MBean を設定します

\$JBOSS_HOME/server/\$PROFILE/deploy/security-service.xml ファイルの JaasSecurityDomain MBean を設定します。

ファイルが存在していない場合は作成する必要があります。例のコードは、ファイルが存在していない場合の必要なコンテンツを説明しています。すでに **security-service.xml** がある場合は、ファイルに **<mbean>** 要素ブロックを追加します。

```
<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>

```

Salt と IterationCount は暗号化されたパスワードの強度を定義する変数であるため、表示されているものから変更できます。新しい値を記録し、暗号化されたパスワードを生成するときに使用するようにしてください。



注記

Salt は 8 文字以上である必要があります。

3. 暗号化されたパスワードを生成します

<mbean> 設定はキーストアが **jboss-as/server/\$PROFILE/conf/localhost.keystore** ファイルに保存されていることを指定します。<mbean> は暗号化されたパスワードファイルが **jboss-as/server/\$PROFILE/conf/keystore.password** ファイルに保存されていることも指定します。

localhost.keystore ファイルを作成する必要があります。

jboss-as/server/\$PROFILE/conf ディレクトリで次のコマンドを実行します。

```

[conf]$ java -cp $JBOSS_HOME/lib/jbosssx.jar
\org.jboss.security.plugins.FilePassword welcometojboss 13 unit-
tests-server keystore.password

```

このコマンドは jbosssx.jar をクラスパス (-cp) として FilePassword をセキュリティプラグインとして使用し、**unit-tests-server** と設定されたパスワードを持つ **keystore.password** ファイルを作成します。**keystore.password** ファイルを作成するパーミッションがあるか確認するには、JaasSecurityDomain の <mbean> <attribute> 要素で設定された salt と iteration パラメータを提供します。

/conf ディレクトリでこのコマンドを実行することで、**keystore.password** ファイルがこのディレクトリに保存されます。

4. Tomcat service MBean を更新します

\$JBOSS_HOME/server/\$PROFILE/deploy/jbossweb.sar/META-INF に移動します。

jboss-beans.xml を開き、次の <depends> タグをファイルの後方に追加します。<depends> タグを追加すると、Tomcat は **jboss.security:service=PBESecurityDomain** の後に開始しなければならないことを指定します。

```

<!-- Transaction manager for unfinished transaction checking in the
CachedConnectionValve -->
<depends>jboss:service=TransactionManager</depends>

<depends>jboss.security:service=PBESecurityDomain</depends>

<!-- Inject the TomcatDeployer -->

```

手順18.1 「Tomcat Container のキーストアのパスワードの暗号化」に基づき、Tomcat Connector で参照された pkcs12 キーストアコンテナは次の例のようになります。

```
<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. 中程度のセキュリティユースケース

ユーザーはキーストアパスワードの暗号化は望みませんが、それを (**server.xml** の外に) 外在させるか事前定義された JaasSecurityDomain の使用を望みます。

手順18.2 事前定義された JaasSecurityDomain

1. **jboss-service.xml** を更新し、コネクタを追加します

\$JBOSS_HOME/server/ \$PROFILE /deploy/jbossweb.sar/META-INF に移動し、次のコードブロックを **jboss-service.xml** ファイルに追加します。

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

2. **<depends>** タグを Tomcat サービスに追加します

\$JBOSS_HOME/server/\$PROFILE/deploy/jbossweb.sar に移動します。

server.xml を開き、次の **<depends>** 要素をファイルの最後に追加します。

```
<depends>jboss.security:service=SecurityDomain</depends>
</mbean>
</server>
```

3. ***-service.xml** ファイルの JaasSecurityDomain MBean を定義します
デプロイディレクトリの **security-service.xml** を例にあげます。

■

```
<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>
```



注記

このエラーが発生した場合は、キーストアファイルは JBoss Enterprise Application Platform を実行しているユーザー ID により書き込み可能であることに注意してください。

第19章 JAASSECURITYDOMAIN での LdapExtLoginModule の使用

本章では JaasSecurityDomain によって復号化する暗号化パスワードで LdapExtLoginModule を使用する方法について説明します。本章では、LdapExtLoginModule が暗号化されていないパスワードですでに正しく実行していることを前提とします。LdapExtLoginModule の詳細については、「[LdapExtLoginModule](#)」を参照してください。

手順19.1

1. JaasSecurityDomain MBean を定義します

パスワードの暗号化されたバージョンを復号化するために使用される JaasSecurityDomain MBean を定義します。MBean を **JBOSS_HOME/server/PROFILE/conf/jboss-service.xml** または **JBOSS_HOME/server/PROFILE/deploy** フォルダの ***-service.xml** デプロイメント記述子に追加できます。

```
<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>
```

注記

JaasSecurityDomain 実装が使用するデフォルトの暗号化方式のアルゴリズムは **PBEwithMD5andDES** です。他の暗号化方式のアルゴリズムには **DES**、**TripleDES**、**Blowfish**、**PBEWithMD5AndTripleDES** があります。すべてのアルゴリズムは対称アルゴリズムです。これらの 1 つの値に設定された **CypherElement** 属性を持つ <attribute> 要素を追加することにより、暗号化方式のアルゴリズムを指定します。

2. パスワード、salt、iteration count を調節します

ステップ 1 にはシンプルな設定があり、暗号化または復号化に必要なパスワード、Salt、Iteration Count が MBean 定義内に含まれます。

KeyStorePass、Salt、IterationCount の値をご使用のデプロイメントに適するよう変更してください。

3. プラットフォームの起動

MBean、パスワード、ソルト、イテレーション数の設定後にサーバーを起動します。この設定を含むサーバープロファイルを指定してプラットフォームを起動してください。

```
[bin]$ ./run.sh -c PROFILE
```

4. JMX コンソールを開く

JMX コンソール (デフォルトで <http://localhost:8080/jmx-console/>) にナビゲートし、**org.jboss.security.plugins.JaasSecurityDomain** MBean を選択します。

5. LdapExtLoginModule の呼び出し

- a. `org.jboss.security.plugins.JaasSecurityDomain` ページで、`encode64(String password)` メソッドを見つけます。
- b. `LdapExtLoginModule` が使用する `password` のプレーンテキストバージョンをこのメソッドに渡します。
- c. `encode64(String password)` メソッドを呼び出します。
- d. 戻り値は、Base64 としてエンコードされたパスワードの暗号化バージョンです。

6. ログインモジュールの設定

ログインモジュール設定内では、次の `module-options` は以下のように設定されるべきです。

```
<module-option
name="jaasSecurityDomain">jboss.security:service=JaasSecurityDomain,
domain=jmx-console</module-option>
<module-option
name="bindCredential">2gx7gcAxcDuaHaJMg05AVo</module-option>
```

最初のオプションでは、手順 1 で設定された `JaasSecurityDomain` がパスワードを復号化するために使用されるよう指定されます。

`bindCredential` は、手順 5 で取得された暗号化 Base64 パスワードと置換されます。

第20章 ファイアウォール

JBoss にはリッスンするポートを開くソケットベースのサービスの多くが同梱されています。ここでは、サービスによって開かれることがあるため、ファイアウォールの背後で JBoss にアクセスする際に設定が必要となるポートの一覧を紹介します。表20.1「デフォルトの JBoss ポート」の表に、ポート、ソケットタイプ、関連サービス、ポートの簡単な説明が記載されています。

表20.1 デフォルトの JBoss ポート

ポート	タイプ	説明	サービス
1090	TCP	JMX MBeanServer へ接続するための RMI/JRMP ポート	jboss.remoting:service=JMXC onnectorServer, protocol=rmi
1098	TCP	クライアントプロキシからの RMI リクエストに対するネーミングサービスポート	jboss:service= Naming
1099	TCP	ネーミングサービスポート	jboss:service= Naming
1100	TCP	HA-JNDI サービスのポート	jboss:service= HAJNDI
1101	TCP	クライアントプロキシからの RMI リクエストに対する HA-JNDI サービスポート	jboss:service= HAJNDI
1102	UDP	自動ディスカバリリクエストの HA-JNDI マルチキャストポート	jboss:service= HAJNDI
1161	UDP	SNMP アダプタ MBean のポート	jboss.jmx:name=SnmpAgent, service=snmp,t ype=adaptor
1162	UDP	SNMP トラップレシーバのポート	jboss.jmx:name=SnmpAgent, service=trapd,t ype=logger
3528	TCP	Corba ORB の IIOP ポート	jboss:service= CorbaORB
3873	TCP	EJB3 リモータリングコネクタポート	jboss.remoting: type=Connecto r,name=Defaul tEjb3Connecto r,handler=ejb3

ポート	タイプ	説明	サービス
4444	TCP	レガシー RMI/JRMP インボーカーのポート	jboss:service=invoker,type=jrmp
4445	TCP	レガシー Pooled インボーカーに対するポート	jboss:service=invoker,type=pooled
4446	TCP	UnifiedInvoker の JBoss Remoting コネクタに対するポート	UnifiedInvoker Connector
4447	TCP	レガシー RMI/JRMP インボーカーの高可用性バージョンに対するポート	jboss:service=invoker,type=jrmpha
4448	TCP	レガシー Pooled インボーカーの高可用性バージョンに対するポート	jboss:service=invoker,type=pooledha
4457	TCP	JBoss Messaging 1.x の ポート	jboss.messaging:service=Connector,transport=bisocket
4712	TCP	JBossTS リカバリマネージャのポート	TransactionManager
4713	TCP	JBossTS トランザクションステータスマネージャのポート	TransactionManager
4714	TCP	JBossTS の固有プロセス ID を提供するためのポート	TransactionManager
5445	TCP	JBoss Messaging 2.x/HornetQ のポート	JBM2/HornetQ
5446	TCP	JBoss Messaging 2.x/HornetQ の SSL ポート	JBM2/HornetQ
5455	TCP	HornetQ バッチポート。 HornetQ User Guide の『Configuring the Netty transport』を参照。	HornetQ
5465	TCP	HornetQ バックアップサーバーポート	HornetQ
5466	TCP	HornetQ の バックアップ SSL ポート	HornetQ
5475	TCP	HornetQ のバックアップバッチポート	HornetQ

ポート	タイプ	説明	サービス
7500	TCP	Probe ユーティリティからの診断リクエストに対し JGroups がリスンするマルチキャストポート	JGroups
7600	TCP	JGroups tcp スタックに使用するポート	JGroups
7650	TCP	JGroups tcp-sync スタックによって使用されるポート	JGroups
7900	TCP	JGroups jbm-data によって使用されるポート。 このポートは、クラスタノード同士が通信するために使用され、通常はファイアウォールが設定されません。このオプションを希望する場合、他の UDP ポートを開く必要がある場合があるため留意してください。	JGroups
8009	TCP	AJP コネクタのポート	jboss:service=WebService
8080	TCP	JBoss Web HTTP コネクタポート (HTTPS および AJP ソケットの値にも関係する)	jboss.web:service=WebServer
8083	TCP	動的クラスとリソースローディングのポート	jboss:service=WebService
8443	TCP	JBoss Web HTTPS コネクタのポート	jboss.web:service=WebServer
45688	UDP	JGroups udp スタックの通信用のマルチキャストポート	JGroups
45689	UDP	JGroups udp-async スタックの通信用のマルチキャストポート	JGroups
45699	UDP	JGroups udp-sync スタックの通信用のマルチキャストポート	JGroups
45700	TCP	JGroups tcp スタックがディスカバリを実行するマルチキャストポート	JGroups
45701	TCP	JGroups tcp-sync スタックがディスカバリを実行するマルチキャストポート	JGroups
45710	UDP	JGroups jbm-data スタックがディスカバリを実行するマルチキャストポート	JGroups
53200	TCP	JGroups jbm_control スタックの FD_SOCKET プロトコルが使用するポート	JGroups

ポート	タイプ	説明	サービス
54200	TCP	JGroups udp スタックの FD_SOCKET プロトコルが使用するポート	JGroups
54225	UDP	JGroups udp-async スタックの FD_SOCKET プロトコルが使用するポート	JGroups
55200	UDP	JGroups udp スタックのポート	JGroups
55225	UDP	JGroups udp-async スタックが使用するポート	JGroups
55250	UDP	JGroups udp-sync スタックが使用するポート	JGroups
57600	UDP	JGroups udp スタックの FD_SOCKET プロトコルが使用するポート	JGroups
57650	TCP	JGroups tcp-sync スタックの FD_SOCKET プロトコルが使用するポート	JGroups
57900	TCP	JGroups jbm-data スタックの FD_SOCKET プロトコルが使用するポート	JGroups

第21章 コンソールとインボーカー

JBoss Enterprise Application Platform にはデプロイメントの管理機能への承認されていないアクセスを防ぐためにセキュアにまたは削除される必要がある管理アクセスポイントが複数同梱されています。本項では様々な管理サービスとそれらをセキュアにする方法を説明します。

21.1. JMX コンソール

deploy ディレクトリにある **jmx-console.war** は JMX マイクロカーネルに HTML ビューを提供します。このように、サーバーのシャットダウン、サービスの停止、新しいサービスのデプロイなどの管理操作へのアクセスを提供します。これは他の Web アプリケーションのようにセキュアにされるかまたは削除されるべきです。

21.2. ADMIN コンソール

Admin コンソールは Web コンソールを置換し、JBoss Operations Network セキュリティ要素を使用してコンソールをセキュアにします。詳細については、『『管理コンソールユーザーガイド (Admin Console User Guide)』』を参照してください。

21.3. HTTP インボーカー

deploy ディレクトリにある **http-invoker.sar** は、EJB および JNDI Naming サービスに RMI/HTTP アクセスを提供するサービスです。これには **MBeanServer** にディスパッチされるべき呼び出しを表すマーシャリングされた **org.jboss.invocation.Invocation** オブジェクトのポストを処理するサーブレットが含まれます。これは効果的に HTTP POST 要求により分離インボーカー操作をサポートする MBean へのアクセスを許可します。このアクセスポイントをセキュアにすることには、**http-invoker.sar/invoker.war/WEB-INF/web.xml** 記述子にある **JMXInvokerServlet** サーブレットをセキュアにすることが関係しています。デフォルトで **/restricted/JMXInvokerServlet** パスに対して定義されたセキュアなマッピングがあります。その他のパスを削除し、**http-invoker.sar/invoker.war/WEB-INF/jboss-web.xml** デプロイメント記述子の **http-invoker** セキュリティドメインの設定を行います。



注記

HTTP インボーカーをセキュアにするさらに詳しい情報は、『『管理コンソールクイックスタートガイド (Admin Console Quick Start Guide)』』を参照してください。

21.4. JMX インボーカー

jmx-invoker-service.xml は RMI/JRMP 分離インボーカーサービスを使用して RMI 互換性のあるインターフェースにより JMX MBeanServer インターフェースを公開する設定ファイルです。

21.5. サービス、分離インボーカーへのリモートアクセス

任意の機能性を統合することが可能な MBean サービスの概念に加えて、JBoss には分離インボーカーの概念もあります。その概念とは MBean サービスがクライアントからのリモートアクセスに対し任意のプロトコルを通じて機能的なインターフェースを公開できるということです。分離インボーカーの概念は、サービスがアクセスされるリモートティンギングとプロトコルがコンポーネントから独立した機能的な側面またはサービスであるということです。従って、ネーミングサービスを RMI/JRMP、RMI/HTTP、RMI/SOAP またはどの任意のカスタムトランスポートを通じて使用可能にできます。

関係するコンポーネントの概要から分離インボーカーアーキテクチャに関する話を始めましょう。分離インボーカーアーキテクチャの主要コンポーネントは 図21.1「分離インボーカーアーキテクチャの主要コンポーネント」 に表示されています。

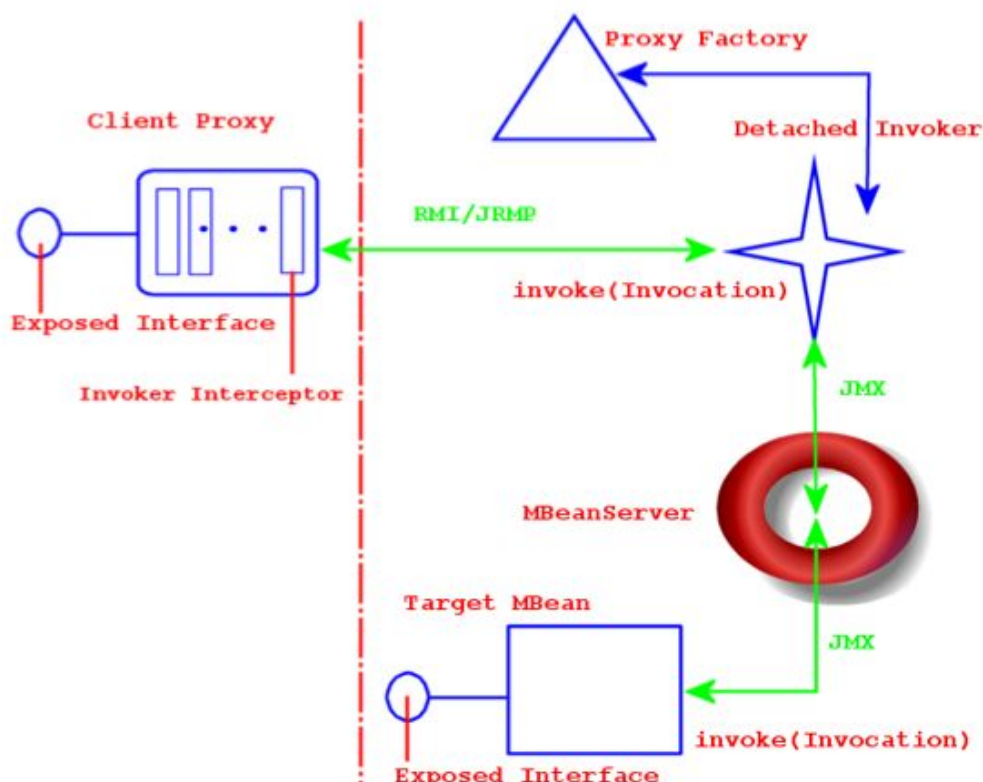


図21.1 分離インボーカーアーキテクチャの主要コンポーネント

クライアント側には、MBean サービスのインターフェースを公開するクライアントプロキシが存在します。これは EJB ホームとリモートインターフェースに使用される同じのスマートなコンパイルのない動的プロキシです。任意のサービスに対するプロキシと EJB との唯一の違いは、公開されたインターフェースのセットだけでなく、プロキシの中にあるクライアント側インターセプタです。クライアントインターセプタはクライアントプロキシの中にある四角形で表示されています。インターセプタはメソッド呼び出し / 戻り値の変換を可能にするパターンのアセンブリラインのタイプです。クライアントは検索メカニズム、通常は JNDI を通じてプロキシを取得します。RMI は 図21.1「分離インボーカーアーキテクチャの主要コンポーネント」 に示されていますが、公開されたインターフェースとそのタイプの唯一の実際の要件は、JNDI のクライアントサーバーだけでなくトランスポートレイヤの間でシリアル化が可能ということです。

トランスポートレイヤの選択は、クライアントプロキシの最後のインターセプタで決まります。それは、図21.1「分離インボーカーアーキテクチャの主要コンポーネント」 でインボーカーインターセプタと呼ばれます。インボーカーインターセプタにはサーバー側の分離インボーカー MBean サービスのトランスポート固有スタブに関する参照が含まれます。インボーカーインターセプタは目的の MBean と同じ VM 内で発生する呼び出しの最適化も処理します。インボーカーインターセプタがこれが該当ケースとして検出すると、単に目的の MBean に呼び出しを渡す参照渡しインボーカーに呼び出しが渡されます。

分離インボーカーサービスの役割は、分離インボーカーが処理するトランスポートにより汎用呼び出し操作を使用可能にすることです。**Invoker** インターフェースは汎用呼び出し操作を図解します。

```
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;
}
```

Invoker インターフェースは **Remote** を拡張し RMI と互換性を持たせますが、これはインボーカーが RMI サービススタブを公開する必要があるという意味ではありません。分離インボーカーサービスは、特定のトランスポートに対して **org.jboss.invocation.Invocation** オブジェクトとして表される呼び出しを受け入れるトランスポートゲートウェイとして働きます。インボーカーサービスは呼び出しをアンマーシャリングし、呼び出しを [図21.1「分離インボーカーアーキテクチャの主要コンポーネント」](#) の **Target MBean** で表されるデスティネーション MBean サービスに転送し、クライアントへの転送コールバックから生じる戻り値または例外をマーシャリングします。

Invocation オブジェクトは単にメソッド呼び出しコンテキストを表したものです。これには目的の MBean 名、メソッド、メソッド引数、プロキシファクトリによりプロキシに関連付けられた情報のコンテキスト、クライアントプロキシインターセプタにより呼び出しと関連付けられたデータの任意のマップが含まれます。

クライアントプロキシの設定はサーバー側プロキシファクトリ MBean サービスで行われ、[図21.1「分離インボーカーアーキテクチャの主要コンポーネント」](#) のプロキシファクトリコンポーネントで示されています。プロキシファクトリは次のタスクを実行します。

- 目的の MBean が公開したいインターフェースを実装する動的プロキシを作成します。
- クライアントプロキシインターセプタと動的プロキシハンドラを関連付けます。
- 呼び出しコンテキストと動的プロキシを関連付けます。これには目的の MBean、分離インボークースタブ、プロキシ JNDI 名が含まれます。
- プロキシを JNDI にバインドすることによりクライアントに対しプロキシを使用可能にします。

[図21.1「分離インボーカーアーキテクチャの主要コンポーネント」](#) の最後のコンポーネントは、呼び出しのインターフェースをリモートクライアントに公開したい**目的の MBean** サービスです。特定のインターフェースを通じて MBean サービスがアクセス可能となるために必要なステップは以下のとおりです。

- シグネチャと一致する JMX 操作を定義します。 **public Object invoke(org.jboss.invocation.Invocation) throws Exception**
- **org.jboss.invocation.MarshalledInvocation.calculateHash** メソッドを使用して、公開されたインターフェース **java.lang.reflect.Method** から長いハッシュ表現にマップする **HashMap<Long, Method>** を作成します。
- **invoke(Invocation)** JMX 操作を実装し、インターフェースメソッドのハッシュマッピングを使用して、呼び出されたメソッドの長いハッシュ表現から公開されたインターフェースの **java.lang.reflect.Method** に変換します。リフレクションを使用して、公開されたイン

ターフェースを実際に実装する MBean サービスと関連付けられたオブジェクトに関する実際の呼び出しを実行します。

21.5.1. 分離インボーカーの例、MBeanServer インボーカーアダプターサービス

MBean サービスにリモートアクセスを提供する必要があるステップの例として、本項では RMI/JRMP によるアクセスのための `org.jboss.jmx.connector.invoker.InvokerAdaptorService` とその設定について記載しています。

例21.1 InvokerAdaptorService MBean

`InvokerAdaptorService` は分離インボーカーパターンで目的の MBean の役割を果たすために存在する簡易 MBean サービスです。

```
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);
        }
    }
}
}

```

InvokerAdaptorServiceMBean を構成するコンポーネントを理解しやすくするために、コードは論理ブロックに分けられ、各ブロックが動作する方法についてのコメントが付いています。

例21.2 ブロック 1

```
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;
    }
    ...

```

InvokerAdaptorService の **InvokerAdaptorServiceMBean** Standard MBean インターフェースには単一の **ExportedInterface** 属性と単一の **invoke(Invocation)** 操作があります。

ExportedInterface

この属性によりサービスがクライアントに公開するインターフェースのタイプのカスタマイズが可能になります。これはメソッド名とシグネチャに関しては **MBeanServer** クラスと互換性がある必要があります。

invoke(Invocation)

この操作は、目的の MBean サービスが分離インボーカーパターンに参加するために公開しなければならない必須のエントリポイントです。この操作は **InvokerAdaptorService** へのアクセスを提供するように設定された分離インボーカーサービスにより呼び出されます。

例21.3 ブロック 2

```

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()));
}

```

このコードブロックは

org.jboss.invocation.MarshalledInvocation.calculateHash(Method) ユーティリティメソッドを使用して **exportedInterface** クラスの **HashMap<Long, Method>** を構築します。

java.lang.reflect.Method インスタンスはシリアル化できないため、シリアル化不可能な **Invocation** クラスの **MarshalledInvocation** バージョンは、クライアントとサーバー間の呼び出しをマーシャリングするために使用されます。**MarshalledInvocation** は **Method** インスタンス

スとそれらに対応するハッシュ表現とを置換します。サーバー側では、Method マッピングへのハッシュが何であるかを **MarshaledInvocation** に伝える必要があります。

このコードブロックは **InvokerAdaptorService** サービス名とそのハッシュコード表現の間のマッピングを作成します。これは分離 invoker により使用され、**Invocation** の目的 MBean **ObjectName** が何であるかを決定します。

ObjectName は作成するのに比較的高価なオブジェクトであるため、目的 MBean 名が **Invocation** に保存されると、その hashCode として保存されます。**org.jboss.system.Registry** はハッシュコードを **ObjectName** マッピングに保存するためにインボーカーが使用するコンストラクタのようなグローバルマップです。

例21.4 ブロック 3

```
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("<div>JMX_OBJECT_NAME</div>");
    if (objectName != null) {
        // Obtain the ClassLoader associated with the MBean
        deployment
        newCL = (ClassLoader)
            server.invoke(mbeanRegistry, "<div>getValue</div>",
                new Object[] { objectName, CLASSLOADER
            },
                new String[] {
            ObjectName.class.getName(),
            "<div>java.lang.String</div>"});
    }

    if (newCL != null && newCL != oldCL) {
        thread.setContextClassLoader(newCL);
    }
}
```

このコードブロックは MBeanServer 操作が実行されている MBean の名前を取得し、MBean の SAR デプロイメントに関連付けられているクラスローダーを検索します。この情報は JBoss JMX 実装固有クラスの **org.jboss.mx.server.registry.BasicMBeanRegistry** で取得可能です。

通常は MBean が適切なクラスローディングコンテキストを確立することが必要です。その理由は、呼び出しと関連付けられるタイプをアンマーシャリングするために必要なクラスローダーへのアクセスを分離インボーカープロトコルレイヤが持っていない場合があるためです。

例21.5 ブロック 4

-

```

...
    try {
        // Set the method hash to Method mapping
        if (invocation instanceof MarshalledInvocation) {
            MarshalledInvocation mi = (MarshalledInvocation)
invocation;
            mi.setMethodMap(marshalledInvocationMapping);
        }
    }
...

```

このコードブロックでは呼び出し引数がタイプ **MarshalledInvocation** である場合 **ExposedInterface** クラスメソッドのハッシュをメソッドマッピングにインストールします。例 21.3「ブロック 2」で計算されるメソッドマッピングはここで使用されます。

第 2 のマッピングは **ExposedInterface** メソッドから **MBeanServer** クラスの一致するメソッドへ実行されます。**InvokerServiceAdaptor** は任意のインターフェースを可能にするため **MBeanServer** クラスからの **ExposedInterface** を切り離します。これが必要なのは標準 **java.lang.reflect.Proxy** クラスはインターフェースをプロキシすることのみ可能であるためです。また、**MBeanServer** メソッドのサブセットを公開し、**java.rmi.RemoteException** などのトランスポート固有の例外を **ExposedInterface** メソッドシングネチャに追加することだけが可能になります。

例21.6 ブロック 5

```

...
    // 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());
        }
    }

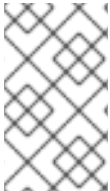
    return value;
} finally {
    if (newCL != null &&& newCL != oldCL) {
        thread.setContextClassLoader(oldCL);
    }
}
}
}

```

このコードブロックは MBeanServer メソッド呼び出しをそれがデプロイされる

InvokerAdaptorService MBeanServer インスタンスにディスパッチします。サーバーのインスタンス変数は **ServiceMBeanSupport** スーパークラスから引き継がれます。

呼び出しにより投げられたすべての宣言された例外のアンラップを含め、反射型呼び出しから生じる例外は処理されます。MBean コードは MBeanServer メソッド呼び出しが成功した結果の戻りにより完了します。



注記

InvokerAdaptorService MBean はトランスポート固有の詳細は直接扱いません。Method マッピングへのメソッドハッシュの計算がありますが、これはトランスポート独自の詳細です。

ここで、「RMI を使用した JMX への接続」で見たように RMI/JRMP により同じ **org.jboss.jmx.adaptor.rmi.RMIAdaptor** インターフェースを公開するために **InvokerAdaptorService** がどのように使用されるかを見てみましょう。

jmx-invoker-adaptor-service.sar デプロイメントのデフォルト設定にあるプロキシファクトリと **InvokerAdaptorService** 設定を表示することから始めます。例21.7「デフォルトの **jmx-invoker-adaptor-server.sar** デプロイメント記述子」はこのデプロイメントに対する **jboss-service.xml** 記述子を示しています。

例21.7 デフォルトの **jmx-invoker-adaptor-server.sar** デプロイメント記述子

```
<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>
    </interceptors>
</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>

```

最初の MBean **org.jboss.invocation.jrmp.server.JRMPProxyFactory** は RMI/JRMP プロトコルに対しプロキシを作成するプロキシファクトリ MBean サービスです。例21.7「デフォルトの **jmx-invoker-adaptor-server.sar** デプロイメント記述子」で示されているようにこのサービスの設定が示していることは次のとおりです。JRMPInvoker が分離インボーカーとして使用され、**InvokerAdaptorService** は要求が転送される目的の Mbean であること、プロキシが **RMIAdaptor** インターフェースに公開され、プロキシが名前 **jmx/invoker/RMIAdaptor** で JNDI にバインドされ、プロキシが3つのインターセプタ **ClientMethodInterceptor**、**InvokerAdaptorClientInterceptor**、**InvokerInterceptor** を含むということです。**InvokerAdaptorService** の設定はサービスが公開している **RMIAdaptor** インターフェースを単に設定することです。

RMI/JRMP による **InvokerAdaptorService** を公開するための設定の最後は分離インボーカーです。使用する分離インボーカーはホームとリモート呼び出しに EJB コンテナが使用する標準 RMI/JRMP インボーカーであり、これは **conf/jboss-service.xml** 記述子で設定された **org.jboss.invocation.jrmp.server.JRMPInvoker** MBean サービスです。同じサービスインスタンスを使用できることはインボーカーの分離という性質を強調しています。プロキシが公開するインターフェースまたはプロキシが活用するサービスに関わらず、JRMPInvoker は単にすべての RMI/JRMP プロキシに対して RMI/JRMP エンドポイントとして働きます。

付録A /usr/sbin/alternatives ユーティリティによるデフォルト JDK の設定

/usr/sbin/alternatives は同じ機能性を提供する異なるソフトウェアパッケージを管理するツールです。**Red Hat Enterprise Linux** は **/usr/sbin/alternatives** を使用して、1 度に 1 つの Java Development Kit だけがシステムデフォルトとして設定されるようにします。



重要

Red Hat Network から Java Development Kit をインストールすると、通常システムは自動的に構成されます。ただし JDK が複数インストールされる場合は、**/usr/sbin/alternatives** に競合する設定が含まれる可能性があります。**/usr/sbin/alternatives** コマンド構文については [手順 A.1 「/usr/sbin/alternatives を使用したデフォルトの JDK 設定」](#) を参照してください。

手順A.1 /usr/sbin/alternatives を使用したデフォルトの JDK 設定

1. ルートユーザーになります

/usr/sbin/alternatives はルート権限で実行する必要があります。**su** コマンドまたは他の方法でこの権限を取得します。

2. java を設定します

コマンド **/usr/sbin/alternatives --config java** を入力します。

次に画面上の指示に従い、**java** の適切なバージョンが選択されていることを確認します。[表 A.1 「java の代替コマンド」](#) では異なる JDK の関連するコマンド設定を示しています。

表A.1 java の代替コマンド

JDK	代替コマンド
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. javac を設定します

コマンド **/usr/sbin/alternatives --config javac** を入力します。

画面上の指示に従い、**javac** の適切なバージョンが選択されていることを確認します。[表 A.2 「javac の代替コマンド」](#) では異なる JDK の適切なコマンド設定を示しています。

表A.2 javac の代替コマンド

JDK	代替コマンド
OpenJDK 1.6	/usr/lib/jvm/java-1.6.0-openjdk/bin/javac
Sun Microsystems JDK 1.6	/usr/lib/jvm/java-1.6.0-sun/bin/javac

4. 追加ステップ : `java_sdk_1.6.0` を設定します

Sun Microsystems JDK 1.6 では以下のコマンドを追加で実行する必要があります。

```
/usr/sbin/alternatives --config java_sdk_1.6.0
```

画面上の指示に従い、適切な `java_sdk`、`/usr/lib/jvm/java-1.6.0-sun` が選択されていることを確認します。

付録B 改訂履歴

改訂 5.1.2-2.400
Rebuild with publican 4.0.0

2013-10-31

Rüdiger Landmann

改訂 5.1.2-2
Rebuild for Publican 3.0

2012-07-18

Anthony Towns

改訂 5.1.2-100 **Thu 8 December 2011** **Russell Dickenson**
JBoss Enterprise Application Platform 5.1.2 GA の変更が含まれます。本ガイドの内容の変更については、『『リリースノート 5.1.2 (Release Notes 5.1.2)』』を参照してください。