BEA WebLogic Server

WebLogic Security
プログラマーズ ガイド

BEA WebLogic Server 6.1
マニュアルの日付：2002 年 6 月 24 日

著作権

限定的権利条項

WebLogic Security プログラマーズ ガイド

マニュアルの版数	マニュアルの日付	ソフトウェアのバージョン
なし	2002 年 6 月 24 日	BEA WebLogic Server バージョン 6.1

目次

このマニュアルの内容

1. WebLogic Security の概要

2. セキュリティの基礎概念

3. WebLogic Server デプロイメントのセキュリティ対策

4. WebLogic Seurity SPI を使用したプログラミング

このマニュアルの内容

このマニュアルでは、BEA WebLogic Server™ のセキュリティ機能に関連する概念を紹介し、その機能を使用して WebLogic Server デプロイメントをセキュリティで保護する方法、および WebLogic Security サービス プロバイダ インタフェース（SPI）のアプリケーション プログラミング インタフェース（API）について説明します。

このマニュアルの内容は以下のとおりです。

- 第 1 章「WebLogic Security の概要」では、WebLogic Security の機能の概要について説明します。
- 第 2 章「セキュリティの基礎概念」では、WebLogic Server のセキュリティ機能に関連する概念の詳細について説明します。
- 第 3 章「WebLogic Server デプロイメントのセキュリティ対策」では、WebLogic Server デプロイメントをセキュリティで保護する方法、および一般的なセキュリティ攻撃を阻止する方法について説明します。
- 第 4 章「WebLogic Seurity SPI を使用したプログラミング」では、WebLogic Server のセキュリティ ポリシーを定義する方法、およびセキュリティ保護された状態で WebLogic Server に接続する方法について説明します。

対象読者

このマニュアルは、WebLogic Server デプロイメントにセキュリティ機能を導入するプログラマを対象としています。

e-docs Web サイト

BEA 製品のドキュメントは、BEA の Web サイトで入手できます。BEA のホーム ページで [製品のドキュメント] をクリックします。

このマニュアルの印刷方法

Web ブラウザの [ファイル | 印刷] オプションを使用すると、Web ブラウザからこのマニュアルのメイン トピックを一度に 1 つずつ印刷できます。

このマニュアルの PDF 版は、Web サイトで入手できます。PDF を Adobe Acrobat Reader で開くと、マニュアルの全体（または一部分）を書籍の形式で印刷できます。PDF を表示するには、WebLogic Server ドキュメントのホーム ページを開き、[ドキュメントのダウンロード] をクリックして、印刷するマニュアルを選択します。

Adobe Acrobat Reader は、Adobe の Web サイト（http://www.adobe.co.jp）から無料で入手できます。

関連情報

BEA の Web サイトでは、WebLogic Server の全マニュアルを提供しています。

サポート情報

BEA のドキュメントに関するユーザからのフィードバックは弊社にとって非常に重要です。質問や意見などがあれば、電子メールで docsupport-jp@bea.com までお送りください。寄せられた意見については、ドキュメントを作成および改訂する BEA の専門の担当者が直に目を通します。

電子メールのメッセージには、ご使用のソフトウェア名とバージョン名、およびマニュアルのタイトルと作成日付をお書き添えください。本バージョンの BEA WebLogic Server について不明な点がある場合、または BEA WebLogic Server のインストールおよび動作に問題がある場合は、BEA WebSUPPORT（www.bea.com）を通じて BEA カスタマ サポートまでお問い合わせください。カスタマ サポートへの連絡方法については、製品パッケージに同梱されているカスタマ サポート カードにも記載されています。

カスタマ サポートでは以下の情報をお尋ねしますので、お問い合わせの際はあらかじめご用意ください。

- お名前、電子メール アドレス、電話番号、ファクス番号
- 会社の名前と住所
- お使いの機種とコード番号
- 製品の名前とバージョン
- 問題の状況と表示されるエラー メッセージの内容

表記規則

このマニュアルでは、全体を通して以下の表記規則が使用されています。

表記法	適用
〔Ctrl〕+〔Tab〕	同時に押すキーを示す。
斜体	強調または本のタイトルを示す。

等幅テキスト	コード サンプル、コマンドとそのオプション、データ構造体とそのメンバー、データ型、ディレクトリ、およびファイル名とその拡張子を示す。等幅テキストはキーボードから入力するテキストも示す。
	例 :
	`import java.util.Enumeration;`
	`chmod u+w *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

斜体の等幅テキスト	コード内の変数を示す。 例 : `String *CustomerName*;`
すべて大文字のテキスト	デバイス名、環境変数、および論理演算子を示す。 例 : LPT1 BEA_HOME OR
{ }	構文内の複数の選択肢を示す。

表記法	適用
[ ]	構文内の任意指定の項目を示す。 例 : `java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]`
|	構文の中で相互に排他的な選択肢を区切る。 例 : `java weblogic.deploy [list|deploy|undeploy|update] password {application} {source}`
...	コマンドラインで以下のいずれかを示す。 ■ 引数を複数回繰り返すことができる。 ■ 任意指定の引数が省略されている。 ■ パラメータや値などの情報を追加入力できる。
. . .	コード サンプルまたは構文で項目が省略されていることを示す。

1 WebLogic Security の概要

以下の節では、WebLogic Security について概説します。

- WebLogic Security の機能
- WebLogic のセキュリティ アーキテクチャ
- BEA Tuxedo に対するクライアントとしての WebLogic Server の使い方

WebLogic Security の機能

セキュリティとは、コンピュータに保存されているデータまたはコンピュータ間でやりとりされるデータが危険にさらされないことを保証する技術です。ほとんどのセキュリティ対策は、証明データとデータ暗号化を利用します。一般に証明データは、ユーザに特定のアプリケーションまたはシステムへのアクセスを許可する秘密の単語または句です。データ暗号化とは、データを解釈不能な形式に変換することです。

電子商取引（e- コマース）向けアプリケーションなどの分散アプリケーションでは、悪意のある何者かがデータを横取りし、処理を混乱させ、不正な入力を行う起点となる可能性があるアクセス ポイントを多数提供します。そのため、企業の分散化が進むにつれて、セキュリティ攻撃を受ける可能性も増大します。したがって、アプリケーションの分散に伴い、その基盤となる分散コンピューティング ソフトウェアによってセキュリティを実現することがますます重要になります。

WebLogic Server のセキュリティ機能を利用すると、Web ブラウザ、Java クライアント、およびほかの WebLogic Server から WebLogic Server にセキュアな接続を確立できます。さらに、セキュアな接続を経由して、WebLogic Server を BEA Tuxedo に対するクライアントとして使用することもできます。

具体的には、WebLogic Server の持つセキュリティ機能は以下のとおりです。

- WebLogic Server リソースの保護を目的とした、ユーザ、グループ、ACL、およびパーミッションの論理グループを表すセキュリティ レルム。Windows NT、UNIX、および LDAP セキュリティ ストアを使用できるように、デフォルトのセキュリティ レルムまたは代替セキュリティ レルムを使用することができます。その他に、独自に作成したカスタム セキュリティ レルムもサポートされます。
- WebLogic Server のリソースへのアクセスを要求しているクライアントの認証。認証を行うには、ユーザ名とパスワードの組み合わせか、またはセキュア ソケット レイヤ（SSL）接続の一部として WebLogic Server に提示される X.509 デジタル証明書内の ID でクライアントを認証する場合は、デジタル証明書を利用します。
- アクセス制御リスト（ACL）を利用したユーザおよびグループの認可。
- 認証用の Java Authentication and Authorization Service（JAAS）アプリケーション プログラミング インタフェース（API）。WebLogic Server の JAAS 実装では、LoginContext 認証と Subject 認可を提供します。JAAS 認可はサポートされていません。
- SSL プロトコルによるデータの整合性と機密性。クライアントは、Hypertext Transfer Protocol（HTTP）、BEA 独自の T3 プロトコル、または Internet Inter-ORB（IIOP）プロトコル上の Remote Method Invocation（RMI）を使用して、WebLogic Server と SSL セッションを確立できます。
- ログイン試行の失敗、認証リクエスト、デジタル証明書の拒否、無効な ACL などのイベントの監査。
- クライアントの出所（ホスト名またはネットワーク アドレス）またはプロトコルに基づいてクライアントのリクエストを受け付けたり拒否したりするためのクライアント接続のフィルタ処理。
- WebLogic Enterprise Connectivity（WLEC）を使用した WebLogic Server セキュリティ レルムから BEA Tuxedo ドメインへのセキュリティ コンテキストの伝播。この機能を使用すると、リクエストを行った WebLogic Server ユーザに関するセキュリティ情報を、信頼された WLEC 接続プールの一部をなすネットワーク接続を介して BEA Tuxedo ドメインに伝播できます。

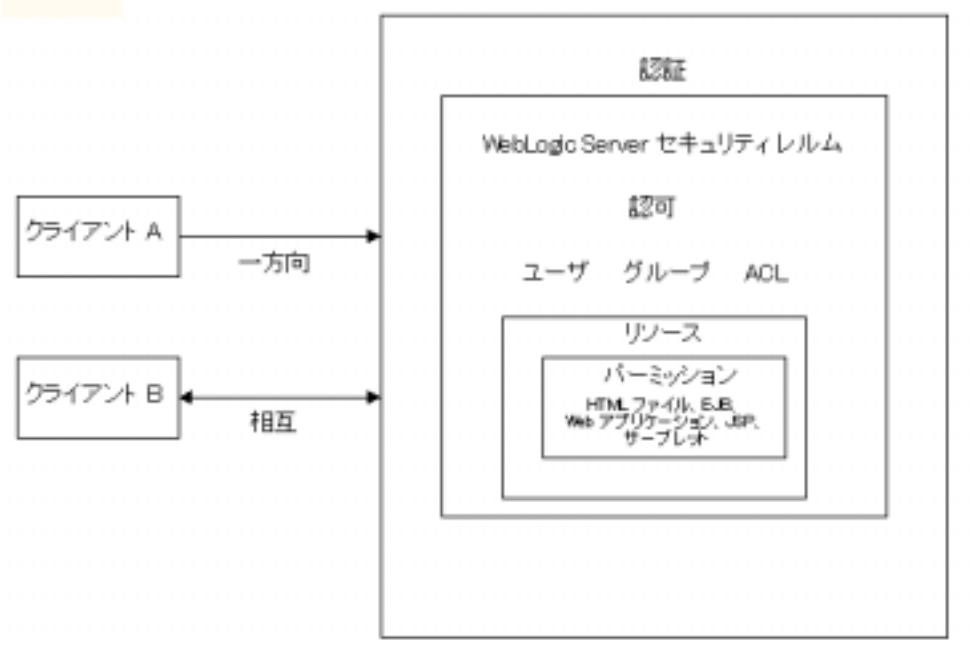
WebLogic EJB のセキュリティの詳細については、『WebLogic エンタープライズ JavaBeans プログラマーズ ガイド』を参照してください。

Web アプリケーションでのセキュリティの詳細については、『Web アプリケーションのアセンブルとコンフィグレーション』を参照してください。

WebLogic のセキュリティ アーキテクチャ

WebLogic Server のセキュリティ アーキテクチャは、ユーザの認可と認証を基本にしています。図 1-1 に、WebLogic Server のセキュリティ アーキテクチャを示します。

図 1-1　WebLogic Server のセキュリティ アーキテクチャ



認証は、WebLogic Server 環境の最初のセキュリティ レイヤです。認証とは、接続完了前に、エンティティの ID を確認するプロセスです。認証は、WebLogic Server 環境にアクセスするユーザを保護します。

WebLogic Server のデフォルトの認証方式は、一方向認証です。一方向認証は、ユーザが個人データを提出する前にセキュアな接続の確立を希望するインターネット上では一般的な方式です。WebLogic Server がクライアントのリクエストを受け取ると、WebLogic Server は提示されたユーザ名とパスワードを、WebLogic Server セキュリティ レルムで定義されているユーザ名とパスワードに照合して認証します。ユーザ名とパスワードが正しいことが確認されると、クライアントは、WebLogic Server 環境へのアクセスを許可されます。

SSL プロトコルが使用されている場合、セキュアな接続を確立するには、WebLogic Server がデジタル証明書チェーンをクライアントに提示して、ID を証明する必要があります。クライアントは信頼された認証局のデジタル証明書一式を使用して、WebLogic Server が提示したデジタル証明書の認証されたものであることを確認します。WebLogic Server 上の SSL プロトコルが相互 SSL 用にコンフィグレーションされている場合は、クライアントも ID の正当性を確認するデジタル証明書のチェーンを渡す必要があります。

WebLogic Server 環境内では、使用可能なリソースにアクセスするユーザを認可によって制御します。認可は、ユーザおよびグループの定義と WebLogic Server 内のリソースに割り当てられたパーミッションに基づいています。リソースには、イベント、サーブレット、JDBC 接続プール、パスワード、JMS 送り先、および JNDI コンテキストがあります。WebLogic Server はセキュリティ レルムを使用して、ユーザ、グループ、ACL、およびパーミッションをリソースとして論理的に編成します。WebLogic Server のリソースは、1 つのセキュリティ レルムの ACL によって保護されます。ユーザはセキュリティ レルムで定義されていないと、そのセキュリティ レルムに属するリソースにアクセスできません。

ユーザがリソースに対してメソッドを実行しようとすると、アクセスを許可するかどうかを決めるために次の手順が取られます。

1. リソースが保護されていて、ユーザが認証を受けていない場合、ユーザは認証を要求されます。認証に失敗した場合、リクエストは拒否されます。
2. WebLogic Server がそのメソッドを呼び出したユーザを識別します。ユーザを識別できない場合、リクエストは拒否されます。
3. WebLogic Server は、必要なパーミッションのセットを調べて、WebLogic Server のリソースに対してメソッドを呼び出します。
4. 呼び出しを行ったユーザが必要なパーミッションのうちの少なくとも 1 つを持っている場合、WebLogic Server はそのメソッドの呼び出しを許可します。

次の節では、WebLogic Server が各種の接続に応じてセキュリティを提供する方法について説明します。

Web ブラウザとの接続

Web ブラウザは、次のように WebLogic Server と対話します。

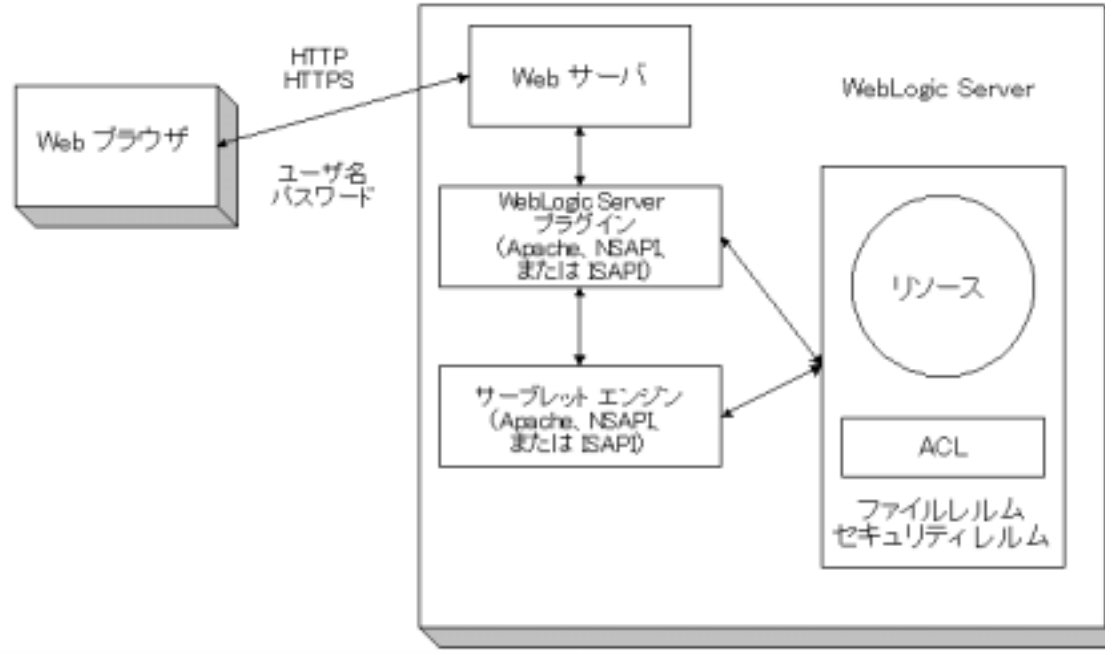
1. ユーザは、Web ブラウザでリソースの URL を入力して、WebLogic Server のリソースを呼び出します。
2. WebLogic Server の Web サーバはリクエストを受け取ります。WebLogic Server では独自の Web サーバを用意していますが、Apache Server、Microsoft Internet Information Server、および Netscape Enterprise Server も Web サーバとして使用できます。
3. Web サーバは、要求された WebLogic Server リソースが ACL によって保護されているかどうかチェックします。WebLogic Server リソースが保護されている場合、Web サーバは確立されている HTTP 接続を使用して、ユーザのユーザ ID とパスワードを要求します。
4. ユーザの Web ブラウザが WebLogic Server からのリクエストを受け取ると、ユーザに対してユーザ ID とパスワードを要求します。
5. Web ブラウザは、ユーザ ID とパスワードと一緒にリクエストを再送信します。
6. Web サーバはリクエストを Web サーバ プラグインに転送します。WebLogic Server では、Web サーバ用に以下のプラグインを提供します。
   - Apache-Weblogic Server プラグイン
   - Netscape Server アプリケーション プログラミング インタフェース（NSAPI）
   - Internet Information Server アプリケーション プログラミング インタフェース（ISAPI）

   Web サーバ プラグインは、HTTP プロトコルを使用して、ユーザから受け取った認証データ（ユーザ ID とパスワード）と一緒に認証リクエストを WebLogic Server 内のリソースに送信することで認証を行います。
7. 認証に成功すると、WebLogic Server は、ユーザがそのリソースへのアクセスに必要なパーミッションを持っているかどうかを調べます。
8. 認可が成功すると、サーブレット エンジンがリクエストを遂行します。サーブレット エンジンは、WebLogic Server に入っています。

9. サーブレットに対してメソッドを呼び出す前に、サーブレット エンジンはセキュリティ チェックを実行します。サーブレット エンジンは、このチェックで、ユーザの資格をセキュリティ コンテキストから取り出し、ユーザがサーブレットに対してメソッドを呼び出す認可を持っていることを確認します。

図 1-2 に、Web ブラウザのセキュア ログイン プロセスを示します。

図 1-2 Web ブラウザのセキュア ログイン



HTTPS プロトコルは、ここで説明している使い方でさらに高いレベルのセキュリティを提供します。SSL プロトコルは、Web ブラウザと WebLogic Server との間で転送されるデータを暗号化するので、ユーザ ID とパスワードはクリア テキストでは転送されません。したがって、SSL プロトコルを使ってユーザのパスワードが保護されている場合、WebLogic Server はユーザを認証できます。

詳細については、以下の節を参照してください。

- 「セキュリティの管理」
- 「Apache サーバ（プロキシ）プラグインのコンフィグレーション」
- 「Microsoft-IIS（プロキシ）プラグインのコンフィグレーション」
- 「Netscape（プロキシ）プラグインのコンフィグレーション」

サーブレット、JSP、EJB、RMI オブジェクト、および Java アプリケーションとの接続

サーブレット、JSP、EJB、RMI オブジェクト、および Java アプリケーションは、Java Authentication and Authorization Service（JAAS）を使用して、WebLogic Server を認証します。JAAS は、Java 2 Software Development Kit の標準拡張機能です。JAAS の認証コンポーネントを使用すると、コードの実行形態が Java アプリケーション、JSP、EJB、RMI オブジェクト、またはサーブレットのいずれであっても、クライアントの ID を安全かつ確実に管理できます。WebLogic Server では、JAAS は既存の Security サービス プロバイダ インタフェース（SPI）の上の層に追加され、レルムベースの認可をそのまま利用できます。これが必要となる理由は、WebLogic Server が JAAS の認可コンポーネントを提供しないからです。すべての認可チェックは、基礎となるセキュリティ レルムを通して行われます。

JAAS 認証を使用する場合、Java クライアントは、Configuration オブジェクトを参照する LoginContext オブジェクトをインスタンス化することで認証プロセスを有効にします。Configuration オブジェクトは、クライアントの認証に用いるコンフィグレーション済み LoginModule を指定します。LoginModule オブジェクトは、クライアントの資格を要求して確認します。WebLogic Server で使用する認証メカニズムのタイプごとに LoginModule オブジェクトを作成する必要があることに注意してください。たとえば相互認証を使用する場合は、資格を要求するとともに提供する LoginModule オブジェクトを作成する必要があります。WebLogic Server では、LoginModule オブジェクトを提供していません。

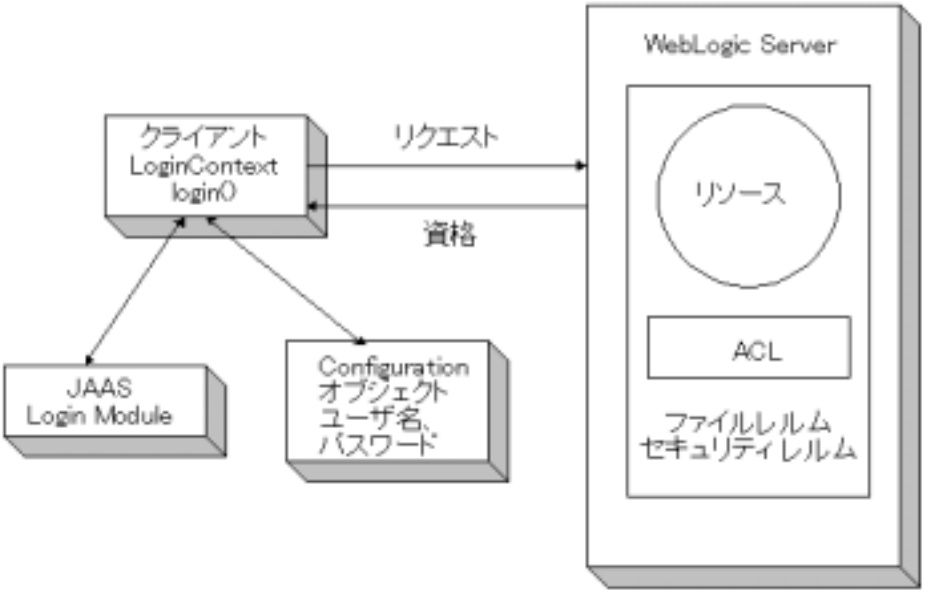
クライアント（サーブレット、JSP、EJB、RMI オブジェクト、および Java アプリケーション）は、次の方法で、WebLogic Server と安全に対話します。

1. クライアントは、LoginModule オブジェクトと CallbackHandler オブジェクトをインスタンス化する LoginContext を作成します。
   - LoginContext が、LoginModule とその実行順序を指定する Configuration オブジェクトを参照します。
   - CallbackHandler オブジェクトが、ユーザからの入力（パスワードやデジタル証明書ファイルの名前など）を集めて、LoginModule に渡します。
2. クライアントは、LoginContext オブジェクトの `login()` メソッドを呼び出します。次に、`login()` メソッドが指定された LoginModule を呼び出します。
3. LoginModule は、クライアントの資格を要求して確認します。
4. 認証に成功すると、WebLogic Server は、クライアントが要求したリソースへのアクセスに必要なパーミッションを持っているかどうかを調べます。パーミッションは、WebLogic Server セキュリティ レルムで定義されたリソースの ACL で調べます。
5. ACL の認可が成功すると、クライアントからのリクエストが WebLogic Server によって遂行されます。

パスワード認証を実装する LoginModule を使用する場合は、SSL プロトコルを利用するよう WebLogic Server をコンフィグレーションできます。SSL プロトコルは、ユーザ ID とパスワードがクリア テキストでは転送されないように、クライアントと WebLogic Server との間で転送されるデータを暗号化します。

図 1-3 に、サーブレット、JSP、EJB、RMI オブジェクト、および Java アプリケーションのセキュア ログイン プロセスを示します。

図 1-3　Java クライアント用のセキュア ログイン



WebLogic Server は、別の WebLogic Server に対するクライアントとして機能できます。この場合、WebLogic Server はクライアントと同じ認証方法を使用します。

注意： WebLogic Server では、認証を渡す方法として JNDI メソッドもサポートしています。ただし、この機能は JAAS 認証に置き換えられます。

詳細については、以下の節を参照してください。

- 「WebLogic Seurity SPI を使用したプログラミング」
- 「セキュリティの管理」

管理サーバとの接続

WebLogic Server では、管理サーバとは、すべてのコンフィグレーション情報の中央ソースとしての役割を持つ WebLogic Server のことです。管理サーバは、1 つの WebLogic Server または複数の WebLogic Server から構成されるクラスタのコンフィグレーション情報を格納できます。管理サーバとその他の WebLogic Server との接続は、盗聴、改ざん、反復、偽装などの攻撃から保護する必要があります。

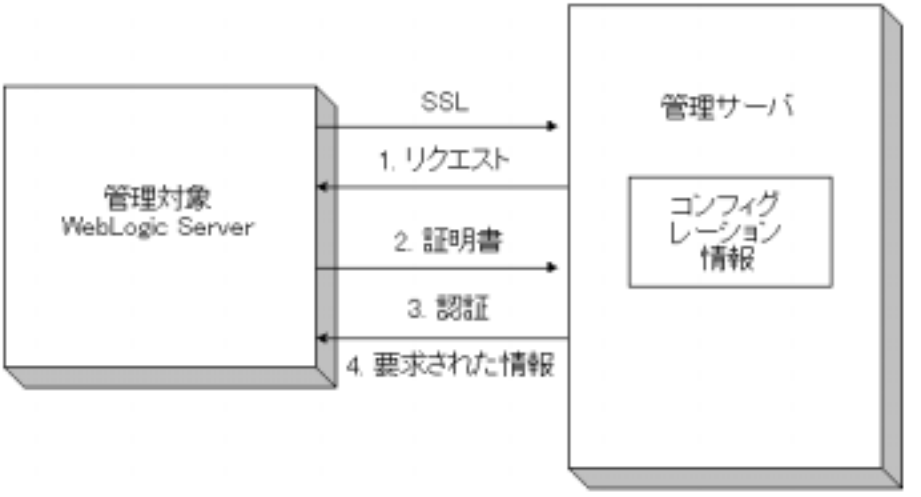
SSL プロトコルと証明書を使用する場合、管理サーバは、管理対象の WebLogic Server が起動されるたびに自身のデジタル証明書をそのサーバ対して提示します。次に管理対象 WebLogic Server は、デジタル証明書内の情報を利用して管理サーバを認証します。

管理サーバのデジタル証明書は、BEA から提供されます。デジタル証明書は、WebLogic Server のインストール時に `\wlserver6.1\config\mydomain` にインストールされます。

デフォルトでは、管理サーバとその他の WebLogic Server との接続は安全ではありません。ユーザ名とパスワードが入ったファイルは暗号化されていません。ユーザ名とパスワードは接続を介してクリア テキストで送信され、コンフィグレーション情報は保護されません。このため、SSL プロトコルと証明書を使用して、管理サーバ内のコンフィグレーション情報を保護することをお勧めします。

図 1-4 に、管理サーバと管理対象 WebLogic Server との間のセキュア ログイン プロセスを示します。

図 1-4　管理対象サーバと管理サーバ用のセキュア ログイン



詳細については、「セキュリティの管理」を参照してください。

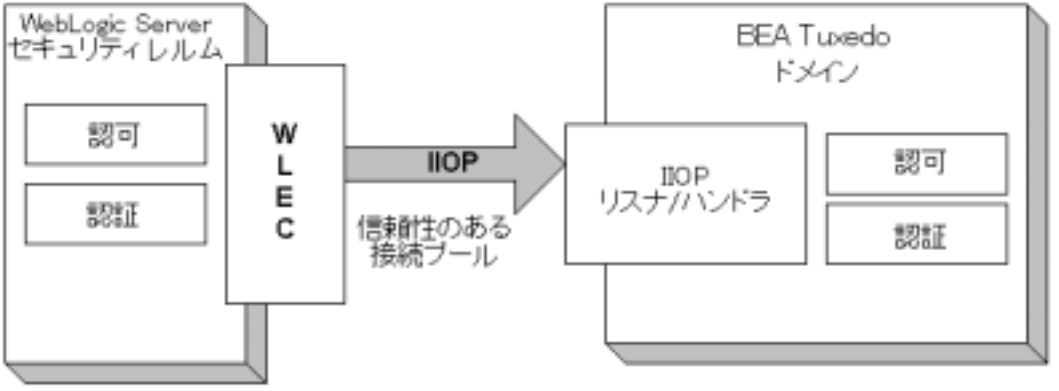
BEA Tuxedo に対するクライアントとしての WebLogic Server の使い方

WebLogic Server セキュリティ レルムのセキュリティのスコープは、BEA Tuxedo ドメイン内のスコープとは異なります。それぞれがユーザの独自のセキュリティ ストアを持ち、独自にアクセスを制御します。ただし、WebLogic Enterprise Connectivity を使用することで、信頼された WLEC 接続プールの一部をなす接続を介して、WebLogic Server セキュリティ レルムで認証済みユーザの ID から BEA Tuxedo ドメイン内の認証済みプリンシパルの ID を形成できます。この機能は、セキュリティ コンテキストの伝播と呼ばれます。

注意： WebLogic Server 製品でのセキュリティ コンテキストの伝播は一方向のみです。つまり、ユーザの ID は、WebLogic Server セキュリティ レルムから BEA Tuxedo ドメインへの方向にのみ伝播できます。

図 1-5 に、WebLogic Server 環境と BEA Tuxedo 環境との間でセキュリティ コンテキストが伝播される仕組みを示します。

図 1-5 WebLogic Server と BEA Tuxedo との間のセキュリティ コンテキストの伝播



セキュリティ コンテキストを伝播する場合、WebLogic Server ユーザのセキュリティ ID が、IIOP リクエストのサービス コンテキストの一部に含まれます。このリクエストは、WLEC 接続プールの一部をなすネットワーク接続を経由して、BEA Tuxedo ドメインに送信されます。WLEC 接続プール内の各ネットワーク接続は、定義済みのユーザ ID を使用して認証されます。WLEC 接続プールを確立するには、パスワードと証明書の両方を使用します。

伝播されたセキュリティ ID は、IIOP リスナ / ハンドラが BEA Tuxedo ドメイン内でユーザの役割を果たすために使用します。ユーザの役割を果たす ID は、認可用と監査用にそれぞれ 1 つのトークンのペアで表されます。これらのトークンは、BEA Tuxedo ドメインの対象 CORBA オブジェクトに伝播され、認可と監査が行われる際に使用されます。

ユーザ ID のマッピングを容易にするため、BEA Tuxedo 内の IIOP リスナ / ハンドラは認証プラグインを使用します。このプラグインは、ユーザ ID を認可および監査トークンにマッピングします。マッピングされたトークンは、対象となる CORBA オブジェクトに転送されるリクエストの一部として伝播されます。対象となる CORBA オブジェクトはトークンを使用して、ユーザの ID やユーザに関連付けられているロールまたはグループの名前など、リクエストの発信元についての情報を調べます。

SSL プロトコルを使用すると、WebLogic Server セキュリティ レルムから送信されるリクエストの機密性と整合性を保護できます。SSL 暗号化は、BEA Tuxedo ドメイン内の CORBA オブジェクトに送信される IIOP リクエストを対象として行われます。リクエストを保護するには、WebLogic Connectivity と BEA Tuxedo CORBA アプリケーションの両方で、SSL プロトコルを使用するようコンフィグレーションする必要があります。

詳細については、以下の節を参照してください。

- 「セキュリティの管理」の「セキュリティ コンテキストの伝播のコンフィグレーション」
- 『WebLogic Enterprise Connectivity ユーザーズ ガイド』

2　セキュリティの基礎概念

この章では、WebLogic Server のセキュリティに関する以下の基礎概念について説明します。

- リソース
- セキュリティ レルム
- ユーザ
- グループ
- ACL とパーミッション
- SSL プロトコル
- 認証メカニズム
- デジタル証明書
- 認証局
- サポートされている公開鍵アルゴリズム
- サポートされている対称鍵アルゴリズム
- サポートされているメッセージ ダイジェスト アルゴリズム
- サポートされている暗号スイート

リソース

リソースとは、イベント、サーブレット、JDBC 接続プール、JMS の送り先、JNDI コンテキスト、接続、ソケット、ファイル、およびデータベースなどのエンタープライズ アプリケーションとそのリソースといった WebLogic Server からアクセス可能なエンティティのことです。

WebLogic Server で保護するリソースごとに、以下の項目を指定しなければなりません。

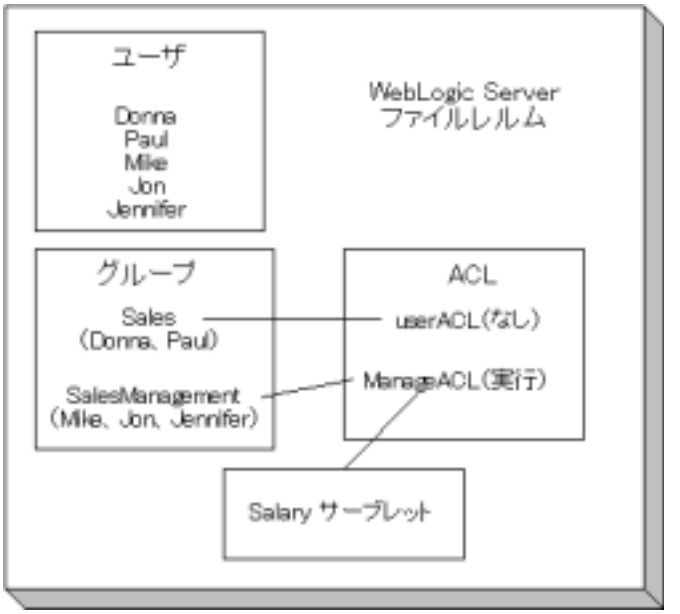
- リソースにアクセスできるユーザを定義する ACL
- リソースの所属先のセキュリティ レルム
- リソースへのアクセスを要求しているユーザを確認する認証メカニズム

詳細については、「セキュリティの管理」を参照してください。

セキュリティ レルム

セキュリティ レルムとは、ユーザ、グループ、および ACL を論理的に分類したものです。WebLogic Server のリソースは、1 つのセキュリティ レルムに属し、そのレルム内の 1 つの ACL によって保護されます。ユーザはセキュリティ レルムで定義されていないと、そのセキュリティ レルムに属するリソースにアクセスできません。ユーザが特定の WebLogic Server リソースにアクセスしようとした場合、WebLogic Server は、関連レルム内で割り当てられている ACL とパーミッションをチェックして、ユーザを認証し、そのリソースに対する認可を持っているかを調べます。図 2-1 は、WebLogic Server でのレルムの仕組みを示しています。

図 2-1 WebLogic Server レルム内のユーザ、グループ、および ACL



WebLogic Server のデフォルトのセキュリティ レルムはファイル レルムです。WebLogic Server が起動すると、WebLogic Server の Administration Console で定義されたプロパティに基づいて、ファイル レルムがユーザ、グループ、および ACL オブジェクトを作成し、`fileRealm.properties` ファイルに保存します。

注意： ファイル レルムは、10,000 以下のユーザを対象としています。ユーザ数が 10,000 より多い場合は、代替セキュリティ レルムを使用することをお勧めします。

また、WebLogic Server は、特殊なセキュリティ状況に対応する必要のある開発者に対するサポートも提供しています。WebLogic Server では、代替セキュリティ レルムをインストールするか、またはカスタム セキュリティ レルムを作成することで、ファイル レルム以外のセキュリティ レルムを使用できます。代替セキュリティ レルムは、WebLogic Server がレルムに関して要求する認証および認可処理をサポートします。

レルムのコンフィグレーションには、次の 2 種類があります。

- ファイル レルムのみ
- 代替セキュリティ レルムまたはカスタム セキュリティ レルムを使用するキャッシング レルム

  後者のコンフィグレーションでは、代替セキュリティ レルムまたはカスタム セキュリティ レルムは、プライマリ レルムとして機能します。ファイル レルムはバックアップ レルムになります。WebLogic Server のプライマリおよびバックアップ レルムは相互に連携して、適切な認証および認可を持つクライアントのリクエストを遂行します。たとえば、LDAP セキュリティ レルムなどの代替セキュリティ レルムを使用する場合は、まず、そのレルム内でユーザが検索されます。ユーザがプライマリ レルム（この場合は LDAP セキュリティ レルム）内に存在しない場合は、バックアップ レルムでユーザが検索されます。

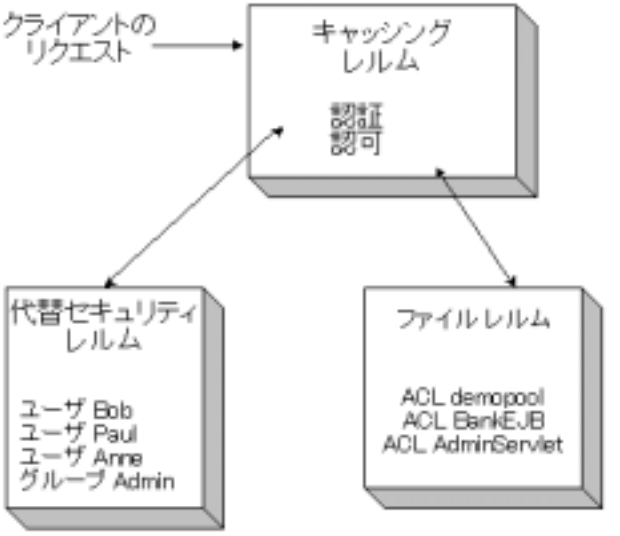
注意： 代替セキュリティ レルムまたはカスタム セキュリティ レルムを使用する場合は、キャッシング レルムをコンフィグレーションする必要があります。

デフォルトの使い方では、クライアントのリクエストは、キャッシング レルムを通じて WebLogic Server に送られます。キャッシング レルムは、リクエストをファイル レルムに転送して、認可および認証処理を行います。ファイル レルムがレルム ルックアップの結果（成功または失敗）を受け取ると、キャッシング レルムがこれらの結果を保存します。キャッシング レルムは、ユーザ、グループ、パーミッション、ACL、および認証ルックアップのキャッシュを別々に保持します。キャッシュの各タイプを選択して有効化したり、キャッシュするオブジェクトの数を設定したり、キャッシュしたオブジェクトの有効期間（秒単位）を指定したりできます。実際には、キャッシング レルムによって、認証および認可処理がより早く、しかも効率的に行われます。

代替セキュリティ レルムまたはカスタム セキュリティ レルムを使用する場合、キャッシング レルムはクライアントのリクエストを評価して、適切なセキュリティ レルムに委任し、結果をキャッシュして次のルックアップを高速化します。たとえば、認証処理だけをサポートする代替セキュリティ レルムを使用しているとします。WebLogic Server がクライアントのリクエストを受け取ると、キャッシング レルムは、認証を行うために代替セキュリティ レルムと、認可を行うためにファイル レルムとそれぞれ通信します。

図 2-2 は、WebLogic Server 環境で代替セキュリティ レルム、キャッシング レルム、およびファイル レルムが相互に連携して、ユーザの認証と認可を処理する仕組みを示しています。

図 2-2 WebLogic Server 内の代替セキュリティ レルム、キャッシング レルム、およびファイル レルム



WebLogic Server では、以下の代替セキュリティ レルムが用意されています。

- LDAP セキュリティ レルム

  Lightweight Directory Access Protocol（LDAP）サーバを通して認証を行います。このレルムを使用すると、ユーザを 1 つの場所、LDAP ディレクトリで管理できます。LDAP セキュリティ レルムを使用すると、LDAP サーバがユーザとグループを認証します。WebLogic Server で SSL プロトコルを使用している場合、LDAP セキュリティ レルムは、デジタル証明書からユーザの共通名を取り出し、その名前を LDAP ディレクトリで検索します。LDAP セキュリティ レルムはデジタル証明書を確認せず、確認は SSL プロトコルによって行われます。

LDAP セキュリティ レルムでは、現在、Open LDAP、Netscape iPlanet、Microsoft Site Server、および Novell NDS がサポートされています。

注意： WebLogic Server のこのリリースでは、更新された LDAP セキュリティ レルムが用意されています。更新された LDAP セキュリティ レルムは、パフォーマンスが向上し、コンフィグレーションが容易になりました。これらの機能を利用するために、更新された LDAP セキュリティ レルムにアップグレードすることをお勧めします。ただし、旧バージョンの LDAP セキュリティ レルムもまだ使用できます。詳細については、「セキュリティの管理」を参照してください。

- Windows NT セキュリティ レルム

  Windows NT のアカウント情報を使用して、ユーザを認証します。Windows NT で定義したユーザおよびグループは、WebLogic Server で使用できます。Administration Console を使用してこのレルムを表示することはできますが、ユーザとグループを定義するには、Windows NT に付属の機能を使用しなければなりません。

- UNIX セキュリティ レルム

  ネイティブ プログラム、`wlauth` を実行し、UNIX ログイン ID とパスワードを使用して、ユーザとグループを認証します。UNIX セキュリティ レルムは、Solaris および Linux プラットフォームでのみサポートされています。`wlauth` では、プラグイン可能な認証モデル（PAM）を使用します。PAM を使用すると、認証サービスを使用する別のアプリケーションに変更せずに、このサービスを UNIX プラットフォーム内でコンフィグレーションできます。Administration Console を使用してこのレルムを表示することはできますが、ユーザとグループを定義するには、UNIX で提供されている機能を使用しなければなりません。

- RDBMS セキュリティ レルム

  ユーザ、グループ、および ACL をデータベースから読み込みます。RDBMS セキュリティ レルムは、WebLogic Server 向けの認証および認可サービスを提供するカスタム レルムの実装例として用意されています。

詳細については、「セキュリティの管理」の以下の節を参照してください。

- 「セキュリティ レルムの指定」
- 「キャッシング レルムのコンフィグレーション」
- 「LDAP セキュリティ レルムのコンフィグレーション」

- 「Windows NT セキュリティ レルムのコンフィグレーション」
- 「UNIX セキュリティ レルムのコンフィグレーション」
- 「RDBMS セキュリティ レルムのコンフィグレーション」

ユーザ

ユーザとは、アプリケーションのエンド ユーザ、クライアント アプリケーション、他の WebLogic Server など、WebLogic Server を使用するエンティティのことです。

ユーザが WebLogic Server にアクセスしようとする場合、ユーザは、プログラムを通じてユーザ名と資格（パスワードまたはデジタル証明書）を WebLogic Server に提示します。WebLogic Server がユーザ名と資格に基づいてユーザの ID を確認できた場合、WebLogic Server はユーザの代わりにコードを実行するスレッドをそのユーザに関連付けます。ただし、スレッドがコードの実行を始める前に、WebLogic Server は適切な ACL をチェックして、ユーザが処理を続行するために必要なパーミッションを持っていることを確認します。

WebLogic Server レルムでユーザを定義する場合は、そのユーザのパスワードも定義します。ユーザ名とパスワードは以前、WebLogic Server セキュリティ レルムにクリア テキストで保存されていました。現在、WebLogic Server では、すべてのパスワードがハッシュ化されています。クライアントのリクエストを受け取ると、WebLogic Server はクライアントが提示するパスワードをハッシュ化して、ハッシュ化済みパスワードと一致するかどうか比較します。

詳細については、「セキュリティの管理」の以下の節を参照してください。

- 「ユーザの定義」
- 「パスワードの保護」

グループ

グループとは、論理的に整理したユーザの集合のことです。グループを管理する方が、多くのユーザを個々に管理するよりも効率的です。たとえば、管理者は、グループを指定することで 50 ユーザのパーミッションを一度に指定することができます。一般に、グループ メンバーどうしは何らかの共通点を持っています。たとえば、職務によって WebLogic Server のリソースへのアクセス レベルが異なるため、企業の販売スタッフを、販売員と販売マネージャという 2 つのグループに分けることができます。

WebLogic Server をコンフィグレーションして、ユーザをグループに割り当てることができます。各グループは、メンバー ユーザによるリソースへのアクセスを管理する共通のパーミッション セットを共有します。ユーザのリストが許可されていれば、グループ名とユーザ名を混合することができます。

ユーザは、個人ユーザとしてもグループ メンバーとしても定義できます。個々のアクセス パーミッションは、グループ メンバーのパーミッションをオーバーライドします。WebLogic Server は、最初にグループを検索してユーザがグループのメンバーになっているかどうかを調べてから、定義済みユーザのリストの中でユーザを検索することで、各ユーザを評価します。

詳細については、「セキュリティの管理」の「グループの定義」を参照してください。

ACL とパーミッション

パーミッションは、リソースにアクセスするために必要な特権を表します。システム管理者は、リソースのアクセスに必要なパーミッションを持つユーザおよびグループのリストを作成することで、リソースを保護します。こうしたリストは、アクセス制御リスト（ACL）と呼ばれます。ACL と ACL によってパーミッションを付与する機能はリソース固有です。たとえば、適切なパーミッションを持つユーザであれば、ファイルの読み取り、書き込み、送信、および受信、サーブレットのロード、ライブラリへのリンクを行うことができます。

ACL ファイルは、それぞれが特定のユーザまたはグループを対象とする一連のパーミッションを格納する AclEntry から構成されます。

WebLogic Server は、Java Development Kit（JDK）1.1 で配布される JavaSoft ACL 標準を利用して、Java のセキュリティ フレームワークを拡張し、エンタープライズ レベルでの実用性を持たせています。WebLogic Server の ACL は、`java.security.acl` パッケージに基づいて実装されています。WebLogic Server の ACL はオープンスタンダードに準拠しているので、独自のソリューションにとらわれません。

パーミッションを設定できる WebLogic Server のリソースは以下のとおりです。

- WebLogic Server
- WebLogic Event
- WebLogic JDBC 接続プール
- WebLogic JMS 送り先
- WebLogic JNDI コンテキスト
- WebLogic MBean

注意： EJB および Web アプリケーションへのアクセスは、ACL で制御するのではありません。それよりも、デプロイメント記述子のなかのセキュリティ要素を使用して、特定の EJB または Web アプリケーションへのアクセスを定義します。

詳細については、「セキュリティの管理」の「ACL の定義」を参照してください。

EJB セキュリティの詳細については、『WebLogic エンタープライズ JavaBeans プログラマーズ ガイド』を参照してください。

Web アプリケーション セキュリティの詳細については、『Web アプリケーションのアセンブルとコンフィグレーション』を参照してください。

SSL プロトコル

SSL プロトコルは、ネットワーク経由で接続された、複数のアプリケーションにセキュリティを提供します。具体的には、SSL プロトコルは以下のものを提供します。

- 互いの ID を認証するためにアプリケーションが利用するメカニズム
- アプリケーション間でやりとりするデータの暗号化

SSL プロトコルを使用する場合、対象は常に発信元に対して自身を認証します。対象が要求した場合は、発信元が自身を対象に対して認証することもあります。ネットワーク経由で送信されるデータは暗号化されるので、予定されている宛先以外には解釈できません。SSL 接続はデジタル証明書をやりとりするアプリケーション間のハンドシェークで始まり、使用する暗号化アルゴリズムの取り決め、そのセッションの残りで使用する暗号キーの生成と続きます。

SSL プロトコルは、以下のセキュリティ機能を提供します。

- サーバ認証 ― WebLogic Server は、信頼された認証局が発行したデジタル証明書を使用して、クライアントに対して認証します。
- クライアント識別確認 ― 場合によっては、WebLogic Server に対するデジタル証明書の提示がクライアントに要求されることもあります。WebLogic Server で、そのデジタル証明書が信頼された認証局によって発行されたものであることが確認されると、SSL 接続が確立されます。デジタル証明書が提示されなかったり、確認されなかったりした場合、SSL 接続は確立されません。こうしたタイプの接続は、「相互 SSL」と呼ばれます。相互 SSL を使用する場合、クライアントによって提示される証明書は WebLogic Server ユーザとは一致しないので、クライアントは認証および認可で使用するユーザ名と資格（パスワードかデジタル証明書）も提示する必要があります。
- 機密性 ― クライアントのリクエストとサーバの応答は暗号化され、ネットワーク経由でやりとりされるデータの機密性が保持されます。
- データの整合性 ― クライアントと WebLogic Server との間のデータ フローを、第三者による改ざんから保護します。

Web ブラウザを使って WebLogic Server と通信する場合は、Hypertext Transfer Protocol with SSL（HTTPS）を利用して、ネットワーク通信の安全性を確保できます。

SSL プロトコルは、IP ベースのプロトコル上をトンネリングされます。トンネリングとは、各 SSL レコードをカプセル化し、別のプロトコル上でレコードを送信するために必要なヘッダと一緒にパッケージ化することです。SSL を使用していることは、WebLogic Server の場所を指定するための URL のプロトコル方式で表されます。

- Web ブラウザと WebLogic Server との SSL 通信は、HTTPS パケットにカプセル化されて転送されます。次に例を示します。

```
https://myserver.com/mypage.html
```

WebLogic Server では、SSL3 をサポートする Web ブラウザで HTTPS を使用できます。WebLogic Server の Java 仮想マシン（JVM）では、現在、HTTPS プロトコル アダプタがサポートされていません。したがって、WebLogic Server は、Web ブラウザでの SSL 実装に依存します。

- SSL プロトコルを使って WebLogic Server に接続する Java クライアントは、BEA の多重化 T3 プロトコル上をトンネリングします。次に例を示します。

  ```
  t3s://myserver.com:7002/mypage.html
  ```

  WebLogic Server 内で動作する Java クライアントは、他の WebLogic Server に T3S 接続するか、または Web サーバや安全なプロキシ サーバなど SSL プロトコルをサポートする他サーバに HTTPS 接続します。

WebLogic Server は、輸出向けと米国内向けのどちらの長さの SSL でも使用できます。

- 輸出向け SSL は、512 ビットの証明書と 40 ビットのバルク データ暗号化をサポートします。
- 米国内向け SSL は、768 ビットと 1024 ビットの証明書、および 56 ビットと 128 ビットのバルク データ暗号化をサポートします。

標準の WebLogic Server 配布キットは、輸出向けの長さの SSL だけをサポートします。米国内向けバージョンについては、BEA の販売代理店を通じてお求めください。米国政府は、2000 年初めに暗号化ソフトウェアの輸出に関する規制を緩和したので、米国内向けバージョンの WebLogic Server はほとんどの国でご使用いただけます。

インストール時に、使用する SSL プロトコルの長さを選択する手順があります。暗号の強度が高いので、米国内向け WebLogic Server 配布キットをお勧めします。輸出向け WebLogic Server 配布キットを使って証明書署名リクエスト（CSR）を生成する場合は、高レベル接続に対応できないので、米国内向けレベルの証明書を提示するクライアントを認証できません。

WebLogic Server が提供する SSL プロトコルの実装は、pure-Java 実装です。Solaris、Windows NT、および IBM AIX プラットフォームでの SSL 処理によっては、ネイティブ ライブラリを使う方がパフォーマンスが向上します。SSL プロトコルでこうしたネイティブ Java ライブラリを使うよう要求するには、Administration Console を使います。

詳細については、「セキュリティの管理」の「SSL プロトコルのコンフィグレーション」を参照してください。

認証メカニズム

WebLogic Server ユーザが、保護されている WebLogic Server リソースへのアクセスを要求する場合、必ず認証を受けなければなりません。このため、各ユーザは資格（ユーザ名 / パスワードの組み合わせまたはデジタル証明書）を WebLogic Server に提示する必要があります。WebLogic Server では、以下のタイプの認証メカニズムがサポートされています。

- パスワード認証 ― ユーザ ID とパスワードをユーザに要求し、クリア テキストで WebLogic Server に送ります。WebLogic Server は受け取った情報をチェックして、信頼性を確認できれば、保護されているリソースへのアクセスを許可します。

  SSL（または HTTPS）プロトコルを使用すると、パスワード認証にさらに高度なレベルのセキュリティを提供できます。SSL プロトコルは、クライアントと WebLogic Server との間で転送されるデータを暗号化するので、ユーザ ID とパスワードはクリア テキストでは転送されません。したがって、WebLogic Server は、ユーザの ID およびパスワードの機密性を保持したままユーザを認証できます。

- 証明書認証 ― SSL または HTTPS クライアントのリクエストが送信されると、WebLogic Server は、クライアントに対してデジタル証明書を提示して応答します。クライアントによってそのデジタル証明書が確認されると、SSL 接続が確立されます。続いて CertAuthenticator クラスはクライアントのデジタル証明書からデータを抽出し、そのデジタル証明書を所有する WebLogic Server ユーザを判別して、WebLogic Server セキュリティ レルムからその認証されたユーザを取り出します。

  相互認証を用いることもできます。この場合、WebLogic Server は自身を認証するだけでなく、要求側のクライアントに対しても認証を要求します。クライアントは、信頼された認証局が発行したデジタル証明書を提出するよう要求されます。相互認証は、アクセスを許可する対象を信頼されたクライアントに制限する場合に便利です。たとえば、管理者が提供したデジタル証明書を持つクライアントだけにアクセスを制限すると便利な場合があります。

詳細については、「セキュリティの管理」の以下の節を参照してください。

- 「SSL プロトコルのコンフィグレーション」
- 「相互認証のコンフィグレーション」

デジタル証明書

デジタル証明書とは、インターネットなどのネットワークを経由して、プリンシパルおよびエンティティのユニークな ID を確認するための電子的なドキュメントのことです。デジタル証明書は、認証局として知られる信頼された第三者によって確認されたユーザまたはエンティティの ID を、特定の公開鍵に安全にバインドします。公開鍵とプライベート キー（秘密鍵）を組み合わせることで、デジタル証明書のオーナにユニークな ID が提供されます。

デジタル証明書を使用すると、特定の公開鍵が実際に特定のユーザまたはエンティティに属しているかどうかを確認できます。デジタル証明書の受信側は、証明書内の公開鍵を使用して、デジタル署名が対応するプライベート キーで作成されたものかどうかを確認します。こうした確認が終わると、この推論のチェーンによって、対応するプライベート キーの所有者がデジタル証明書に名前のある主体であること、そしてそのデジタル署名を作成したのがその主体であることを確認できます。

一般に、デジタル証明書には以下のようにさまざまな情報が入っています。

- 主体（保持者、オーナ）の名前と、デジタル証明書を使用する Web サーバの URL や個人の電子メール アドレスなど、その主体の ID をユニークに確認するために必要なその他の情報
- 主体の公開鍵
- デジタル証明書を発行した認証局の名前
- シリアル番号
- （開始日と終了日で定義される）デジタル証明書が有効性を持つ期間（有効期間）

最も広く使われているデジタル証明書のフォーマットは、ITU-T X.509 国際標準で定義されているものです。X.509 標準に準拠していればどのアプリケーションを使っても、デジタル証明書を読み書きできます。WebLogic Server の公開鍵イ

ンフラストラクチャ（PKI）では、X.509 バージョン 3 または X.509v3 に準拠したデジタル証明書が認識されます。Verisign や Entrust などの認証局から証明書を取得することをお勧めします。

WebLogic Server では、X.509 標準によって提供される拡張子がサポートされていますが、`weblogic.security.X509` クラスには特定のデジタル証明書で使用される拡張子に関する情報を提供するアクセサが用意されていません。その情報を入手するには、`weblogic.security.X509` オブジェクトを `java.security.cert.X509Certificate` オブジェクトに変換します。次のコード例には、この変換を行うコードが含まれています。

```
X509[] wlCerts=...
X509Certificate [] javaCerts = new X509Certificate[wlCerts.length];
try{
    CertificateFactory cf =
                      java.security.cert.CertificateFactory.getInstance("X.509");
    for(int i=0; i<wlCerts.length; i++){
       ByteArrayOutputStream bos = new ByteArrayOutputStream();
       wlcerts[i].output(bos);
       ByteArrayInputStream bis = new
    ByteArrayInputStream(bos.toByteArray());
       javaCerts[i] = (X509Certificate)cf.generateCertificate(bis);
       }
    }
```

詳細については、「セキュリティの管理」の「SSL プロトコルのコンフィグレーション」を参照してください。

認証局

デジタル証明書は、認証局によって発行されます。デジタル証明書および公開鍵の発行先の ID を保証する信頼された第三者組織または企業はすべて、認証局になることができます。認証局は、デジタル証明書を作成する際に、証明書が改ざんされればわかるようにプライベート キーを使って署名します。認証局は、署名したデジタル証明書を要求主体に返します。

主体は、認証局の公開鍵を使って発行認証局の署名を確認します。認証局の公開鍵は、低レベル認証局の公開鍵の妥当性を保証する高レベル認証局から発行されたデジタル証明書を提供することで利用できるようになります。この方式によって、認証局の階層が構築されます。この階層は、最終的に、ルート キーと呼ばれる自己署名デジタル証明書にたどり着きます。

暗号化されたメッセージの受信側は、受信側がすでに信頼性を確認済みで該当認証局よりも高レベルの認証局が署名し、該当認証局の公開鍵が入ったデジタル証明書を持っている場合は、認証局のプライベート キーを再帰的に信頼します。この点では、デジタル証明書はデジタルの信頼関係の踏み台です。結局、信頼する必要があるのは、ごく少数の最上位認証局の公開鍵だけです。デジタル証明書のチェーンを通じて、多数のユーザのデジタル署名の信頼を確立できます。

通信中のエンティティの ID は、このようにしてデジタル署名によって確立できますが、デジタル署名による信頼は、信頼性を確認するための公開鍵の範囲にとどまります。

詳細については、「セキュリティの管理」の「SSL プロトコルのコンフィグレーション」を参照してください。

サポートされている公開鍵アルゴリズム

公開鍵（または非対称鍵）アルゴリズムは、数学的には互いに関連を持ちながらも異なる 2 つの鍵を利用します。

- デジタル証明書を確認したり、データを外見上解釈不能な形式に変換したりするための公開鍵（一般に配布）
- デジタル証明書を作成したり、データを元の形式に戻したりするためのプライベート キー（非公開のまま保管）

WebLogic Server の PKI もデジタル署名のアルゴリズムをサポートします。デジタル署名アルゴリズムは、デジタル署名を生成するためだけの公開鍵アルゴリズムのことです。

WebLogic Server では、Rivest、Shamir、Adelman（RSA）アルゴリズムをサポートしています。

サポートされている対称鍵アルゴリズム

対称鍵アルゴリズムでは、メッセージの暗号化と解読に同じ鍵を使用します。公開鍵暗号化システムは対称鍵アルゴリズムを使用して、通信中の 2 つのエンティティ間でやりとりされるメッセージを暗号化します。対称暗号化は、公開鍵暗号作成法よりも少なくとも 1000 倍の早さで処理を実行できます。

ブロック暗号は、プレーン テキスト（暗号化されていないテキスト）の固定長ブロックを同じ長さの暗号テキスト（暗号化済みテキスト）データ ブロックに変換する対称鍵アルゴリズムの一種です。この変換は、ランダムに生成したセッション キーの値に従って発生します。固定長はブロック サイズと呼ばれます。

WebLogic Server の PKI は以下の対称鍵アルゴリズムをサポートしています。

- DES-CBC（Data Encryption Standard - Cipher Block Chaining）

  DES-CBC は、Cipher Block Chaining（CBC）モードの 64 ビット ブロックの暗号文です。56 ビットの鍵（8 パリティ ビットをフル 64 ビット鍵から除去します）を使用します。

- 2 キー形式トリプル DES（Data Encryption Standard）

  2 キー形式トリプル DES は、Encrypt-Decrypt-Encrypt（EDE）モードの 128 ビット ブロックの暗号です。2 キー形式トリプル DES では、2 つの 56 ビット鍵（実効では 112 ビットの鍵 1 つ）を利用します。

  現在では、トリプル DES による DES 暗号鍵を保護および転送する方式が一般的となっています。つまり、入力データ（この場合には シングル DES 鍵）を暗号化し、解読してから、もう一度暗号化します（暗号化 解読 暗号化処理）。同じキーを 2 回の暗号化処理に使います。

- RC4（Rivest's Cipher 4）

  RC4 は、サイズが 40 ～ 128 ビットの鍵を使用する可変キー サイズ ブロック暗号です。DES よりも処理が高速で、キー サイズを 40 ビットにして輸出可能です。米国企業の海外子会社および海外支社には、56 ビットのキー サイズが許可されています。WebLogic Server PKI によってキー長が 128 ビットに制限されていますが、米国では、事実上キー長の制限なしに RC4 を使用できます。

WebLogic Server ユーザは、このアルゴリズムのリストを拡張したり変更したりすることができません。

サポートされているメッセージ ダイジェスト アルゴリズム

WebLogic Server は、MD5 および SHA（Secure Hash Algorithm）メッセージ ダイジェスト アルゴリズムをサポートしています。MD5 も SHA も、広く知られた一方向のハッシュ アルゴリズムです。一方向のハッシュ アルゴリズムは、メッセージを、メッセージ ダイジェストまたはハッシュ値と呼ばれる数字の固定文字列に変換します。

MD5 は高速処理可能な 128 ビット ハッシュで、32 ビット マシンで使用することを想定しています。MD5 と比べて、SHA は 160 ビット ハッシュを用いてより高度なセキュリティを提供しますが、処理速度は低下します。

サポートされている暗号スイート

暗号スイートとは、通信の整合性を保護するために使用する鍵交換アルゴリズム、対称暗号化アルゴリズム、およびセキュア ハッシュ アルゴリズムを含む SSL 暗号方式の一種です。たとえば、`RSA_WITH_RC4_128_MD5` という暗号スイートは、鍵交換用に RSA、バルク暗号化用に 128 ビット キーを使う RC4、およびメッセージ ダイジェスト用に MD5 を使用します。

WebLogic Server でサポートされている暗号スイートは、表 2-1 のとおりです。

表 2-1 WebLogic Server でサポートされている SSL 暗号スイート

暗号スイート	鍵交換タイプ	対称鍵強度
SSL_RSA_WITH_RC4_128_SHA	RSA	128
SSL_RSA_WITH_RC4_128_MD5	RSA	128
SSL_RSA_WITH_DES_CBC_SHA	RSA	56
SSL_RSA_EXPORT_WITH_RC4_40_MD5	RSA	40
SSL_RSA_EXPORT_WITH_DES_40_CBC_SHA	RSA	40
SSL_RSA_WITH_3DES_EDE_CBC_SHA	RSA	112
SSL_NULL_WITH_NULL_NULL		
SSL_RSA_WITH_NULL_SHA	RSA	0
SSL_RSA_WITH_NULL_MD5	RSA	0

WebLogic Server のライセンスによって、通信を保護するために使用する暗号スイートの強度（米国内向けまたは輸出向け）が決まります。config.xml ファイルで定義されている暗号スイートの強度がライセンスで指定されている強度を超える場合は、ライセンスで指定されている強度が使用されます。たとえば、ライセンスが輸出向け強度であるにもかかわらず、米国内向け強度の暗号スイートを使うよう config.xml で定義した場合は、米国内向け強度の暗号スイートは拒否されて、輸出向け強度の暗号スイートが使用されます。

3 WebLogic Server デプロイメントのセキュリティ対策

以下の節では、WebLogic Server のセキュリティ機能を使用して、デプロイメントを保護する方法を説明します。

- WebLogic Server のセキュリティが重要な理由
- WebLogic Server デプロイメントに必要なセキュリティの決定
- WebLogic Server を実行するマシンのセキュリティ対策
- UNIX 上の保護されたポートへのアクセス
- ネットワーク接続の設計
- WebLogic Server の開発環境とプロダクション環境の管理
- 暗号化の使用
- SSL プロトコルの使用
- 介在者の攻撃の防止
- サービス拒否攻撃の防止
- HTTP 応答ヘッダのセキュリティ保護
- ユーザ アカウントの保護
- アプリケーションのコンテンツの保護
- ユーザ入力データ中の HTML 特殊文字の置換
- 保護された EJB によるビジネス ロジックへのアクセス制限
- ACL の使用
- 適切なセキュリティ レルムの使用
- データベースのセキュリティ対策

- 監査の利用

WebLogic Server のセキュリティが重要な理由

アプリケーション サーバは、エンド ユーザと貴重なデータやリソースとの間の重要なレイヤに位置しています。WebLogic Server は、リソースの保護に関して認証および暗号化サービスを提供します。しかし、こうしたサービスでは、デプロイメント環境の弱点を見つけ出して悪用することでアクセスを取得した侵入者から、リソースを守ることはできません。

インターネットまたはイントラネット上で WebLogic Server をデプロイする場合には、独立したセキュリティ専門家に依頼して、セキュリティ プランと手順を検討してもらい、インストール済みシステムの監査を受け、改善点のアドバイスを受けるとよいでしょう。

セキュリティ問題についてできる限り知識を増やすことも重要です。Web サーバのセキュリティ対策の最新情報については、カーネギー メロン大学が運営する CERT(TM) Coordination Center が公開している「Security Improvement Modules, Security Practices, and Technical Implementations」に目を通すことをお勧めします。

BEA の「security advisories」ページで推奨されている対策は是非、実行してください。また、リリースされている各サービス パックの適用もお勧めします。サービス パックには、製品の各バージョンおよび以前にリリースされた各サービス パックのすべてのバグの修正が含まれています。BEA 製品にセキュリティ関連の問題が発生した場合は、BEA から、その報告と適切な対策を示した指示が配信されます。サイトのセキュリティ問題を担当されている方は、今後、BEA からセキュリティ関連の問題の通知を受信できるよう、登録を行ってください。BEA では、BEA 製品に関するセキュリティ問題をご報告いただくための電子メール アドレス（`security-report@bea.com`）も用意しています。

さらに、WebLogic Server のプロダクション環境の保護に役に立つ、パートナ製品もあります。詳細については、BEA パートナのページを参照してください。

この章では、WebLogic Server に関する具体的なヒントと一般的なヒントを取り上げます。

WebLogic Server デプロイメントに必要なセキュリティの決定

WebLogic Server デプロイメントのセキュリティ対策を行う前に、WebLogic Server 環境で必要なセキュリティを把握してくおくことが重要です。セキュリティ ニーズを明確に把握するために、次の質問の回答を検討してください。

- 保護対象とする WebLogic Server リソースは何か

  WebLogic Server 環境には、WebLogic Server がアクセスするデータベース内の情報、Web サイトの可用性、パフォーマンス、および整合性を始めとして、保護可能な数多くのリソースがあります。セキュリティのレベルを決める際に、保護するリソースを十分検討します。

- WebLogic Server リソースを何から保護するのか

  ほとんどの Web サイト リソースの場合は、インターネット上のすべてのユーザから保護しなければなりません。しかし、社内のイントラネット上で従業員から Web サイトを保護する必要がありますか。従業員がすべての WebLogic Server リソースにアクセスする必要がありますか。システム管理者がすべての WebLogic Server リソースにアクセスする必要がありますか。システム管理者はすべてのデータにアクセスできる必要がありますか。機密性の高いデータや重要なリソースへのアクセスは、十分に信頼できるごく一部のシステム管理者に限定した方がよい場合もあります。また、データやリソースには、システム管理者がアクセスできないようにする方がよい場合もあります。

- 重要なリソースの保護に失敗した場合に何が起こるか

  場合によっては、セキュリティ スキーマの欠点が簡単に見つけられても迷惑なだけということもあります。また、欠点が原因で、Web サイトを利用する会社や個人の顧客に大きな損害を与えることもあります。各リソースのセキュリティが失敗した場合の結果を理解しておくと、適切な保護のレベルを判断する参考になります。

サイトのセキュリティ対策に関する以下のヒントに目を通す際には、上記の質問に対する回答を念頭に置いてください。

WebLogic Server を実行するマシンのセキュリティ対策

WebLogic Server デプロイメントのセキュリティは、デプロイメントが実行されるマシンのセキュリティと同じです。したがって、物理的なマシン、オペレーティング システム、およびホスト マシン上にインストールされたその他のすべてのソフトウェアのセキュリティ対策を取ることが重要です。ただし、以下のヒントはデプロイメント用マシンのセキュリティ対策に関するもので、マシン、オペレーティング システム、インストール済みソフトウェアに関しては、製造元に確認する必要があります。

- ハードウェアは、無認可のユーザがデプロイメント用マシンやネットワーク接続に不用意に触れることがないように、安全な場所に設置します。
- 電子メール プログラムやディレクトリ サービスなどのネットワーク サービスを専門家に評価してもらい、悪意ある攻撃者がオペレーティング システムまたはシステムレベルのコマンドにアクセスするために悪用できる弱点がないことを確認します。
- ディレクトリとファイルへのアクセスを、モニタ可能なごく少数のユーザ アカウントに限定して、デプロイメント用マシン上のファイル システムのセキュリティ対策を講じます。WebLogic Server コンフィグレーション データの一部（とアクセス制御リスト（ACL）で保護された Java Server Pages（JSP）および HTML ページを含むアプリケーションの一部）は、ファイル システム上にクリア テキストで保存されます。。ユーザまたは侵入者がファイルとディレクトリの読み取りアクセスを持っていれば、WebLogic Server の認証および許可スキーマに施したセキュリティ メカニズムを簡単に破ることができます。
- デプロイメント用マシン上では複数のユーザ アカウントを作成せず、ファイル システムをエンタープライズ ネットワーク内の他のマシンと共有しません。
- WebLogic ユーザをオペレーティング システム内に作成し、オペレーティング システムのセキュリティ制御機能を使用して、このユーザに WebLogic Server デプロイメント内のすべてのファイルおよびディレクトリの所有権と独占的アクセスを割り当てます。その他のユーザには、WebLogic Server デ

プロイメント内のどのファイルに対しても書き込みアクセスを割り当てません。

- アクティブなユーザ アカウントを定期的に再検討します。また、退職者があった場合にも見直します。パスワードの有効期間が一定期間で切れるようにポリシーを設定します。パスワードをクライアント アプリケーションにコーディングしないようにしてください。

UNIX 上の保護されたポートへのアクセス

UNIX システムでは、特権のあるユーザ アカウント (普通は root) で実行されるプロセスだけが、1024 未満のポートにバインドできます。

ただし、WebLogic Server のように長時間実行されるプロセスは、特権のあるアカウントで実行することはできません。代わりに、次のいずれかの方法を使用します。

- 特権のあるポートにアクセスする必要のある WebLogic Server インスタンスごとに、特権のあるユーザ アカウントで起動し、特権のあるポートにバインドし、ユーザ ID を特権のないアカウントに変更するよう、サーバをコンフィグレーションします。

  ノード マネージャを使用してサーバ インスタンスを起動する場合は、セキュア ポートだけで、単一の既知のホストだけからリクエストを受け取るように、ノード マネージャをコンフィグレーションします。

  『Administration Console オンライン ヘルプ』の「UNIX 上の保護されたポートへのバインド」を参照してください。

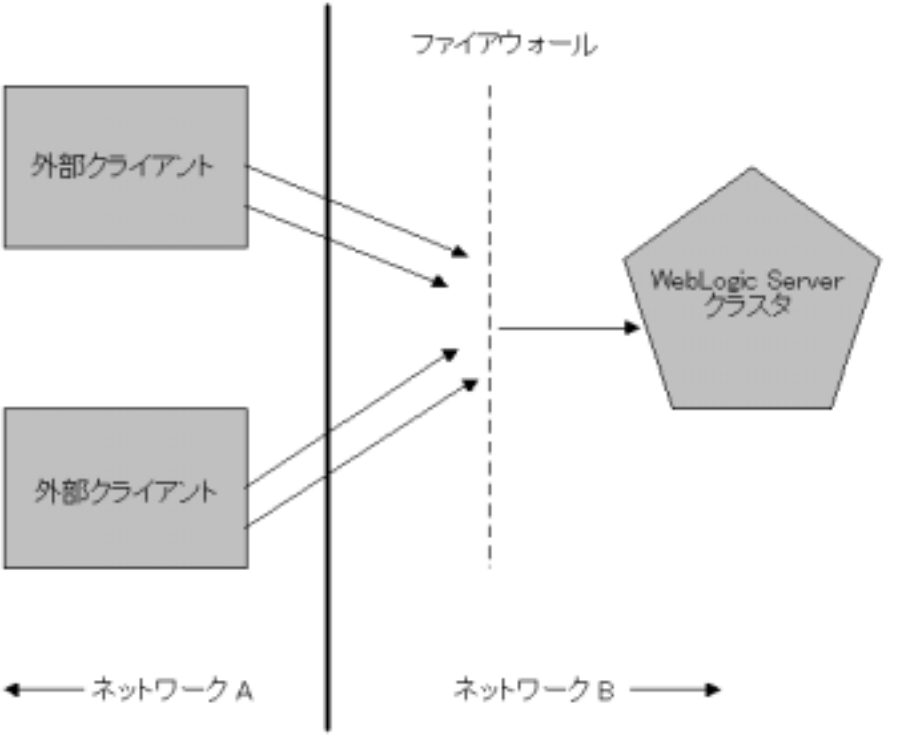
- 特権のないアカウントから WebLogic Server インスタンスを起動し、Network Address Translation (NAT) ソフトウェアを使って保護されたポートを保護されないポートにマップするよう、ファイアウォールをコンフィグレーションします。BEA は、NAT ソフトウェアを提供していません。

ネットワーク接続の設計

ネットワーク接続を設計する場合は、管理が最も簡単なソリューションの使用を検討します。ソリューションを選ぶには、保護を必要とする重要な WebLogic Server リソースとのバランスを検討しなければなりません。WebLogic Server リソースの設置場所が潜在的な侵入者から遠いほど、セキュリティ侵害の危険性が抑えられます。社内にファイアウォールを導入すると、セキュリティが強固になり、小さなセキュリティの欠点があっても危機的な状況にならずに済みます。たとえば、重要なデータを格納するデータベースはファイアウォールの内側で保護するとよいでしょう。また、データベースのホスト マシンだけでなく、データベース用のユーザ名とパスワードも保護します。誰かがデータベース用のユーザ名とパスワードを入手した場合でも、データベースがファイアウォールで保護されており、インターネット上のコンピュータからの接続を受け取れなければ、損害はほとんどありません。

ファイアウォール、WebLogic Server、およびネットワーク サーバを組み合わせるさまざまな方法があります。図 3-1 は、WebLogic Server クラスタ宛てのトラフィックをフィルタ処理するファイアウォールを使った代表的な設定を示しています。

図 3-1 代表的なファイアウォールの設定

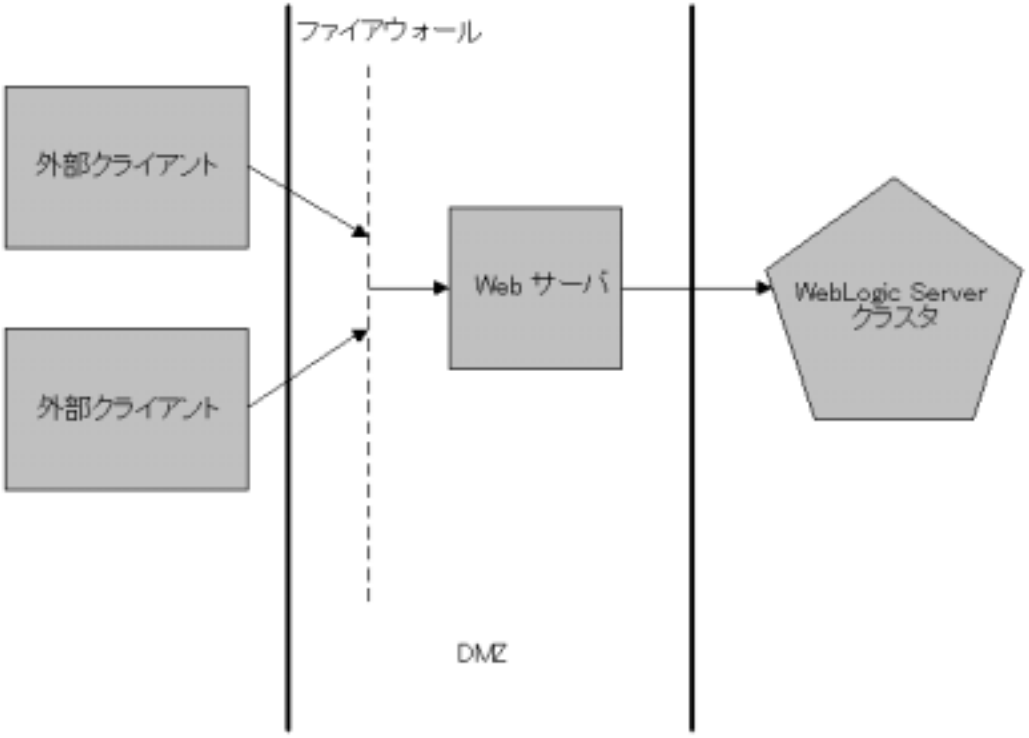


HTTP または HTTPS の Web 接続だけにアクセスを制限するファイアウォール コンフィグレーションも一般的です。ファイアウォールを使用すると、クライアントは、一般に標準の HTTP ポート 80 または HTTPs ポート 443 で動作する Web サーバにのみアクセスを許可されます。Web サーバは、WebLogic Server でも、WebLogic Server へのリクエストをプロキシするサードパーティ製 Web サーバであってもかまいません。たとえば、Netscape Enterprise Server、Microsoft Internet Server、および Apache Server を設定すると、静的な Web ページを提供して、WebLogic Server に対するサーブレットおよび JSP リクエストをプロキシできます。図 3-2 はこのコンフィグレーションを示しています。

図 3-2 では、Web サーバは非武装地帯（DMZ）で動作するゲートウェイです。コンピュータ ネットワークに関して DMZ とは、会社のプライベート ネットワークと外部のパブリック ネットワークとの間の中立ゾーンとして配置されたコンピュータ ホストまたは小規模ネットワークのことです。これにより、外部

のユーザが会社のデータを納めたサーバに直接アクセスすることはできなくなります。DMZ はファイアウォールのオプションの実装方法で、ファイアウォールがプロキシ サーバとしての役割も果たし、セキュリティはさらに強固になります。WebLogic Server の接続は、プロキシされた Web サーバのリクエストだけに基づいているので、WebLogic Server アプリケーションおよびバックエンド リソースのセキュリティが向上します。図 3-2 のコンフィグレーションでは、クライアントは Web サーバとだけ対話し、WebLogic Server の接続はプロキシされた Web サーバのリクエストだけに基づいて作成されます。その結果、この方法でコンフィグレーションされている WebLogic Server アプリケーションとバックエンド リソースのセキュリティが向上します。

図 3-2 Web サーバ ゲートウェイとファイアウォール



ファイアウォールを設定する以外にも、`weblogic.security.net.ConnectionFilter` インタフェースを実装することで、WebLogic Server デプロイメントへのアクセスを制限できます。このインタ

フェースを実装すると、接続の起点のマシンのホスト名およびネットワーク アドレスと使用プロトコルに基づいて、ネットワーク接続を受け付けたり拒否したりできます。

WebLogic Server の開発環境とプロダクション環境の管理

さまざまな理由により、プロダクション環境に近いマシンで開発した方が開発もプロダクションも容易です。ただし、セキュリティを考慮して、開発環境とプロダクション環境は以下のように区別します。

- プロダクション用マシン上では開発しないでください。開発用マシン上でまず開発して、コードが完了し、テストしてから、プロダクション用マシンにコードを移植します。この手順を取ることで、開発環境のバグがプロダクション環境のセキュリティに影響を与えるのを防ぎます。
- プロダクション用マシンに WebLogic Server サンプル アプリケーションをインストールしないでください。
- プロダクション用マシンのシステム パスワードはドメイン内でユニークでなければならず、また、慎重に保護する必要があります。
- 開発ツールをプロダクション用マシンに持ち込まないでください。こうした開発ツールには、javac、rmic、および ejbc コンパイラなどの開発用製品のコンポーネントだけでなく、その他に使用する開発ツールなども含まれます。開発ツールをプロダクション用マシンに持ち込まないことで、侵入者が WebLogic Server のプロダクション用マシンに部分的にアクセスできた場合でも、悪用されるおそれが少なくなります。
- ソース コードを保護します。ソース コードにアクセスできれば、侵入者がセキュリティ ホールを見つけ出します。必ず、ソース コードはプロダクション用マシンから遠ざけておきます。エンド ユーザを対象としていない JSP ファイル内のコメントには、HTML 構文の `<!-- ... -->` ではなく、JSP 構文の `<%/* ... */%>` を使用します。JSP のコンパイル時に JSP のコメントが削除され、参照できなくなるからです。また、JSP ソースの保護強化のために、Administration Console の [ファイル] タブの [大文字 / 小文字を区別する] フィールドを無効にします。

- プロダクション環境では Servlet サーブレットを使用しないことをお勧めします。プロダクション環境で Web アプリケーションを使用する前には、すべての Web アプリケーションから、WebLogic サーブレットと Servlet サーブレットの間の既存のマッピングを全部、削除する必要があります。
- プロダクション環境では File サーブレットをデフォルト サーブレットにしないでください。

暗号化の使用

暗号化とは、理解できないようにテキストなどのデータの形式を変換する処理のことです。解読とは、逆に、テキストなどのデータを理解可能な形式に変換する処理のことです。解読処理には、プライベート キーまたはパスワードが必須です。プライベート キーはビットからなる長い文字列で、解読アルゴリズムが正しく機能するための引数として必要になります。暗号化アルゴリズムの強度は、鍵に含まれるビットの数を基準に評価されます。

暗号化のタイプは数多く、それぞれがさまざまな強度で機能します。各アルゴリズムで最も大きな違いは、データの解読にどれだけの CPU 時間が必要で、何個の鍵を使うか（対称鍵アルゴリズムでは暗号化と解読に鍵を 1 個しか使わず、公開鍵アルゴリズムでは暗号化用の 1 個と解読用の 1 個の合わせて 2 個の鍵を使います）という点です。

一般に暗号化は、重要な情報を保存したりやりとりしたりする場所で利用されます。こうした場所は、ネットワーク上のマシン、ディスク、データベース、メモリ、レガシー システムなどさまざまです。

暗号化には以下の欠点があります。

- 暗号化と解読は、CPU 時間を多く使い、コンピュータの負荷が大きなアルゴリズムです。
- 暗号化されたデータを評価してデータが正しいことを確認できないので、暗号化によってデバッグが困難になります。
- プライベート キーを紛失すると、暗号化されたデータがすべて無駄になるおそれがあります。プライベート キーが一時的に利用できない場合（プライベート キーを知る担当者が休暇中など）でも、プライベート キーを取り戻すまで、Web サイトが無用になることもあります。

- 鍵の管理は注意を要します。一部を挙げるだけでも、プライベート キーを誰が知り、どこに保管するか、鍵自体を暗号化するかといった問題があります。

WebLogic Server デプロイメントに関して暗号化を計画する際には、以下の質問の回答を検討してください。

- 何を暗号化するのか
- データの暗号化にどのアルゴリズムを使用して、どの程度の強度にするのか
- 鍵をどこに保管するのか

SSL プロトコルの使用

ネットワーク（インターネットまたはイントラネット）上を送信されるデータは、ネットワーク上の相手方が参照できます。ネットワークの目的からすると、これは避けることができません。重要なデータを危険から守るには、データを暗号化する必要があります。

インターネット上で暗号化済みデータを送信するには、HTTP プロトコルではなく、HTTPS プロトコル（セキュア ソケット レイヤ（SSL）上の HTTP）を使用すべきです。SSL プロトコルに合わせて Web アプリケーションをコンフィグレーションするには、`web.xml` ファイル内で `user-data-constraint` タグを使用し、transport-guarantee を `CONFIDENTIAL` に設定しなければなりません。

WebLogic Server 付属のデモ用デジタル証明書はテスト目的でのみ使用してください。WebLogic Server をダウンロードすると、これらのデジタル証明書用のプライベート キーが必ず付属します。これらのデジタル証明書は、デプロイ済みの WebLogic Server では使用しないでください。`filerealm.properties` ファイルをチェックして、デモ用デジタル証明書がデプロイ済み WebLogic Server から削除されていることを確認してください。

WebLogic Server によってサポートされている最もセキュリティの高い暗号化、1024 ビット キーと 128 バルク データ暗号化をデータに対して使用します。ダウンロード可能なバージョンの WebLogic Server では、512 ビット キーと 40 ビット バルク暗号化のみを使用できます。強度の高いバージョンをお求めになる場合は、BEA 販売代理店にお問い合わせください。

介在者の攻撃の防止

SSL プロトコルを使用している場合は、通信グループ間で送信されたデータが、介在者の攻撃に対して無防備になっていることがあります。介在者の攻撃は、ネットワークに配置されたマシンによって、無防備な通信先に対するメッセージが取り込まれたり、変更されたり、再転送されたりして発生します。介在者の攻撃を回避する 1 つの方法は、接続先のホストが予定していた通信先、または許可された通信先であることを確認することです。SSL クライアントでは、SSL サーバのホスト名と SSL サーバのデジタル証明書を比較して、接続を検証できます。WebLogic Server には、SSL 接続を介在者の攻撃から保護するためのホスト名検証が用意されています。

デフォルトでは、ホスト名検証は有効になっています。ただし、サイトへの WebLogic Server の実装時に、この機能が無効になっていた可能性もあります。WebLogic Server デプロイメントでホスト名検証が使用されるようにするには、Administration Console の [サーバ] ノードの [SSL] タブにある [ホスト名検証を無視] 属性がチェックされていないことを確認します。

サービス拒否攻撃の防止

サービス拒否攻撃を受けると、Web サイトは動作していても使用不能になります。ハッカーは、Web サイトの 1 つまたは複数の重要なリソースを消耗させたり、削除したりすることで、この攻撃を仕掛けます。サービス拒否攻撃は、ハッカーが WebLogic Server に対する管理者特権を得た場合に起こることもありますが、一般には、特権を持たないユーザが必要なリソースを WebLogic Server デプロイメントから削除した場合に発生します。

侵入者は WebLogic Server に対してサービス拒否攻撃を仕掛けるために、サイズが非常に大きく送信が終了するまでに時間がかかるリクエストや、リクエストが終了する前にクライアントがデータの送信を止めてしまうので完了することのないリクエストを大量に送信します。サービス拒否攻撃を防止するために、WebLogic Server では、メッセージのサイズだけでなく、メッセージの到着にかかる最長時間も制限することができます。この情報は、WebLogic Server が使用する 3 つのプロトコル、T3、HTTP、および IIOP のそれぞれに対して個別に設定できます。[最大 T3 メッセージ サイズ]、[T3 メッセージ タイムアウト]、[

最大 HTTP メッセージ サイズ]、[HTTP メッセージ タイムアウト]、[最大 IIOP メッセージ サイズ]、および [IIOP メッセージ タイムアウト] フィールドの設定の詳細については、Administration Console のオンライン ヘルプを参照してください。これらのフィールドには、最大メッセージ サイズとして 2GB、完了メッセージ タイムアウトとして 480 秒がデフォルト設定されています。どのフィールドでも値に 0 を指定すると、制限がチェックされなくなります。

HTTP 応答ヘッダのセキュリティ保護

WebLogic Server にサーバ自身の名前とバージョン番号を HTTP 応答で送信させないようにすることを検討しましょう。

デフォルトでは、WebLogic Server のインスタンスが HTTP リクエストに応答する際には、そのサーバの名前と WebLogic Server バージョン番号が HTTP 応答ヘッダに含まれています。そのため、WebLogic Server のそのバージョンに存在する何らかの脆弱性を攻撃者が知っていた場合には、セキュリティ上のリスクにつながるおそれがあります。

WebLogic Server インスタンスにサーバ自身の名前とバージョン番号を送信させないようにするには、Administration Console で [Send Server Header を有効化] 属性を無効にします。この属性は、[サーバ | *ServerName* | コンフィグレーション | プロトコル | HTTP] タブにあります。

ユーザ アカウントの保護

辞書攻撃では、ハッカーは「辞書」から取り出したパスワードを使用してログインを試みるスクリプトをセットアップします。WebLogic Server は、辞書攻撃からユーザ アカウントを守るコンフィグレーション可能な属性を提供します。このような属性のコンフィグレーションは、さまざまな方法で行うことができます（たとえば、すべての属性を無効にする、アカウントをロックするまでの不正なログインの試行回数を増やす、アカウントをロックするまでの不正なログインの試行が行われる期間を増やす、ユーザ アカウントのロック回数を変更するなど）。これらの属性をどう設定するかの判断は、サイト管理者の責任です。この

機能を使用して、最大限のセキュリティをかけてアカウントを保護します。WebLogic Server の出荷時には、セキュリティを最高レベルにするようになっています。

注意： 開発中に、これらの属性を変更してセキュリティ レベルを下げる場合、デプロイする前にその属性を戻すことを忘れないでください。

詳細については、「パスワードの保護」を参照してください。

アプリケーションのコンテンツの保護

デフォルトでは、WebLogic Server では Web ドキュメント ルート ディレクトリと呼ばれる 1 つのディレクトリが、静的なアプリケーションのコンテンツ（HTML ファイルと画像）および動的なアプリケーションのコンテンツ（JSP ファイルと jHTML ファイル）の格納場所として使用されます。Web ドキュメント ルート ディレクトリ内の、動的なコンテンツを含むファイルの作成または変更がアプリケーションで可能な場合には、攻撃される危険性があります。

アプリケーションで Web ドキュメント ルート ディレクトリ内の既存のファイルを変更できる場合には、既存のファイル内に JSP タグまたは jHTML タグの形式で実行可能コードを挿入されるおそれがあります。特定のファイルによって動的なコンテンツが提供される場合、次回、そのファイルがクライアントに提供されると、挿入されたコードが実行されます。

攻撃される危険性のある状況を回避するために、以下のような追加のセキュリティ対策を行うことをお勧めします。

- 1 つまたは複数の特定のユーザ アカウントに対する、特定のディレクトリおよびファイルへのアクセス制御機能がサポートされているディスクにのみ、WebLogic Server をインストールします。暗号化ファイル システムを使用すれば、セキュリティのレベルを高めることができます。ただし、パフォーマンスは犠牲になります。
- オペレーティング システム固有の特別なユーザ アカウント（`wls_owner` など）を WebLogic Server の実行用に設定します。このユーザ アカウントには、最小限のオペレーティング システムの権限およびアプリケーションの正常な実行に不可欠な特権だけを許可する必要があります。

- オペレーティング システム固有のユーザ アカウント（wls_owner）を、Web ドキュメント ルート ディレクトリ内のファイルにアクセスし、ファイルの作成または変更を行うことができる唯一のユーザ アカウントにします。このセキュリティ対策を行うと、WebLogic Server と同じマシンで実行されている他のアプリケーションの Web ルート ディレクトリへのアクセス機能を制限できます。
- WebLogic Server が実行されているオペレーティング システム固有のユーザ アカウント（wls_owner）によってのみアクセスおよび変更できるように、JSP ファイルまたは jHTML ファイルを格納しているディレクトリを保護します。root や Administrator などの管理アカウントには、アーカイブ化用に読み込み専用のアクセス権を付与できます。
- JSP ファイルまたは jHTML ファイルの作成に使用される、オペレーティング システム固有のユーザ アカウント（wls_owner）には、JSP ファイルまたは jHTML ファイルに対する読み込みおよび実行パーミッションのみを付与します。このセキュリティ対策によって、オペレーティング システム固有のユーザ アカウントによる、これらのファイルへの故意の書き込みを回避できます。
- WebLogic Server の実行に使用されているマシンから不要なアプリケーションを削除します。アプリケーションを削除できない場合は、そのアプリケーションが実行されるセキュリティ環境を見直します。特権を持つユーザ アカウント、setuid 特権を持つアプリケーションなどで実行されるアプリケーションがアクセスできるディレクトリを理解しておく必要があります。それ以外のアプリケーションでは、WebLogic Server が実行されるオペレーティング システム固有のユーザ アカウント（wls_owner）を使用しないことをお勧めします。
- WebLogic Server が実行されるオペレーティング システムで、ファイルおよびディレクトリへのアクセスのセキュリティ監査がサポートされている場合は、監査ログを使用して、拒否されたディレクトリまたはファイルへのアクセス違反を追跡することをお勧めします。
- プロダクション環境を変更する攻撃を検出するには、Intrusion Detection System（IDS）の使用を検討してください。

ユーザ入力データ中の HTML 特殊文字の置換

ユーザが入力したデータを返す機能があると、**クロスサイト スクリプティング**と呼ばれるセキュリティ上の弱点が発生し、ユーザのセキュリティ認可を盗むために利用される危険があります。クロスサイト スクリプティングの詳細については、http://www.cert.org/tech_tips/malicious_code_mitigation.html の「Understanding Malicious Content Mitigation for Web Developers」(CERT によるセキュリティ勧告) を参照してください。

このようなセキュリティ上の弱点を排除するには、ユーザが入力したデータを返す前に、HTML の特殊文字がデータに含まれるかどうかを調べます。特殊文字がある場合は、それを HTML の実体参照または文字参照に置き換えます。文字を置き換えることで、ユーザが入力したデータをブラウザが HTML として実行することを防止できます。

詳細については、「JSP でのユーザ入力データのセキュリティ対策」および「サーブレットでのクライアント入力のセキュリティ対策」を参照してください。

保護された EJB によるビジネス ロジックへのアクセス制限

Web アプリケーションには他よりも重要な部分があります。たとえば、アプリケーションの中で HTML を変更する部分は、キー データベース テーブルにアクセスする部分ほど重要ではありません。Web アプリケーションの重要部分の保護に関しては、さらに配慮が必要です。Web アプリケーションの重要部分を保護する方法の 1 つは、Web アプリケーションの該当部分をエンタープライズ JavaBean（EJB）でラップし、ACL を使ってその EJB を保護するというものです。この手法により、適切な認証を受け、認可を持つユーザだけがその EJB を実行できるようになります。

次の例は、EJB と ACL を使用して Web アプリケーションの重要部分を保護する方法を示します。

- ユーザが Web サイト上で発注できるようにするコードを、Web サイトの登録済みユーザだけにアクセスを限定した ACL で保護されている EJB に入れます。
- Web サイト上で製品カタログを検索して表示するコードを、すべてのユーザがアクセス可能な EJB に入れます。
- 商品の返品を認可するコードを、顧客サービス担当者だけがアクセス可能な ACL で保護された EJB に入れます。

何を保護して、特定の処理へのアクセスを誰に許可するかという判断は、アプリケーションごとに行う必要があります。

Web アプリケーションは徐々に発展していくものです。このため、方式を理解することだけでなく、管理することも困難になっていきます。後で混乱しないようにする方法の 1 つは、セキュリティをパッケージ単位でまとめておくことです。たとえば、あるパッケージでは、登録済みユーザがすべてのクラスのすべてのメソッドを使用でき、別のパッケージでは、顧客サービス担当者だけがすべてのクラスのすべてのメソッドを使用できるようにします。誰にどのアクセスを許可するかを最終的に決めるのは EJB デプロイヤですが、パッケージに基づくセキュリティ方式は、デプロイヤにとって実行しやすいメカニズムです。

ACL の使用

ACL は、WebLogic Server リソースへのアクセスを保護する複数のエントリを持つデータ構造です。WebLogic Server で提供する ACL は、以下の WebLogic Server リソースを保護します。

- WebLogic Server
- WebLogic Event
- WebLogic HTTP サーブレット /JSP/HTML ページ
- WebLogic JDBC 接続プール
- WebLogic JMS 送り先
- WebLogic JNDI コンテキスト
- WebLogic MBean

提供されている ACL を使用すると、WebLogic Server デプロイメントのリソースを保護できます。

WebLogic EJB の ACL およびパーミッションは、その他の WebLogic Server リソースの ACL およびパーミッションとは以下の点で異なります。

- EJB の ACL は、EJB のデプロイメント記述子のアクセス制御プロパティでコンフィグレーションされます。
- パーミッションは、Bean の個々のメソッドに対して付与され、あらかじめ定義されていません。
- EJB に対するパーミッションは、WebLogic Server 内のグループにマップされるロールに対して付与されます。

詳細については、以下の節を参照してください。

- 「ACL とパーミッション」
- 「セキュリティの管理」の「ACL の定義」
- 『WebLogic エンタープライズ JavaBeans プログラマーズ ガイド』で説明されている WebLogic EJB への ACL の割り当て方法
- 「Web アプリケーションでのセキュリティのコンフィグレーション」で説明されている WebLogic Web アプリケーションでのセキュリティ ロールの割り当て方法

適切なセキュリティ レルムの使用

WebLogic Server のセキュリティは、セキュリティ レルムという概念に基づいています。WebLogic Server 内のセキュリティ レルムは、ユーザ、グループ、ユーザとグループの関連データ、および WebLogic Server リソースに関してユーザとグループに割り当てられた許可からなるコレクションです。WebLogic Server には、さまざまなタイプのセキュリティ レルムが用意されています。

デフォルトのセキュリティ レルム、ファイル レルムは、最もセキュリティの低いレルムです。デプロイ済み WebLogic Server にはファイル レルムを使用しないでください。WebLogic Server には、セキュリティ レルムの機能を使って認証と許可を行う代替セキュリティ レルムもあります。使用可能なセキュリティ レルムの詳細については、第 2 章「セキュリティの基礎概念」を参照してください。

データベースのセキュリティ対策

ほとんどの Web アプリケーションはデータベースを使用してデータを保存します。WebLogic Server で使われるデータベースとしては、Oracle、MicroSoft の SQL Server、および Informix が一般的です。データベースには、顧客リスト、顧客の連絡先、クレジット カード情報、その他の独自の情報など、Web アプリケーションの重要なデータを保存することがよくあります。Web アプリケーションを作成する際には、データベースに保存するデータの種類とデータのセキュリティ レベルを考慮する必要があります。また、データベース製造元によるセキュリティ メカニズムを理解し、セキュリティ ニーズに十分に対応できるかどうかを判断することも必要です。メカニズムが十分でない場合は、他のセキュリティ手法を用いて、データベースのセキュリティを向上することができます。一般的な方法の 1 つは、重要なデータをデータベースに書き込む前に暗号化することです。たとえば、クレジット カード情報だけを暗号化して、その他の顧客データはプレーン テキストのままデータベースに保存するという方法でもよいでしょう。

監査の利用

監査とは、WebLogic Server 環境での重要なセキュリティ イベントを記録する処理のことです。通常、監査レコードは、WebLogic Server ログ ファイルとは別に保存されます。監査レコードを参照すると、セキュリティ侵害やその試みがあったかどうかを判断する手がかりになります。何度もログオンしようとして失敗しているとか、変わったパターンでセキュリティ イベントが起こっているといったレコードは、重要な問題を防止するヒントになる可能性があります。監査を使用するには、`weblogic.security.audit` パッケージを実装します。詳細については、「セキュリティの管理」の「監査プロバイダのインストール」を参照してください。

4 WebLogic Seurity SPI を使用したプログラミング

以下の節では、WebLogic Security SPI を使用したプログラミング方法について説明します。

- 始める前に
- WebLogic Security SPI
- JAAS 認証の使い方
- JNDI 認証の使い方
- SSL 対応 Web ブラウザとの安全な通信
- 相互認証の使い方
- カスタム ホスト名検証の使い方
- トラスト マネージャの使用
- SSL コンテキストの使用
- カスタム ACL の使い方
- カスタム セキュリティ レルムの作成
- セキュリティ イベントの監査
- ネットワーク接続のフィルタ処理

始める前に

この章では、WebLogic Server で提供されている Security サービス プロバイダ インタフェース（SPI）のアプリケーション プログラミング インタフェース（API）によるプログラミングについて説明します。この章で説明するプログラミング タスクを実行する前に、以下のコンフィグレーション タスクが完了している必要があります。

1. セキュリティ レルムの指定（デフォルトでは、代替セキュリティ レルムかカスタム セキュリティ レルム）
2. セキュリティ レルムへのユーザとグループの追加
3. セキュリティ レルム内のリソースへの ACL とパーミッションの割り当て
4. SSL プロトコルのコンフィグレーション（省略可能。ネットワーク接続の保護を追加する場合、または証明書認証を使用する場合）
5. 相互 SSL のコンフィグレーション（省略可能）
6. 証明書認証のコンフィグレーション（省略可能）

以上のコンフィグレーション タスクの詳細については、『管理者ガイド』の「セキュリティの管理」を参照してください。

WebLogic EJB のセキュリティの詳細については、『WebLogic エンタープライズ JavaBeans プログラマーズ ガイド』を参照してください。

WebLogic Security SPI

WebLogic Security SPI は、Java Developer's Kit（JDK）のセキュリティ SPI を基にしています。必要に応じた実装と拡張、およびセキュリティ API を WebLogic Server の認証および認可サービスに収集するレルム インタフェースを提供します。WebLogic Server の認証方式は、Java Authentication and Authorization Service（JAAS）に基づいています。JAAS では、WebLogic Server へのセキュリティ保護された接続を行う際に、ユーザ名と資格（パスワードかデジタル証明書）を提出するのに必要なサポートが提供されます。

表 4-1 は、WebLogic Server 環境でのセキュリティに使用されるパッケージを示しています。

表 4-1 WebLogic Security パッケージ

パッケージ名	使用目的
javax.security.auth	JAAS スタイルの LoginContext と主体ベースの認証を実行する場合。
weblogic.security	Web ブラウザと Java クライアントから送信されたデジタル証明書を WebLogic Server にマッピングする場合。このクラスを使用すると、有効なデジタル証明書を持っていれば、WebLogic Server のリソースにアクセスするときにユーザ名とパスワードを入力する必要がなくなる。
weblogic.security.acl	外部ストアから WebLogic Server のユーザ、グループ、または ACL にアクセスするためのカスタム セキュリティ レルムを作成する場合。このパッケージは、サーバサイド プログラムのカスタム ACL のテストにも使用する。
weblogic.security.audit	セキュリティ イベントを監査する場合。WebLogic Server は、認証および認可リクエストに関する情報を基に Audit クラスを呼び出す。このパッケージは、認証および認可リクエストをフィルタ処理してログ ファイルなどの管理機能に送信するのに使用できる。
weblogic.security.net	ネットワーク接続を開始したクライアントの IP アドレス、ドメイン、プロトコルなどの属性を基に接続を許可または拒否するために、WebLogic Server への接続を調べる場合。
weblogic.net.http.HTTPsURLConnection	別の WebLogic Server に対するクライアントとして機能している WebLogic Server から発信 SSL 接続を確立する場合。

表 4-1 WebLogic Security パッケージ （続き）

パッケージ名	使用目的
weblogic.security.SSL.HostNameVerifier	クライアント認証をサポートする。ホスト名検証クラス、トラスト マネージャ クラス、および SSL コンテキスト クラスが含まれる。

JAAS 認証の使い方

JAAS は、Java Development Kit バージョン 1.3 のセキュリティに対する標準拡張です。JAAS では、コードを実行するものに基づいてアクセス制御を実行できます。JAAS は、JNDI 認証メカニズムの代わりとして WebLogic Server で提供されています。WebLogic Server の認証方法としては、この JAAS が最も望ましいものです。JAAS を使用するには、Java SDK バージョン 1.3 をインストールする必要があります。

注意： JAAS の認可コンポーネントは WebLogic Server では提供されていません。

表 4-2 は、WebLogic Server でサポートされている JAAS クラスを示しています。

表 4-2 JAAS クラス

クラス	説明
javax.security.auth.Subject	リクエストの送信元を表す。どのエンティティ（個人またはクライアントなど）でも構わない。Subject オブジェクトは、ユーザ認証またはログインが完了すると作成される。
javax.security.auth.login.LoginContext	LoginContext オブジェクトを介して、アプリケーションはログインとログアウトを行い、認可チェックのために、認証された主体を取得する。
javax.security.auth.login.Configuration	WebLogic Server の特定の実装で提供される LoginModule のリストを取得するための getConfiguration() メソッドを提供する。

表 4-2 JAAS クラス （続き）

クラス	説明
`javax.security.auth.spi.LoginModule`	異なる種類の認証技術を WebLogic Server に実装できるようにする。たとえば、ある LoginModule オブジェクトではパスワードによる認証、別の LoginModule オブジェクトでは証明書による認証が可能になる。
`javax.security.auth.callback.Callback`	パスワードやデジタル証明書ファイルの名前など、ユーザからの入力をまとめて Java クライアントに渡す。
`javax.security.auth.callback.Callback.CallbackHandler`	LoginModule から主体に通信して認証情報を取得できるようにする。CallbackHandler インタフェースを実装して LoginContext に渡す。LoginContext は、基本となる LoginModule に直接それを転送する。LoginModule では、ユーザ入力（パスワードなど）の収集とユーザへの情報（ステータス情報など）提供の両方を CallbackHandler で行う。CallbackHandler を使用すると、各 LoginModule は、他の通信方法から独立した形で WebLogic Server とユーザとの通信を制御できる。

Java クライアントで JAAS を使用して主体を認証するには、以下の手順を実行します。

1. WebLogic Server で使用する認証メカニズム用に LoginModule クラスを実装します。認証メカニズムのタイプごとに LoginModule クラスが必要です。1 つの WebLogic Server デプロイメントに対して複数の LoginModule クラスを割り当てることができます。

   WebLogic Server では、LoginModule クラスの記述を簡便化することを目的にしたヘルパー クラス `weblogic.security.auth.Authenticate` が提供されています。`weblogic.security.auth.Authenticate` クラスでは、JNDI Environment オブジェクトを使用して認証された主体を返します。JNDI Environment オブジェクトには、表 4-3 で示されているプロパティが指定されている必要があります。

2. WebLogic Server でどの LoginModule クラスを使用し、どの順番で LoginModule クラスを呼び出すかを指定するために、Configuration クラスを実装します。
3. Java クライアントで、LoginContext をインスタンス化します。

   LoginContext は、Configuration にアクセスして、WebLogic Server 用にコンフィグレーションされた LoginModule をすべてロードします。
4. LoginContext の `login()` メソッドを呼び出します。

   `login()` メソッドによって、ロードされた LoginModule がすべて呼び出されます。各 LoginModule で主体の認証が試みられます。

   コンフィグレーションされたログイン条件に一致しなかった場合、LoginContext では LoginException が送出されます。
5. LoginContext から認証された主体を取得します。
6. 主体の認証に成功すると、`javax.security.auth.Subject` クラスの `doAs()` メソッドを呼び出せば、アクセス制御をその主体に設定できます。`doAs()` メソッドによって、指定した主体が現在のスレッドの ACL に関連付けられ、アクションが実行されます。主体に必要なアクセス制御がある場合、アクションは完了しますが、主体にそのアクセス制御がない場合、セキュリティ例外が送出されます。

`samples\examples\security` ディレクトリにある、WebLogic Server 付属の `examples.security.jaas` サンプルでは、Java クライアントの JAAS 認証の方法が示されています。

コード リスト 4-1 は、パスワード認証を実行する `javax.security.auth.spi.LoginModule` クラスの実装を示しています。コード リスト 4-1 内のコードは、`examples.security.jaas` パッケージの SampleLoginModule からの抜粋です。

コード リスト 4-1 パスワード認証用の LoginModule の例

```
...

// 適切なクラスをインポートする //
import java.util.Map;
import java.io.IOException;
import java.net.MalformedURLException;
import java.rmi.RemoteException;
```

```
import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.LoginException;
import javax.security.auth.login.FailedLoginException;
import javax.security.auth.spi.LoginModule;
import weblogic.security.auth.Authenticate;
import weblogic.jndi.Environment;

public class SampleLoginModule implements LoginModule
{
       private Subject subject = null;
       private CallbackHandler callbackHandler = null;
       private Map sharedState = null;
       private Map options = null;
       private String url = null;

// 認証ステータス
       private boolean succeeded = false;
       private boolean commitSucceeded = false;

// ユーザ名とパスワード
       private String username = null;
       private String password = null;

// 初期化

  public void initialize(Subject subject,
                    CallbackHandler callbackHandler,
                    Map sharedState,
                    Map options)

  {

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

// WebLogic Server URL 文字列を取得

              url = System.getProperty
                     ("weblogic.security.jaas.ServerURL");

// 渡されたユーザ名とパスワードを使用するユーザを認証
// 成功の場合は true を返す
// 認証に失敗した場合は FailedLoginException を送出
// LoginModule が認証を実行できなかった場合は、LoginException を
// 送出

public boolean login() throws LoginException
{
       // クライアントによってコールバック ハンドラが提供されたことを確認
```

```
        if(callbackHandler == null)
        throw new LoginException("No CallbackHandler Specified");

        // コールバック リストを指定 //
        Callback[] callbacks = new Callback[2];
        callbacks[0] = new NameCallback("username: ");
        callbacks[1] = new PasswordCallback("password: ", false);

        // ユーザ名とパスワードの入力を要求
        callbackHandler.handle(callbacks);

        // ユーザ名を取得
        username = ((NameCallback) callbacks[0]).getName();

        // パスワードを取得し、char[] を String に変換
        char[] charPassword = ((PasswordCallback)
        callbacks[1]).getPassword();

        if(charPassword == null)
        {
// NULL パスワードを NULL ではなく空のパスワードとして扱う
        charPassword = new char[0];
        }
                password = new String(charPassword);
        }

// WebLogic 環境と認証を指定
        Environment env = new Environment();
        env.setProviderUrl(url);
        env.setSecurityPrincipal(username);
        env.setSecurityCredentials(password);

// ユーザの資格を認証し、主体を指定
        Authenticate.authenticate(env, subject);

// 指定した情報で主体の認証に成功
        succeeded = true;
        return succeeded;

...
```

WebLogic Server では、デフォルトの LoginModule（`weblogic.security.internal.ServerLoginModule`）を使用して、サーバの初期化時に認証情報を収集します。デフォルトの Login モジュールを変更するには、`Server.policy` ファイルを編集し、デフォルト Login モジュール名をカスタム Login モジュール名に置き換えます。

WebLogic Server の JAAS 実装では、NameCallback、PasswordCallback、および TextInputCallback の各コールバックを、提供された LoginModule で使用できます。

server.policy ファイルで、カスタム Login モジュール名をデフォルト LoginModule の前に記述して指定することもできます。WebLogic Server の JAAS 実装では、server.policy ファイルで定義されている順番に従って Login モジュールが使用されます。デフォルト Login モジュールでは、実行の前に既存のシステム ユーザ認証が定義されているかどうかがチェックされ、既に定義されている場合は何の処理も行われません。

デフォルト Login モジュールでは、システム ユーザ名とパスワードの両方に対して JVM プロパティを定義する必要があります。それぞれ、weblogic.management.username と weblogic.management.password で指定します。カスタム Login モジュールを使用するには、各モジュールの仕様に従ってこのプロパティを設定します。

コード リスト 4-2 は、javax.security.auth.login.Configuration クラスの実装を示しています。コード リスト 4-2 内のコードは、examples.security.jaas パッケージの SampleConfig からの抜粋です。

コード リスト 4-2 コンフィグレーション実装の例

```
...
import java.util.Hashtable;
import javax.security.auth.login.Configuration;
import javax.security.auth.login.AppConfigurationEntry;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class SampleConfig extends Configuration

{
       String configFileName = null;

// Configuration オブジェクトを作成する
       public SampleConfig()

// アプリケーション名をインデックスとして Configuration からエントリを
// 取得する。このコードでは、単独の LoginModule を指定した

       public AppConfigurationEntry[]
              getAppConfigurationEntry(String applicationName)
       {

              AppConfigurationEntry[] list =
```

```
                                new AppConfigurationEntry[1];
                AppConfigurationEntry entry = null;

// 指定したコンフィグレーション ファイルを取得
        configFileName =
                System.getProperty("weblogic.security.jaas.Policy");
        System.out.println("Using Configuration File: " +
                configFileName);

        try
        {
                FileReader fr = new FileReader(configFileName);
                BufferedReader reader = new BufferedReader(fr);
                String line;

        line = reader.readLine();
        while(line != null)
        {

// 「{」で始まる行まで行をスキップ
        if(line.length() == 0 || line.charAt(0) != '{')
                {
                        line = reader.readLine();
                        continue;
        }
// コンフィグレーションした LoginModule を含む次の行を読み取る
        line = reader.readLine();

        int i;
        for(i = 0; i < line.length(); i++)
        {
                char c = line.charAt(i);
                if(c != ' ')
        break;
        }
                int sep = line.indexOf(' ', i);

        String LMName = line.substring(0, sep).trim();
        String LMFlag = line.substring(sep + 1)line.indexOf
                                            (' ', sep + 1));

        System.out.println("Login Module Name: " + LMName);
        System.out.println("Login Module Flag: " + LMFlag);

        if(LMFlag.equalsIgnoreCase("OPTIONAL"))
        {
                entry = new AppConfigurationEntry(LMName,
                        AppConfigurationEntry.LoginModuleControlFlag.
                        OPTIONAL,new Hashtable());
        list[0] = entry;
        }


        else if(LMFlag.equalsIgnoreCase("REQUIRED"))
        {
                entry = new AppConfigurationEntry(LMName,
                        AppConfigurationEntry.LoginModuleControlFlag.
```

```
                    REQUIRED, new Hashtable());
        list[0] = entry;
        }

        else if(LMFlag.equalsIgnoreCase("REQUISITE"))
        {
               entry = new AppConfigurationEntry(LMName,
                      AppConfigurationEntry.LoginModuleControlFlag.
                      REQUISITE, new Hashtable());
        list[0] = entry;
        }

        else if(LMFlag.equalsIgnoreCase("SUFFICIENT"))

         {
               entry = new AppConfigurationEntry(LMName,
                      AppConfigurationEntry.LoginModuleControlFlag.
                      SUFFICIENT,new Hashtable());
        list[0] = entry;

...
// 全 Login コンフィグレーションを更新および再ロード
        public void refresh()

...
```

コード リスト 4-3 は、JAAS 認証を使用する Java クライアントの例を示しています。コード リスト 4-3 内のコードは、examples.security.jaas パッケージの SampleClient からの抜粋です。Java クライアントには、javax.security.auth.callback.Callback.CallbackHandler クラスの実装があります。

コード リスト 4-3　JAAS 認証を使用する Java クライアントの例

```
// 必要なクラスをインポートする
import java.io.*;
import java.util.*;
import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.TextOutputCallback;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;
import javax.security.auth.login.FailedLoginException;
```

```
import javax.security.auth.login.AccountExpiredException;
import javax.security.auth.login.CredentialExpiredException;

public class SampleClient
{

// ユーザ認証を試行
       LoginContext loginContext = null;

// JAAS サーバ URL システム プロパティを設定し、LoginContext を作成
{

// SampleLoginModule のサーバ URL を設定、JAAS のコード例の
// LoginModule
       Properties property = new
                             Properties(System.getProperties());
       property.put("weblogic.security.jaas.ServerURL", args[0]);
       System.setProperties(property);

// コンフィグレーション クラス名を設定して、SampleConfiguration をロード
// JAAS コード例の Configuration
       property = new Properties(System.getProperties());
       property.put("weblogic.security.jaas.Configuration",
                    "examples.security.jaas.SampleConfig");
       System.setProperties(property);

// Configuration ファイル名を設定してサンプル コンフィグレーション ポリ
// シー ファイルをロード
       property = new Properties(System.getProperties());
       property.put("weblogic.security.jaas.Policy",
                    "Sample.policy");
       System.setProperties(property);

// LoginContext を作成
       loginContext = new LoginContext("SampleLoginModule", new
                     MyCallbackHandler());
}

// 認証を試行
       loginContext.login();

// 認証された主体を取得し、SampleAction を主体として実行
       Subject subject = loginContext.getSubject();
       SampleAction sampleAction = new SampleAction();
       Subject.doAs(subject, sampleAction);

// CallbackHandler インタフェースの実装
class MyCallbackHandler implements CallbackHandler

{
       public void handle(Callback[] callbacks) throws IOException,
       UnsupportedCallbackException
{
       for(int i = 0; i < callbacks.length; i++)
{
       if(callbacks[i] instanceof TextOutputCallback)
{
```

```
// 指定したタイプに従ってメッセージを表示
       TextOutputCallback toc = (TextOutputCallback) callbacks[i];
       switch(toc.getMessageType())
       {
              case TextOutputCallback.INFORMATION:
              System.out.println(toc.getMessage());
       break;
              case TextOutputCallback.ERROR:
              System.out.println("ERROR: " + toc.getMessage());
       break;
              case TextOutputCallback.WARNING:
              System.out.println("WARNING: " + toc.getMessage());
       break;
              default:
              throw new IOException("Unsupported message type: " +
              toc.getMessageType());
              }
       }
       else if(callbacks[i] instanceof NameCallback)

      {
// ユーザ名の入力を要求
       NameCallback nc = (NameCallback) callbacks[i];
       System.err.print(nc.getPrompt());
       System.err.flush();
       nc.setName((new BufferedReader(new
       InputStreamReader(System.in))).readLine());
       }else if(callbacks[i] instanceof PasswordCallback)

      {
// パスワードの入力を要求
       PasswordCallback pc = (PasswordCallback) callbacks[i];
       System.err.print(pc.getPrompt());
       System.err.flush();

// JAAS は、パスワードが String ではなく char[] であるように指定
       String tmpPassword = (new BufferedReader(new
       InputStreamReader(System.in))).readLine();

              int passLen = tmpPassword.length();
              char[] password = new char[passLen];
              for(int passIdx = 0; passIdx < passLen; passIdx++)
              password[passIdx] = tmpPassword.charAt(passIdx);
              pc.setPassword(password);
       }
       else
       {
       throw new UnsupportedCallbackException(callbacks[i],
       "Unrecognized Callback");
       }
...
```

JAAS 使用の詳細については、『Java Authentication and Authorization Service Developer's Guide』を参照してください。

JNDI 認証の使い方

Java クライアントでは JNDI を使用して資格を渡すこともできます。Java クライアントは、JNDI InitialContext を取得して WebLogic Server との通信を確立します。その後、InitialContext を使用して、WebLogic Server JNDI ツリーで必要なリソースをルックアップします。

ユーザとユーザの資格を指定するには、次の表で示されている JNDI プロパティを設定します。

表 4-3 認証に使用される JNDI プロパティ

プロパティ	意味
`INITIAL_CONTEXT_FACTORY`	エントリ ポイントを WebLogic Server 環境に提供する。`weblogic.jndi.WLInitialContextFactory` クラスは WebLogic Server 用の JNDI SPI。
`PROVIDER_URL`	WebLogic Server のホストとポートを指定する。例：`t3://weblogic:7001`

表 4-3 認証に使用される JNDI プロパティ （続き）

プロパティ	意味
SECURITY_AUTHENTICATION	使用する認証のタイプを示す。以下の値を使用できる。 ■ None の場合、認証は実行されない。 ■ Simple の場合、パスワードによる認証が実行される。 ■ Strong の場合、証明書による認証が実行される。 **注意：** WebLogic Server 上のセキュア コンポーネントへのアクセスを試みる場合、SECURITY_AUTHENTICATION の設定によって指定される認証のタイプに関係なく、WebLogic Server でユーザ認証が必須となる。たとえば、SECURITY_AUTHENTICATION を None に設定すると、セキュア コンポーネントにアクセスする場合にも正しいパスワードの指定が必要になる。
SECURITY_PRINCIPAL	ユーザが WebLogic Server セキュリティ レルムに対して認証されているときのユーザの ID を指定する。
SECURITY_CREDENTIALS	ユーザが WebLogic Server セキュリティ レルムに対して認証されているときのユーザの資格を指定する。 ■ SECURITY_AUTHENTICATION="simple" で有効化するパスワード認証の場合、このプロパティは、ユーザのパスワードを表す文字列か、または資格の検証に使用される User オブジェクトを表す文字列を指定する。 ■ SECURITY_AUTHENTICATION="strong" で有効化する証明書認証の場合、このプロパティは、WebLogic Server のデジタル証明書とプライベート キーを格納する X509 オブジェクトの名前を指定する。

これらのプロパティは、InitialContext コンストラクタに渡されるハッシュテーブルに格納されます。

コード リスト 4-4 は、Java クライアントでパスワード認証を使用する方法を示しています。コード リスト 4-4 のコードは、samples\examples\security ディレクトリにある、WebLogic Server 付属の examples.security.acl サンプルの Client からの抜粋です。

コード リスト 4-4 パスワード認証の例

```
...
Hashtable env = new Hashtable();
       env.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.jndi.WLInitialContextFactory");
       env.put(WLContext.PROVIDER_URL, "t3://weblogic:7001");
       env.put(WLContext.SECURITY_AUTHENTICATION "simple");
       env.put(Context.SECURITY_PRINCIPAL, "javaclient");
       env.put(Context.SECURITY_CREDENTIALS, "password");

       ctx = new InitialContext(env);
```

コード リスト 4-5 は、Java クライアントで証明書認証を使用する方法を示しています。SSL プロトコルを介した WebLogic Server 独自のプロトコルである T3S プロトコルの使用に注意してください。SSL プロトコルは、WebLogic Server と Java クライアントの間の接続および通信を保護します。

コード リスト 4-5 証明書認証の例

```
...
Hashtable env = new Hashtable();
       env.put(Context.INITIAL_CONTEXT_FACTORY,
           "weblogic.jndi.WLInitialContextFactory");
       env.put(WLContext.PROVIDER_URL, "t3s://weblogic:7001");
       env.put(WLContext.SECURITY_AUTHENTICATION "strong");
       env.put(Context.SECURITY_PRINCIPAL, "javaclient");
       env.put(Context.SECURITY_CREDENTIALS, "certforclient");

       ctx = new InitialContext(env);
```

コード リスト 4-4 と コード リスト 4-5 のコードでは、ユーザ名とパスワードが正しい場合や、デジタル証明書が有効な場合に User オブジェクトを返す、`weblogic.security.acl.Security.getUser()` の呼び出しが生成されます。WebLogic Server では、この認証された User オブジェクトは WebLogic Server 内の Java クライアントのスレッドに格納されます。格納された User オブジェクトは、以降の認可リクエストで、スレッドが ACL で保護されているリソースにアクセスする際に使用されます。

注意： JNDI コンテキストとスレッドについて、および JNDI コンテキストで発生する可能性のある問題を避ける方法については、『WebLogic JNDI プログラマーズ ガイド』の「JNDI コンテキストとスレッド」および「JNDI コンテキストの潜在的な問題を避ける方法」を参照してください。

SSL 対応 Web ブラウザとの安全な通信

URL オブジェクトを使用すると、別の WebLogic Server に対するクライアントとして機能している WebLogic Server から発信 SSL 接続を確立することができます。`weblogic.net.http.HttpsURLConnection` クラスを使用して、プライベート キーとデジタル証明書を含む、クライアントのセキュリティ コンテキスト情報を指定できます。

`weblogic.net.http.HttpsURLConnection` クラスは、ネゴシエーションされた暗号スイートの判別、ホスト名検証の取得と設定、サーバの認証チェーンの取得、および新しい SSL ソケットを作成するための SSLSocketFactory の取得と設定を行うメソッドを提供します。

SSL クライアントのコード例には、`weblogic.net.http.HttpsURLConnection` クラスを使用して発信 SSL 接続を確立する方法が示されています。さらに、SSL クライアントのコード例には、Java Secure Socket Extension（JSSE）アプリケーション プログラミング インタフェース（API）を使用して、発信 SSL 接続を確立する方法も示されています。SSL クライアントのコード例は、`\samples\examples\security\sslclient` ディレクトリの `examples.security.sslclient` パッケージにあります。

相互認証の使い方

証明書認証では、WebLogic Server は、リクエストを発したクライアントにデジタル証明書を送信します。クライアントは、デジタル証明書を調べて、本物かどうか、期限切れでないかどうか、認証元の WebLogic Server に一致しているかどうかを確認します。

相互認証では、リクエストを発したクライアントの側も WebLogic Server にデジタル証明書を送信します。Administration Console の各フィールドを設定して WebLogic Server のコンフィグレーションを変更すると、指定した認証局からデジタル証明書を送信するようにリクエスト元のクライアントに要求できるようになります。WebLogic Server では、指定した認証局のルート証明書によって署名されたデジタル証明書のみが受け付けられます。詳細については、「セキュリティの管理」の「SSL プロトコルのコンフィグレーション」を参照してください。

以降の節では、相互認証を WebLogic Server に実装するさまざまな方法について説明します。

注意： Java クライアントの認証で JAAS を使用する場合、相互認証を実行する LoginModule クラスを記述します。

JNDI による相互認証

Java クライアントの認証で JNDI を使用する場合、WebLogic JNDI Environment クラスの `setSSLClientCertificate()` メソッドを使用します。このメソッドは、クライアント認証に対して、X.509 デジタル証明書のプライベート キーと認証チェーンを設定します。Java クライアントのデジタル証明書とプライベート キーを提供するには、デジタル証明書とプライベート キーを含む Definite Encoding Rules（DER）ファイルを X509 オブジェクトに読み込んで、JNDI ハッシュ テーブルにその X509 オブジェクトを設定します。「JNDI 認証の使い方」で説明されている JNDI プロパティを使用して、認証に必要な情報を指定します。

デジタル証明書を JNDI に渡すには、DER エンコードされたデジタル証明書を格納するファイルで開かれている `InputStreams` の配列を作成し、JNDI ハッシュ テーブルにその配列を設定します。配列内の最初の要素には、Java クライアントのプライベート キー ファイルで開かれている `InputStream` が格納されている

必要があります。配列内の 2 番目の要素には、Java クライアントのデジタル証明書ファイルで開かれている InputStream が格納されている必要があります（このファイルには Java クライアントの公開鍵が含まれています）。追加要素として、ルート認証局のデジタル証明書、認証チェーン内のデジタル証明書の署名者を格納できます。デジタル証明書が fileRealm.properties ファイルの Java クライアントで登録された認証局によって直接発行されない場合、WebLogic Server は、認証チェーンを使用して Java クライアントのそのデジタル証明書を認証できます。

weblogic.security.PEMInputStream クラスを使用すると、Privacy Enhanced Mail（PEM）ファイルに保存されているデジタル証明書を読み込むことができます。このクラスでは、ベース 64 でエンコードされた DER 認証を PEM ファイルにデコードするフィルタが提供されます。

コード リスト 4-6 は、Java クライアントで相互認証を使用する方法を示しています。コード リスト 4-6 のコードは、samples\examples\security ディレクトリにある、WebLogic Server 付属の examples.security.acl サンプルの AltClient からの抜粋です。

コード リスト 4-6 相互認証の例

```
package examples.security.acl;

import java.io.FileInputStream;
import java.io.InputStream;
import javax.naming.Context;
import weblogic.jndi.Environment;
import weblogic.security.PEMInputStream;


public class AltClient
{

  public static void main(String[] args)
  {
    Context ctx = null;

    String url = args[0];
    try
    {
      Environment env = new Environment();

      env.setProviderUrl(url);

      // 2 番目と 3 番目の引数はユーザ名とパスワード
      if (args.length >= 3)
      {
```

```
env.setSecurityPrincipal(args[1]);
env.setSecurityCredentials(args[2]);
      }

      // 4 番目と 5 番目の引数はプライベート キーと
      // 公開鍵
      if (url.startsWith("t3s") && args.length >= 5)
      {
InputStream[] certs = new InputStream[args.length - 3];
for (int q = 3; q < args.length; q++)
{
  String file = args[q];
  InputStream is = new FileInputStream(file);

  if (file.toLowerCase().endsWith(".pem"))
  {
    is = new PEMInputStream(is);
  }
  certs[q - 3] = is;
}
env.setSSLClientCertificate(certs);
      }
      ctx = env.getInitialContext();
      ...
```

JNDI の `getInitialContext()` メソッドが呼び出されると、Java クライアントと WebLogic Server は、Web ブラウザが相互認証を実行してセキュリティ保護された Web サーバ接続を取得するのと同じ方法で相互認証を実行します。デジタル証明書を確認できない場合や Java クライアントのデジタル証明書をセキュリティ レルムで認証できない場合、例外が送出されます。認証された User オブジェクトは Java クライアントのサーバ スレッドに格納され、ACL 保護された WebLogic Server リソースへの Java クライアントのアクセスを管理するパーミッションのチェックに使用されます。

WebLogic JNDI Environment クラスを使用する場合、`getInitialContext()` メソッドの呼び出しごとに Environment オブジェクトを作成する必要があります。ユーザとセキュリティ資格を一度指定すると、ユーザおよびユーザに関連する資格は Environment オブジェクトの設定に残ります。再設定を試みて JNDI `getInitialContext()` メソッドを呼び出した場合は、元のユーザと資格が使用されます。

Java クライアントによる相互認証を使用する場合、WebLogic Server は、クライアント JVM ごとにユニークな Java 仮想マシン（JVM）ID を取得し、Java クライアントと WebLogic Server の間の接続が切断されないようにします。処理がないために接続がタイムアウトになるまで、Java クライアントの JVM が実行されて

いる間は接続が続行されます。Java クライアントが新しい SSL 接続を確実にネゴシエーションできる唯一の方法は、その JVM を停止して JVM の他のインスタンスを実行することです。

SSL 接続の JVM で動作している Java クライアントは、新しい JNDI `InitialContext` を作成し、JNDI `SECURITY_PRINCIPAL` と `SECURITY_CREDENTIALS` プロパティで新しいユーザ名とパスワードを指定して WebLogic Server ユーザの ID を変更できます。SSL 接続後に Java クライアントによって渡されたデジタル証明書は使用されません。新しい WebLogic Server ユーザは、最初のユーザのデジタル証明書でネゴシエーションした SSL 接続を使用し続けます。

CertAuthenticator インタフェースを実装している場合、WebLogic Server は、Java クライアントに対するデジタル証明書を CertAuthenticator クラスの実装に渡します。CertAuthenticator クラスは、デジタル証明書を WebLogic Server ユーザにマッピングします。デジタル証明書は、JVM からの最初の接続リクエスト時にのみ処理されるため、CertAuthenticator クラスを使用するときに新しいユーザ ID 設定することはできません。

注意： JNDI コンテキストとスレッドについて、および JNDI コンテキストで発生する可能性のある問題を避ける方法については、『WebLogic JNDI プログラマーズ ガイド』の「JNDI コンテキストとスレッド」および「JNDI コンテキストの潜在的な問題を避ける方法」を参照してください。

WebLogic Server ユーザへのデジタル証明書のマッピング

相互認証を実行すると、WebLogic Server は、SSL 接続を確立するために Web ブラウザまたは Java クライアントのデジタル証明書を認証します。ただし、デジタル証明書は Web ブラウザまたは Java クライアントを WebLogic Server セキュリティ レルムのユーザとしては認識しません。Web ブラウザまたは Java クライアントが ACL で保護された WebLogic Server リソースをリクエストすると、WebLogic Server は Web ブラウザまたは Java クライアントにユーザ名とパスワードを指定するように要求します。

Web ブラウザまたは Java クライアントを WebLogic Server セキュリティ レルム内のユーザにマップするには、`weblogic.security.acl.CertAuthenticator` インタフェースを実装します。SSL 接続が確立されると、CertAuthenticator クラスが呼び出されます。このクラスは、デジタル証明書からデータを抽出し、どのユーザがデジタル証明書のオーナかを決定します。次に、CertAuthenticator クラスは `weblogic.security.acl.getUser()` メソッドを呼び出して認証された User オブジェクトを WebLogic Server セキュリティ レルムから取得します。

CertAuthenticator クラスがインストールされると、Web ブラウザで WebLogic Server ユーザ名の入力を要求する必要はなくなります。また、Java アプリケーションの JNDI `SECURITY_CREDENTIALS` プロパティでパスワードを設定する必要もなくなります。詳細については、「セキュリティの管理」の「SSL プロトコルのコンフィグレーション」を参照してください。

Java クライアント アプリケーションで CertAuthenticator クラスを使用する場合、SSL 接続が確立されると Java クライアントはユーザ ID を変更できません。新しいデジタル証明書を提供するには、Java クライアントの JVM を停止して、新しい JVM インスタンスでクライアントを再起動し、SSL 接続をネゴシエーションできるようにします。

いくつかあるメソッドのいずれかを使用すると、デジタル証明書をユーザにマップできます。たとえば、ユーザのパスワードをユーザのデジタル証明書のフィンガープリントに設定するという方法があります。次に、デジタル証明書からユーザ名を抽出し、フィンガープリントを計算し、ユーザがユーザ名とパスワードを提出してリソースにアクセスするときに WebLogic Server が行うのと同じ方法で `weblogic.security.acl.getUser()` メソッドを呼び出すことができます。

注意： デジタル証明書のフィンガープリントは証明書の一部ではありませんが、証明書から計算できます。フィンガープリントは、DER でエンコードした `CertificateInfo` の MD5 ダイジェストです。これは、X.509 仕様に含まれている ANS.1 タイプです。

CertAuthenticator クラスは、引数なしのパブリック コンストラクタを持っており、`authenticate()` メソッドを呼び出します。WebLogic Server は、ユーザ名（空のままでも構いません）、Java クライアントまたは認証チェーンのデジタル証明書を格納した `Certificate` 配列、SSL ハンドシェークが成功した場合に `true` となるブール値を指定した `authenticate()` メソッドを呼び出します。`Certificate` 配列でメソッドを呼び出してデジタル証明書からデータを取得できます。

コード リスト 4-7 は、CertAuthenticator インタフェースの実装方法を示しています。このインタフェースでは、デジタル証明書の電子メール アドレスからユーザ名を抽出し、weblogic.security.acl.getUser() メソッドを呼び出して WebLogic Server セキュリティ レルムから認証された User オブジェクトを取得します。コード例では、電子メール アドレスの一部のみを調べています。そのため、セキュリティについては万全ではありません。異なるドメインで同じ電子メール アドレスを持つ複数のデジタル証明書は、同じユーザにマップされ、追加認証が実行されない場合があります。この機能を実装する場合、クライアントの ID を完全に確立するコードを追加することもできます。

コード リスト 4-7 は、WebLogic Server セキュリティ レルムのユーザにデジタル証明書をマップする方法も示しています。コード リスト 4-7 のコードは、samples\examples\security ディレクトリにある、WebLogic Server 付属の examples.security.cert サンプルの SimpleCertAuthenticator からの抜粋です。

コード リスト 4-7 WebLogic Server ユーザへのデジタル証明書のマッピングの例

```
package examples.security.cert;


import weblogic.security.Certificate;
import weblogic.security.Entity;
import weblogic.security.X500Name;
import weblogic.security.acl.CertAuthenticator;
import weblogic.security.acl.BasicRealm;
import weblogic.security.acl.Realm;
import weblogic.security.acl.User;

public class SimpleCertAuthenticator
  implements CertAuthenticator
{
  private BasicRealm realm;

  public SimpleCertAuthenticator()
  {
    realm = Realm.getRealm("weblogic");
  }


  /**
   * リモート ユーザの認証を試行
   *
   * @ この例では、param userName は無視
   * @ param certs を使用して 電子メール アドレスから
   * @ WebLogic ユーザへのマップを試行
```

```
   * @ param ssl が false の場合、この例では null を返す
   * @ 認証されたユーザまたは null（認証失敗の場合）を返す
   */
  public User authenticate(String userName, Certificate[] certs,
boolean ssl)
  {
    // この実装では、成功した相互 SSL ハンドシェークによる
    // 証明書のみを信頼する
    if (ssl == false)
    {
      return null;
    }

    User result = null;
    Certificate cert = certs[0];
    Entity holder = cert.getHolder();

    if (holder instanceof X500Name)
    {
      X500Name x500holder = (X500Name) holder;
      String email = x500holder.getEmail();

      if (email != null)
      {
int at = email.indexOf("@");

if (at > 0)
{
  String name = email.substring(0, at);

  // 電子メール アドレスから抽出したユーザが
  // 実際に存在することを確認
  result = realm.getUser(name);
}
      }
    }

    return result;
  }
}
```

コード リスト 4-7 のコンストラクタでは、サーバサイド クラスで WebLogic Server セキュリティ レルムにアクセスする方法が示されています。`weblogic.security.acl.realm` クラスの `getRealm("weblogic")` メソッドは、ファイル レルムか代替セキュリティ レルム（LDAP セキュリティ レルムなど）に関係なく、WebLogic Server で使用されるレルムを返します。

weblogic.security.X500Name クラスには、weblogic.security.Certificate クラスからフィールドを取得するためのアクセサ メソッドが含まれています。コード リスト 4-7 では、Certificate オブジェクトを X500Name オブジェクトにキャストし、X500Name オブジェクトの getEmail() メソッドを呼び出して、電子メール アドレスの最初の部分文字列を取得しています。weblogic.security.acl.AbstractableRealm クラスの getUser(String) メソッドは、指定したユーザ名の WebLogic Server ユーザを取得します。ユーザが存在しない場合は、weblogic.security.acl.AbstractableRealm クラスの authenticate() メソッドは null を返します。

他の WebLogic Server との相互認証の使い方

一方の WebLogic Server がもう一方の WebLogic Server のクライアントとして機能するサーバ間通信で相互認証を使用できます。サーバ間通信で相互認証を使用すると、一般的なクライアント / サーバ環境でない場合でも、高度なセキュリティの接続に依存することができます。

コード リスト 4-8 は、WebLogic Server で動作しているサーブレットから server2.weblogic.com という 2 番目の WebLogic Server に対して、セキュリティ保護された接続を確立しています。

コード リスト 4-8　他の WebLogic Server へのセキュリティ保護された接続の確立

```
...

FileInputStream [] f = new FileInputStream[3];
   f[0]= new FileInputStream("demokey.pem");
   f[1]= new FileInputStream("democert.pem");
   f[2]= new FileInputStream("ca.pem");

Environment e = new Environment ();
e.setProviderURL("t3s://server2.weblogic.com:443");
e.setSSLClientCertificate(f);
e.setSSLServerName("server2.weblogic.com");
e.setSSLRootCAFingerprints("ac45e2d1ce492252acc27ee5c345ef26");

T3Client t3c = e.createProviderClient();
t3c.connect();
```

```
e.setInitialContextFactory
      (“weblogic.jndi.WLInitialContextFactory”);
Context ctx = new InitialContext(e.getProperties())

...
```

コード リスト 4-8 では、WebLogic JNDI Environment クラスは、以下のパラメータを格納するハッシュ テーブルを作成します。

- `setProviderURL` ― クライアントとして機能する WebLogic Server の URL。URL では、SSL プロトコルを基にした、WebLogic Server 独自のプロトコルである T3S プロトコルを指定します。SSL プロトコルは、2 つの WebLogic Server 間の接続および通信を保護します。
- `setSSLClientCertificate` ― 接続に使用する認証チェーンを指定します。認証チェーンは、信頼できる認証局のプライベート キー、それに対応する公開鍵、およびデジタル証明書のチェーンを含む配列です。各認証局は前のデジタル証明書の発行元です。
- `setSSLServerName` ― デジタル証明書を提示する WebLogic Server の名前を指定します。名前は、WebLogic Server のデジタル証明書の共通名フィールドと比較されます。このパラメータは、介在者の攻撃を防ぐために使用されます。
- `setSSLRootCAFingerprint` ― 指定したフィンガープリントを基にデジタル証明書が認証局によって発行される必要があることを指定します。このパラメータは、介在者の攻撃を防ぐために使用されます。

注意： JNDI コンテキストとスレッドについて、および JNDI コンテキストで発生する可能性のある問題を避ける方法については、『WebLogic JNDI プログラマーズ ガイド』の「JNDI コンテキストとスレッド」および「JNDI コンテキストの潜在的な問題を避ける方法」を参照してください。

サーブレットによる相互認証の使い方

Java クライアントをサーブレット（または他のサーバサイド Java クラス）で認証するには、クライアントがデジタル証明書を提供したかどうかをチェックする必要があります。提供した場合は、証明書が信頼できる認証局で発行されたかどうかをさらにチェックします。サーブレットの作成者には、Java クライアント

が有効なデジタル証明書を持っているかどうかを尋ねる役割があります。WebLogic Servlet API でサーブレットを作成する場合、`HTTPServletRequest` オブジェクトの `getAttribute()` メソッドで SSL 接続に関する情報にアクセスする必要があります。

以下の属性が、WebLogic Server サーブレットでサポートされています。

- `javax.net.ssl.cipher_suite` ― 使用中の暗号スイートを文字列で返します。
- `javax.net.ssl.session` ― 暗号スイートを含む SSL Session オブジェクト、および SSL Session オブジェクトが作成された日付と最後に使用された日付を返します。
- `javax.net.ssl.rootCA` ― 指定した認証局からデジタル証明書を文字列で返します。
- `javax.net.ssl.rootCADigest` ― 指定した認証局から証明書のダイジェストを文字列で返します。
- `javax.net.ssl.peer_certificates` ― クライアントの認証チェーンに関する情報を取得し、デジタル証明書の配列をオブジェクトとして返します。すると、その配列を `weblogic.security.X509` にキャストできるようになります。

デジタル証明書に定義されているユーザ情報にアクセスできます。デジタル証明書には、以下のような情報が指定されています。

- 主体（保持者、オーナ）の名前と、デジタル証明書を使用する Web サーバの URL や個人の電子メール アドレスなど、その主体の ID をユニークに確認するために必要なその他の情報
- 主体の公開鍵
- デジタル証明書を発行した認証局の名前
- シリアル番号
- デジタル証明書の有効期間（開始日と終了日で定義）

デジタル証明書を提供する X509 オブジェクトで `to_string()` メソッドを呼び出すと、各デジタル証明書の使用可能な情報を参照できます。

X509 オブジェクトをコンストラクタに渡すと、デジタル証明書に対する weblogic.security.JDK11Certificate オブジェクトを作成できます。weblogic.security.JDK11Certificate クラスは、java.security.Certificate インタフェースを実装します。

コード リスト 4-9 は、X509 オブジェクトの配列内の最初の X509 オブジェクトを JDK111Certificate オブジェクトに変換しています。

コード リスト 4-9 X509 オブジェクトの JDK111Certificate オブジェクトへの変換

```
weblogic.security.JDK11Certificate jdk11cert =
       new weblogic.security.JDK111Certificate(X509certs [0]);

print(out, "jdk11cert.getPrincipal().getName() -",
            jdk11cert.getPrincipal().getName() );
print(out, "jdk11cert.getGuarantor().getName() -",
            jdk11cert.getGuarantor().getName() );
```

weblogic.security.JDK11Certificate クラスには、デジタル証明書に関する情報を提供する以下のメンバー関数があります。

- getIssuerCertificate() ― 発行元のデジタル証明書を java.security.Certificate オブジェクトとして返します。
- getFingerprint() ― デジタル証明書のフィンガープリントを返します。フィンガープリントは、DER エンコードされたデジタル証明書の MD5 です。同じフィンガープリントを持つ複数のデジタル証明書を構築することは困難になります。
- getSubjectOrgUnit() ― デジタル証明書内の指定した名前の Organizational ユニットを含む文字列を返します。

カスタム ホスト名検証の使い方

ホスト名検証は、SSL 接続先のホストが予定していた通信先、または許可された通信先であることを確認します。WebLogic Server または WebLogic クライアントが別のアプリケーション サーバの SSL クライアントとして機能している場合には、介在者の攻撃を防ぐことができて便利です。

WebLogic Server の SSL ハンドシェーク機能としてのデフォルトの動作は、SSL サーバのデジタル証明書の SubjectDN にある共通名と、SSL 接続の開始に使用する SSL サーバのホスト名を比較することです。これらの名前が一致しない場合は SSL 接続が中断されます。

SSL 接続の中断は、サーバのホスト名をデジタル証明書と照らし合わせて有効性を検証する SSL クライアントによって実行されます。デフォルト以外の動作が必要な場合は、ホスト名検証を無効にするか、カスタム ホスト名検証を登録します。ホスト名検証を無効にすると、WebLogic Server は介在者の攻撃に対して無防備な状態になります。

注意： ホスト名検証は、WebLogic Server に付属のデモ用デジタル証明書を使用しているときに無効にします。

ホスト名検証は、以下の方法で無効にできます。

- Administration Console で、[サーバ] ノードの [SSL] タブで [ホスト名検証を無視] 属性をチェックします。
- SSL クライアントのコマンドラインで、次の引数を入力します。

```
-Dweblogic.security.SSL.ignoreHostnameVerification=true
```

カスタム ホスト名検証を記述できます。`weblogic.security.SSL.HostnameVerifier` インタフェースではコールバック メカニズムが提供されるため、接続先のサーバ名がサーバのデジタル証明書の SubjectDN にあるサーバ名と一致しない場合を処理するためのポリシーを定義できます。

カスタム ホスト名検証を使用するには、`weblogic.security.SSl.HostnameVerifier` インタフェースを実装するクラスを作成し、サーバのセキュリティ ID に関する情報を取得するメソッドを定義します。

カスタム ホスト名検証を使用する前に、以下の方法で実装するクラスを定義する必要があります。

- Administration Console で、[サーバ] ノードの [SSL] タブで [ホスト名の検証] 属性のホスト名検証のクラスを定義します。
- コマンドラインに、次の引数を入力します。

  `-Dweblogic.security.SSL.HostnameVerifier=`*`hostnameverifier`*

  `hostnameverifier` は、カスタム ホスト名検証を実装するクラスの名前です。

デフォルトでない JDK プロトコル ハンドラを使用して接続を確立するには、以下の 2 つの関数を呼び出してハンドラを初期化する必要があります。

```
weblogic.net.http.Handler.init();
weblogic.management.application.Handler.init();
```

カスタム ホスト名検証の例については、`\samples\examples\security\sslclient` ディレクトリの `examples.security.sslclient` パッケージにある SSL クライアントのコード例を参照してください。このコード例には、比較のために常に true を返す NullHostnameVerifier クラスが含まれています。このサンプルでは、WebLogic SSL クライアントは、サーバのホスト名とデジタル証明書の SubjectDN との比較に関係なく、どの SSL サーバにも接続できます。

ホスト名検証のインスタンスは、`setHostnameVerifier` メソッドを使用して SSL コンテキストに関連付けることができます。次に例を示します。

```
public void setHostnameVerifier (HostnameVerifier hv)
```

詳細については、「SSL コンテキストの使用」を参照してください。

トラスト マネージャの使用

`weblogic.security.SSL.TrustManager` クラスを使用すると、ピアのデジタル証明書内での検証エラーをオーバーライドし、SSL ハンドシェークを継続できます。また、サーバのデジタル証明書チェーンで付加的な検証を実行することで、SSL ハンドシェークを中止することもできます。

SSL クライアントが SSL サーバに接続すると、SSL サーバは認証のためにデジタル証明書チェーンをクライアントに提示します。提示されたチェーンに無効なデジタル証明書が含まれている場合もあります。SSL 仕様では、クライアントが無効な証明書を検出した場合、SSL 接続が中断されることになっています。しかし、Web ブラウザは、証明書チェーン内の残りの証明書を使用して SSL サーバを認証できるかどうかを判別するため、無効な証明書を無視してチェーンの検証を継続します。トラスト マネージャを使用すると、どのような場合に SSL 接続を継続するか（または中止するか）を制御でき、上記のような矛盾した動作をなくすことができます。

トラスト マネージャを作成するには、`weblogic.security.SSL.TrustManager` クラスを使用します。このクラスには、証明書を検証するための一連のエラーコードが含まれています。また、必要に応じて、ピア証明書での付加的な検証を実行したり、SSL ハンドシェークを中断したりできます。デジタル証明書の検証が済むと、`weblogic.security.SSL.TrustManager` クラスがコールバック関数を使用してデジタル証明書の検証結果をオーバーライドします。トラスト マネージャのインスタンスは、`setTrustManager()` メソッドを使用して SSL コンテキストに関連付けることができます。
`weblogic.security.SSL.TrustManager` クラスは、JSSE 仕様に準拠しています。トラスト マネージャを使用しても、パフォーマンスには影響ありません。トラスト マネージャはプログラムでのみ設定できます。その使用は、Administration Console やコマンドラインでは定義できません。

トラスト マネージャの使用例は、`\samples\examples\security\sslclient` ディレクトリに格納されています。

- 次の例では、トラスト マネージャで SSL コンテキストを使用して新しい SSL 接続を設定する方法を示しています。

  `\samples\examples\security\sslclient\SSLSocketClient`

- 次の例では、常に true を返すカスタム トラスト マネージャを示しています。

  `\samples\examples\security\sslclient\NulledTrustManager`

SSL コンテキストの使用

SSL コンテキストを使用すると、特定の SSL 接続のホスト名検証やトラスト マネージャなど、さまざまな情報を格納するセキュア ソケット プロトコルを実装できます。SSL コンテキスト クラスのインスタンスは、SSL ソケット用のファクトリとして使用します。たとえば、SSL コンテキストが提供するソケット ファクトリによって作成されたすべてのソケットは、SSL コンテキストに関連付けられたハンドシェーク プロトコルを使用することによってセッション ステートを一致させることができます。各インスタンスは、認証の実行に必要なキー、認証チェーン、および信頼されたルート CA を使用してコンフィグレーションできます。これらのセッションはキャッシュされます。このため、同じ SSL コンテキストで作成された他のソケットは、その後もセッションを再利用できます。セッションのキャッシュについての詳細は、『管理者ガイド』の「SSL セッション キャッシングのパラメータの変更」を参照してください。トラスト マネージャ クラスのインスタンスを SSL コンテキストに関連付けるには setTrustManager メソッドを使用します。

SSL コンテキストは、プログラムでのみ設定できます。Administration Console やコマンドラインでは設定できません。SSL コンテキスト オブジェクトは、Java の new 式または SSL コンテキスト クラスの getInstance() ファクトリ メソッドで作成できます。getInstance() ファクトリ メソッドは静的で、セキュア ソケット プロトコルを実装するためのインスタンスを返します。SSL コンテキストの使用例は、`\samples\examples\security\sslclient` ディレクトリに格納されています。

- 次の例では、SSL コンテキストを使用して新しい SSL ソケットを作成する SSL ソケット ファクトリの新規作成方法について示しています。

  `\samples\examples\security\sslclient\SSLSocketClient`

SSL コンテキストは、Sun Microsystems の Java Secure Socket Extension（JSEE）に準拠した上位互換コードです。

カスタム ACL の使い方

WebLogic Server では、リソースを保護するアクセス制御リスト（ACL）の標準セットが定義されます。リソースに対して ACL を作成すると、WebLogic Server では、ユーザがアクセスする前にそのリソースに対するパーミッションが自動的にチェックされます。WebLogic Server のほとんどのリソースは、この標準 ACL によって完全に保護されます。

ただし、一部のリソースでは、標準 ACL よりも詳細な保護が必要になります。WebLogic Server では、そのようなリソースに対するセキュリティを強化できます。たとえば、特定のデータを Web ページに書き込む前にユーザ パーミッションをチェックするサーブレットを作成することが可能です。`weblogic.security.acl.Security` クラスは、ACL のチェックなど、レルムの操作に対するアクセス権を提供します。このクラスはサーバサイド コードに対してのみ利用できます。

`weblogic.security.acl.Security` クラスの `Security.hasPermission()` メソッドと `Security.checkPermission()` メソッドは、リソースへのアクセスに必要なパーミッションをユーザが持っているかどうかをテストします。この 2 つのメソッドはよく似ていますが、`Security.hasPermission()` メソッドではブール値（ユーザがパーミッションを持っている場合は `true`）が返されるのに対して、`Security.checkPermission()` メソッドでは、ユーザがパーミッションを持っていない場合は `java.lang.SecurityException` が送出されるという違いがあります。

WebLogic Server 付属の `examples.security.acl` サンプル（`samples\examples\security` ディレクトリにある）では、独自の ACL を作成してサーバサイド クラスでテストする方法が示されています。この例では、`frob` というパーミッションを持つカスタム ACL `aclexample` で、RMI クラス `FrobImpl` が保護されます。

注意： カスタム ACL を使用する前に、WebLogic Server で使用するセキュリティ レルムに ACL を組み込む必要があります。

カスタム ACL を使用するには、Java クライアント アプリケーションは以下の作業を行う必要があります。

1. JNDI InitialContext を WebLogic Server から取得します。

2. パーミッション名で、WebLogic Server JNDI ツリー内の保護されているリソースをルックアップします。たとえば、Java クライアントはパーミッション名 frob で FrobImpl をルックアップします。
3. RMI スタブの frob() メソッドを実行します。

FrobImpl クラスには、ACL をテストするサーバサイド コードがあります。コード リスト 4-10 に、weblogic.security.acl.realm クラスの静的メソッド checkPermission() でカスタム ACL をテストする方法を示します。

コード リスト 4-10 デフォルト セキュリティ レルムでのカスタム ACL のテスト

```
Security.checkPermission(Security.getCurrentUser(),
          "aclexample",
          Security.getRealm().getPermission("frob"),
          '.');
```

カスタム ACL を代替セキュリティ レルムでテストすることもできます。

1. weblogic.security.acl.realm クラスの getRealm(realm_name) メソッドで代替セキュリティ レルムを取得します。
2. weblogic.security.acl.realm クラスの getACL() メソッドと getPermission() メソッドでカスタム ACL とパーミッションを取得します。
3. acl.checkPermission() メソッドを呼び出してパーミッションをテストします。

コード リスト 4-11 は、この方法でカスタム ACL をテストしています。

コード リスト 4-11 代替セキュリティ レルムでのカスタム ACL のテスト

```
    User p = Security.getCurrentUser();
    BasicRealm realm = Realm.getRealm(realm_name);
    Acl acl = realm.getAcl(acl_name);
    Permission perm = realm.getPermission(permission_name);
    boolean result = acl == null || !acl.checkPermission(p, perm);
```

コード リスト 4-11 の最後の行では、カスタム ACL があったかどうか、ユーザ p が frob パーミッションを持っているかどうかをテストしています。戻り値は通常の感覚と逆です。ACL が存在し、ユーザが frob パーミッションを持っている場合、結果は false になります。

セキュリティ レルムが getACL() 機能を実装していない場合、java.lang.UnsupportedOperationException 例外を送出します。これで、ACL ルックアップはセカンダリ レルムに返ります。セカンダリ レルムがコンフィグレーションされていない場合には、ランライム エラーが発生します。

セキュリティ イベントを監査し、コード リスト 4-12 の方法でパーミッションをテストする場合、パーミッション テストを監査するには、weblogic.security.audit パッケージの静的クラス Audit を明示的に呼び出す必要があります。Audit クラスを呼び出すと、WebLogic Server の AuditProvider クラスに対してパーミッション チェック イベントの通知が生成されます。

コード リスト 4-12 パーミッションのテスト

```
Audit.checkPermission("Frob", acl, p, perm, !result);
```

カスタム ACL を使用する完全なコード例については、samples\examples\security ディレクトリにある、WebLogic Server 付属の examples.security.acl パッケージを参照してください。

カスタム セキュリティ レルムの作成

ネットワーク上のディレクトリ サーバなどの環境にある既存のセキュリティ ストアを基に独自のセキュリティ レルムを作成できます。認証をサポートするカスタム セキュリティ レルムを作成するには、以下のコードを記述する必要があります。

1. カスタム セキュリティ レルムの User クラスを定義するコード
2. カスタム セキュリティ レルムの Group クラスを定義するコード

3. セキュリティ ストアのユーザとグループを返し、完了時にセキュリティ ストアのリソースを解放する列挙値クラスを定義するコード
4. カスタム セキュリティ レルムのクラスを定義するコード
5. セキュリティ ストアに関するコンフィグレーション データを取得するコード
6. ユーザを認証するコード
7. グループのメンバーを返し、グループのメンバーを含むハッシュ テーブルを作成するコード
8. ユーザ名が指定された User オブジェクトを返すコード
9. グループ名が指定された Group オブジェクトを返すコード
10. ユーザの列挙値を使用してセキュリティ ストアの全ユーザの User オブジェクトを返すコード
11. グループの列挙値を使用してセキュリティ ストアの全グループの Group オブジェクトを返すコード

認可をサポートするカスタム セキュリティ レルムを記述することもできます。詳細については、「カスタム セキュリティ レルムでの認可の使い方」を参照してください。

注意： WebLogic Server では、WebLogic Server Administration Console で管理可能なカスタム セキュリティ レルムを作成することもできます。詳細については、`weblogic.security.acl` パッケージの Javadoc を参照するか、BEA プロフェッショナル サービスにお問い合わせください。

カスタム セキュリティ レルムから別のサーバに対する RMI 呼び出しを実行しないでください。RMI 呼び出しを使うと、サーバのソケット リーダー スレッドが不足するおそれがあります。

表 4-4 は、カスタム セキュリティ レルムの作成に使用される WebLogic クラスを示しています。

表 4-4 カスタム セキュリティ レルムの作成に使用される WebLogic クラス

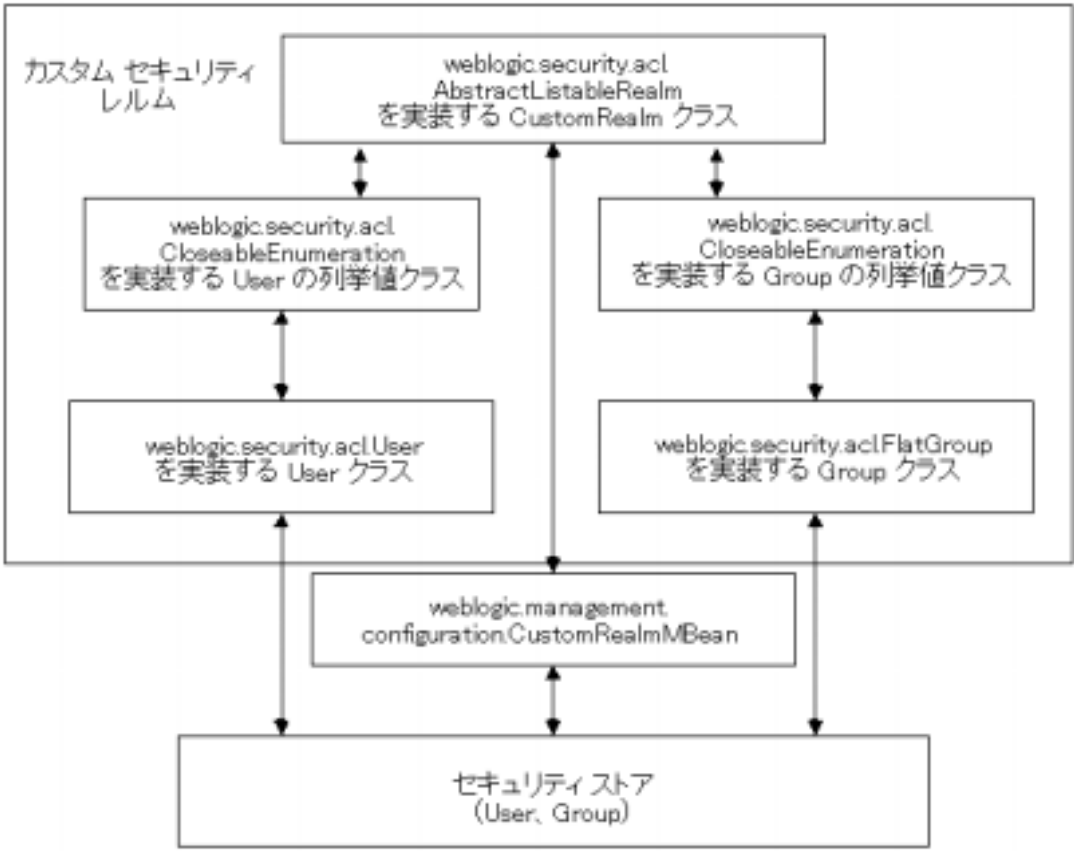
クラス	定義
`weblogic.security.acl.User`	ローカル セキュリティ ストアから取得されるユーザを定義する。

表 4-4 カスタム セキュリティ レルムの作成に使用される WebLogic クラス

クラス	定義
weblogic.security.acl.FlatGroup	ローカル セキュリティ ストア更新時にメンバシップが更新されるグループを定義する。
weblogic.security.acl.CloseableEnumeration	閉じた後にリソースを解放できる列挙値を定義する。
weblogic.security.acl.AbstractListableRealm	ユーザ、グループ、ACL、およびパーミッションを Admission Console で表示できるセキュリティ レルムの作成を可能にする。ただし、使用しているセキュリティ ストアによって提供される機能を使用して、ユーザ、グループ、ACL を追加および削除し、ユーザとグループにパーミッションを割り当てる必要がある。
weblogic.security.acl.RefreshableRealm	Administration Console に表示されるユーザ、グループ、ACL、パーミッションに関する情報と、ローカル セキュリティ ストアの情報との同期を取る。
weblogic.management.configuration.CustomRealmMbean	カスタム セキュリティ レルムからアクセスされるセキュリティ ストアに関するコンフィグレーション情報を取得する。

図 4-1 は、これらのクラスを組み合わせてどのようにカスタム セキュリティ レルムを作成するかを示しています。

図 4-1 カスタム セキュリティ レルムの作成に使用する WebLogic クラス



以降の節では、セキュリティ レルムの記述とカスタマイズに必要なプログラミング タスクについて説明します。

User クラスの定義

`weblogic.security.acl.User` クラスを拡張してカスタム セキュリティ レルムの User クラスを作成します。

コード リスト 4-13 は、User クラスを定義するコードを示しています。

コード リスト 4-13 User クラスの定義

```
// 必要なクラスをインポートする
import weblogic.security.acl.user;
...

// カスタム レルムのカスタム User クラスを作成
   /*package */class CustomRealmUser
     extends User
  {
    // ユーザのカスタム レルムを追跡
    CustomRealm realm = null;

    // コンストラクタを実装する
    /*package*/ CustomRealmUser(String name, CustomRealm realm);

      {
        // ユーザの名前を渡す基本コンストラクタを
        // 呼び出す
        super(name);

        // このユーザのレルムを追跡
        this.realm = realm;
      }
        // ユーザのカスタム レルムを返す
        public BasicRealm getRealm()
         {
            return realm;
          }
    }
...
```

Group クラスの定義

グループによってセキュリティの管理が容易になります。カスタム セキュリティ レルムの内部では、グループは、ユーザまたはグループになるメンバーのリストを格納するハッシュ テーブルとして表されます。

カスタム セキュリティに Group クラスを実装するには、`weblogic.security.acl.Flatgroup` クラスを拡張し、メンバシップ情報が定義されていない新しい Group クラスを作成します。クラスのコンストラクタは、カスタム セキュリティ レルムに対応する目的のグループとレルム オブジェクトの名前を入力値として取得します。

Flatgroup クラスは、カスタム セキュリティ レルムで動作するように特に設計されています。カスタム セキュリティ レルムは定期的にグループ メンバシップを更新します。FlatGroup では、静的セットの代わりにキャッシュにグループ メンバシップを保持します。キャッシュの期限が切れると、グループ実装はセキュリティ ストアに問い合わせて最新のメンバシップ情報を取得します。デフォルトのグループのキャッシュ時間は 5 分です。つまり、基になるストアを変更すると、5 分以内にカスタム セキュリティ レルムで認識されます。この値は、Administration Console の [グループ メンバシップ キャッシュ 生存時間] フィールドの値を、グループがキャッシュに保持される秒数に設定して調整します。

コード リスト 4-14 は、Group クラスを定義するコードを示しています。

コード リスト 4-14 Group クラスの定義

```
// 必要なクラスをインポートする
import weblogic.security.acl.FlatGroup;
...

/*package*/ class CustomRealmGroup
extends FlatGroup
{
   // コンストラクタを実装する
   /*package*/ CustomRealmGroup(String name,
      CustomRealm realm);
     {
       // グループとカスタム セキュリティ レルムの名前を
       // 渡す基本クラスのコンストラクタを呼び出す
       super(name, source);
     }
       // カスタム セキュリティ レルムの User クラスを返す
       // Group クラスを使用
       protected Class getUserClass()
     {
        return CustomRealmUser.class;
     }
}
...
```

ユーザとグループの列挙値クラスの定義

ユーザとグループの列挙値クラスを記述します。列挙値が、列挙の完了時に解放する必要があるリソース（たとえば、データベース カーソル）を保持している場合は、weblogic.security.acl.CloseableEnumeration クラスを実装します。それ以外の場合は java.util.Enumeration インタフェースを実装します。列挙値のコンストラクタでは、セキュリティ ストアへのアクセスに必要な引数を渡します。

カスタム セキュリティ レルムでユーザまたはグループごとに User オブジェクトまたは Group オブジェクトを作成し、それを列挙するハッシュ テーブルに配置しないでください。デプロイされた WebLogic Server のカスタム セキュリティ レルムは、メモリの許容分より多くのユーザとグループを保持することがあります。代わりに、必要に応じて徐々に User オブジェクトまたは Group オブジェクトを作成できるデータベース カーソルを使用します。

コード リスト 4-15 は、ユーザとグループの列挙値クラスを定義するコードを示しています。

コード リスト 4-15 ユーザとグループの列挙値クラスの定義

```
// 必要なクラスをインポートする
import weblogic.security.acl.FlatGroup;
import weblogic.security.acl.ClosableEnumeration;

...
// ユーザの列挙値クラスを定義
/*package*/ class CustomRealmUsersEnumeration
 implements CloseableEnumeration
{
    // ここにデータ メンバー（データベース カーソルなど）を保持

      // 列挙値のセキュリティ レルムを追跡
      // （User コンストラクタで使用する場合）
      private CustomRealm realm      = null;

      // コンストラクタを実装する
      /*package */ CustomRealmUsersEnumeration(...,
                           CustomRealm realm)
      {
        this.realm     = realm;
      }

      // より多くのユーザがいるかどうかを判別するメソッドを実装
```

```
        public boolean hasMoreElements()
        {
          // たとえば、データベース カーソルを使用して
          // より多くのユーザがいるかどうかを確認
          return (there are more users ...) ? true : false;
        }

          // 次のユーザを返すメソッドを実装
          // メソッドは、カスタム セキュリティ レルムの
          // User クラスを使用する User オブジェクトを返す
          public Object nextElement()
          {
            // たとえば、データベース カーソルを使用して
            // 次のユーザの名前を取得
            return new CustomRealmUser(next user name ..., realm);
          }

            // 列挙を終了するメソッドを実装
            // この手順は省略可能
            public void close()

          {
            // この列挙が、閉じる必要があるイテレータに
            // 委託している場合（データベース カーソルなど）、
            // ここでリソースを解放
          }
}

// カスタム セキュリティ レルムの Group クラスを作成
/*package*/class CustomRealmGroupsEnumeration
   implements CloseableEnumeration
{
   // ここにデータ メンバー（データベース カーソルなど）を保持

     // 列挙値のセキュリティ レルムを追跡
     // （Group コンストラクタで使用する場合）
     private CustomRealm realm      = null;

       // コンストラクタを実装する
       /*package */ CustomRealmUsersEnumeration(...,
                          CustomRealm realm)
       {
         this.realm     = realm;
       }

       // より多くのグループがあるかどうかを判別するメソッドを実装
       public boolean hasMoreElements()
       {
         // たとえば、データベース カーソルを使用して
         // より多くのグループがあるかどうかを確認
```

```
        return (there are more groups ...) ? true : false;
      }

        // 次のグループを返すメソッドを実装
        // メソッドは、カスタム セキュリティ レルムの
        // Group クラスを使用する Group オブジェクトを返す
        public Object nextElement()
        {
          // たとえば、データベース カーソルを使用して
          // 次のグループの名前を取得する
          return new CustomRealmGroup(next group name ..., realm);
        }

        // 列挙を終了するメソッドを実装
        // この手順は省略可能
        public void close()

        {
          // この列挙が、閉じる必要があるイテレータに
          // 委託している場合（データベース カーソルなど）、
          // ここでリソースを解放
        }
}
...
```

カスタム セキュリティ レルムのクラスの定義

weblogic.security.acl.AbstractListableRealm クラスを拡張して、カスタム セキュリティ レルム用に新しいクラスを定義し、カスタム セキュリティ レルムを作成するコンストラクタを実装します。このクラスでは、以下の作業を行う必要があります。

1. セキュリティ ストアに関するコンフィグレーション データを取得します。
2. ユーザを認証します。
3. グループのメンバーを決定します。
4. セキュリティ ストアからユーザとグループを取得します。

WebLogic Server では、ユーザとグループの情報がメモリにキャッシュされます。したがって、カスタム セキュリティ レルムでは、ユーザとグループの情報が必要になるたびにディスクを参照するようお勧めします。WebLogic Server システム管理者は、セキュリティ ストアに付属のツールでストア内の情報を編集

してカスタム セキュリティ レルムのユーザまたはグループを変更することのみが可能です。システム管理者は、セキュリティ ストア内のユーザまたはグループ情報を変更したら、Administration Console の [リセット] ボタンをクリックして情報を更新する必要があります。これによって、メモリに保持されているユーザとグループの情報が自動的にフラッシュされます。

カスタム セキュリティ レルムでユーザまたは グループの情報をメモリにキャッシュする場合は、`weblogic.security.acl.RefreshableRealm` クラスとこのクラスの `refresh()` メソッドを実装します。システム管理者がレルムをリセットすると、`refresh()` メソッドが呼び出されます。`refresh()` メソッドを使用すると、キャッシュされているユーザまたはグループ情報が破棄されます。

コード リスト 4-16 は、カスタム セキュリティ レルムのクラスを定義するコードを示しています。

コード リスト 4-16 カスタム セキュリティ レルムのクラスの定義

```
...
// 必要なクラスをインポート
import weblogic.security.acl.AbstractListableRealm;
import weblogic.security.acl.BasicRealm;
import weblogic.security.acl.RefreshableRealm;
import weblogic.server.Server;

// カスタム セキュリティ レルムのクラスを作成
public class CustomRealm
   extends AbstractListableRealm // 必須
   implements Refreshable Realm // 省略可能

// カスタム セキュリティ レルムを作成するコンストラクタを実装
public CustomRealm()
{
       super("Custom Realm");
...

public void refresh()
{
       // メモリ内のユーザおよびグループ情報を破棄
}
```

セキュリティ ストアに関するコンフィグレーション データの取得

セキュリティ ストアに接続するには、カスタム セキュリティ レルムで使用されるセキュリティ ストアにアクセスする方法を指定するコンフィグレーション プロパティ（URL を格納するプロパティやディレクトリ パスを指定するプロパティなど）を定義する必要があります。このプロパティを定義したら、WebLogic Server のシステム管理者は Administration Console の [新しい Custom Realm の作成] ウィンドウの [コンフィグレーション] タブでプロパティを設定します。セキュリティ ストアのプロパティを WebLogic Server の管理環境で定義してからカスタム セキュリティ レルムを使用する必要があります。

たとえば、ユーザとグループ情報を含むファイルへのパスを指定する `userInfoFileName` と `groupInforFileName` という 2 つのプロパティを定義します。システム管理者は、Administration Console の [新しい Custom Realm の作成] ウィンドウの [コンフィグレーション] タブでそれらのプロパティを入力します。

カスタム セキュリティ レルムのコードで、`weblogic.management.Helper` および `weblogic.management.configuration.DomainMbean` クラスを使用してカスタム セキュリティのコンフィグレーション プロパティを取得し、そのプロパティの値を基にセキュリティ ストアに接続します。

コード リスト 4-17 は、セキュリティ ストアのコンフィグレーション プロパティを取得してセキュリティ ストアに接続するコードを示しています。

コード リスト 4-17 セキュリティ ストアへのアクセス

```
...
// 必要なクラスをインポート
import weblogic.management.Helper;
import weblogic.management.configuration.DomainMBean;

MBeanHome mHome = Helper.getMBeanHome(user, password, url,
                                      servername);
DomainMBean domainMBean = mHome.getActiveDomain();
SecurityMBean secMBean = domainMBean.getSecurity();
BasicRealmMBean basicRealmMBean =
              secMBean.getRealm().getCachingRealm.getBasicRealm();

CustomRealmMBean customRealmMBean =
  (CustomRealmMBean)basicRealmMBean;
```

```
// CustomRealmMBean からコンフィグレーション データを取得
Properties configData = customRealmMBean.getConfigurationData();

// カスタム セキュリティ ストアのプロパティを取得
String userInfoFileName =
       configData.getProperty("UserInfoFileName);
String groupInfoFileName =
       configData.getProperty("GroupInfoFileName);
```

ユーザの認証

カスタム セキュリティ レルムでユーザをパスワード認証する場合は、authUserPassword() メソッドを明示的に実装する必要があります。authUserPassword() メソッドで指定したユーザが指定したパスワードで存在することを確認するためにセキュリティ ストアをチェックするコードを記述します。

- ユーザが存在し、パスワードが有効な場合、User オブジェクトを返します。カスタム セキュリティ レルムの User クラスを使用します。
- ユーザが存在しないか、パスワードが不正な場合、認証は失敗します。この場合、認証失敗を示す null を返します。

コード リスト 4-18 は、ユーザを認証するコードを示しています。

コード リスト 4-18 ユーザの認証

```
...
// ユーザを認証するメソッドを実装
protected User authUserPassword(String name, String password)
{
   // セキュリティ ストアをチェックし、
   // パスワードが password の name というユーザがいるかどうかを確認

   if (...) {
            return new CustomRealmUser(name, this);
   }  else   {
       return null;
  }
}
```

グループのメンバーの決定

weblogic.security.acl.AbstractListableRealm クラスの getGroupMembersInternal() メソッドを使用してグループのメンバーを取得し、グループのメンバーでハッシュ テーブルを構築します。

コード リスト 4-19 は、グループのメンバーを取得するコードを示しています。

コード リスト 4-19 グループのメンバーの取得

```
...
// グループのメンバーを返すメソッドを実装
protected Hashtable getGroupMembersInternal(String name)
{
       // セキュリティ ストアをチェックし、指定した名前の
       // グループがあるかどうかを確認
       if (!...)  {

       // グループが存在しない場合は、例外を送出
       throw new CustomRealmException("No such group : " + name);
    }

       // グループ メンバーを格納してから返されるハッシュ テーブルを
       // 作成

       Hashtable members = new Hashtable();

       // セキュリティ ストアからグループのメンバーを取得
       for (...) {// メンバーをループ処理
          if (...) {
          // このメンバーがユーザの場合、このユーザ用の
          // User オブジェクトを作成して、それをメンバーのリストに追加
          // カスタム セキュリティ レルム用に作成された
          // User クラスを使用
          members.put(memberName, new CustomRealmUser
                    (memberName, this));
          }  else if
          // このメンバーがグループの場合、このグループ用の
          // Group オブジェクトを作成して、それをメンバーのリストに追加
          // カスタム セキュリティ レルム用に作成された
          // User クラスを使用
          members.put(memberName, new CustomRealmGroup
                    (memberName, this));
          }
    }
    // グループのメンバーを格納するハッシュ テーブルを返す
    return members;
  }
...
```

セキュリティ ストアからのユーザとグループの取得

weblogic.security.acl.AbstractListableRealm クラスの getUser() メソッドと getGroup() メソッドを実装して、セキュリティ ストアのユーザとグループを取得し、カスタム セキュリティ レルムに取り込みます。weblogic.security.acl.AbstractListableRealm クラスの getUsers() メソッドと getGroups() メソッドを使用して、セキュリティ ストアのユーザとグループに対する User オブジェクトと Group オブジェクトを返す列挙オブジェクトを返します。セキュリティ ストア内のユーザとグループは Administration Console で表示できます。

コード リスト 4-20 は、セキュリティ ストアからユーザとグループを取得するコードを示しています。

コード リスト 4-20 セキュリティ ストアからのユーザとグループの取得

```
...
// 指定したユーザ名の User オブジェクトを返すメソッドを実装
public User getUser(String name)
{
  // セキュリティ ストアをチェックし、指定した名前の
  // グループがあるかどうかを確認
  if (...) {
  // ユーザがいる場合は、そのユーザの User オブジェクトを返す
  // カスタム セキュリティ レルム用に作成された User クラスを使用
  return new CustomRealmUser(name, this);
  } else {
  // ユーザがいない場合は null を返す
  return null;
  }
}

// 指定したグループ名の Group オブジェクトを返すメソッドを実装
public Group getGroup(String name)
{
  // セキュリティ ストアをチェックし、指定した名前で
  // グループがあるかどうかを確認
  if (...) {
  // グループがある場合は、そのグループの Group オブジェクトを返す
  // カスタム セキュリティ レルム用に作成された Group クラスを使用
  return new CustomRealmGroup(name, this
  } else {
  // グループがない場合は null を返す
  return null;
  }
}
```

```
// カスタム セキュリティ レルムで全グループに繰り返し処理を行うのに
// 使用できる列挙オブジェクトを返すメソッドを実装
public Enumeration getUsers()
{
  // セキュリティ ストア内のユーザの User オブジェクトを返す列挙オブ
  // ジェクトを返す
  // カスタム セキュリティ レルム用に作成されたユーザ列挙クラスを使用
  // 列挙値からセキュリティ ストアにアクセスできるようにすること
  // それによって、複数のユーザに対して繰り返し処理が可能になる
  return new CustomRealmUsersEnumeration(..., this);
}

// カスタム セキュリティ レルムで全グループに繰り返し処理を行うのに
// 使用できる列挙オブジェクトを返すメソッドを実装
public Enumeration getGroups()
{
  // セキュリティ ストア内のグループの Group オブジェクトを返す列挙オブ
  // ジェクトを返す
  // カスタム セキュリティ レルム用に作成されたグループ列挙クラスを使用
  // 列挙クラスからセキュリティ クラスにアクセスできるようにすること
  // それによって複数のグループに対して繰り返し処理が可能になる
  return new CustomRealmGroupsEnumeration(..., this);
}
...
```

注意： パフォーマンスを向上させるには、`weblogic.security.acl.FlatGroup` クラスの `isMember()` メソッドを使用します。このメソッドは、すべてのグループ メンバーをフェッチする代わりに、個別のユーザがグループのメンバーであるかどうかのチェックを行います。

カスタム セキュリティ レルムでの認可の使い方

カスタム セキュリティ レルムで ACL を作成するには、新しい AclImpl オブジェクトを作成し、名前を設定して、ユーザまたはグループごとに AclEntryImpl オブジェクトを追加します。各 AclEntry オブジェクトには、ユーザまたはグループを表す特定のプリンシパルに関連付けられた一連のパーミッションが格納されています。

AclImpl インタフェースでは、ACL の作成時に順序付け制約が強制的に適用されます。AclEntryImpl オブジェクトを AclImpl オブジェクトに追加する前に、AclEntryImpl オブジェクトにすべてのパーミッションを追加する必要があります。

カスタム セキュリティ レルムのクラスでは、`getAcl()` メソッドを使用してセキュリティ ストアから ACL を取得します。`getAclInternal()` メソッドでは、結果セットを基に AclImpl オブジェクトを作成し、ユーザとグループごとに AclEntryImpl オブジェクトを作成して、パーミッションを AclEntryImpl オブジェクトに追加します。AclEntryImpl オブジェクトが作成されると、AclImpl オブジェクトに追加されます。AclImpl オブジェクトが作成されると、`getAclInternal()` メソッドは完成した ACL を返します。

セキュリティ イベントの監査

`weblogic.security.audit` パッケージを使用すると、WebLogic Server セキュリティ レルムで発生するイベントに対して監査 SPI を使用できるようになります。パッケージには、インタフェース AuditProvider、および静的クラス Audit があります。WebLogic Server では、監査可能なセキュリティ イベントをこの 2 つに送信します。

監査を有効にするには、AuditProvider インタフェースを実装するクラス、および監査するセキュリティ イベントを示すメソッドを作成します。WebLogic Server では、ユーザが認証を試行する場合、パーミッションがテストされる場合、無効なデジタル証明書またはルート デジタル証明書が提示された場合に、クラスでメソッドが呼び出されます。AuditProvider クラスは、イベントのタイプごとに情報を受信し、ユーザが指定した方法でイベントを処理します。たとえば、失敗した認証リクエストだけを WebLogic Server ログ ファイルに記録することや、データベース テーブルに監査可能なイベントをすべて記録することなどが可能です。

AuditProvider クラスを使用する前に、Administration Console でクラスをインストールする必要があります。詳細については、「セキュリティの管理」の「監査プロバイダのインストール」を参照してください。

LoginAuditProvider サンプルは、\samples\examples\security\audit ディレクトリに収められている、WebLogic Server 付属の examples.security.audit パッケージにあります。サンプルでは、WebLogic Server ログ ファイルで受信する全イベントが記述されています。また、イベントのタイプごとにフィルタ メソッドを定義し、そのフィルタを呼び出して特定のイベントを記録するかどうかを決定しています。サンプル コードでは、フィルタ メソッドは全イベントが記録されるように常に true を返します。このサンプルを拡張する場合、記録するイベントを選択するメソッドでフィルタ メソッドをオーバーライドできます。ロギング以外の作業を行う場合は、LogAuditProvider サンプルを基に独自のプロバイダを作成できます。

ネットワーク接続のフィルタ処理

パスワード、ACL、デジタル証明書を使用すると、WebLogic Server リソースに対してユーザの性質に応じたセキュリティを適用できます。ネットワーク接続をフィルタ処理してセキュリティのレイヤを追加できます。たとえば、ユーザの企業のネットワーク外部からの非 SSL 接続を拒否できます。

ネットワーク接続をフィルタ処理するには、weblogic.security.net.ConnectionFilter インタフェースを実装するクラスを作成し、WebLogic Server にそのクラスをインストールします。それによって、受信したリクエストを調べ、それを受け付けるか拒否するかを決定できるようになります。

接続フィルタを使用する前に、Administration Console でクラスをインストールする必要があります。詳細については、「セキュリティの管理」の「接続フィルタのインストール」を参照してください。

Java クライアントまたは Web ブラウザ クライアントが WebLogic Server への接続を試行すると、WebLogic Server は ConnectionEvent オブジェクトを作成して、ConnectionFilter クラスの accept() メソッドに渡します。ConnectionEvent オブジェクトには、リモート IP アドレス（java.net.InetAddress という形式）、リモート ポート番号、ローカル WebLogic Server のポート番号、プロトコルを示す文字列（HTTP、HTTPS、T3、T3S、または IIOP）が格納されています。

ConnectionFilter クラスでは、ConnectionEvent オブジェクトを調べて、接続を受け付ける場合はオブジェクトを返し、接続を拒否する場合は FilterException を送出できます。

samples\examples\security\net ディレクトリにある、WebLogic Server 付属の examples.security.net.SimpleConnectionFilter サンプルでは、ルール ファイルで接続がフィルタ処理されています。SimpleConnectionFilter サンプルでは、ルール ファイルを解析してルールに適合するアルゴリズムを設定することで、WebLogic Server 接続のフィルタ処理のオーバーヘッドが最小化されています。SimpleConnectionFilter サンプルは、効率性を考慮した汎用接続フィルタです。必要に応じて、このコードを変更できます。たとえば、ローカルまたはリモートのポート番号を、フィルタ、またはフィルタ処理のオーバーヘッドを軽減するサイト固有のアルゴリズムに合わせることができます。

コード リスト 4-21 では、WebLogic Server は、ConnectionEvent を指定した SimpleConnectionFilter.accept() メソッドを呼び出しています。SimpleConnectionFilter.accept() メソッドは、リモート アドレスとプロトコルを取得してプロトコルをビットマスクに変換し、ルール適合時に文字列が比較されることを防ぎます。次に、SimpleConnectionFilter.accept() メソッドは、ルールごとにリモート アドレスとプロトコルを比較して、一致するものを見つけます。

コード リスト 4-21 ネットワーク接続のフィルタ処理の例

```
public void accept(ConnectionEvent evt)
  throws FilterException
{
  InetAddress remoteAddress = evt.getRemoteAddress();
  String protocol = evt.getProtocol().toLowerCase();
  int bit = protocolToMaskBit(protocol);

  // この特殊なビットマスクは、
  // 認識されたプロトコルのいずれかが接続で使用されないことを
  // 示す
  if (bit == 0xdeadbeef)
  {
    bit = 0;
  }

  // 記述された順でルールをチェック

  for (int i = 0; i < rules.length; i++)
  {
    switch (rules[i].check(remoteAddress, bit))
    {
```

```
      case FilterEntry.ALLOW:
return;
      case FilterEntry.DENY:
throw new FilterException("rule " + (i + 1));
      case FilterEntry.IGNORE:
break;
      default:
throw new RuntimeException("connection filter internal error!");
      }
    }

    // 一致したルールがない場合でも、接続を成功させることができる

    return;
  }
```

SSL を使用した RMI over IIOP の使い方

SSL プロトコルを使用すると、RMI または EJB リモート オブジェクトへの IIOP 接続を保護できます。SSL プロトコルは、認証を通じて接続を保護し、オブジェクト間のデータ交換を暗号化します。WebLogic Server では、以下の方法で SSL を使用した RMI over IIOP を使用できます。

- CORBA クライアント Object Request Broker（ORB）による方法
- Java クライアントによる方法

どちらの場合でも、SSL プロトコルを使用するよう、WebLogic Server をコンフィグレーションする必要があります。詳細については、「SSL プロトコルのコンフィグレーション」を参照してください。

CORBA クライアント ORB による SSL を使用した RMI over IIOP を使用するには、次の手順を行います。

1. SSL プロトコルを使用するよう、CORBA クライアント ORB をコンフィグレーションします。SSL プロトコルのコンフィグレーションの詳細については、クライアント ORB の製品マニュアルを参照してください。
2. `host2ior` ユーティリティを使用して、WebLogic Server IOR をコンソールに出力します。`host2ior` ユーティリティによって、SSL 接続用と非 SSL 用の 2 種類の IOR が出力されます。

3. SSL IOR は、WebLogic Server JNDI ツリーにアクセスする CosNaming サービスへの初期参照を取得するときに使用します。

RMI over IIOP の使い方の詳細については、『WebLogic RMI Over IIOP プログラマーズ ガイド』を参照してください。

Java クライアントによる SSL を使用した RMI over IIOP を使用するには、次の手順を行います。

1. コールバックを使用する場合は、Java クライアントのプライベート キーとデジタル証明書を取得します。
2. SSL ソケット接続を処理するよう、`java.rmi.server.RMISocketFactory` クラスを拡張します。WebLogic Server が SSL 接続をリスンするポートを指定してください。`java.rmi.server.RMISocketFactory` クラスを拡張したクラスの例については、コード リスト 4-22 を参照してください。
3. `-d` オプションを付けて ejbc コンパイラを実行します。
4. `java.rmi.server.RMISocketFactory` クラスの拡張を、Java クライアントの `CLASSPATH` に追加します。
5. Java クライアントを起動する場合は、次のコマンド オプションを使用します。

   `-xbootclasspath/a:%CLASSPATH%`
   `-Dorg.omg.CORBA.ORBSocketFactoryClass=`*`java.rmi.server.RMISocketFactory`* の実装
   `-Dssl.certs=`*`Java`* クライアントのデジタル証明書のパス
   `-Dssl.key=`*`Java`* クライアントのプライベート キーのパス

Java クライアントでは、`CLASSPATH` に含まれる SSL プロトコル用に WebLogic Server が使用するクラスが必要になります。

着信接続の場合（コールバックのための WebLogic Server から Java クライアントへの接続）、Java クライアントのデジタル証明書とプライベート キーをコマンドラインで指定する必要があります。`ssl.certs` および `ssl.key` というコマンドライン オプションを使用して、情報を入力します。コード リスト 4-22 内の Java クライアントは、WebLogic Server の SSL ライブラリを使用して、SSL ソケットを提供しています。または、Sun Microsystems の JSSE などの SSL プロバイダを SSL ソケットとして使用することもできます。

コード リスト 4-22 java.rmi.server.RMISocketFactory の例

```
package examples.rmi_iiop.ejb.rmi_iiop;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.rmi.server.RMISocketFactory;
import java.util.StringTokenizer;
import java.util.Vector;
import weblogic.security.PEMInputStream;
import weblogic.security.RSAPrivateKey;
import weblogic.security.SSL.SSLCertificate;
import weblogic.security.SSL.SSLParams;
import weblogic.security.SSL.SSLServerSocket;
import weblogic.security.SSL.SSLSocket;
import weblogic.security.X509;

/**
*SSL プロトコルを使用するには、org.omg.CORBA.ORBSocketFactoryClass
* システム プロパティを examples.rmi_iiop.ejb.rmi_iiop.SSLSocketFactory
* に設定する WebLogic Server から Java クライアントへの通信が必要になる場合
*（WebLogic Server で呼び出す必要のあるリモート オブジェクトを Java クラ
* イアントがエクスポートする場合など）もあるので、WebLogic Server で Java
* クライアントとの SSL 接続を確立できるよう、SSL プライベート キーとデジタル
* 証明書の入力が必要になる場合もある **/

public class SSLSocketFactory extends RMISocketFactory

{
    static int sslPort = 7002;

    SSLCertificate cert;
    RSAPrivateKey key;

    private static InputStream getDERStream(String fileName)
    throws IOException
    {
      InputStream is = new FileInputStream(fileName);

      if (fileName.toLowerCase().endsWith(".pem")) {
         is = new PEMInputStream(is);
    }

    return is;
    }

    public SSLSocketFactory()
    {
      String certFiles = System.getProperty("ssl.certs");
      String keyFile = System.getProperty("ssl.key");

      if (certFiles == null) {
      System.err.println("Warning: no server certs (ssl.certs)
```

```
  provided!");
  System.err.println("Warning: incoming server connections
  may fail!");
  return;
}

  if (keyFile == null) {
  System.err.println("Warning: no server private key (ssl.key)
  provided!");
  System.err.println("Warning: incoming server connections
  may fail!");
  }

  StringTokenizer toks = new StringTokenizer(certFiles,
                        System.getProperty("path.separator",
                         ","));
  cert = new SSLCertificate();
  cert.certificateList = new Vector();

  try {
    if (keyFile != null) {
    cert.privateKey = new
                      RSAPrivateKey(getDERStream(keyFile));
  }

     while (toks.hasMoreTokens()) {
     InputStream is = getDERStream(toks.nextToken());
     cert.certificateList.addElement(new X509(is));
     is.close();
     }
  }

  catch (Exception e) {
    e.printStackTrace();
    System.exit(1);
  }
}

public Socket createSocket(String host, int port)
  throws IOException
{
  Socket sock = null;

  System.out.println("*** connecting to " + host + ":" + port);

  if (port == sslPort) {
     try {
     SSLParams p = new SSLParams();

     sock = new SSLSocket(host, port, p);
     }
     catch (Exception e) {
     e.printStackTrace();
     }
  }
  else {
     sock = new Socket(host, port);
  }
```

```
        return sock;
    }

      public ServerSocket createServerSocket(int i)
         throws IOException
      {
        ServerSocket sock = null;

         if (true) {
           try {
           SSLParams p = new SSLParams();

         if (cert != null) {
           p.setServerCert(cert);
          }
          else {
          System.err.println
          ("**** Listening for SSL connections without server
           private key or certs!");

          System.err.println
          ("**** THIS MAY CAUSE FAILURES IF THE SERVER
          CONNECTS TO US!");
          }

         sock = new SSLServerSocket(i, p);
         }
         catch (Exception e) {
         e.printStackTrace();
         }
      }
      else {
        sock = new ServerSocket(i);
      }

      int lp = sock.getLocalPort();
      if (i != lp) {
         System.out.println("*** listening on any port -
         got " + lp);
      }
      else {
        System.out.println("*** listening on port " + lp);
      }

      return sock;
    }
}
```