JBoss 4.0. Podręcznik administratora

Autor: The JBoss Group
Tłumaczenie: Adam Bochenek, Piotr Rajca
ISBN: 83-246-0092-2
Tytuł oryginału: JBoss 4.0 – The Official Guide
Format: B5, stron: 616

Kompendium wiedzy o profesjonalnym serwerze aplikacji

- Proces instalacji i konfiguracji
- Tworzenie i udostępnianie aplikacji
- Administrowanie serwerem i zabezpieczanie go

Technologia J2EE święci triumfy. Programiści na całym świecie stosują ją do tworzenia rozbudowanych aplikacji korporacyjnych i e-commerce. Jednym z integralnych elementów systemu zbudowanego w tej technologii jest odpowiedni serwer aplikacji. Na rynku dostępnych jest kilka platform komercyjnych i zyskujący na popularności produkt open-source – JBoss. JBoss to w pełni profesjonalny serwer aplikacji J2EE, który dzięki bezpłatnemu dostępowi znacznie redukuje koszty wdrożenia systemów informatycznych. Oczywiście to nie jedyna zaleta JBossa – trudno pominąć jego stabilność i bezpieczeństwo, wsparcie ze strony tysięcy użytkowników z całego świata i modułową budowę, która pozwala na szybkie dodawanie kolejnych usług.

„JBoss 4.0. Podręcznik administratora" to wyczerpujące źródło informacji o najnowszej edycji JBossa. Autorami są twórcy JBossa, co gwarantuje wysoki poziom merytoryczny. Znajdziesz tu omówienie wszystkich zastosowań serwera oraz poznasz sposoby tworzenia i wdrażania aplikacji J2EE wykorzystujących komponenty EJB, serwlety, JMS i usługi sieciowe. Przeczytasz również o bezpieczeństwie serwera i aplikacji oraz obsłudze baz danych i transakcji. Książka zawiera szczegółowy opis jądra JBossa, technologii Hibernate oraz programowania aspektowego.

- Instalacja serwera
- Domyślna struktura katalogów
- Pliki konfiguracyjne JBossa
- Zastosowanie mechanizmów JNDI
- Obsługa transakcji
- EJB i serwlety
- Stosowanie usługi JMS
- Zabezpieczanie serwera JBoss
- Korzystanie z usługi Tomcat
- Mapowanie tabel baz danych na obiekty za pomocą Hibernate
- Programowanie aspektowe

Poznaj architekturę serwera JBoss i skonfiguruj go tak, aby pracował z maksymalną wydajnością

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 8.

Bezpieczeństwo

Zapewnienie odpowiedniego poziomu bezpieczeństwa jest jednym z fundamentalnych zadań aplikacji korporacyjnych. Musimy być w stanie precyzyjnie określić, kto może mieć dostęp do aplikacji i w jakim zakresie może z oferowanych przez system operacji korzystać. Specyfikacja J2EE definiuje prosty, korzystający z pojęcia roli model bezpieczeństwa dla komponentów sieciowych i EJB. Bezpieczeństwem serwera JBoss zajmuje się podsystem o nazwie JBossSX. Obsługuje on zarówno bazujący na rolach deklaratywny model bezpieczeństwa, jak i pozwala na integrację za pomocą pośrednika ze specjalizowanym modułem bezpieczeństwa. Domyślna implementacja deklaratywnego modelu bezpieczeństwa korzysta z technologii JAAS (ang. *Java Authentication and Authorization Service*). Warstwa pośrednika bezpieczeństwa pozwala na podłączenie dowolnego, specjalizowanego modułu obsługi kwestii bezpieczeństwa, którego nie jesteśmy w stanie zdefiniować za pomocą modelu deklaratywnego. Odbywa się to w sposób przezroczysty z punktu widzenia komponentów biznesowych. Zanim przejdziemy do szczegółów implementacyjnych serwera JBoss, przypomnimy opisane w specyfikacji modele bezpieczeństwa dotyczące serwletów i komponentów EJB oraz podstawy technologii JAAS.

Deklaratywny model obsługi bezpieczeństwa J2EE

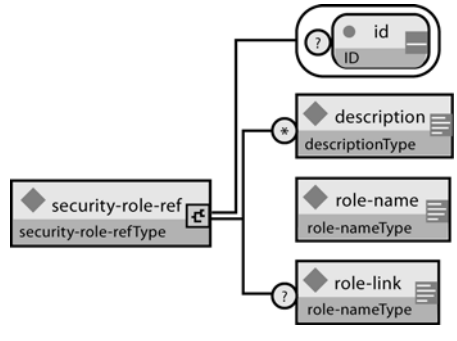
Specyfikacja J2EE zaleca stosowanie deklaratywnego modelu obsługi bezpieczeństwa. Nosi on nazwę deklaratywnego, ponieważ odpowiednie role i uprawnienia definiujemy za pomocą standardowego deskryptora XML, informacji tych nie wstawiamy natomiast do wnętrza komponentów biznesowych. W ten sposób izolujemy kwestie bezpieczeństwa od kodu warstwy logiki biznesowej. Mechanizmy zapewniające bezpieczeństwo powinny być funkcją kontenera, w którym komponent jest wdrożony, nie zaś częścią samego komponentu. Wyobraźmy sobie komponent, który reprezentuje i zapewnia dostęp do konta bankowego. Sprawy związane z zapewnieniem bezpieczeństwa, rolami, prawami dostępu są zupełnie niezależne od logiki, jaką obiekt posiada. Zależą one przede wszystkim od specyfiki miejsca, w którym komponent jest wdrożony.

Zabezpieczenie aplikacji J2EE zależy od konfiguracji wymagań, a ta znajduje się w standardowych deskryptorach wdrożenia. Za dostęp do komponentów EJB odpowiada plik deskryptora *ejb-jar.xml*. W przypadku komponentów warstwy sieciowej jest to deskryptor *web.xml*. W poniższych punktach omówimy znaczenie i sposób użycia podstawowych elementów konfigurujących sposób zabezpieczenia aplikacji.

Referencje — element security-role-ref

Zarówno komponentom EJB, jak też serwletom możemy przypisać jeden lub wiele elementów `security-role-ref` (rysunek 8.1). Stosując ten element, deklarujemy, że komponent korzysta z wartości `role-name`, traktując ją jako argument wywołania metody `isCallerInRole(String)`. Za pomocą metody `isCallerInRole` komponent może sprawdzić, czy obiekt wywołujący występuje w roli, która została zadeklarowana przez element `security-role-ref/role-name`. Wartość elementu `role-name` musi być związana z elementem `security-role` za pośrednictwem elementu `role-link`. Typowe wykorzystanie metody `isCallerInRole` polega na dokonaniu operacji weryfikacji, której nie da się przeprowadzić przy użyciu standardowych, związanych z rolą elementów `method-permissions`.

Rysunek 8.1. *Struktura elementu security-role-ref*



Na listingu 8.1 prezentujemy sposób użycia elementu `security-role-ref` w pliku deskryptora *ejb-jar.xml*.

Listing 8.1. *Fragment deskryptora ejb-jar.xml ilustrujący sposób użycia elementu security-role-ref*

```
<!-- Przykładowy fragment deskryptora ejb-jar.xml -->
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>ASessionBean</ejb-name>
      ...
      <security-role-ref>
          <role-name>TheRoleICheck</role-name>
          <role-link>TheApplicationRole</role-link>
      </security-role-ref>
    </session>
  </enterprise-beans>
  ...
</ejb-jar>
```

Sposób wykorzystania elementu `security-role-ref` w pliku *web.xml* demonstruje listing 8.2

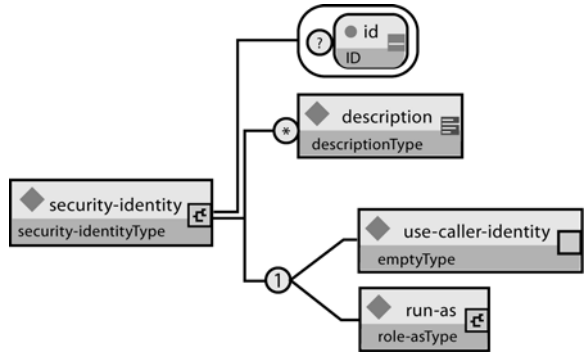
Listing 8.2. *Fragment deskryptora web.xml ilustrujący sposób użycia elementu security-role-ref*

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

Tożsamość — element security-identity

Za pomocą elementu `security-identity` istnieje możliwość określenia tożsamości, którą posługuje się komponent EJB w chwilach, w których wywołuje metody innych komponentów. Strukturę elementu `security-identity` pokazuje rysunek 8.2.

Rysunek 8.2. *Struktura elementu security-identity*



Tożsamość komponentu wywołującego metodę może być taka sama jak aktualna tożsamość klienta, który z tego komponentu korzysta. Możemy jednak posłużyć się inną rolą. Osoba konstruująca aplikację używa elementu `security-identity` wraz z elementem podrzędnym `use-caller-identity`, chcąc wskazać, że aktualna tożsamość komponentu EJB ma być propagowana w ramach wywołań metod innych komponentów. Opcja ta jest domyślna i zostanie zastosowana także w sytuacjach, w których element `security-identity` nie będzie w ogóle zdefiniowany.

Rozwiązanie alternatywne polega na użyciu elementu `run-as/role-name`. Określamy w ten sposób inną rolę, wskazaną właśnie przez wartość elementu `role-name`, która będzie odpowiadała tożsamości wysyłanej podczas wywoływania metod innych komponentów EJB. Zauważ, że nie zmieniamy w ten sposób tożsamości obiektu wywołującego, co pokazuje metoda `EJBContext.getCallerPrincipal`. Oznacza to natomiast, że rolom obiektu wywołującego przypisywana jest pojedyncza rola określona przez

wartość elementu `run-as/role-name`. Jednym z zastosowań elementu `run-as` jest uniemożliwienie zewnętrznym klientom dostępu do komponentów EJB, z których chcemy korzystać wyłącznie wewnątrz aplikacji. Osiągamy to na zasadzie przypisania elementom `method-permission` komponentu EJB takich ról, mających zezwolenie na wywołanie metody, które nigdy nie zostaną przydzielone klientom zewnętrznym. Następnie komponentom EJB, które upoważnimy do wywoływania metod komponentów wewnętrznych, przypisujemy tę rolę za pomocą elementu `run-as/role-name`. Listing 8.3 zawiera fragment deskryptora *ejb-jar.xml*, w którym użyty został element `security-identity`.

Listing 8.3. *Fragment deskryptora ejb-jar.xml, który jest przykładem wykorzystania elementu security-identity*

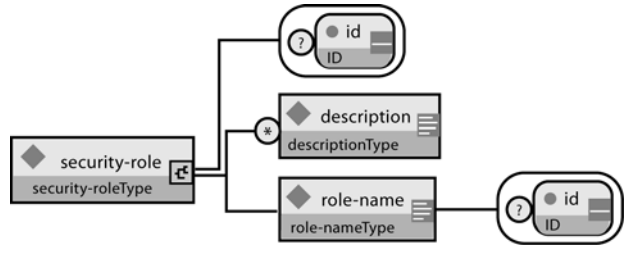
```
<!-- Przykładowy fragment deskryptora ejb-jar.xml -->
<ejb-jar>
    <enterprise-beans>
        <session>
            <ejb-name>ASessionBean</ejb-name>
            <!-- ... -->
            <security-identity>
                <use-caller-identity/>
            </security-identity>
        </session>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

Role — element security-role

Nazwa roli, do której odwołujemy się z poziomu elementów `security-role-ref` lub `security-identity` musi odpowiadać którejś z ról zdefiniowanych na poziomie aplikacji. Osoba wdrażająca aplikację tworzy takie role logiczne za pomocą elementów `security-role` (rysunek 8.3). Wartość elementu `role-name` jest nazwą roli logicznej zdefiniowanej w ramach aplikacji. Przykładem może być rola `Administrator`, `Architekt` czy `Sprzedawca`.

W specyfikacji J2EE wyraźnie podkreśla się, by nie zapominać o tym, że role wymienione w deskryptorze wdrożenia tworzą pewien logiczny widok aplikacji, który odpowiada przyjętej strategii zapewnienia jej bezpieczeństwa. Role zdefiniowane na poziomie deskryptora nie powinny być mylone z grupami, użytkownikami i uprawnieniami istniejącymi na poziomie systemu operacyjnego, w ramach którego pracuje serwer

Rysunek 8.3. *Struktura elementu security-role*



aplikacyjny. Role z deskryptora wdrożenia to składniki aplikacji, taki jest też ich zasięg. W przypadku aplikacji bankowej rolami takimi może być np. `KierownikZmiany`, `Kasjer` czy `Klient`.

W serwerze JBoss elementy `security-role` używane są jedynie w celu przyporządkowania wartości `security-role/role-name` do roli logicznej, do której odwołuje się komponent. Role przypisywane użytkownikowi są dynamiczną funkcją menedżera bezpieczeństwa aplikacji, co pokażemy podczas omawiania szczegółów implementacyjnych podsystemu JBossSX. JBoss nie wymaga definicji elementów `security-role` w celu określenia uprawnień zezwalających na wywoływanie metod. Jednakże tworzenie elementów `security-role` jest ciągle zalecaną praktyką, gwarantującą przenośność aplikacji pomiędzy różnymi serwerami aplikacyjnymi. Na listingu 8.4 widzimy element `security-role` użyty wewnątrz deskryptora *ejb-jar.xml*.

Listing 8.4. *Fragment deskryptora ejb-jar.xml, który jest przykładem wykorzystania elementu security-role*

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

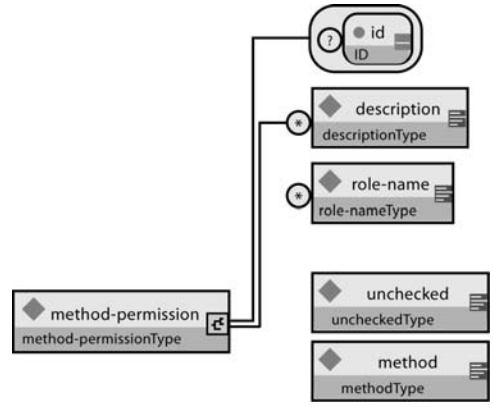
Na listingu 8.5 widzimy element `security-role` użyty w deskryptorze *web.xml*.

Listing 8.5. *Fragment deskryptora web.xml, który jest przykładem wykorzystania elementu security-role*

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

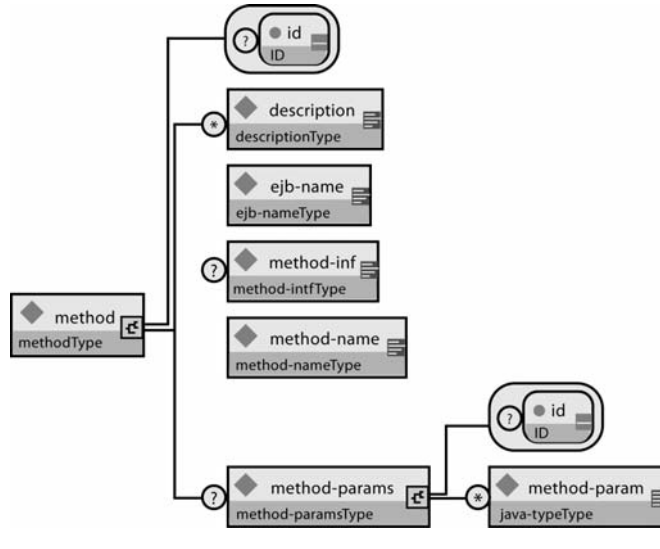
Uprawnienia wywoływania metod

Osoba wdrażająca aplikację może za pomocą elementów `method-permission` wskazać role, które uprawnione są do wywoływania metod zawartych w domowych i zdalnych interfejsach komponentów EJB. Strukturę elementu `method-permission` widzimy na rysunku 8.4.



Rysunek 8.4. *Struktura elementu method-permission*

Każdy element `method-permission` posiada jeden lub wiele elementów zagnieżdżonych `role-name` wskazujących na role posiadające uprawnienia, które pozwalają wywołać zdefiniowane za pomocą elementów zagnieżdżonych `method` metody komponentu EJB (rysunek 8.5). Zamiast elementu `role-name` możemy zdefiniować element `unchecked`, deklarując w ten sposób, że dostęp do metod określonych za pomocą elementów `method` mają wszyscy klienci, którzy zostali przez aplikację uwierzytelnieni. Istnieje także element `exclude-list` definiujący metody, których nie można wywołać. Metody komponentu EJB, które nie zostały wymienione na liście metod dostępnych, traktowane są domyślnie w taki sam sposób, jakby były wykluczone za pomocą elementu `exclude-list`.



Rysunek 8.5. *Struktura elementu method*

Istnieją trzy dopuszczalne sposoby deklaracji elementu `method`:

- Sposób pierwszy stosujemy wówczas, gdy chcemy wskazać wszystkie metody interfejsu domowego i zdalnego danego komponentu EJB:

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

- Sposób drugi stosujemy, gdy chcemy wskazać metodę o podanej nazwie wchodzącą w skład interfejsu domowego lub zdalnego danego komponentu EJB:

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

  Jeśli istnieje kilka przeciążonych metod o tej samej nazwie, to element dotyczy wszystkich.

- Trzeci sposób jest najbardziej dokładny. Pozwala wskazać konkretną metodę nawet wówczas, gdy jej nazwa jest przeciążona.

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

  Wskazana metoda musi być składnikiem interfejsu domowego lub zdalnego danego komponentu. Każdy element `method-param` posiada wartość, którą jest pełna nazwa typu kolejnego parametru metody.

Opcjonalny element `method-intf` pozwala nam na dokonanie rozróżnienia w przypadkach, gdy metody o identycznej sygnaturze wchodzą w skład zarówno interfejsu domowego, jak i zdalnego.

Poniżej, na listingu 8.6, prezentujemy przykład użycia elementu `method-permission`.

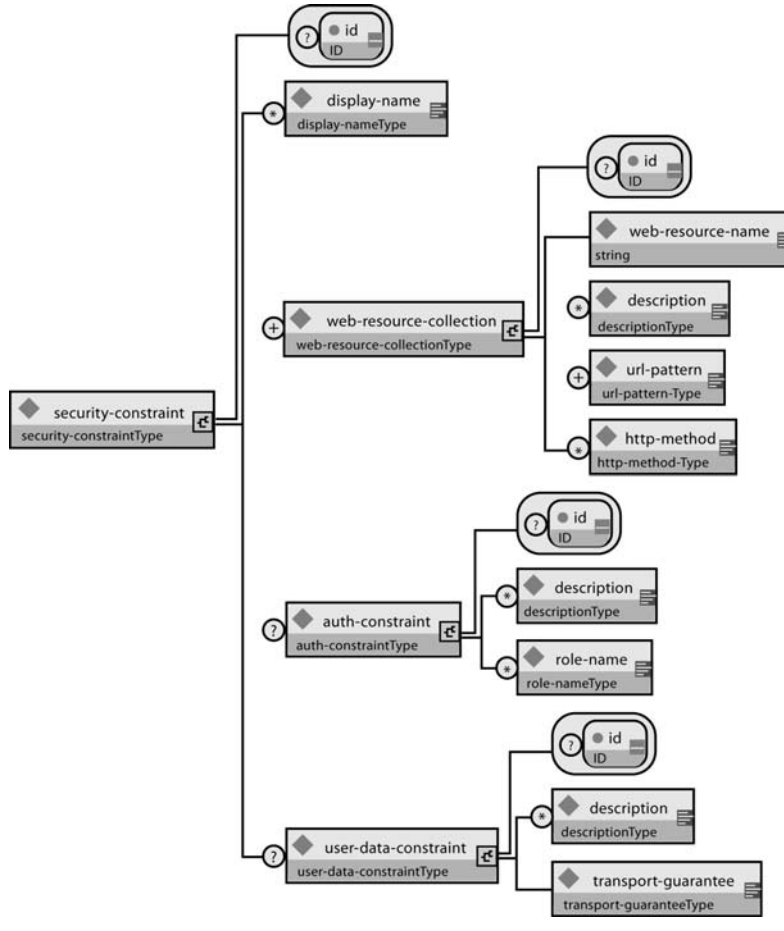
Listing 8.6. *Fragment deskryptora ejb-jar.xml, który jest ilustracją użycia elementu method-permission*

```
<ejb-jar>
    <assembly-descriptor>
        <method-permission>
            <description>Role employee oraz temp-employee pozwalają na dostęp
               do metod komponentu EmployeeService</description>
            <role-name>employee</role-name>
            <role-name>temp-employee</role-name>
            <method>
                <ejb-name>EmployeeService</ejb-name>
                <method-name>*</method-name>
            </method>
```

```
        </method-permission>
        <method-permission>
            <description>Rola employee zezwala na wywołanie metod findByPrimaryKey,
               getEmployeeInfo oraz updateEmployeeInfo(String) komponentu</description>
            <role-name>employee</role-name>
            <method>
                <ejb-name>AardvarkPayroll</ejb-name>
                <method-name>findByPrimaryKey</method-name>
            </method>
            <method>
                <ejb-name>AardvarkPayroll</ejb-name>
                <method-name>getEmployeeInfo</method-name>
            </method>
            <method>
                <ejb-name>AardvarkPayroll</ejb-name>
                <method-name>updateEmployeeInfo</method-name>
                <method-params>
                    <method-param>java.lang.String</method-param>
                </method-params>
            </method>
        </method-permission>
        <method-permission>
            <description>Rola admin pozwala na wywołanie dowolnej metody komponentu
               EmployeeServiceAdmin</description>
            <role-name>admin</role-name>
            <method>
                <ejb-name>EmployeeServiceAdmin</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>
        <method-permission>
            <description>Każdy uwierzytelniony użytkownik posiada prawo wywoływania
               dowolnej metody komponentu EmployeeServiceHelp
            </description>
            <unchecked/>
            <method>
                <ejb-name>EmployeeServiceHelp</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>
        <exclude-list>
            <description>Wdrożenie nie zezwala na wywołanie żadnej z metod fireTheCTO
               komponentu EmployeeFiring
            </description>
            <method>
                <ejb-name>EmployeeFiring</ejb-name>
                <method-name>fireTheCTO</method-name>
            </method>
        </exclude-list>
    </assembly-descriptor>
</ejb-jar>
```

Uprawnienia dotyczące aplikacji sieciowych

W aplikacjach sieciowych zasady bezpieczeństwa i dostępu do zasobów również określane są przy użyciu ról. W tym przypadku jednak prawa dostępu do zasobów przypisujemy rolom, korzystając z wzorców URL-i. Wskazują one te treści, do których nie można się swobodnie odwoływać. Całość definiujemy w pliku deskryptora *web.xml* za pomocą elementu `security-constraint` (rysunek 8.6).



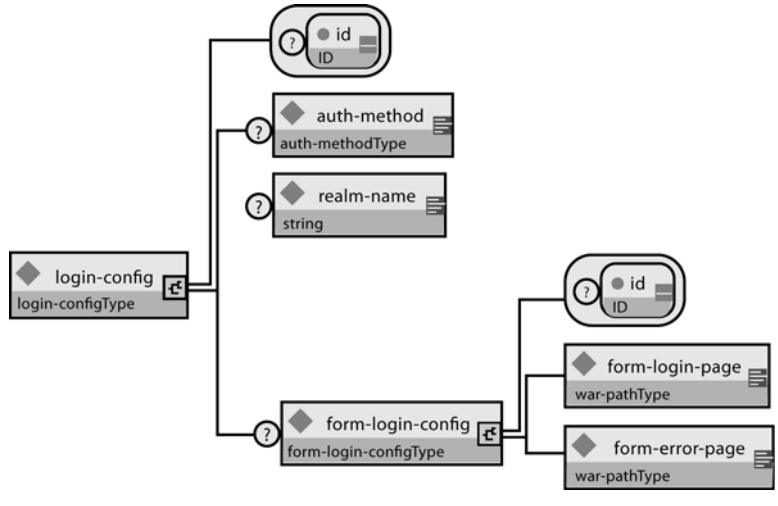
Rysunek 8.6.
Struktura elementu security-constraint

Treść, która ma być chroniona, deklarujemy za pomocą jednego lub wielu elementów `web-resource-collection`. Każdy taki element zawierać może listę elementów podrzędnych `url-pattern` oraz listę elementów `http-method`. Wartość elementu `url-pattern` zawiera wzorzec URL. Dostępne będą tylko te zasoby wskazane w żądaniu, które odpowiadają wzorcowi. Wartość elementu `http-method` wskazuje natomiast, jakie rodzaje żądań HTTP będą obsługiwane.

Opcjonalny element `user-data-constraint` określa wymagania, jakie stawiamy warstwie transportowej nadzorującej połączenie pomiędzy klientem a serwerem. Możemy wybrać wariant gwarantujący integralność przesyłanych danych (chroniący dane przed ich modyfikacją w czasie transmisji) bądź poufność danych (osoby trzecie nie będą w stanie odczytać przesyłanych danych). Elementu `transport-guarantee` określa sposób ochrony danych w czasie komunikacji na linii klient-serwer. Możemy wybrać jedną z trzech wartości tego elementu: `NONE`, `INTEGRAL` lub `CONFIDENTIAL`. Wybór pierwszej opcji, `NONE`, oznacza, że aplikacja nie wymaga żadnej ochrony przesyłanych danych. Opcję `INTEGRAL` stosujemy wówczas, gdy chcemy, by dane pomiędzy serwerem a klientem nie mogły być podczas transportu zmodyfikowane. Wartość `CONFIDENTIAL` zapewni z kolei, że transmitowane dane są chronione przed ewentualną obserwacją prowadzoną przez osoby trzecie. W większości przypadków wybór opcji `INTEGRAL` bądź `CONFIDENTIAL` wiąże się z koniecznością włączenia obsługi bezpiecznego protokołu SSL.

Opcjonalny element `login-config` służy do definicji metody uwierzytelnienia, dziedziny, która ma być związana z aplikacją, oraz dodatkowych atrybutów, które są wymagane w czasie uwierzytelnienia z użyciem formularza. Model zawartości tego elementu został przedstawiony na rysunku 8.7.

Rysunek 8.7.
Element login-config



Element podrzędny `auth-method` określa mechanizm uwierzytelniania klientów aplikacji sieciowej. Uzyskanie dostępu do dowolnego zasobu sieciowego, który jest chroniony, musi być poprzedzone zakończoną powodzeniem operacją uwierzytelnienia. Istnieją cztery dopuszczalne wartości elementu `auth-metod`: `BASIC`, `DIGEST`, `FORM` i `CLIENT--CERT`. Element podrzędny `realm-name` określa nazwę dziedziny używanej podczas stosowania metod `BASIC` (podstawowa metoda uwierzytelniania) oraz `DIGEST` (odmiana metody podstawowej z zastosowaniem funkcji jednokierunkowej). Element podrzędny `form-login-config` wskazuje stronę logowania i stronę błędu, które użyte zostaną w czasie użycia metody uwierzytelniania bazującej na formularzu. Jeśli natomiast wartość elementu `auth-method` jest inna niż `FORM`, to cały węzeł `form-login-config` jest ignorowany. Na listingu 8.7 prezentujemy fragment deskryptora *web.xml*, w którym ustalamy, że wszystkie zasoby znajdujące się w katalogu */restricted* wymagają, by użytkownik,

który chce się do nich odwoływać, miał przypisaną rolę `AuthorizedUser`. Nie ma natomiast żadnych wymagań odnośnie warstwy transportowej, a uwierzytelnienie dokonywane jest przy użyciu metody podstawowej.

Listing 8.7. *Fragment deskryptora web.xml, który jest ilustracją użycia elementu security-constraint*

```
<web-app>
    <!-- ... -->
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Secure Content</web-resource-name>
            <url-pattern>/restricted/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>AuthorizedUser</role-name>
        </auth-constraint>
        <user-data-constraint>
            <transport-guarantee>NONE</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
    <!-- ... -->
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>The Restricted Zone</realm-name>
    </login-config>
    <!-- ... -->
    <security-role>
        <description>Rola wymagana w celu uzyskania dostępu do zabezpieczonej treści
      </description>
        <role-name>AuthorizedUser</role-name>
    </security-role>
</web-app>
```

Obsługa bezpieczeństwa deklaratywnego w serwerze JBoss

Przedstawione dotychczas elementy opisują kwestie bezpieczeństwa jedynie z perspektywy aplikacji. Ich wartości definiują pewne logiczne role, które osoba zajmująca się wdrożeniem musi przenieść do środowiska, w którym aplikacja ma funkcjonować. Specyfikacja J2EE nie omawia tych zagadnień szczegółowo, wychodząc z założenia, że szczegóły zależą już od implementacji samego serwera aplikacyjnego. W JBossie związanie logicznych ról ze środowiskiem serwera wiąże się ze wskazaniem menedżera bezpieczeństwa, który jest zgodny z modelem bezpieczeństwa J2EE. Odbywa się to za pomocą specjalnego deskryptora wdrożenia. Więcej szczegółów na temat konfigurowania warstwy bezpieczeństwa omawiamy w podrozdziale „Model bezpieczeństwa serwera JBoss”.

Wprowadzenie do JAAS

Podsystem JBossSX bazuje na API JAAS (ang. *Java Authentication and Aauthorization Service*). Zrozumienie działania implementacji JBossSX wymaga znajomości podstaw JAAS API. Dlatego zaczniemy od krótkiego wykładu na temat JAAS, który stanowi przygotowanie do dalszej dyskusji na temat JBossSX.

Czym jest JAAS?

JAAS API 1.0 to zestaw pakietów i klas, które zaprojektowano z myślą o operacjach uwierzytelniania i autoryzacji. Stanowią one napisaną w języku Java implementację standardowego, wymiennego modułu uwierzytelniającego PAM (ang. *Pluggable Authentication Module*). Rozwiązanie to jest zgodne i rozszerza standardową architekturę bezpieczeństwa oraz zawarte tam mechanizmy uwierzytelniania użytkowników. Moduł JAAS był początkowo dostarczany w postaci rozszerzenia dla JDK 1.3. Począwszy od wersji 1.4, jest już standardowym składnikiem JDK. A ponieważ JBossSX używa wyłącznie tych funkcji JAAS, które wiążą się z uwierzytelnianiem, to od tej chwili koncentrować się będziemy jedynie na zagadnieniach z tym związanych.

Uwierzytelnianie w JAAS odbywa się na zasadzie dołączenia odpowiedniego modułu. Pozwala to aplikacjom Javy pozostawać niezależnymi od aktualnie stosowanej technologii i mechanizmów uwierzytelniania. Dzięki temu JBossSX może pracować w oparciu o różne infrastruktury bezpieczeństwa. Integracja z określoną infrastrukturą bezpieczeństwa może odbywać się bez konieczności dokonywania zmian w implementacji menedżera bezpieczeństwa JBossSX. Jedynym elementem, który wymaga modyfikacji, jest konfiguracja stosu uwierzytelniania używanego przez JAAS.

Podstawowe klasy JAAS

Podstawowe klasy wchodzące w skład JAAS dzielą się na trzy kategorie: klasy ogólne, klasy odpowiadające za uwierzytelnianie i klasy odpowiadające za autoryzację. Poniżej prezentujemy jedynie klasy kategorii pierwszej i drugiej. To one implementują funkcjonalność modułu JBossSX, którą opisujemy w rozdziale.

Klasy ogólne:

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

Klasy związane z uwierzytelnianiem:

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

Klasy Subject i Principal

Zanim nastąpi autoryzacja, zezwalająca na dostęp do zasobów, aplikacja musi najpierw uwierzytelnić źródło, z którego pochodzi żądanie. W JAAS zdefiniowane zostało pojęcie *podmiotu* (ang. *subject*), który reprezentuje źródło żądania. Odpowiadająca mu klasa `Subject` jest kluczowym elementem JAAS. Obiekt `Subject` zawiera informacje o pojedynczej encji, którą może być osoba lub usługa. Informacje te obejmują tożsamość encji, a także jej publiczne i prywatne dane uwierzytelniające. JAAS API używa istniejącego już w Javie interfejsu `java.security.Principal`, który służy do reprezentowania tożsamości. Obiekt tego typu przechowuje po prostu nazwę.

W trakcie procesu uwierzytelnienia podmiotowi przypisuje się tożsamość (klasa `Principal`) lub tożsamości, ponieważ może on mieć ich kilka. Na przykład jedna osoba może posiadać obiekt `Principal` odpowiadający jej nazwisku (np. Jan Kowalski), numerowi identyfikacyjnemu (71061421421) czy też nazwie użytkownika, jaką osoba się posługuje (jank). Wszystkie te dane pozwalają nam skutecznie odróżniać od siebie poszczególne obiekty typu `Subject`. Istnieją dwie metody, które zwracają obiekty typu `Principal` związane z obiektem `Subject`:

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

Pierwsza z nich zwraca wszystkie tożsamości podmiotu. Metoda druga zwraca natomiast tylko te obiekty `Principal`, które są instancjami typu `c` lub po `c` dziedziczą. Gdy obiektów takich nie ma, zwracany jest zbiór (`Set`) pusty. Zwróć uwagę, że interfejs `java.security.acl.Group` jest rozszerzeniem interfejsu `java.security.Principal`. Dzięki temu, stosując odpowiednie typy, możemy logicznie grupować obiekty reprezentujące tożsamość.

Uwierzytelnienie podmiotu

Uwierzytelnienie podmiotu wiąże się z operacją logowania do JAAS. Procedura logowania składa się z następujących etapów:

1. Aplikacja tworzy instancję typu `LoginContext`, przekazując nazwę konfiguracji logowania oraz obiekt implementujący interfejs `CallbackHandler`, któremu zostanie przekazana tablica obiektów typu `Callback`, ich zestaw odpowiada konfiguracji modułu `LoginModule`.
2. Obiekt `LoginContext` na podstawie obiektu `Configuration` ustala listę wszystkich modułów `LoginModule`, które zgodnie z konfiguracją należy załadować. Przy braku wskazanej konfiguracji domyślnie użyta zostanie konfiguracja o nazwie `other`.
3. Aplikacja wywołuje metodę `LoginContext.login`.
4. Metoda `login` odwołuje się kolejno do wszystkich załadowanych modułów typu `LoginModule`. Każdy z nich dokonuje próby uwierzytelnienia podmiotu, wywołując metodę `handle` odpowiedniego obiektu `CallbackHandler`, która zwraca informacje wymagane w procesie uwierzytelnienia. Parametrem metody `handle` jest tablica obiektów typu `Callback`. Jeśli operacja zakończy się sukcesem, `LoginModule` przypisują podmiotowi odpowiednią tożsamość i dane uwierzytelniające.

5. Obiekt `LoginContext` zwraca aplikacji status operacji uwierzytelnienia. Sukcesowi odpowiada prawidłowe zakończenie metody `login`. Nieudane logowanie sygnalizowane jest przez zgłoszenie wyjątku `LoginException`.
6. Jeśli operacja uwierzytelnienia się powiedzie, dostęp do uwierzytelnionego podmiotu aplikacja uzyskuje za pomocą metody `LoginContext.getSubject`.
7. Po opuszczeniu zakresu, w którym potrzebny jest uwierzytelniony podmiot, możemy usunąć wszystkie związane z nim tożsamości i inne stowarzyszone z podmiotem informacje. Służy do tego metoda `LoginContext.logout`.

Klasa `LoginContext` udostępnia zestaw podstawowych metod służących do uwierzytelniania podmiotów i pozwala na tworzenie aplikacji w sposób niezależny od używanej w danej chwili technologii obsługującej warstwę bezpieczeństwa. `LoginContext` określa aktualnie skonfigurowane usługi odpowiadające za uwierzytelnianie na podstawie danych zawartych w obiekcie `Configuration`. Poszczególnie usługi uwierzytelniające reprezentowane są przez obiekty typu `LoginModule`. Jak widać, istnieje możliwość włączania różnych modułów logowania i nie wiąże się to z koniecznością modyfikowania kodu aplikacji. Poniższy fragment kodu prezentuje kolejne kroki, które pozwalają aplikacji na uwierzytelnienie podmiotu, czyli obiektu `Subject`:

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

try {
    lc.login();
    Subject subject = lc.getSubject();
} catch(LoginException e) {
    System.out.println("operacja uwierzytelnienia nie powiodła się");
    e.printStackTrace();
}

// Obszar zawierający operacje
// wymagające uwierzytelnionego obiektu Subject
// ...

// Koniec obszaru, wywołujemy metodę logout
try {
    lc.logout();
} catch(LoginException e) {
    System.out.println("błąd podczas wylogowywania");
    e.printStackTrace();
}

// Przykładowa klasa MyHandler
class MyHandler
    implements CallbackHandler
{
    public void handle(Callback[] callbacks) throws
        IOException, UnsupportedCallbackException
    {
        for (int i = 0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof NameCallback) {
                NameCallback nc = (NameCallback)callbacks[i];
                nc.setName(username);
```

```
35:             } else if (callbacks[i] instanceof PasswordCallback) {
36:                 PasswordCallback pc = (PasswordCallback)callbacks[i];
37:                 pc.setPassword(password);
38:             } else {
39:                 throw new UnsupportedCallbackException(callbacks[i],
40:                                                 "Nieznany obiekt Callback");
41:             }
42:         }
43:     }
44: }
```

Istnieje możliwość stworzenia własnej implementacji interfejsu `LoginModule`. Dzięki temu da się integrować aplikacje z różnymi zewnętrznymi technologiami uwierzytelniania. Można także stworzyć łańcuch wielu modułów typu `LoginModule`; w ten sposób w procesie uwierzytelnienia brać może udział wiele różnych technologii. Na przykład jeden moduł `LoginModule` może dokonać sprawdzenia nazwy użytkownika i jego hasła, podczas gdy inny dokona uwierzytelnienia na podstawie wartości podanych przez czytnik kart bądź urządzenie biometryczne.

Cykl życia modułu `LoginModule` określony jest przez obiekt `LoginContext`, który jest tworzony przez klienta w celu wywołania jego metody `LoginContext.login`. Ten dwufazowy proces składa się następujących kroków:

1. `LoginContext` tworzy wymienione w konfiguracji obiekty typu `LoginModule`, korzystając z ich bezparametrowego konstruktora.
2. Każdy `LoginModule` jest następnie inicjalizowany za pomocą metody `initialize`. Jej argument typu `Subject` nie może wskazywać wartości `null`. Oto pełna sygnatura tej metody: `public void initialize(Subject subject, CallbackHandler callbackHandler, Map sharedState, Map options)`.
3. Wywoływana jest metoda `login`, która rozpoczyna proces uwierzytelnienia. Może ona na przykład wyświetlić okienko służące do wprowadzenia nazwy użytkownika i hasła, a następnie dokonać weryfikacji tych danych za pomocą odwołania do usługi NIS lub LDAP. Z kolei inna implementacja tej metody może korzystać z czytnika kart lub czytnika danych biometrycznych. Możemy także pobrać informacje o użytkowniku z systemu operacyjnego. Sprawdzenie poprawności tożsamości użytkownika przez każdy obiekt `LoginModule` nazywamy pierwszą fazą uwierzytelnienia JAAS. Sygnatura metody `login` jest następująca: `boolean login() throws LoginException`. Zwrócenie wyjątku `LoginException` sygnalizuje błąd logowania. Zwrócenie przez metodę wartości `true` oznacza natomiast, że operacja zakończyła się sukcesem. Zwrócenie wartości `false` informuje nas, że moduł logowania powinien być zignorowany.
4. Gdy wszystkie operacje uwierzytelnienia w ramach obiektu `LoginContext` zakończą się sukcesem, wywoływana jest metoda `commit` każdego modułu `LoginModule`. Jeśli więc faza pierwsza przebiegnie pomyślnie, przechodzimy do drugiej fazy uwierzytelnienia, w której obiektowi typu `Subject` przypisywane są obiekty typu `Principal`, a także publiczne i prywatne dane uwierzytelniające.

Niepowodzenie w fazie pierwszej także wiązać się będzie z wywołaniem metody `commit`. W takim przypadku jednak jej działanie będzie inne, dokona ona bowiem usunięcia wcześniej podanych danych, takich jak identyfikator i hasło. Sygnatura metody `commit` jest następująca: `boolean commit() throws LoginException`. Zgłoszenie wyjątku `LoginException` oznacza problemy z wykonaniem fazy drugiej. Wartość `true` wskazuje na pomyślne wykonanie metody `commit`, podczas gdy zwrócona wartość `false` mówi o tym, że moduł logowania powinien być zignorowany.

5. Gdy proces uwierzytelnienia za pomocą obiektu `LoginContext` zakończy się niepowodzeniem, w każdym z obiektów `LoginModule` wywoływana jest metoda `abort`. Metoda `abort` usuwa wszystkie związane z uwierzytelnieniem dane, które powstały w wyniku działanie metod `login` i `initialize`. Oto sygnatura metody `abort`: `boolean abort() throws LoginException`. Zgłoszenie wyjątku `LoginException` oznacza problemy związane z wykonaniem czynności zawartych w metodzie `abort`. Wartość `true` wskazuje na pomyślne wykonanie metody `abort`, podczas gdy zwrócona wartość `false` mówi o tym, że moduł logowania powinien być zignorowany.

6. Wywołanie metody `logout` obiektu `LoginContext` również oznacza usunięcie informacji dotyczących procesu uwierzytelnienia, tym razem jednak dzieje się to po poprawnie zakończonej operacji logowania. Wywołanie `LoginContext.logout` powoduje rozpoczęcie procesu, w którym wywoływane są metody `logout` kolejnych obiektów typu `LoginModule`. Usuwane są wówczas dane przyporządkowane podmiotowi przez metodę `commit`. Sygnatura metody `logout` to: `boolean logout() throws LoginException`. Zgłoszenie wyjątku `LoginException` oznacza problemy związane z operacją wylogowywania. Wartość `true` wskazuje na pomyślne wykonanie metody, podczas gdy wartość `false` mówi o tym, że moduł logowania powinien być zignorowany.

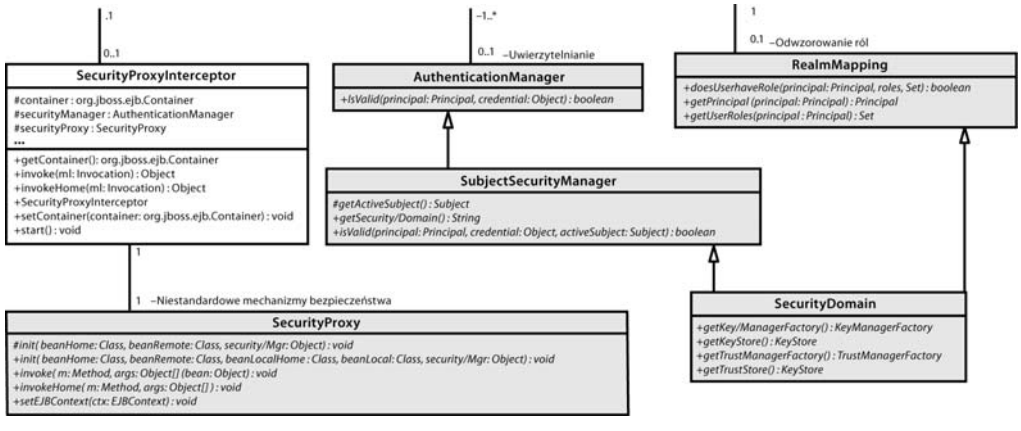
Obiekt `LoginModule` wymaga informacji uwierzytelniających, a w celu ich pobrania musi komunikować się z użytkownikiem. Korzysta wówczas z obiektu typu `CallbackHandler`. Jednak to aplikacje implementują interfejs `CallbackHandler`, instancję tego typu przekazują obiektowi `LoginContext`, który przesyła ją dalej, do modułów `LoginModule`. Moduł `LoginModule` używa obiektu `CallbackHandler` zarówno w celu pobrania danych użytkownika (takich jak hasło czy numer PIN), ale również w celu wysłania użytkownikowi pewnych informacji (np. statusu operacji logowania). Ponieważ to aplikacja wskazuje obiekt `CallbackHandler`, moduły `LoginModule` pozostają niezależne od tego, w jaki sposób przebiegała będzie interakcja z użytkownikiem. Na przykład implementacja obiektu `CallbackHandler` aplikacji posiadającej graficzny interfejs użytkownika wyświetli zapewne okienko dialogowe i poprosi użytkownika o wprowadzenie danych. W przypadku środowiska bez interfejsu graficznego, którym jest na przykład serwer aplikacyjny, dane uwierzytelniające mogą być pobrane za pomocą API serwera. Interfejs `CallbackHandler` definiuje tylko jedną metodę, którą musimy zaimplementować:

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

Ostatnim elementem procesu uwierzytelnienia, który chcemy opisać, jest interfejs `Callback`. Moduły typu `LoginModule` używają obiektów typu `Callback` w celu pobrania informacji wymaganej przez mechanizm uwierzytelnienia. Istnieje kilka predefiniowanych implementacji tego typu, m.in. `NameCallback` i `PasswordCallback`, które widzieliśmy w zaprezentowanym przed chwilą przykładzie. Obiekt `LoginModule` przekazuje w czasie procesu uwierzytelniania tablicę obiektów typu `Callback` bezpośrednio do obiektu `CallbackHandler` jako parametr metody `handle`. Jeśli aktualnie używany obiekt typu `CallbackHandler` nie wie, jak obsłużyć któryś z przekazanych mu obiektów typu `Callback`, zgłaszany jest wyjątek `UnsupportedCallbackException` i proces logowania jest przerywany.

Model bezpieczeństwa serwera JBoss

Podobnie jak pozostałe fragmenty architektury serwera JBoss model bezpieczeństwa również zdefiniowany jest w postaci zestawu interfejsów, co umożliwia alternatywne implementacje tego podsystemu. Istnieją trzy interfejsy, które stanowią szkielet warstwy bezpieczeństwa serwera JBoss: `org.jboss.security.AuthenticationManager`, `org.jboss.security.RealmMapping` oraz `org.jboss.security.SecurityProxy`. Diagram prezentujący interfejsy podsystemu bezpieczeństwa i ich relacje z kontenerem EJB widzimy na rysunku 8.8.

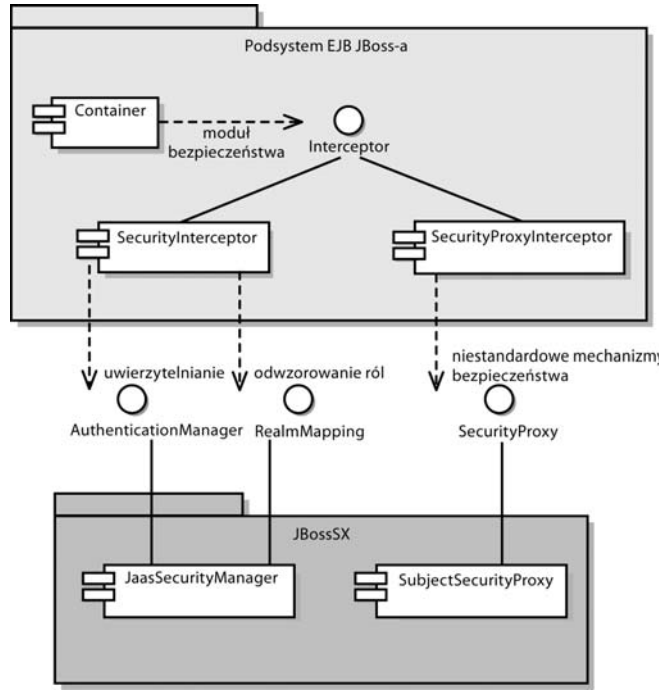


Rysunek 8.8. *Kluczowe interfejsy modelu bezpieczeństwa i ich związek z elementami kontenera EJB serwera JBoss*

Klasy zaznaczone na rysunku 8.8 kolorem szarym to interfejsy warstwy bezpieczeństwa, element niezaznaczony w ten sposób jest składnikiem kontenera EJB. Model bezpieczeństwa J2EE wymaga implementacji co najmniej dwóch interfejsów: `org.jboss.security.AuthenticationManager` i `org.jboss.security.RealMapping`. Charakterystykę wszystkich interfejsów z rysunku 8.8 zamieszczamy poniżej:

- `AuthenticationManager` — interfejs ten odpowiada za sprawdzenie poprawności danych uwierzytelniających przypisanych do obiektów `Principal`. Obiekty `Principal` określają tożsamość, przykładem może być nazwa użytkownika, numer pracownika, numer polisy ubezpieczeniowej. Zadaniem danych uwierzytelniających jest potwierdzenie tej tożsamości, przykładem może być hasło, klucz sesji, podpis cyfrowy. W celu sprawdzenia, czy podana tożsamość odpowiada danym uwierzytelniającym, wywołujemy metodę `isValid`.
- `RealmMapping` — interfejs zajmuje się powiązaniem tożsamości z rolą. Metoda `getPrincipal` na podstawie tożsamości użytkownika, znanej w środowisku operacyjnym, podaje rolę znaną w domenie aplikacji. Metoda `doesUserHaveRole` sprawdza natomiast, czy określonej tożsamości ze środowiska operacyjnego jest na poziomie aplikacji przypisana rola.
- `SecurityProxy` — interfejs opisujący wymagania, jakie stawiane są modułom typu `SecurityProxyInterceptor`. `SecurityProxy` pozwala przenieść na zewnątrz mechanizm kontroli wywołań metod obiektu domowego i zdalnego EJB.
- `SubjectSecurityManager` — ten interfejs wywodzi się bezpośrednio z interfejsu `AuthenticationManager`, a dodatkowo pojawia się w nim metoda zwracająca nazwę domeny menedżera bezpieczeństwa oraz metoda uwierzytelnionego w ramach bieżącego wątku obiektu `Subject`.
- `SecurityDomain` — rozszerzenie interfejsów `AuthenticationManager`, `RealmMapping` i `SubjectSecurityManager`. Jest to próba stworzenia w miarę kompletnego interfejsu związanego z bezpieczeństwem. Zawiera on w sobie również obiekt JAAS typu `Subject`, bazuje także na interfejsach `java.security.KeyStore` oraz zdefiniowanych w ramach JSSE interfejsach `com.sun.net.ssl.KeyManagerFactory` i `com.sun.net.ssl.TrustManagerFactory`. Interfejs ten jest próbą stworzenia bazy dla wielodomenowej architektury bezpieczeństwa, która będzie w stanie lepiej obsłużyć aplikacje i zasoby wdrażane w stylu ASP.

Zauważ, że interfejsy `AuthenticationManager`, `RealmMapping` oraz `SubjectSecurityManager` nie mają swojego odpowiednika po stronie JAAS. Tak więc, choć podsystem JBossSX w dużym stopniu zależy od JAAS, nie dotyczy to podstawowych interfejsów, które podczas tworzenia modelu obsługi bezpieczeństwa są implementowane. A podsystem JBossSX to implementacja interfejsów dołączanego modułu bezpieczeństwa, które bazują na JAAS. Najlepszą ilustracją tego faktu jest diagram zaprezentowany na rysunku 8.9. Skutkiem architektury, która opiera się na dołączanym module, jest możliwość implementacji takiej wersji podsystemu JBossSX, który korzysta z naszego własnego menedżera bezpieczeństwa nie używającego JAAS. Przyglądając się komponentom MBean podsystemu JBoss pokazanym na rysunku 8.9, dochodzimy do wniosku, że jest to możliwe.



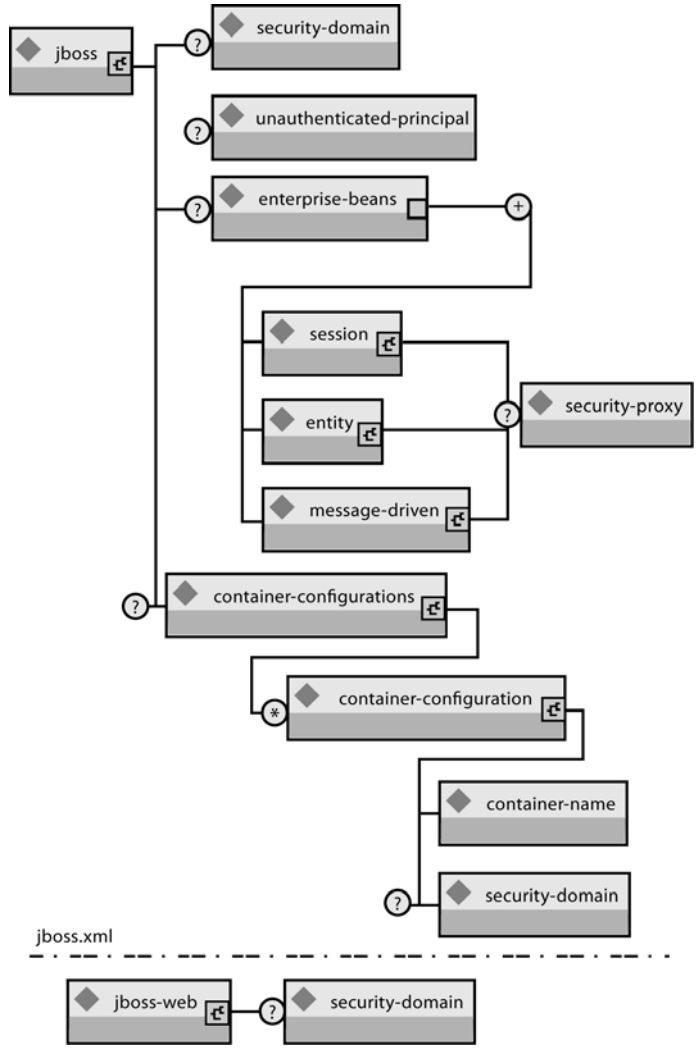
Rysunek 8.9. *Związek pomiędzy implementacją podsystemu JBossSX a warstwą kontenera EJB serwera JBoss*

Obsługa bezpieczeństwa deklaratywnego w serwerze JBoss, odsłona druga

Poprzedni punkt o takim samym tytule, który znajduje się we wcześniejszej części niniejszego rozdziału, zakończyliśmy stwierdzeniem, że za włączenie obsługi bezpieczeństwa odpowiedzialny jest specyficzny dla serwera JBoss deskryptor wdrożenia. Teraz zaprezentujemy szczegóły tej konfiguracji, ponieważ jest ona częścią ogólnego modelu bezpieczeństwa JBoss. Na rysunku 8.10 widzimy strukturę tych elementów specyficznych dla serwera JBoss deskryptorów *jboss.xml* oraz *jboss-web.xml*, które związane są z konfiguracją warstwy bezpieczeństwa.

Wartość elementu `security-domain` określa nazwę JNDI implementacji menedżera bezpieczeństwa, który jest używany przez serwer JBoss do obsługi kontenera EJB i kontenera aplikacji sieciowych. Musi to być obiekt, który implementuje dwa interfejsy: `AuthenticationManager` i `RealmMapping`. Element zdefiniowany na najwyższym poziomie deskryptora staje się domeną bezpieczeństwa dla wszystkich komponentów EJB wdrażanego modułu. I jest to standardowy sposób jego użycia, ponieważ łączenie jednego modułu z kilkoma menedżerami bezpieczeństwa powoduje wiele komplikacji dotyczących współpracy obiektów, utrudnia też administrację.

Wskazanie domeny bezpieczeństwa dla pojedynczego komponentu EJB polega na zdefiniowaniu elementu `security-domain` na poziomie konfiguracji kontenera. Możemy w ten sposób przedefiniować wartość zdefiniowaną za pomocą elementu `security-domain` znajdującego się na najwyższym poziomie deskryptora.



Rysunek 8.10. *Podzbiór elementów deskryptorów jboss.xml oraz jboss-web.xml odpowiadających za konfigurację warstwy bezpieczeństwa*

Element `unauthenticated-principal` określa nazwę obiektu `Principal` zwróconego przez metodę `EJBContext.getUserPrincipal`, gdy metodę obiektu EJB wywołuje użytkownik, który nie został uwierzytelniony. Zauważ, że nie chodzi tutaj o przydzielenie specjalnych praw nieuwierzytelnionym obiektom wywołującym. Celem takiego rozwiązania jest zezwolenie niechronionym serwletom i stronom JSP na dostęp do niezabezpieczonych komponentów EJB. Również dzięki temu docelowy obiekt EJB może uzyskać dostęp do niepustej referencji typu `Principal` obiektu wywołującego, co umożliwia metoda `getUserPrincipal`.

Element `security-proxy` wskazuje na niestandardową implementację obiektu pośrednika, który pozwala dokonywać weryfikacji żądań na zewnątrz deklaratywnego modelu bezpieczeństwa, bez konieczności włączania tej dodatkowej logiki do implementacji warstwy EJB. Może to być implementacja interfejsu `org.jboss.security`.

SecurityProxy lub po prostu obiekt, który posiada implementację metod interfejsu domowego, zdalnego, lokalnego domowego i lokalnego. Nie jest więc wymagana implementacja żadnego dodatkowego specjalnego interfejsu. Jeśli klasa nie implementuje interfejsu SecurityProxy, jej instancja musi zostać „owinięta" przez implementację SecurityProxy, która oddeleguje wywołania do odpowiednich metod obiektu. Klasa org.jboss.security.SubjectSecurityProxy jest przykładem implementacji interfejsu SecurityProxy i jest używana w domyślnej konfiguracji podsystemu JBossSX.

Spójrzmy teraz na przykład własnej implementacji interfejsu SecurityProxy w kontekście bardzo prostego bezstanowego komponentu sesyjnego. Zadaniem naszego obiektu będzie nie dopuścić do wywołania metody echo w przypadkach, gdy jej argumentem jest wyraz składający się z czterech liter. Taki rodzaj weryfikacji nie jest możliwy za pomocą standardowego modelu bezpieczeństwa bazującego na rolach, nie jest bowiem związany z prawami przypisanymi danemu użytkownikowi, dotyczy natomiast argumentów wywoływanej przez niego metody. Kod źródłowy naszego pośrednika o nazwie EchoSecurityProxy znajduje się na listingu 8.8, cały przykład znajduje się wśród przykładów dołączonych do książki, w katalogu *src\main\org\jboss\chap8\ex1*.

Listing 8.8. *Klasa EchoSecurityProxy będąca implementacją interfejsu SecurityProxy. Zadaniem klasy jest weryfikacja wartości argumentów metody echo*

```
 1: package org.jboss.chap8.ex1;
 2:
 3: import java.lang.reflect.Method;
 4: import javax.ejb.EJBContext;
 5: import org.apache.log4j.Category;
 6: import org.jboss.security.SecurityProxy;
 7:
 8: /**
 9:  * Prosty przykład pośrednika typu SecurityProxy; demonstruje on weryfikację
10:  * wartości atrybutów wywoływanych metod
11:  *
12:  * @author Scott.Stark@jboss.org
13:  * @version $Revision: 1.2 $
14:  */
15: public class EchoSecurityProxy implements SecurityProxy {
16:     Category log = Category.getInstance(EchoSecurityProxy.class);
17:
18:     Method echo;
19:
20:     public void init(Class beanHome, Class beanRemote, Object securityMgr)
21:             throws InstantiationException {
22:         init(beanHome, beanRemote, null, null, securityMgr);
23:     }
24:
25:     public void init(Class beanHome, Class beanRemote, Class beanLocalHome,
26:             Class beanLocal, Object securityMgr) throws InstantiationException {
27:         log.debug("init, beanHome=" + beanHome + ", beanRemote=" + beanRemote
28:                 + ", beanLocalhome=" + beanLocalHome + ", beanLocal="
29:                 + beanLocal + ", securityMgr=" + securityMgr);
30:         // pobranie metody echo, której wywołania będziemy testować
31:         try {
32:             Class[] params = { String.class };
33:             echo = beanRemote.getDeclaredMethod("echo", params);
```

```
34:          } catch (Exception e) {
35:              String msg = "Nie znaleziona metoda echo(String)";
36:              log.error(msg, e);
37:              throw new InstantiationException(msg);
38:          }
39:      }
40:
41:      public void setEJBContext(EJBContext ctx) {
42:          log.debug("setEJBContext, ctx=" + ctx);
43:      }
44:
45:      public void invokeHome(Method m, Object[] args) throws SecurityException {
46:          // Nie weryfikujemy dostępu do metod domowych
47:      }
48:
49:      public void invoke(Method m, Object[] args, Object bean)
50:              throws SecurityException {
51:          log.debug("invoke, m=" + m);
52:          // Sprawdzenie wywołania metody echo
53:          if (m.equals(echo)) {
54:              // Kontrola, czy argument składa się z czterech znaków
55:              String arg = (String) args[0];
56:              if (arg == null || arg.length() == 4) {
57:                  throw new SecurityException(
58:                          "Nieakceptowane są 4-literowe słowa");
59:              }
60:              // Nie wywołujemy metody invoke
61:          }
62:      }
63:  }
```

Obiekt typu `EchoSecurityProxy` sprawdza, czy wywoływaną metodą komponentu jest metoda `echo(String)`. Jeśli tak, pobieramy wartość argumentu wywołania i sprawdzamy, czy jest ona wartością `null` lub czy jej długość wynosi 4. W obu przypadkach generujemy wyjątek `SecurityException`. Oczywiście przykład może wydawać się nieco sztuczny, ale chodzi wyłącznie o demonstrację opisywanego mechanizmu. W rzeczywistości aplikacje dosyć często dokonują kontroli wartości przesyłanych argumentów pod kątem bezpieczeństwa. Zaprezentowany przykład stworzyliśmy jednak przede wszystkim po to, by pokazać, jak można stworzyć dodatkową warstwę zabezpieczeń wychodzącą poza standardowy model deklaratywny, która nie wchodzi jednak w skład implementacji komponentów biznesowych. W ten sposób zadania związane z dodatkowymi zabezpieczeniami możemy przekazać ekspertom od tych spraw, a samemu skoncentrować się wyłącznie na logice biznesowej. Obiekty związane z tymi dwoma zagadnieniami mogą być rozwijane niezależnie od siebie.

Na listingu 8.9 prezentujemy deskryptor *jboss.xml*, w którym instalujemy obiekt `EchoSecurityProxy` w roli pośrednika bezpieczeństwa komponentu `EchoBean`.

Listing 8.9. *Deskryptor jboss.xml, który wiąże obiekt pośrednika bezpieczeństwa EchoSecurityProxy z komponentem EchoBean*

```
<?xml version="1.0"?>
<jboss>
    <security-domain>java:/jaas/chap8-ex1</security-domain>

    <enterprise-beans>
        <session>
            <ejb-name>EchoBean</ejb-name>
            <jndi-name>chap8.EchoBean</jndi-name>
            <security-proxy>org.jboss.chap8.ex1.EchoSecurityProxy</security-proxy>
        </session>
    </enterprise-beans>
</jboss>
```

Teraz stworzony przez nas obiekt pośrednika przetestujemy za pomocą klienta, który wywoła metodę `echo` komponentu `EchoBean` z dwoma argumentami: `"Witaj"` i `"1234"`. Program klienta widzimy poniżej:

```
 1:  package org.jboss.chap8.ex1;
 2:
 3:  import javax.naming.InitialContext;
 4:  import org.apache.log4j.Logger;
 5:
 6:  /**
 7:   * @author Scott.Stark@jboss.org
 8:   * @version $Revision: 1.2 $
 9:   */
10:  public class ExClient {
11:      public static void main(String args[]) throws Exception {
12:          Logger log = Logger.getLogger("ExClient");
13:          log.info("Wyszukanie komponentu EchoBean");
14:
15:          InitialContext iniCtx = new InitialContext();
16:          Object ref = iniCtx.lookup("chap8.EchoBean");
17:          EchoHome home = (EchoHome) ref;
18:          Echo echo = home.create();
19:
20:          log.info("Tworzymy obiekt Echo");
21:          log.info("Echo.echo('Witaj') = " + echo.echo("Witaj"));
22:          log.info("Echo.echo('1234') = " + echo.echo("1234"));
23:      }
24:  }
```

Pierwsze wywołanie metody `echo` powinno się udać. Drugie natomiast zawiedzie, ponieważ argument składa się w tym przypadku z czterech znaków. Klienta uruchamiamy za pomocą pokazanego poniżej polecenia, wchodząc wcześniej do katalogu *przyklady*:

```
c:\JBoss\przyklady>ant -Dchap=chap8 -Dex=1 run-example
...
run-example1:
     [copy] Copying 1 file to c:\jboss-4.0.1\server\default\deploy
     [echo] Waiting for 5 seconds for deploy...
     [java] [INFO,ExClient] Wyszukanie komponentu EchoBean
     [java] [INFO,ExClient] Tworzymy obiekt Echo
```

```
     [java] [INFO,ExClient] Echo.echo('Witaj') = Witaj
     [java] java.rmi.ServerException: RemoteException occurred in server thread;
➥nested exception is:
     [java]     java.rmi.AccessException: SecurityException; nested exception is:
     [java]     java.lang.SecurityException: Nieakceptowane są 4-literowe słowa
...
     [java]     at org.jboss.chap8.ex1.ExClient.main(ExClient.java:25)
     [java] Caused by: java.rmi.AccessException: SecurityException; nested
➥exception is:
     [java]     java.lang.SecurityException: Nieakceptowane są 4-literowe słowa
...
```

Wywołanie metody `echo` z argumentem `"Witaj"` zgodnie z przewidywaniami kończy się sukcesem. Wywołanie z argumentem `"1234"`, także zgodnie z oczekiwaniami, powoduje wyświetlenie komunikatów wskazujących na to, że zgłoszony został wyjątek (lista komunikatów celowo została skrócona).

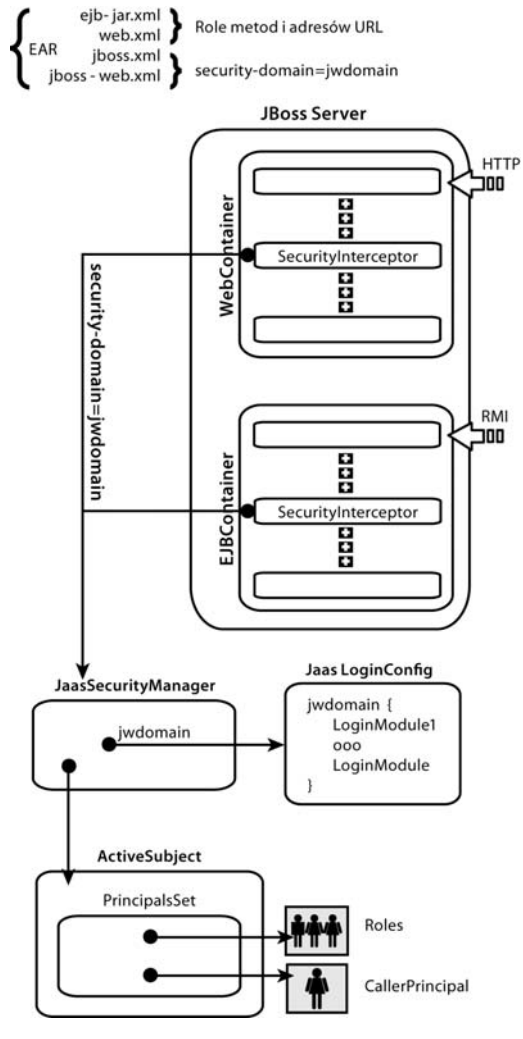
Kluczowym błędem jest oczywiście wyjątek `SecurityException`, mówiący o tym, że „Nieakceptowane są 4-literowe słowa". Jest on generowany z wnętrza klasy `EchoSecurityProxy` w przypadku, gdy pojawia się wywołanie, do realizacji którego nie chcemy dopuścić.

Architektura JBossSX

Z dotychczasowej dyskusji na temat ogólnych zagadnień dotyczących bezpieczeństwa serwera JBoss dowiedziałeś się już, że podsystem JBossSX stanowi implementację interfejsów definiujących warstwę bezpieczeństwa. Ich poprawna implementacja jest podstawowym zadaniem JBossSX. A szczegóły tej implementacji są bardzo interesujące, ponieważ oferują nam wielką swobodę w integrowaniu z istniejącą infrastrukturą, którą może być zarówno baza danych czy serwer LDAP, jak też wyjątkowo zaawansowany zestaw oprogramowania nadzorujący kwestie bezpieczeństwa. Tę elastyczność integracji osiągnięto poprzez zastosowanie modelu modułów uwierzytelniania, który dostępny jest w ramach szkieletu JAAS.

Sercem struktury JBossSX jest klasa `org.jboss.security.plugins.JaasSecurityManager`. To domyślna implementacja interfejsów `AuthenticationManager` i `RealmMapping`. Na rysunku 8.11 widać sposób, w jaki `JaasSecurityManager` współpracuje z warstwą kontenera EJB i kontenera sieciowego, bazując na elemencie `security-domain` deskryptora wdrożenia odpowiedniego komponentu.

Rysunek 8.11 przedstawia aplikację, która składa się z komponentów sieciowych oraz obiektów EJB, a związane są one z domeną bezpieczeństwa o nazwie `jwdomain`. Kontener sieciowy i kontener EJB korzystają z usług odpowiadającego za bezpieczeństwo obiektu przechwytującego, który chroni zawarte w nich komponenty. Instancja używanego menedżera bezpieczeństwa ustalana jest w momencie wdrażania aplikacji, na podstawie wartości elementu `security-domain`, który znajduje się w deskryptorach *jboss.xml* oraz *jboss-web.xml*. Na tej podstawie obiekt przechwytujący odwołuje się do



Rysunek 8.11. *Zależności między wartością security--domain deskryptora, kontenerami oraz menedżerem bezpieczeństwa JaasSecurityManager*

menedżera bezpieczeństwa. Gdy pojawia się żądanie dotyczące chronionego obiektu, obiekt przechwytujący deleguje zadanie weryfikacji żądania do stowarzyszonego z kontenerem menedżera bezpieczeństwa.

Zawarta w JBossSX implementacja `JaasSecurityManager` dokonuje weryfikacji praw dostępu na podstawie informacji związanych z obiektem `Subject`, który zostaje odpowiednio skonfigurowany przez moduły logowania odpowiadające domenie wskazanej za pomocą elementu `security-domain`. Już za chwilę zajmiemy się szczegółami implementacyjnymi obiektu `JaasSecurityManager`.

W jaki sposób JaasSecurityManager korzysta z JAAS

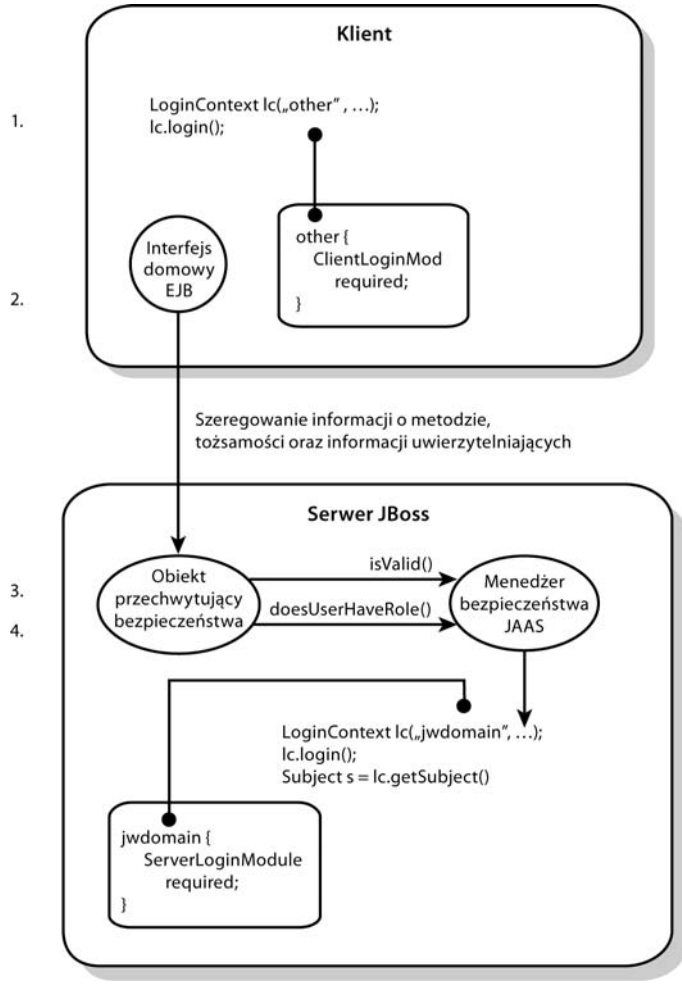
`JaasSecurityManager` korzysta z pakietów JAAS w celu implementacji interfejsów `AuthenticationManager` i `RealmMapping`. Mówiąc dokładnie, zachowanie obiektu ma związek z działaniem modułów logowania, które zostały skonfigurowane w ramach domeny bezpieczeństwa, której przypisana jest instancja typu `JaasSecurityManager`. Moduły logowania biorą na siebie zadanie uwierzytelnienia tożsamości oraz przyporządkowanie klientowi odpowiedniej roli. Tak więc możemy posługiwać się obiektem `JaasSecurityManager` w ramach różnych domen, zmieniając jedynie konfiguracje włączanych modułów logowania.

Dobrą ilustracją szczegółów działania obiektu `JaasSecurityManager` w czasie procesu uwierzytelniania będzie dokładny opis sytuacji, w której klient wywołuje metodę obiektu domowego komponentu EJB. Na początku założymy, że komponent EJB został już wdrożony, a metody jego obiektu domowego zostały zabezpieczone przez odpowiednią definicję elementów `method-permission` znajdujących się w deskryptorze *ejb-jar.xml*. Komponentowi została też przydzielona domena bezpieczeństwa o nazwie `jwdomain`, a odpowiada za to element `security-domain` deskryptora *jboss.xml*.

Schemat omawianej w niniejszym punkcie komunikacji pomiędzy klientem i serwerem przedstawiony został na rysunku 8.12.

Oto opis kolejnych kroków pokazanych na rysunku 8.12:

1. Klient wykonuje operację logowania w celu ustalenia tożsamości i danych uwierzytelniających. Sposób przeprowadzenia czynności związanych z logowaniem zależy od aktualnej konfiguracji warstwy bezpieczeństwa. Logowanie przy użyciu JAAS zakłada stworzenie instancji `LoginContext` i przekazanie jej nazwy konfiguracji, która ma obowiązywać. W naszym przypadku korzystamy z konfiguracji domyślnej o nazwie `other`. Pojedyncze logowanie związuje tożsamość i dane uwierzytelniające z wszystkimi mającymi nastąpić za chwilę wywołaniami metod komponentu EJB. Zwróć uwagę na to, że opisywany proces może nie uwierzytelniać klienta. Natura logowania po stronie klienta zależy od konfiguracji modułu logowania używanego przez klienta. W opisywanym przykładzie konfiguracja o nazwie `other` oznacza, że użyty zostanie moduł `ClientLoginModule` (`org.jboss.security.ClientLoginModule`). Jest to domyślny moduł logowania stosowany po stronie klienta, jego zadaniem jest jedynie przekazanie warstwie wywołującej EJB nazwy użytkownika i hasła. Proces uwierzytelnienia odbędzie się później, po stronie serwera. Tożsamość użytkownika nie jest weryfikowana po stronie klienta.
2. Klient uzyskuje dostęp do obiektu domowego EJB i za jego pomocą dokonuje próby stworzenia komponentu, a to oznacza, że wywołanie metody obiektu domowego jest przesyłane do serwera. Wywołanie to zawiera argumenty podane przez klienta, jak również tożsamość i dane uwierzytelniające ustalone podczas realizacji pierwszego kroku.



Rysunek 8.12. *Kroki składające się na proces uwierzytelnienia i autoryzacji dotyczący wywołania metody obiektu domowego komponentu EJB*

3. Działający po stronie serwera obiekt przechwytujący najpierw musi uwierzytelnić użytkownika, który jest odpowiedzialny za wywołanie. Podobnie jak po stronie klienta, tutaj również musi zostać wywołana metoda logowania JAAS.

4. Domena bezpieczeństwa, która odpowiada za ochronę komponentu EJB, odpowiada za wybór modułów logowania. Nazwa domeny używana jest do wskazania konfiguracji w konstruktorze obiektu `LoginContext`. W naszym przykładzie jest to domena `jwdomain`. Jeśli uwierzytelnienie użytkownika zakończy się sukcesem, stworzona zostanie instancja typu `Subject`, z którą związane zostaną następujące obiekty:

   ♦ Obiekt `java.security.Principal`, który określa tożsamość klienta znaną w środowisku warstwy bezpieczeństwa wdrożenia.

- Obiekt typu `java.security.acl.Group` o nazwie `Roles` reprezentujący zdefiniowane na poziomie domeny role przypisane użytkownikowi. Obiekty `org.jboss.security.SimplePrincipal` odpowiadają nazwom ról. Każdy obiekt tego typu jest bazującą na łańcuchu znaków implementacją typu `Principal`. Role te porównywane są z rolami przypisanymi do metod na poziomie deskryptora *ejb-jar.xml*, korzystamy z nich w implementacji metody `EJBContext.isCallerInRole(String)`.
- Opcjonalny obiekt typu `java.security.acl.Group` o nazwie `CallerPrincipal`, który zawiera pojedynczy obiekt `org.jboss.security.SimplePrincipal` odpowiadający tożsamości obiektu wywołującego domeny aplikacji. Jedynym elementem grupy `CallerPrincipal` będzie wartość zwracana przez metodę `EJBContext.getCallerPrincipal()`. Celem takiego przyporządkowania jest związanie obiektu `Principal` znanego w warstwie bezpieczeństwa wdrożenia z obiektem `Principal` o nazwie znanej aplikacji. W przypadku braku obiektu `CallerPrincipal`, środowisko wdrożenia używa nazwy zwróconej przez metodę `getCallerPrincipal`, czyli wartości odpowiadającej tożsamości znanej na poziomie domeny aplikacji.

Ostatnim krokiem wykonywanym przez obiekt przechwytujący jest sprawdzenie, czy uwierzytelniony użytkownik posiada odpowiednie uprawnienia pozwalające mu wywołać określoną metodę. Proces autoryzacji przebiega następująco:

- Z kontenera EJB pobierane są nazwy ról zezwalających na wywołanie metody komponentu EJB. Role te zdefiniowane są w deskryptorze *ejb-jar.xml* za pomocą elementów `role-name` wchodzących w skład wszystkich elementów `method--permission` związanych z daną metodą.
- Jeśli nie znaleziono żadnych przypisanych metodzie ról, lub gdy nazwa metody zawarta jest w ramach elementu `exclude-list`, to dostęp do metody jest blokowany. W przeciwnym wypadku obiekt przechwytujący wywołuje metodę `doesUserHaveRole` menedżera bezpieczeństwa, sprawdzając, czy wśród upoważnionych ról znajduje się rola przypisana użytkownikowi. Proces weryfikacji polega na tym, że przeglądane są kolejne role i sprawdza się, czy grupa `Roles` podmiotu `Subject` zawiera obiekt `SimplePincipal` wskazujący na rolę o odpowiedniej nazwie. Zezwolenie na dostęp do metody jest udzielane wówczas, gdy dowolna rola jest elementem grupy `Roles`. W przeciwnym razie następuje odmowa dostępu.
- Jeśli komponentowi EJB przypisano dodatkowy obiekt pośrednika bezpieczeństwa, wywołanie metody jest do niego delegowane. Jeśli pośrednik stwierdzi, że dostęp do metody powinien być zabroniony, zgłosi wyjątek `java.lang.SecurityException`. Nie generując wyjątku tego typu, wyrażamy zgodę, by wywołanie doszło do skutku. Sam obiekt wywołania trafia do następnego obiektu przechwytującego. Zwróć uwagę na to, że obiekt `SecurityProxyInterceptor` obsługuje ten etap, a dodatkowy obiekt przechwytujący nie jest na rysunku 8.12 pokazany.

Każde wywołanie metody chronionego komponentu EJB lub też żądanie dostępu do zabezpieczonego składnika aplikacji sieciowej wymaga uwierzytelnienia i autoryzacji klienta wysyłającego to żądanie. Informacje związane z bezpieczeństwem są traktowane

jako bezstanowy atrybut żądania, który musi być przekazany i sprawdzony przy każdej operacji. A w czasie intensywnej komunikacji pomiędzy klientem i serwerem czynności te mogą okazać się kosztowne. Dlatego też obiekt `JaasSecurityManager` wprowadza pojęcie bufora uwierzytelniania, w którym przechowywane są informacje dotyczące poprzednich, zakończonych powodzeniem operacji logowania. Możemy nawet w ramach konfiguracji `JaasSecurityManager` wskazać obiekt, który będzie pełnił funkcję takiego bufora, piszemy o tym w kolejnym punkcie. Jeśli elementu tego nie zdefiniujemy, użyty będzie bufor domyślny przechowujący wspomniane dane przez określony czas.

Komponent JaasSecurityManagerService

Komponent MBean `JaasSecurityManagerService` to usługa zarządzająca menedżerami bezpieczeństwa. I choć nazwa komponentu zaczyna się od *„Jaas”*, to zarządzane przez nią menedżery bezpieczeństwa nie muszą w swej implementacji odwoływać się do usług JAAS. Nazwa wzięła się stąd, że domyślnym menedżerem bezpieczeństwa jest obiekt `JaasSecurityManager`. Podstawowym zadaniem usługi `JaasSecurityManagerService` jest możliwość użycia zewnętrznej implementacji menedżera bezpieczeństwa. Wystarczy w tym celu zaproponować nową, alternatywną implementację interfejsów `AuthenticationManager` i `RealmMapping`.

Drugim, fundamentalnym zadaniem usługi `JaasSecurityManagerService` jest zaoferowanie implementacji typu `javax.naming.spi.ObjectFactory`, która pozwala na proste i niewymagające pisania kodu tworzenie na poziomie JNDI przyporządkowań miedzy nazwą a implementacją menedżera bezpieczeństwa. Dzięki temu, jak już wspominaliśmy wcześniej, istnieje możliwość włączenia mechanizmu bezpieczeństwa poprzez wskazanie nazwy JNDI konkretnej implementacji menedżera bezpieczeństwa, a wszystko to na poziomie elementu `security-domain` deskryptora wdrożenia. Jednak użycie JNDI wymaga poprawnego przyporządkowania nazwy do obiektu. By uprościć operację nadawania nazwy menedżerowi bezpieczeństwa, usługa `JaasSecurityManagerService` obsługuje proces wiązania tego obiektu z instancją `ObjectFactory` w ramach kontekstu `java:/jaas`. Pozwala to stosować konwencję, w której przypisanie elementowi `security-element` nazwy `java:/jaas/XYZ` wiązać się będzie ze stworzeniem menedżera bezpieczeństwa w domenie `XYZ`. Menedżer bezpieczeństwa dla domeny `XYZ` powstanie w momencie wystąpienia pierwszego odwołania do nazwy `java:/jaas/XYZ`, a stworzona zostanie instancja klasy określonej atrybutem `SecurityManagerClassName`. Argumentem jej konstruktora będzie nazwa domeny bezpieczeństwa. Spójrz na fragment konfiguracji kontenera:

```
<jboss>
    <!-- Wszystkie kontenery należą do domeny "hades" -->
    <security-domain>java:/jaas/hades</security-domain>
    <!-- ... -->
</jboss>
```

Dowolna operacja wyszukania obiektu przyporządkowanego nazwie `java:/jaas/hades` spowoduje zwrócenie instancji menedżera bezpieczeństwa, który będzie związany z domeną bezpieczeństwa o nazwie `hades`. Menedżer bezpieczeństwa musi implementować interfejsy `AuthenticationManager` oraz `RealmMapping` i jest instancją klasy wskazanej przez atrybut `SecurityManagerClassName` usługi `JaasSecurityManagerService`.

Komponent JaasSecurityManagerService jest domyślnie skonfigurowany w ramach standardowej dystrybucji serwera JBoss. I często korzystamy z tej właśnie konfiguracji. Cały czas jednak istnieje możliwość modyfikacji tych ustawień, dlatego też poniżej prezentujemy listę atrybutów komponentu JaasSecurityManagerService:

- SecurityManagerClassName — nazwa klasy będącej implementacją menedżera bezpieczeństwa. Klasa ta musi obsługiwać interfejsy org.jboss.security.AuthenticationManager oraz org.jboss.security.RealmMapping. Gdy nie zdefiniujemy wartości tego atrybutu, domyślnie użyta zostanie bazująca na JAAS klasa org.jboss.security.plugins.JaasSecurityManager.
- CallbackHandlerClassName — nazwa klasy implementującej interfejs javax.security.auth.callback.CallbackHandler, która używana będzie przez obiekt JaasSecurityManager. Domyślnie używaną klasę (czyli org.jboss.security.auth.callback.SecurityAssociationHandler) możemy zamienić, jeśli nie w pełni odpowiada naszym potrzebom. Przypadki takie zdarzają się jednak stosunkowo rzadko i wymagają sporej wiedzy oraz doświadczenia.
- SecurityProxyFactoryClassName — nazwa klasy implementującej interfejs org.jboss.security.SecurityProxyFactory. Brak definicji atrybutu oznacza, że domyślnie użyta będzie klasa org.jboss.security.SubjectSecurityProxyFactory.
- AuthenticationCacheJndiName — lokalizacja reguł bufora przechowującego dane uwierzytelniające. Wartość tę traktujemy jako nazwę obiektu typu ObjectFactory, który zwraca instancje typu CachePolicy, z których każda przypada jednej domenie bezpieczeństwa. W czasie wyszukiwania obiektu CechePolicy do wartości atrybutu dodawana jest nazwa domeny bezpieczeństwa. Jeśli jednak operacja wykonana w taki sposób zawiedzie, to wartość atrybutu traktowana jest jako lokalizacja pojedynczego obiektu CachePolicy używanego przez wszystkie domeny. Domyślnie przyjmuje się, że dane znajdujące się w buforze ważne są tylko przez pewien czas.
- DefaultCacheTimeout — czas przechowywania danych w buforze, wyrażony w sekundach. Domyślną wartością jest 1800 (30 minut). Wartość podana w tym miejscu zależy od dwóch czynników: częstotliwości operacji uwierzytelniania oraz tego, jak długo trwać może okres braku synchronizacji danych uwierzytelniających. Jeśli chcemy zrezygnować z mechanizmu buforowania tych danych, należy wstawić wartość 0. Wówczas kompletna operacja weryfikacji będzie miała miejsce przy każdym żądaniu. Wartość atrybutu nie będzie brana pod uwagę, jeśli zmienimy ustawienia domyślne AuthenticationCacheJndiName.
- DefaultCacheResolution — wartość atrybutu określa odstęp czasu mówiący o tym, jak często bufor danych uwierzytelniających ma być odświeżany. Wartość ta powinna być mniejsza od wartości atrybutu DefaultCacheTimeout, inaczej pojęcie okresu ważności nie będzie precyzyjne. Domyślną wartością jest 60 (1 minuta). Atrybut nie będzie brany pod uwagę, jeśli zmienimy ustawienia domyślne AuthenticationCacheJndiName.
- DefaultUnauthenticatedPrincipal — tożsamość reprezentująca użytkowników nieuwierzytelnionych. Atrybut ten pozwala przydzielić im domyślne prawa.

`JaasSecurityManagerService` oferuje także zbiór przydatnych operacji. Wśród nich znajdziemy funkcję pozwalającą wyzerować bufor danych uwierzytelniających w czasie działania aplikacji oraz funkcję zwracającą listę użytkowników, których dane znajdują się w buforze. Istnieje też możliwość wywołania dowolnej metody zdefiniowanej w ramach interfejsu menedżera bezpieczeństwa.

Opróżnienie bufora związanego z określoną domeną bezpieczeństwa wiąże się ze skasowaniem wszystkich danych uwierzytelniających przechowywanych w pamięci. Operacja taka może być potrzebna na przykład w chwili, gdy zmieniliśmy pewne ustawienia związane z bezpieczeństwem i chcemy, by zaczęły one obowiązywać natychmiast. W takich właśnie przypadkach możemy wywołać metodę o następującej sygnaturze: `public void flushAuthenticationCache(String securityDomain)`. Poniżej prezentujemy fragment kodu, w którym korzystamy z tej operacji:

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

Pobranie listy aktywnych użytkowników pozwala stworzyć migawkę, która zawierała będzie listę obiektów `Principal` znajdujących się w danej chwili w buforze danych uwierzytelniających. Sygnatura tej metody jest następująca: `public List getAuthenticationCachePrincipals(String securityDomain)`. A oto przykład jej wywołania:

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

A oto lista kilku kolejnych metod, które oferuje nam menedżer bezpieczeństwa:

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

Są to metody odpowiadające metodom zdefiniowanym w ramach interfejsów `AuthenticationManager` i `RealmMapping` i dotyczą domeny bezpieczeństwa określonej wartością parametru `securityDomain`.

Komponent JaasSecurityDomain

Komponent `org.jboss.security.plugins.JaasSecurityDomain` stanowi rozszerzoną wersję `JaasSecurityManager`, którą uzupełniono o obsługę magazynu kluczy, zgodną z JSSE fabrykę `KeyManagerFactory`, fabrykę `TrustManagerFactory` obsługującą protokół SSL oraz inne funkcje związane z szyfrowaniem danych. Pojawiły się też nowe atrybuty pozwalające konfigurować usługę `JaasSecurityDomain`:

- `KeyStoreType` — typ implementacji obiektu `KeyStore`. Argument odpowiada wartości przekazywanej w wywołaniu metody fabrykującej `java.security.KeyStore.getInstance(String type)`. Wartością domyślną jest `JKS`.
- `KeyStoreURL` — URL określający lokalizację bazy danych obiektu `KeyStore`. Służy do tworzenia strumienia `InputStream` inicjalizującego magazyn kluczy `KeyStore`. Jeśli łańcuch będący wartością atrybutu nie jest poprawnym URL-em, wówczas traktujemy go jak nazwę pliku.
- `KeyStorePass` — hasło bazy danych obiektu `KeyStore`. Atrybut `KeyStorePass` jest używany również w kombinacji z atrybutami `Salt` i `IterationCount` do utworzenia klucza PBE (ang. *Password Based Encryption*) używanego podczas operacji kodowania i dekodowania. Atrybut `KeyStorePass` występuje w jednym z następujących formatów:
  - **Hasło magazynu danych wyrażone w postaci zwykłego tekstu** — wartość `toCharArray()` łańcucha znaków używana bez żadnych dodatkowych manipulacji.
  - **Polecenie, po wywołaniu którego otrzymamy hasło w postaci tekstu** — w tym przypadku mamy do czynienia z formatem `{EXT}...`, gdzie `...` jest poleceniem, które zostanie przekazane do metody `Runtime.exec(String)` i wywoła specyficzną dla danej platformy komendę systemu operacyjnego. Pierwszy wiersz zwrócony przez wywołane polecenie traktowany jest jako hasło.
  - **Klasa zwracająca hasło w postaci tekstu** — wartość argumentu ma format `{CLASS}classname[:ctorarg]`, gdzie `[:ctorarg]` to opcjonalny łańcuch, który będzie argumentem konstruktora tworzącego instancję klasy `classname`. Hasło zostanie przekazane za pomocą wywołania metody `toCharArray()`, jeśli taka metoda zostanie znaleziona. W przeciwnym przypadku korzystamy z metody `toString()`.
- `Salt` — wartość domieszki `PBEParameterSpec`.
- `IterationCount` — ilość iteracji w czasie generacji klucza `PBEParameterSpec`.
- `TrustStoreType` — typ implementacji obiektu `TrustStore`. Argument odpowiada wartości przekazywanej w wywołaniu metody fabrykującej `java.security.KeyStore.getInstance(String type)`. Wartością domyślną jest `JKS`.
- `TrustStoreURL` — URL określający lokalizację bazy danych obiektu `TrustStore`. Służy do tworzenia strumienia `InputStream` inicjalizującego magazyn kluczy `KeyStore`. Jeśli łańcuch będący wartością atrybutu nie jest poprawnym URL-em, wówczas traktujemy go jak nazwę pliku.
- `TrustStorePass` — hasło bazy danych obiektu `TrustStore`. Wartość atrybutu jest po prostu hasłem, nie występują tutaj takie opcje jak w przypadku atrybutu `KeyStorePass`.

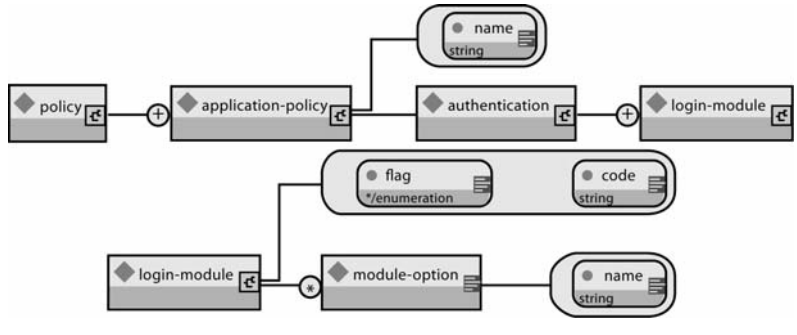
- `ManagerServiceName` — nazwa obiektu JMX (typu `ObjectName`) usługi MBean menedżera bezpieczeństwa. W ten sposób komponent `JaasSecurityDomain` rejestrowany jest jako menedżer bezpieczeństwa, a odpowiadać mu będzie nazwa `java:/jaas/<domain>`, gdzie `<domain>` to nazwa, która przekazywana jest konstruktorowi obiektu MBean. Wartość domyślna to `jboss.security:service=JaasSecurityManager`.

Komponent ładujący plik XML z konfiguracją logowania

`XMLLoginConfig` to usługa, która ładuje standardową konfigurację aplikacji JAAS z lokalnego pliku konfiguracyjnego. Komponent konfigurujemy za pomocą następujących atrybutów:

- `ConfigURL` — atrybut określa URL wskazujący na plik XML z konfiguracją logowania, który ładowany jest podczas uruchamiania. Wartością atrybutu powinien być łańcuch będący poprawną reprezentacją URL-a.
- `ConfigResource` — atrybut wskazuje nazwę zasobu zawierającego plik XML z konfiguracją logowania. Nazwa ta traktowana jest jako ścieżka dostępu w ramach lokalizowanego URL-a, używamy przy tym związanego kontekstowego classloadera wątku
- `ValidateDTD` — wartość logiczna mówiąca o tym, czy zawartość pliku konfiguracyjnego XML będzie weryfikowana przy użyciu DTD. Wartością domyślną jest `true`.

Struktura pliku konfiguracyjnego XML powinna odpowiadać schematowi DTD pokazanemu na rysunku 8.13. Definicję DTD znajdziemy w pliku *docs\dtd\security_config.dtd*.



Rysunek 8.13. *Struktura DTD XMLLoginConfig*

Atrybut `name` elementu `application-policy` to nazwa konfiguracji logowania. Odpowiada ona wartości elementu `security-domain` deskryptorów *jboss.xml* i *jboss-web.xml* — chodzi o wartość, która poprzedzona jest prefiksem `java:/jaas/`. Atrybut `code` elementu `login-module` wskazuje nazwę klasy implementującej moduł logowania. Atrybut `flag` steruje ogólnym zachowaniem stosu uwierzytelniania. Oto możliwe jego wartości wraz z krótką charakterystyką:

- `required` — wymagany, by operacja realizowana za pomocą modułu `LoginModule` się powiodła. Niezależnie jednak od tego, czy zakończy się ona sukcesem czy też nie, w dalszym ciągu będą wywoływane dalsze, znajdujące się na liście, moduły logowania.
- `requisite` — wymagany, by operacja realizowana za pomocą modułu `LoginModule` się powiodła. Jeśli zakończy się sukcesem, proces uwierzytelnienia będzie kontynuowany zgodnie z listą zawierającą obiekty `LoginModule`. W przeciwnym natomiast przypadku sterowanie wraca natychmiast do aplikacji (pozostałe obiekty `LoginModule` znajdujące się na liście są pomijane).
- `sufficient` — niewymagamy, by operacja realizowana za pomocą modułu `LoginModule` zakończyła się sukcesem. Jeśli tak się jednak stanie, sterowanie wraca natychmiast do aplikacji (pozostałe obiekty `LoginModule` znajdujące się na liście są pomijane). W przeciwnym razie proces uwierzytelniania za pomocą modułów będących na liście jest kontynuowany.
- `optional` — niewymagamy, by operacja realizowana za pomocą modułu `LoginModule` zakończyła się sukcesem. Niezależnie od wyniku tej operacji proces uwierzytelnienia będzie kontynuowany zgodnie z listą modułów `LoginModule`.

Element `login-module` może posiadać zero lub więcej elementów podrzędnych `module--option`. Każdy z nich jest parą nazwa/wartość, korzystamy z nich podczas inicjalizowania modułu logowania. Nazwą opcji jest wartość atrybutu `name`, natomiast wartością opcji jest wartość samego elementu. Przykład konfiguracji logowania pokazany jest na listingu 8.10.

Listing 8.10. *Przykładowa konfiguracja modułu logowania używana przez komponent XMLLoginConfig*

```
<policy>
    <application-policy name="srp-test">
        <authentication>
            <login-module code="org.jboss.security.srp.jaas.SRPCacheLoginModule"
                          flag="required">
                <module-option name="cacheJndiName">
                    srp-test/AuthenticationCache
                </module-option>
            </login-module>

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

Opisywany komponent MBean oferuje również dostęp do operacji, które pozwalają modyfikować konfigurację logowania w czasie działania aplikacji. Zauważ jednak, że dowolna operacja próbująca dokonać zmiany konfiguracji wymaga prawa `javax.security.auth.AuthPermission("refreshLoginConfiguration")`. Przykład `org.jboss.chap8.service.SecurityConfig` demonstruje, w jaki sposób możemy dynamicznie wpływać na konfigurację obsługi bezpieczeństwa. Oto lista metod:

- `void addAppConfig(String appName, AppConfigurationEntry[] entries)` — metoda dodaje do bieżącej konfiguracji stos modułów logowania, któremu nadana zostanie nazwa `appName`. Jeśli z nazwą tą wiąże się już jakaś konfiguracja, to zostanie ona usunięta.
- `void removeAppConfig(String appName)` — w ten sposób usuwamy konfigurację modułów logowania zarejestrowaną pod podaną nazwą.
- `String[] loadConfig(URL configURL) throws Exception` — metoda pozwala załadować jedną lub więcej konfiguracji logowania z lokalizacji określonej przez URL, który wskazuje na plik XML bądź wychodzący już z użycia plik konfiguracyjny w formacie zaproponowanym przez firmę Sun. Warto wiedzieć, że dodane będą albo wszystkie konfiguracje logowania, albo żadna z nich. Zwracaną wartością jest tablica zawierająca nazwy dodanych konfiguracji.
- `void removeConfigs(String[] appNames)` — metoda usuwa konfiguracje logowania o nazwach wymienionych w tabeli `appNames`.
- `String displayAppConfig(String appName)` — operacja wyświetla informację na temat konfiguracji o podanej nazwie, jeśli oczywiście taka istnieje.

Komponent zarządzający konfiguracją logowania

Za instalację specjalizowanej klasy `javax.security.auth.login.Configuration` odpowiada usługa MBean o nazwie `org.jboss.security.plugins.SecurityConfig`. Posiada ona tylko jeden atrybut, który wolno nam konfigurować:

- `LoginConfig` — wartością atrybutu jest nazwa JMX (typu `ObjectName`) komponentu MBean, który obsługuje domyślną konfigurację logowania JAAS. W chwili, gdy działanie rozpoczyna obiekt `SecurityConfig`, wskazany za pomocą opisywanego atrybutu komponent MBean proszony jest o zwrócenie instancji typu `javax.security.auth.login.Configuration`, a służy do tego metoda `getConfiguration(Configuration currentConfig)`. Gdy nie podamy wartości atrybutu `LoginConfig`, użyta zostanie domyślna implementacja interfejsu `Configuration` firmy Sun, jej opis znajduje się w dokumentacji JavaDoc.

Opisywana usługa, poza opcją pozwalającą na dołączanie specjalizowanej implementacji konfiguracji logowania, oferuje nam również możliwość łączenia konfiguracji w stos. Dzięki temu wolno nam dany obiekt konfiguracji odłożyć na stos, a następnie go stamtąd pobrać. W ten sposób do domyślnej instalacji serwera JBoss możemy dołączyć własne moduły konfiguracji logowania. Wstawienie nowej konfiguracji na stos odbywa się za pomocą wywołania następującej metody:

```
public void pushLoginConfig(String objectName)
    throws JMException,
           MalformedObjectNameException;
```

Parametr `objectName` wskazuje odpowiednią usługę MBean na tej samej zasadzie jak atrybut `LoginConfig`. Do usunięcia bieżącej konfiguracji służy metoda:

```
public void popLoginConfig() throws JMException;
```

Używanie i tworzenie modułów logowania JBossSX

Implementacja `JaasSecurityManager` pozwala na dowolną modyfikację mechanizmu uwierzytelniania, a wszystko to za sprawą modułów logowania JAAS. Nową, własną implementację uwierzytelniania definiujemy za pomocą odpowiedniej konfiguracji, która odpowiada nazwie domeny bezpieczeństwa chroniącej nasze komponenty aplikacji J2EE.

Podsystem JBossSX zawiera kilka predefiniowanych modułów logowania, które nadają się do integracji ze standardową infrastrukturą bezpieczeństwa i obsługują takie interfejsy jak LDAP lub JDBC. JBossSX oferuje także standardową implementację klas, które pomogą nam wykorzystać wzorzec opisany w dalszej części rozdziału, w punkcie „Pisanie niestandardowych modułów logowania". W ten sposób w prosty sposób dokonamy integracji własnego protokołu, jeśli żadne z gotowych rozwiązań nie spełnia naszych wymagań. Niniejszy punkt rozpoczynamy od opisu gotowych, dostępnych modułów logowania i sposobu ich konfiguracji. Następnie przechodzimy do dyskusji na temat tworzenia własnej implementacji interfejsu `LoginModule` i jej integracji w serwerem JBoss.

org.jboss.security.auth.spi.IdentityLoginModule

`IdentityLoginModule` jest prostym modułem logowania, który łączy tożsamość określoną w opcjach z dowolnym podmiotem uwierzytelnionym w ramach modułu. Tworzona jest instancja `SimplePrincipal`, która posługuje się nazwą określoną przez parametr `principal`. Oczywiście opisywane rozwiązanie nie nadaje się do zastosowania w wymagającym wysokiego stopnia bezpieczeństwa środowisku produkcyjnym, możemy jednak wykorzystać je w czasie tworzenia i testowania aplikacji, gdy chcemy sprawdzić zabezpieczenia związane z określoną tożsamością i przypisanymi jej rolami.

Oto lista opcji, za pomocą których konfigurujemy opisywany moduł:

- `principal` — nazwa używana przez `SimplePrincipal`, za pomocą której uwierzytelniani są wszyscy użytkownicy. Jeśli nie zdefiniujemy tego atrybutu, domyślnie przyjmuje się wartość `guest`.
- `roles` — nazwy ról, które przypisane będą tożsamości użytkownika. Wartością parametru jest lista ról oddzielonych przecinkami.
- `password-stacking` — jeśli opcji przypiszemy wartość `useFirstPass`, moduł będzie w pierwszej kolejności szukał wspólnej nazwy użytkownika, którą powinna wskazywać właściwość `javax.security.auth.login.name`. Jeśli zostanie ona znaleziona, będzie traktowana jako nazwa tożsamości. W przeciwnym razie nazwa tożsamości ustalona przez moduł logowania zostanie przypisana właściwości `javax.security.auth.login.name`.

Poniżej prezentujemy przykład konfiguracji `XMLLoginConfig`. Na jej podstawie wszystkim użytkownikom przypisana zostanie tożsamość `jduke`, z którą wiążą się role `TheDuke` oraz `AnimatedCharacter`:

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

org.jboss.security.auth.spi.UsersRolesLoginModule

`UsersRolesLoginModule` to nieskomplikowany moduł logowania obsługujący wielu użytkowników i wiele ról. Dane te znajdują się w plikach właściwości Javy. Hasła przypisane poszczególnym użytkownikom znajdują się w pliku o nazwie *users.properties*, natomiast ich role zdefiniowane są w pliku *roles.properties*. Zawartość obu plików ładowana jest w czasie inicjalizacji, w ramach metody `initialize` wywoływanej przez kontekstowego classloadera wątku. Oznacza to, że wymienione pliki mogą być umieszczone w archiwum JAR wraz z aplikacją J2EE, w katalogu konfiguracyjnym JBoss, bądź też w dowolnym innym katalogu w ramach serwera lub ścieżki poszukiwania. Głównym zadaniem modułu jest umożliwienie łatwego testowania bezpieczeństwa aplikacji w sytuacji, w której istnieje wielu zdefiniowanych użytkowników i wiele ról. Wszystko to za pomocą plików właściwości dołączonych do aplikacji.

W pliku *users.propeties* każdemu użytkownikowi odpowiada osobna linia, w której mieści się nazwa użytkownika i hasło (w formacie `użytkownik=hasło`):

```
użytkownik1=hasło1
użytkownik2=hasło2
...
```

Kolejne wiersze pliku *roles.properties* przyjmują następującą postać: `użytkownik=role1,rola2,...`;opcjonalnie możemy podać nazwę grupy. Oto przykład:

```
użytkownik1=rola1,rola2,...
użytkownik1.Grupa1=rola3,rola4,...
użytkownik2=rola1,rola3,...
```

Nazwa w formacie `użytkownik.XXX` oznacza, że role przypisujemy określonej grupie ról, nazwą grupy jest w tym przypadku `XXX`. A zapis `użytkownik=...` jest skróconą formą zapisu `użytkownik.Role`, gdzie `Role` to standardowa nazwa grupy, której spodziewa się `JaasSecurityManager` i która posiada przypisane uprawnienia.

Dwie poniższe definicje są jednoznaczne:

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

Opisywany moduł logowania konfigurować możemy za pomocą następujących parametrów:

- `unauthenticatedIdentity` — parametr ten pozwala zdefiniować tożsamość, która przypisana zostanie do żądań nieposiadających żadnych informacji uwierzytelniających. Za pomocą tej opcji możemy zezwolić niezabezpieczonym serwletom na wywoływanie tych metod komponentów EJB, które nie wymagają określonej roli. Podana w opisywany sposób tożsamość nie posiada bowiem żadnych przypisanych ról, tak więc pozwala wyłącznie na dostęp do niezabezpieczonych komponentów EJB bądź metod obiektów EJB, dla których nie zdefiniowano reguł bezpieczeństwa.
- `password-stacking` — jeśli opcji przypiszemy wartość `useFirstPass`, moduł będzie w pierwszej kolejności szukał wspólnej nazwy użytkownika i jego hasła, na które powinny wskazywać odpowiednio właściwości `javax.security.auth.login.name` oraz `javax.security.auth.login.password`. Jeśli zostaną one znalezione, będą traktowane jako nazwa tożsamości i odpowiadające jej hasło. W przeciwnym razie nazwa tożsamości i hasło ustalone przez moduł logowania zostaną przypisane właściwościom `javax.security.auth.login.name` oraz `javax.security.auth.login.password`.
- `hashAlgorithm` — opcja pozwala wskazać algorytm `java.security.MessageDigest`, który służyć będzie do haszowania haseł. Ponieważ nie jest przewidziana żadna wartość domyślna, włączenie haszowania wiąże się z koniecznością ustalenia wartości parametru. Gdy parametr `hashAlgorithm` jest ustalony, hasło w postaci otwartego tekstu otrzymane za pomocą metody `callbackHandler` jest haszowane przed dalszym przekazaniem go w roli argumentu `inputPassword` metody `UsernamePasswordLoginModule.validatePassword`. Wartość `expectedPassword`, znajdująca się w pliku właściwości *users.properties*, musi być haszowana w ten sam sposób.
- `hashEncoding` — format haszowanego hasła. Parametr może być jedną z następujących wartości: `base64` lub `hex`. Wartość domyślna to `base64`.
- `hashCharset` — opcja określa sposób kodowania hasła z postaci tekstowej do tablicy bajtów. Wartością domyślną są ustawienia systemowe.
- `userProperties` — opcja pozwala wskazać inny plik właściwości zawierający przyporządkowanie nazw i haseł użytkowników. Przypominamy, że domyślnie jest to plik *users.properties*.
- `rolesProperties` — opcja pozwala wskazać inny plik właściwości zawierający przyporządkowanie nazw i ról. Domyślnie jest to plik *roles.properties*.

Poniżej prezentujemy przykład konfiguracji `XMLLoginConfig`, na podstawie której nieuwierzytelnionym użytkownikom przypisywana jest tożsamość o nazwie `nobody`, a hasła w postaci haszowanej za pomocą algorytmu MD5 i w formacie base64 znajdują się w pliku o niestandardowej nazwie *usersb64.properties*:

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

org.jboss.security.auth.spi.LdapLoginModule

`LdapLoginModule` to implementacja modułu logowania, która w czasie uwierzytelniania klienta używa serwera LDAP. Korzystamy w tym przypadku z loginu JNDI oraz odpowiednich ustawień konfiguracyjnych modułu. Modułem `LdapLoginModule` należy posługiwać się wówczas, gdy dane użytkownika i dotyczące go informacje uwierzytelniające są przechowywane na serwerze LDAP, a dostęp do niego zapewnia określony dostawca JNDI.

Informacje dotyczące sposobu łączenia się z serwerem LDAP zawarte są w opcjach konfiguracyjnych i służą do prawidłowego stworzenia kontekstu początkowego JNDI. Poniżej widzimy listę parametrów dotyczących łączenia z LDAP za pomocą JNDI:

- `java.naming.factory.initial` — nazwa klasy będącej implementacją interfejsu `InitialContextFactory`. Domyślnie jest to dostawca LDAP firmy Sun, klasa `com.sun.jndi.ldap.LdapCtxFactory`.
- `java.naming.provider.url` — URL serwera LDAP.
- `java.naming.security.authentication` — poziom określający stopień bezpieczeństwa. Wartością domyślną jest `simple`.
- `java.naming.security.protocol` — parametr wskazuje na protokół stosowany przy bezpiecznym dostępie, np. `ssl`.
- `java.naming.security.principal` — tożsamość uwierzytelniająca klienta odwołującego się do usługi. Parametr ten opiszemy dokładnie już za chwilę.
- `java.naming.security.credentials` — wartość tego parametru zależy od wariantu uwierzytelniania. Może to być haszowane hasło, hasło w postaci otwartego tekstu, klucz, certyfikat itp.

A oto pozostałe parametry konfiguracyjne modułu logowania:

- `principalDNPrefix` — opcja określa prefiks dodawany do nazwy użytkownika w celu stworzenia rozróżnialnej nazwy. Więcej szczegółów znajduje się w opisie kolejnej opcji.

- `principalDNSuffix` — opcja określa wartość dodawaną na końcu nazwy użytkownika. Z opcji tej korzystamy w przypadkach, gdy nie chcemy zmuszać użytkownika do wprowadzania pełnej, rozróżnialnej nazwy. Tak więc ostateczna wartość `userDN` wyznaczona jest na podstawie następującego wzoru: `principalDNPrefix + <nazwa użytkownika> + principalDNSuffix`.
- `useObjectCredential` — parametr przyjmuje wartość `true` lub `false`. Użycie pierwszej z tych wartości wskazuje, że dane uwierzytelniające będą przekazane w postaci instancji typu `Object` za pomocą typu `org.jboss.security.auth.callback.ObjectCallback`. W drugim przypadku będzie to hasło w postaci tablicy typu `char[]` uzyskane za pomocą JAAS `PasswordCallback`. Istnienie parametru pozwala na przesyłanie danych do serwera LDAP w formie innej niż `char[]`.
- `rolesCtxDN` — opcja określa stałą nazwę rozróżnialną odpowiadającą kontekstowi, w ramach którego szukamy ról użytkownika.
- `userRolesCtxDNAttributeName` — ten parametr określa nazwę atrybutu obiektu użytkownika zawierającego rozróżnialną nazwę kontekstu, w jakim należy poszukiwać ról użytkownika. Różni się on od parametru `rolesCtx` tym, iż kontekst, jaki należy przeszukiwać może być inny dla każdego użytkownika.
- `roleAttributreID` — ten parametr określa nazwę atrybutu zawierającego role użytkowników. Jeśli nie zostanie on określony jawnie, to przyjmie domyślną wartość `roles`.
- `roleAttributeIsDN` — ten parametr wskazuje flagę określającą, czy parametr `roleAttributeID` zawiera w pełni rozróżnialną nazwę obiektu roli czy też nazwę roli. Jeśli przyjmie on wartość `false`, to nazwa roli zostanie odczytana z wartości parametru `roleAttributeID`. Z kolei, jeśli atrybut ten będzie miał wartość `true`, to będzie to oznaczać, iż atrybut roli reprezentuje rozróżnialną nazwę obiektu roli. Nazwą roli jest wartość atrybutu `roleNameAttributeId` nazwy kontekstu określonej przez rozróżnialną nazwę zapisaną w atrybucie `roleAttributeId`. W niektórych schematach katalogów (na przykład, w Active Dirctory firmy Microsoft) atrybuty roli przechowywane jako w pełni rozróżnialne nazwy obiektów ról, a nie jako zwyczajne nazwy. W takich przypadkach właściwość `roleAttributeIsDN` powinna przyjąć wartość `true`. Jej domyślną wartością jest `false`.
- `roleNameAttributeID` — ta opcja określa nazwę atrybutu kontekstu wskazywanego przez rozróżnialną nazwę zapisaną w atrybucie `roleCtxDN` zawierającego nazwę roli. Jeśli atrybut `roleAttributeIsDN` przyjmie wartość `true`, to atrybutu `roleNameAttributeID` można użyć do odszukania atrybutu `name` obiektu. Jego domyślną wartością jest `group`.
- `uidAttributeID` — ta opcja określa nazwę atrybutu obiektu, który zawiera role odpowiadające identyfikatorowi użytkownika. Służy ona od odnajdywania ról użytkownika. W przypadku gdy opcja ta nie zostanie jawnie podana, to przyjmuje wartość domyślną `uid`.

- `matchOnUserDN` — ta opcja jest flagą logiczną (przyjmującą wartości `true` lub `false`) określającą, czy podczas poszukiwania ról użytkowników należy uwzględniać ich w pełni rozróżnialne nazwy. Jeśli atrybut ten przyjmie wartość `false`, to z wartością atrybutu `uidAttributeName` będzie porównywana nazwa użytkownika. W przeciwnym razie, kiedy atrybut przyjmie wartość `true`, podczas porównywania będzie używana pełna wartość `userDN`.
- `unauthenticatedIdentity` — ta opcja określa główną nazwę, jaką należy przypisać do żądania, które nie zawiera żadnych informacji uwierzytelniających. Jej działanie jest dziedziczone po klasie bazowej `UserPasswordLoginModule`.
- `password-stacking` — kiedy tej opcji zostanie przypisana wartość `useFirstPass`, to moduł logowania w pierwszej kolejności będzie poszukiwać nazwy użytkownika oraz jego hasła we właściwościach o nazwach `javax.security.auth.login.name` oraz `javax.security.auth.login.password` przechowywanych we wspólnej mapie stanu modułu logowania. W razie ich odnalezienia, nazwy te zostaną zastosowane jako nazwa tożsamości oraz jej hasło. W przeciwnym razie nazwa tożsamości oraz jej hasło zostaną określone przez dany moduł logowania i zapisane odpowiednio we właściwościach `java.security.auth.login.name` oraz `javax.security.auth.login.password`.
- `allowEmptyPasswords` — ta opcja określa flagę informującą, czy puste hasła (czyli hasła o zerowej długości) należy przekazywać do serwera LDAP. Niektóre serwery LDAP traktują takie hasła jak próbę logowania anonimowego, co nie zawsze jest pożądane. Przypisanie tej opcji wartości `false` sprawia, że wszelkie próby logowania z wykorzystaniem hasła pustego będą odrzucane; z kolei, zastosowanie wartości `true` spowoduje, że serwer LDAP będzie weryfikować puste hasła. Domyślną wartością tej opcji jest `true`.

Uwierzytelnianie użytkownika odbywa się poprzez nawiązanie połączenia z serwerem LDAP zgodnie z informacjami konfiguracyjnymi modułu logowania. Nawiązanie połączenia polega na utworzeniu obiektu `InitialLdapContext`, którego otoczenie jest inicjowane właściwościami LDAP JDNI opisanymi we wcześniejszej części tego rozdziału. We właściwości `Context.SECURITY_PRINCIPAL` zapisywana jest rozróżnialna nazwa użytkownika, określana przy wykorzystaniu metody zwrotnej, opcji `principalDNPrefix`, `principalDNSuffix`. Z kolei właściwości `Context.SECURITY_CREDENTIALS`, w zależności od opcji `useObjectCredential`, przypisywany jest łańcuch znaków zawierający hasło bądź też obiekt zawierający odpowiednie informacje.

Po poprawnym uwierzytelnieniu użytkownika, w wyniku którego zostanie utworzona instancja obiektu `InitialLdapContext`, następuje sprawdzenie ról danego użytkownika. W tym celu przeszukiwana jest lokalizacja określona jako `rolesCtxDN`, przy czym w parametrach wyszukiwania zapisywane są wartości opcji `roleAttributeName` oraz `uidAttributeName`. Nazwy zwróconych ról można pobrać, wywołując metodę `toString` atrybutów ról zapisanych w zbiorze wyników wyszukiwania.

Poniżej przedstawiony został fragment pliku `login-config.xml`:

```
<application-policy name="testLDAP">
    <authentication>
        <login-module code="org.jboss.security.auth.spi.LdapLoginModule"
                      flag="required">
            <module-option name="java.naming.factory.initial">
                com.sun.jndi.ldap.LdapCtxFactory
            </module-option>
            <module-option name="java.naming.provider.url">
                ldap://ldaphost.jboss.org:1389/
            </module-option>
            <module-option name="java.naming.security.authentication">
                simple
            </module-option>
            <module-option name="principalDNPrefix">uid=</module-option>
            <module-option name="principalDNSuffix">
                ,ou=People,dc=jboss,dc=org
            </module-option>
            <module-option name="rolesCtxDN">
                ou=Roles,dc=jboss,dc=org
            </module-option>
            <module-option name="uidAttributeID">member</module-option>
            <module-option name="matchOnUserDN">true</module-option>
            <module-option name="roleAttributeID">cn</module-option>
            <module-option name="roleAttributeIsDN">false </module-option>
        </login-module>
    </authentication>
</application-policy>
```

Poniżej przedstawiony został plik LDIF reprezentujący strukturę katalogu, na którym operują powyższe dane:

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

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

dn: uid=jduke,ou=People,dc=jboss,dc=org
objectclass: top
objectclass: uidObject
objectclass: person
uid: jduke
cn: Java Duke
sn: Duke
userPassword: theduke

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

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

Przeglądając przedstawioną we wcześniejszej części rozdziału konfigurację modułu logowania `testLDAP`, można zauważyć, iż zgodnie z wartościami opcji `java.naming.factory.initial`, `java.naming.factory.url` oraz `java.naming.security` zostanie wykorzystana implementacja dostawcy LDAP JNDI firmy Sun, serwer LDAP dostępny pod adresem *ldaphost.jboss.org* i działający na porcie 1389, a do nawiązania połączenia z serwerem LDAP zostanie użyta prosta metoda uwierzytelniania (`simple`).

Podczas prób nawiązania połączenia z serwerem LDAP moduł logowania korzysta z w pełni rozróżnialnej nazwy reprezentującej użytkownika starającego się o uwierzytelnienie. Nazwa ta tworzona jest na podstawie opcji `principalDNPrefix` przekazanej w nazwie użytkownika oraz opcji `principalDNSuffix` opisanej we wcześniejszej części rozdziału. W przedstawionym przykładzie użytkownik o nazwie `jduke` zostały skojarzony z wpisem `uid=jduke,ou=People,dc=jboss,dc=org`. Zakładamy przy tym, że serwer LDAP przeprowadza uwierzytelnianie użytkowników, korzystając przy tym z atrybutu `userPassword` (który w powyższym przykładzie ma wartość `theduke`). W taki sposób działa przeważająca większość serwerów LDAP, niemniej jednak, jeśli używany serwer LDAP działa inaczej, konieczne będzie ustawienie opcji uwierzytelniania w odpowiedni sposób.

Po udanym uwierzytelnieniu użytkownika należy pobrać role, na podstawie których zostanie przeprowadzona autoryzacja. W tym celu przeprowadzana jest analiza poddrzewa `roleCtxDN` w poszukiwaniu wpisów, w których wartość `uidAttributeID` odpowiada danemu użytkownikowi. Jeśli opcja `matchOnUserDN` ma wartość `true`, to wyszukiwanie bazuje na rozróżnialnej nazwie użytkownika. W przeciwnym razie używana jest faktyczna, podana nazwa. W powyższym przykładzie analizowany będzie fragment poniżej `ou=Roles,dc=jboss,dc=org`, a poszukiwane w nim będą wszelkie wpisy, w których wartość atrybutu `member` będzie wynosić `uid=duke,ou=People,dc=jboss,dc=org`. W tym przykładzie zwrócona zostanie wartość `cn=JBossAdmin` we wpisie `roles`.

W wyniku wyszukiwania zwracany jest atrybut określony przez opcję `roleAttributeID`. W powyższym przykładzie atrybutem tym jest `cn`. Ponieważ wyszukiwanie zwróci wartość `JBossAdmin`, użytkownik `jduke` zostanie powiązany z rolą `JBossAdmin`.

Często zdarza się, że lokalny serwer LDAP udostępnia usługi związane z określaniem tożsamości oraz uwierzytelnianiem, lecz nie jest w stanie korzystać z usług autoryzacji. Wynika to z faktu, iż role używane w aplikacji nie zawsze dobrze odpowiadają grupom LDAP, a administratorzy zazwyczaj nie są chętni do umieszczania na centralnych serwerach LDAP zewnętrznych danych, z których korzystają aplikacje. Właśnie z tych powodów bardzo często moduły przeprowadzające uwierzytelnianie w oparciu o serwery LDAP są używane wraz z dodatkowymi modułami logowania, takimi jak moduły logujące wykorzystujące bazy danych, które są w stanie dostarczać informacji o rolach odpowiadających potrzebom tworzonej aplikacji.

org.jboss.security.auth.spi.DatabaseServerLoginModule

`DatabaseServerLoginModule` jest implementacją modułu logowania wykorzystującego JDBC, który obsługuje zarówno uwierzytelnianie, jak i przypisywanie ról. Można z niego korzystać, jeśli informacje o nazwach użytkowników, ich hasłach oraz skojarzonych z nimi rolach są przechowywane w relacyjnej bazie danych. Moduł ten wykorzystuje dwie logiczne tabele:

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

Pierwsza z tych tabel — `Principals` — kojarzy `PrincipalID` użytkownika z poprawnym hasłem. Z kolei tabela `Roles` kojarzy `PrincipalID` użytkownika ze zbiorem ról. Role służące do określania uprawnień użytkownika muszą być zapisywane w wierszach, w kolumnie `RoleGroup`. Tabele te są logiczne w tym sensie, iż istnieje możliwość podania zapytania SQL, które będzie używane przez moduł logowania. Jedyny wymóg sprowadza się do tego, by zwracany obiekt `java.sql.ResultSet` miał identyczną strukturę jak przedstawione powyżej dwie tabele: `Principals` oraz `Roles`. Rzeczywiste nazwy tabel oraz ich kolumn nie mają tu większego znaczenia, gdyż wyniki są odczytywane przy wykorzystaniu indeksów kolumn, a nie ich nazw. Aby dokładniej wyjaśnić, co to oznacza, przeanalizujemy przykład bazy danych zawierającej dwie tabele — `Principals` oraz `Roles` — o przedstawionej wcześniej strukturze. Pierwsze z przedstawionych poniżej poleceń SQL zapisuje w tabeli `Principals` wiersz danych, który w kolumnie `PrincipalID` zawiera wartość `java`, a w kolumnie `Password` wartość `echoman`. Pozostałe dwa polecenia zapisują dwa wiersze w tabeli `Roles`. W pierwszym z nich w polu `PrincipalID` zostaje umieszczona wartość `java`, w polu `Roles` określającym nazwę roli wartość `Echo`, a w polu `RoleGroup` wartość `Roles`; w drugim z rekordów w tych samych kolumnach zostaną odpowiednio zapisane wartości: `java`, `caller_java` oraz `CallerPrincipal`.

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

Poniżej przedstawione zostały opcje konfiguracyjne tego modułu logowania:

- `dsJndiName` — ta opcja określa nazwę JNDI źródła danych (`DataSource`) reprezentującego bazę danych zawierającą logiczne tabele `Principals` oraz `Roles`. Jeśli opcja ta nie zostanie jawnie określona, to przyjmie wartość domyślną `java:/DefaultDS`.
- `principalsQuery` — ta opcja określa przygotowane zapytanie SQL stanowiące odpowiednik zapytania `select Password from Principals where PrincipalID=?`. To polecenie zostanie zastosowane, jeśli wartość opcji nie będzie jawnie podana.
- `rolesQuery` — ta opcja określa przygotowane zapytanie SQL stanowiące odpowiednik zapytania `select Role, RoleGroup from Roles where PrincipalID=?`. To polecenie zostanie zastosowane, jeśli wartość opcji nie zostanie jawnie podana.
- `unauthenticatedIdentity` — ta opcja określa tożsamość przypisywaną żądaniom, które nie zawierają żadnych informacji uwierzytelniających.

- `password-stacking` — jeśli ta opcja przyjmie wartość `useFirstPass`, to moduł logowania w pierwszej kolejności spróbuje odczytać ze wspólnej mapy stanu wartości właściwości `javax.security.auth.login.name` oraz `javax.security.auth.login.password`, określające odpowiednio nazwę użytkownika oraz jego hasło. Jeśli wartości te zostaną odczytane, to moduł użyje ich podczas uwierzytelniania. W przeciwnym przypadku tożsamość oraz hasło użytkownika zostaną określone przez moduł logowania i zapisane we odpowiednio we właściwościach `javax.security.auth.login.name` oraz `javax.security.auth.login.password`.
- `hashAlgorithm` — opcja określa nazwę algorytmu `java.security.MessageDigest`, który zostanie użyty do haszowania hasła. Opcja ta nie ma wartości domyślnej, zatem aby haszowanie zostało włączone, należy ją jawnie podać. W przypadku określenia wartości tej opcji, zanim hasło podane otwartym tekstem (zwrócone przez metodę zwrotną `callbackhandler`) będzie przekazane jako argument `inputPassword` do metody `UsernamePasswordLoginModule.validatePassword`, zostanie ono zahaszowane. Wartość `expectedPassword` odczytywana przez moduł logowania z bazy danych musi być zahaszowana w identyczny sposób.
- `hashEncoding` — ta opcja określa format łańcucha znaków, w jakim zostanie zapisane zahaszowane hasło. Opcja ta może przyjmować tylko dwie wartości: `base64` lub `hex`, przy czym jej domyślną wartością jest `base64`.
- `hashCharset` — ta opcja określa sposób kodowania, jaki będzie używany do przekształcenia tekstowego hasła na postać tablicy bajtów. Domyślnie używane jest domyślne kodowanie używanej platformy systemowej.
- `ignorePasswordCase` — ta opcja służy do podania wartości flagi logicznej określającej, czy podczas sprawdzania hasła należy uwzględniać wielkość liter. Może ona być przydatna w przypadku kodowania zahaszowanego hasła w sytuacjach, gdy wielkość liter uzyskanego łańcucha znaków nie ma znaczenia.
- `principalClass` — opcja określa nazwę klasy implementującej interfejs `Principal`. Klasa ta musi udostępniać konstruktor pobierający jeden argument będący łańcuchem znaków, który reprezentuje nazwę użytkownika.

W ramach przykładu konfiguracji modułu `DatabaseServerLoginModule` przeanalizujmy tabele o następującej strukturze:

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

Poniżej przedstawiono fragment pliku konfiguracyjnego *login-config.xml*, który operuje na tych tabelach:

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

```
                <module-option name="rolesQuery">
                    select userRoles, 'Roles' from UserRoles where username=?</module-
➥option>
            </login-module>
        </authentication>
    </application-policy>
```

BaseCertLoginModule

`BaseCertLoginModule` jest modułem, który przeprowadza uwierzytelnianie użytkowników na podstawie certyfikatów X509. Zazwyczaj moduł ten jest używany przez warstwę sieciową korzystającą z metody uwierzytelniania `CLIENT-CERT`. Moduł ten służy wyłącznie do uwierzytelniania, dlatego też, w celu pełnego zdefiniowania dostępu do zabezpieczonego komponentu sieciowego lub komponentu EJB, należy wykorzystywać go wspólnie z innym modułem logowania, który będzie w stanie pobierać informacje o rolach konieczne do autoryzacji użytkownika. Dwie klasy pochodne tego modułu, `CertRolesLoginModule` oraz `DatabaseCertLoginModule`, rozszerzają jego działanie i zapewniają możliwość pobierania ról z właściwości lub z bazy danych.

Do przeprowadzenia weryfikacji użytkownika moduł `BaseCertLoginModule` musi korzystać z obiektu `KeyStore`. Obiekt ten jest pobierany przy użyciu implementacji interfejsu `org.jboss.security.SecurityDomain`. Przeważnie implementacja klasy `SecurityDomain` jest konfigurowana przy użyciu komponentu MBean `org.jboss.security.plugins.JaasSecurityDomain`, przedstawionego na poniższym przykładzie pliku `jboss-serice.xml`:

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

Powyższy fragment tworzy domenę bezpieczeństwa o nazwie `jmx-console`, której klasa implementująca interfejs `SecurityDomain` jest dostępna za pomocą JNDI pod nazwą `java:/jaas/jmx-console`, zgodną ze wzorcem nazw domen bezpieczeństwa stosowanych przez JBossSX. Aby zabezpieczyć aplikację sieciową taką jak `jmx-console.war` przy wykorzystaniu certyfikatów klienta oraz autoryzacji bazującej na rolach, w pierwszej kolejności należałoby zmodyfikować plik *web.xml* i zadeklarować w nim zabezpieczane zasoby, role, jakie mają do nich dostęp, oraz dziedzinę zabezpieczeń, jakiej należy używać podczas uwierzytelniania i autoryzacji. Poniżej przedstawiony został fragment pliku *web.xml* określający wszystkie te informacje:

```
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC
                  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
                  "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    ...
```

```
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>HtmlAdaptor</web-resource-name>
          <description>Przykładowa konfiguracja bezpieczeństwa pozwalająca
➥użytkownikom posiadającym rolę JBossAdmin na dostęp do aplikacji sieciowej
                JMX console </description>
            <url-pattern>/*</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>JBossAdmin</role-name>
        </auth-constraint>
    </security-constraint>
    <login-config>
        <auth-method>CLIENT-CERT</auth-method>
        <realm-name>JBoss JMX Console</realm-name>
    </login-config>
    <security-role>
        <role-name>JBossAdmin</role-name>
    </security-role>
</web-app>
```

Następnie w pliku *jboss-web.xml* należy określić dziedzinę bezpieczeństwa JBossa:

```
<jboss-web>
    <security-domain>java:/jaas/jmx-console</security-domain>
</jboss-web>
```

Ostatnim krokiem jest zdefiniowanie konfiguracji modułu logowania dla określonej powyżej domeny bezpieczeństwa `jmx-console`. Niezbędne informacje są umieszczane w pliku *conf/login-config.xml*:

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

W tym przypadku moduł `BaseCertLoginModule` jest używany do uwierzytelniania certyfikatu użytkownika, natomiast ze względu na zastosowanie opcji `password-stacking=useFirstPass` moduł `UserRolesLoginModule` jest używany wyłącznie do autoryzacji.

Zarówno `localhost.keystore`, jak i `jmx-console-roles.properties` wymagają wpisu, który określi odwzorowanie z tożsamością skojarzoną z certyfikatem klienta. Domyślnie tożsamość jest tworzona przy wykorzystaniu rozróżnialnej nazwy podanej w certyfikacie klienta. Przeanalizujmy następujący certyfikat:

```
[starksm@banshee9100 conf]$ keytool -printcert -file unit-tests-client.export
Owner: CN=unit-tests-client, OU=JBoss Inc., O=JBoss Inc., ST=Washington, C=US
Issuer: CN=jboss.com, C=US, ST=Washington, L=Snoqualmie Pass, EMAILADDRESS=admin
@jboss.com, OU=QA, O=JBoss Inc.
Serial number: 100103
Valid from: Wed May 26 07:34:34 PDT 2004 until: Thu May 26 07:34:34 PDT 2005
Certificate fingerprints:
         MD5: 4A:9C:2B:CD:1B:50:AA:85:DD:89:F6:1D:F5:AF:9E:AB
         SHA1: DE:DE:86:59:05:6C:00:E8:CC:C0:16:D3:C2:68:BF:95:B8:83:E9:58
```

Na potrzeby `localhost.keystore` certyfikat ten powinien zostać zapisany z aliasem o postaci `CN=unit-tests-client, OU=JBoss Inc.,O=JBoss Inc., ST=Washington, C=US`; z kolei `jmx-console-roles.properties` także wymaga wpisu dla tego samego wpisu. Ponieważ wpis DN zawiera wiele znaków, które normalnie są traktowane jako separatory, zatem konieczne będzie poprzedzenie ich znakami odwrotnego ukośnika, jak w poniższym przykładzie:

```
# Przykładowy plik roles.properties używany przez moduł logowania UsersRolesLoginModule
CN\=unit-tests-client,\ OU\=JBoss\ Inc.,\ O\=JBoss\ Inc.,\ ST\=Washington,\
➥C\=US=JBossAdmin
admin=JBossAdmin
```

org.jboss.security.auth.spi.RunAsLoginModule

JBoss posiada pomocniczy moduł logowania o nazwie `RunAsLoginModule`, który pozwala na zastosowanie roli pomocniczej podczas trwania fazy uwierzytelniania i zaprzestanie stosowania tej roli w momencie zakończenia uwierzytelniania (niezależnie od jego wyniku). Moduł ten został stworzony w celu udostępnienia roli, z jakiej mogłyby korzystać inne moduły logowania, które podczas przeprowadzania uwierzytelniania muszą mieć dostęp do zasobów chronionych. Na przykład z modułu `RunAsLoginModule` może korzystać moduł logowania korzystający z chronionego komponentu EJB. Moduł `RunAsLogingModule` musi zostać skonfigurowany przed zastosowaniem innych modułów logowania korzystających z roli pomocniczej.

`Jedyną opcją konfiguracyjną tego modułu jest roleName`; zawiera ona nazwę roli pomocniczej, która będzie używana podczas fazy logowania. Jeśli nazwa tej roli nie zostanie jawnie podana, przyjęta zostanie wartość domyślna `nobody`.

org.jboss.security.ClientLoginModule

Moduł `ClientLoginModule` jest implementacją interfejsu `LoginModule` używanego przez klientów JBossa do określenia tożsamości oraz informacji uwierzytelniających klienta. Moduł ten zapisuje we właściwości `org.jboss.security.SecurityAssociation.principal` wartość typu `NameCallback` zwróconą przez metodę `callbackhandler`, a we właściwości `org.jboss.security.SecurityAssociation.credential` zwróconą wartość typu `PasswordCallback`. Jest to jedyny dostępny mechanizm pozwalający klientowi na

określenie, kto odwołał się do bieżącego wątku. Modułu tego muszą używać zarówno niezależne aplikacje klienckie, jak i aplikacje działające w środowisku serwera, które funkcjonują jako klienty EJB w sytuacjach, gdy środowisko zabezpieczeń nie zostało skonfigurowane w sposób zapewniający możliwość niezauważalnego korzystania z podsystemu JBossSX. Oczywiście zawsze można bezpośrednio określić informacje `org.jboss.security.SecurityAssociation`, niemniej jednak interfejs ten jest uważany za API wewnętrzne, które w każdej chwili może ulec zmianie bez wcześniejszych zapowiedzi.

Należy zauważyć, że ten moduł logowania nie przeprowadza uwierzytelniania, a jedynie kopiuje przekazane do niego informacje dotyczące logowania do warstwy wywołań EJB serwera JBoss w celu ich późniejszego wykorzystania podczas uwierzytelniania. Jeśli niezbędne jest przeprowadzenie uwierzytelniania użytkownika po stronie klienta, to oprócz modułu `ClientLoginModule` należy wykorzystać także inny moduł logowania.

Poniżej zostały opisane dostępne opcje konfiguracyjne tego modułu:

- `multi-threaded` — jeśli tej opcji zostanie przypisana wartość `true`, to każdy wątek logowania będzie dysponować własnym obszarem służącym do przechowywania danych o tożsamości i informacji uwierzytelniających. Możliwość ta jest przydatna w środowiskach klienckich, w których, w osobnych wątkach, są aktywne dane wielu różnych użytkowników. Jeśli opcja ta przyjmie wartość `true`, to każdy wątek musi osobno przeprowadzać logowanie. W przeciwnym przypadku, kiedy opcja przyjmie wartość `false`, zarówno tożsamość, jak i inne informacje uwierzytelniające będą zmiennymi globalnymi, dostępnymi i używanymi przez wszystkie wątki działające na danej wirtualnej maszynie Javy. Wartością domyślną tej opcji jest `false`.
- `password-stacking` — jeśli ta opcja przyjmie wartość `useFirstPass`, to moduł logowania w pierwszej kolejności spróbuje odczytać ze wspólnej mapy stanu wartości właściwości `javax.security.auth.login.name` oraz `javax.security.auth.login.password`, określające odpowiednio nazwę użytkownika oraz jego hasło. W ten sposób moduły, które są używane, mogą przed tym określić poprawną nazwę użytkownika oraz hasło, które zostaną przekazane do serwera JBoss. Opcja ta jest używana w przypadkach, gdy chcemy przeprowadzić uwierzytelnianie aplikacji po stronie klienta, używając przy tym innego modułu logowania, jak na przykład `LdapLoginModule`.
- `restore-login-identity` — w przypadku, gdy opcja ta przyjmie wartość `true`, informacje zapisane w obiekcie `SecurityAssociation` i przekazywane do metody `login()` są zapisywane, a następnie odtwarzane w momencie przerywania logowania lub wylogowywania klienta. Jeśli opcja ta przyjmie domyślną wartość `false`, przerwanie operacji logowania lub wylogowanie klienta powoduje usunięcie wszelkich informacji zapisanych w obiekcie `SecurityAssociation`. Przypisanie tej opcji wartości `true` jest konieczne w przypadkach, gdy trzeba zmienić, a następnie przywrócić oryginalną tożsamość.

Przedstawiony poniżej fragment konfiguracji modułu `ClientLoginModule` przedstawia domyślny wpis konfiguracyjny skopiowany z pliku `client/auth.conf`, który można znaleźć w dystrybucji serwera JBoss:

```
other {
    // Put your login modules that work without jBoss here[1]

    // jBoss LoginModule
    org.jboss.security.ClientLoginModule required;

    // Put your login modules that need jBoss here[2]
};
```

Pisanie niestandardowych modułów logowania

Jeśli moduły logowania, w jakie domyślnie wyposażony jest podsystem JBossSX, nie są w stanie współpracować z używanym systemem zabezpieczeń, to można napisać własne moduły implementujące wszystkie niezbędne możliwości.

Czytelnik zapewne pamięta, że klasa `JaasSecurityManager` przedstawiona wcześniej w podrozdziale „Architektura JBossSX" oczekuje, że obiekty `Subject` będą wykorzystywane w ściśle określony sposób. Koniecznie należy zrozumieć sposoby przechowywania informacji przez obiekty tego typu, ich cechy oraz oczekiwane sposoby ich stosowania. Wiedza ta jest konieczna do tworzenia modułów logowania, które będą w stanie współpracować z klasą `JaasSecurityManager`. W tej części rozdziału zostały przedstawione wymienione wcześniej zagadnienia, a oprócz tego podano w niej informacje o dwóch abstrakcyjnych klasach bazowych implementujących interfejs `LoginModule`, które mogą pomóc w tworzeniu własnych modułów logowania.

JBoss udostępnia sześć metod służących do pobierania informacji przechowywanych w obiektach `Subject`:

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

Jeśli chodzi o informacje uwierzytelniające oraz informacje o rolach udostępniane przez obiekty `Subject`, to w podsystemie JBossSX dokonano najbardziej naturalnego wyboru: są one zwracane w formie zbioru (obiektu typu `Set`) przez metody `getPrincipals()` oraz `getPrincipals(java.lang.Class)`. Poniżej przedstawiony został sposób ich wykorzystania:

[1] Tutaj zapisz moduły logowania, które nie wymagają JBossa.

[2] Tutaj zapisz moduły logowania, które wymagają JBossa.

- Informacje o tożsamości użytkownika (nazwa użytkownika, numer PESEL, identyfikator pracownika i tak dalej) są przechowywane w zbiorze `Subject Principals` w formie obiektów typu `java.security.Principal`. Klasa, która reprezentuje tożsamość użytkownika, implementująca interfejs `java.security.Principal` musi zapewniać operacje porównania oraz równości bazujące na nazwie tożsamości uwierzytelniającej. JBoss udostępnia przydatną implementację takiej klasy o nazwie `org.jboss.security.SimplePrincipal`. W razie potrzeby do zbioru `Subject Principals` można dodawać inne instancje typu `Principal`.
- Przypisane role użytkowników także są przechowywane w zbiorze `Principals`, jednak są one grupowane na podstawie nazwy roli przy wykorzystaniu obiektów `java.security.acl.Group`. Interfejs `Group` definiuje kolekcję obiektów `Principals` i (bądź) `Groups`. `Group` jest interfejsem pochodnym interfejsu `java.security.Principal`. Dowolną ilość ról można przypisać do obiektu `Subject`. Obecnie JBossSX używa dwóch dobrze znanych zbiorów ról noszących nazwy `Roles` oraz `CallerPrincipal`. Grupa `Roles` jest kolekcją obiektów `Principal` reprezentujących nazwane role dostępne w domenie aplikacji, w której `Subject` został uwierzytelniony. Ten zbiór ról jest używany między innymi przez metodę `EJBContext.isCallerInRole(String)`, której komponenty EJB mogą używać do sprawdzenia, czy wywołujący je klient należy do nazwanej roli domeny aplikacji. Z tego zbioru korzystają także mechanizmy obiektu przechwytującego odpowiadającego za bezpieczeństwo, realizujące sprawdzenie uprawnień. Z kolei grupa `CallerPrincipal` zawiera pojedynczy obiekt `Principal` przypisany użytkownikowi w domenie aplikacji. Jest ona wykorzystywana przez metodę `EJBContext.getCallerPrincipal()` w celu zapewnienia domenie aplikacji możliwości odwzorowania tożsamości użytkownika stosowanej w środowisku pracy na tożsamość, która będzie przydatna dla samej aplikacji. Jeśli obiekt `Subject` nie zawiera grupy `CallerPrincipal`, oznacza to, że tożsamość używana w aplikacji jest taka sama jak tożsamość używana w środowisku pracy.

Obsługa sposobu wykorzystania obiektu Subject

W celu uproszczenia implementacji opisanego powyżej sposobu wykorzystania obiektu `Subject` podsystem JBossSX udostępnia dwa abstrakcyjne moduły logowania, które obsługują zapisywanie informacji w uwierzytelnionym obiekcie `Subject`, a jednocześnie wymuszają jego poprawne zastosowanie. Bardziej ogólnym z nich jest klasa `org.jboss.security.auth.spi.AbstractServerLoginModule`. Zapewnia ona konkretną implementację interfejsu `javax.security.auth.spi.LoginModule` i udostępnia abstrakcyjne metody służące do realizacji wszystkich kluczowych operacji typowych dla infrastruktury zabezpieczeń środowiska pracy. Poniższy fragment kodu źródłowego klasy przedstawia jej kluczowe szczegóły, a zamieszczone w nim komentarze opisują przeznaczenie poszczególnych metod:

```
1: package org.jboss.security.auth.spi;
2: /**
3:  * Ta klasa implementuje wspólne możliwości funkcjonalne, które muszą
4:  * posiadać działające na serwerze moduły JAAS LoginModule, oraz
5:  * implementuje narzucany przez JBossSX schemat wykorzystania obiektów
```

```
6:  * Subject do przechowywania informacji o tożsamości i rolach.
7:  * Klasę tę można wykorzystać do stworzenia własnych modułów logowania
8:  * LoginModule, w których zostaną przesłonięte metody login(),
9:  * getRoleSets() oraz getIdentity().
10:  */
11: public abstract class AbstractServerLoginModule
12:     implements javax.security.auth.spi.LoginModule
13: {
14:     protected Subject subject;
15:     protected CallbackHandler callbackHandler;
16:     protected Map sharedState;
17:     protected Map options;
18:     protected Logger log;
19:
20:     /** Flaga określająca, czy należy używać wspólnych danych
21:      *  o tożsamości */
22:     protected boolean useFirstPass;
23:     /**
24:      * Flaga informująca o tym, czy faza logowania zakończyła się
25:      * powodzeniem. Klasy pochodne przesłaniające metodę login()
26:      * muszą przypisywać tej składowej wartość true, jeśli logowanie
27:      * zakończyło się pomyślnie.
28:      */
29:     protected boolean loginOk;
30:
31:     // ...
32:
33:     /**
34:      * Metoda inicjalizuje moduł logowania. Zapisuje obiekt
35:      * Subject, CallbackHandler, mapę sharedState oraz mapę
36:      * zawierającą opcje sesji logowania. Metodę tę należy przesłaniać,
37:      * jeśli klasy pochodne muszą przetwarzać własne opcje sesji.
38:      * W takim przypadku konieczne jest wywołanie tej metody w klasie
39:      * bazowej - super.initialize(...).
40:      *
41:      * <p>
42:      * W opcjach poszukiwany jest parametr <em>password-stacking</em>.
43:      * Jeśli przypisano mu wartość "useFirstPass", to tożsamość używana podczas
44:      * logowania zostanie pobrana z wartości
45:      * <code>javax.security.auth.login.name</code> mapy sharedState, natomiast
46:      * informacja potwierdzająca tożsamość z wartości
47:      * <code>javax.security.auth.login.password</code> tej samej mapy.
48:      *
49:      * @param subject obiekt Subject, który należy zaktualizować po pomyślnym
50:      * logowaniu
51:      * @param callbackHandler obiekt CallbackHandler, który zostanie użyty w celu pobrania
52:      * informacji o tożsamości użytkownika oraz danych niezbędnych do jego uwierzytelnienia
53:      * @param sharedState obiekt Map używany przez wszystkie skonfigurowane moduły logowania
54:      * @param options parametry przekazywane do modułu logowania
55:      */
56:     public void initialize(Subject subject,
57:                            CallbackHandler callbackHandler,
58:                            Map sharedState,
59:                            Map options)
60:     {
61:         // ...
62:     }
```

```
63:
64:     /**
65:      * Jeśli użyto wartości "useFirstPass", to metoda ta poszukuje w mapie
66:      * sharedState wartości javax.security.auth.login.name oraz
67:      * javax.security.auth.login.password i zwraca wartość true, jeśli
68:      * zostały one odnalezione. Jeśli wartości te nie istnieją lub wynoszą
69:      * null, to metoda zwraca wartość false.
70:      *
71:      * Należy zauważyć, że w przypadku pomyślnego logowania klasy pochodne
72:      * przesłaniające metodę login() muszą przypisywać składowej loginOk
73:      * wartość true, aby w późniejszej fazie logowania w obiekcie
74:      * Subject zostały poprawnie zapisane informacje. Ta implementacja
75:      * przypisuje składowej loginOk wartość true, jeśli metoda login() zwróci
76:      * wartość true; w przeciwnym razie składowej loginOk przypisywana jest
77:      * wartość false.
78:      */
79:     public boolean login()
80:          throws LoginException
81:     {
82:        // ...
83:     }
84:
85:     /**
86:      * Ta metoda jest przesłaniana przez klasy pochodne, tak by zwracała
87:      * obiekt Principal odpowiadający podstawowej tożsamości użytkownika.
88:      */
89:     abstract protected Principal getIdentity();
90:
91:     /**
92:      * Ta metoda, przesłaniana przez klasy pochodne, zwraca obiekt Groups
93:      * zawierający zbiór ról przypisanych danemu użytkownikowi. W najprostszym
94:      * przypadku klasy pochodne powinny utworzyć obiekt Group skojarzony z
95:      * nazwą "Roles" i zawierający role przypisane danemu użytkownikowi.
96:      * Drugą powszechnie używaną grupą jest "CallerPrincipal" określająca
97:      * tożsamość użytkownika w danej aplikacji, a nie w domenie bezpieczeństwa.
98:      *
99:      * @return Group[] containing the sets of roles
100:      */
101:     abstract protected Group[] getRoleSets() throws LoginException;
102: }
```

Należy zwrócić uwagę na zmienną `loginOk`. Wszystkie klasy pochodne przesłaniające metodę `login()` muszą przypisać jej wartość `true` w przypadku udanego logowania lub wartość `false` w przeciwnym razie. W przypadku, gdy wartość tej zmiennej nie zostanie poprawnie określona, może się okazać, że metoda `commit()` bądź to nie zaktualizuje informacji o podmiocie (obiektu `Subject`), bądź też zaktualizuje je w nieodpowiednim momencie. Możliwość śledzenia wyników fazy logowania została dodana po to, by połączone w łańcuch moduły logowania nie musiały zgłaszać poprawności logowania w ściśle określonej kolejności, aby cała operacja logowania zakończyła się pomyślnie.

Drugi z abstrakcyjnych bazowych modułów logowania nadających się do wykorzystania podczas tworzenia własnych modułów nosi nazwę `org.jboss.security.auth.spi.UsernamePasswordLoginModule`. Moduł ten wymaga, by tożsamość użytkownika była określana przy użyciu jego nazwy zapisanej jako łańcuch znaków (`String`), a hasło

potwierdzające tę tożsamość było przekazywane jako tablica znaków (`char[]`). Takie rozwiązanie dodatkowo upraszcza implementację niestandardowych modułów logowania. Oprócz tego moduł `UserPasswordLoginModule` zapewnia możliwość kojarzenia użytkowników anonimowych (identyfikowanych przy użyciu pustej nazwy oraz hasła) z tożsamością, która nie posiada żadnych ról. Poniższy fragment kodu źródłowego klasy przedstawia jej kluczowe szczegóły, a zamieszczone w nim komentarze opisują przeznaczenie poszczególnych metod.

```
1: package org.jboss.security.auth.spi;
2: /**
3:  * Abstrakcyjna klasa pochodna klasy AbstractServerLoginModule wymagająca,
4:  * by tożsamość była określana poprzez podanie nazwy użytkownika zapisanej
5:  * jako łańcuch znaków, a informacje uwierzytelniające poprzez podanie hasła
6:  * zapisanego jako tablica znaków. Klasy pochodne muszą przesłaniać metody
7:  * getUsersPassword() oraz getUsersRoles(), które będą odpowiednio zwracać
8:  * oczekiwaną nazwę użytkownika oraz hasło.
9:  */
10: public abstract class UsernamePasswordLoginModule
11:     extends AbstractServerLoginModule
12: {
13:     /** Tożsamość używana podczas logowania */
14:     private Principal identity;
15:     /** Potwierdzenie tej tożsamości */
16:     private char[] credential;
17:     /** Tożsamość, jakiej należy użyć w przypadku przekazania
18:      *  pustej nazwy użytkownika i hasła */
19:     private Principal unauthenticatedIdentity;
20:
21:     /**
22:      * Algorytm używany do zahaszowania hasła; jeśli zostania użyta wartość
23:      * null, to hasła będą przekazywane w niezmienionej postaci */
24:     private String hashAlgorithm = null;
25:
26:     /**
27:      * Nazwa zbioru znaków (lub sposobu kodowania), jaki należy wykorzystać
28:      * podczas konwersji hasła podanego w formie łańcucha znaków do postaci
29:      * tablicy bajtów. Domyślnie stosowane jest domyślne kodowanie używane
30:      * przez platformę systemową.
31:      */
32:     private String hashCharset = null;
33:
34:     /** Format kodowania łańcuchów znaków, jaki należy zastosować.
35:      *  Domyślnie jest to base64. */
36:     private String hashEncoding = null;
37:
38:     // ...
39:
40:     /**
41:      * Ta metoda przesłania metodę klasy bazowej poszukującej wartości
42:      * właściwości unauthenticatedIdentity. W pierwszej kolejności wywoływana
43:      * jest metoda klasy bazowej.
44:      *
45:      * @param options,
46:      * @option unauthenticatedIdentity: nazwa tożsamości, jaką należy zastosować
47:      * w przypadku podania pustej nazwy użytkownika oraz hasła.
48:      */
```

```
49:     public void initialize(Subject subject,
50:                            CallbackHandler callbackHandler,
51:                            Map sharedState,
52:                            Map options)
53:     {
54:         super.initialize(subject, callbackHandler, sharedState,
55:                          options);
56:         // Sprawdzenie opcji unauthenticatedIdentity.
57:         Object option = options.get("unauthenticatedIdentity");
58:         String name = (String) option;
59:         if (name != null) {
60:             unauthenticatedIdentity = new SimplePrincipal(name);
61:         }
62:     }
63:
64:     // ...
65:
66:     /**
67:      * Metoda pozwalająca klasom pochodnym na zmianę weryfikacji podanego
68:      * hasła na podstawie hasła oczekiwanego. Ta wersja metody sprawdza,
69:      * czy żadne z przekazanych haseł nie jest równe null, a zwraca wartość
70:      * logiczną stanowiącą wynik porównania obu przekazanych łańcuchów znaków.
71:      *
72:      * @return zwracana jest wartość true, jeśli hasło inputPassword jest
73:      * poprawne, w przeciwnym razie zwracana jest wartość false.
74:      */
75:     protected boolean validatePassword(String inputPassword,
76:                                        String expectedPassword)
77:     {
78:         if (inputPassword == null || expectedPassword == null) {
79:             return false;
80:         }
81:         return inputPassword.equals(expectedPassword);
82:     }
83:
84:     /**
85:      * Metoda pobiera oczekiwane hasło dla bieżącej nazwy użytkownika,
86:      * która jest określana przy użyciu metody getUsername(). Jest ona
87:      * wywoływana przez metodę login(), kiedy metoda CallbackHandler
88:      * zwróci już nazwę użytkownika oraz podane przez niego hasło.
89:      *
90:      * @return poprawne hasło zwrócone jako String
91:      */
92:     abstract protected String getUsersPassword()
93:         throws LoginException;
94: }
```

Decyzja, którą z klas bazowych — `AbstractServerLoginModule` bądź `UsernamePasswordLoginModule` — należy zastosować, sprowadza się do określenia, czy technologia uwierzytelniania, na potrzeby której tworzony jest nowy moduł logowania, zezwala na podawanie nazwy użytkownika oraz danych potwierdzających jego tożsamość w formie łańcuchów znaków. Jeśli dane łańcuchowe mogą być używane, to należy tworzyć moduł logowania w oparciu o klasę `UsernamePasswordLoginModule`; w pozostałych sytuacjach należy zastosować klasę `AbstractServerLoginModule`.

Czynności, jakie należy wykonać podczas tworzenia niestandardowego modułu logowania, zależą od zastosowanej abstrakcyjnej klas bazowej. Pracę nad modułem logowania, który będzie się integrować z infrastrukturą zabezpieczeń, należy rozpocząć od stworzenia klasy dziedziczącej po `AbstracServerLoginModule` lub `UsernamePasswordLoginModule`; dzięki temu można mieć pewność, że informacje o uwierzytelnionym użytkowniku zwrócone w formie obiektu `Principal` będą mogły zostać wykorzystane przez JBossSX.

W przypadku tworzenia modułu logowania dziedziczącego po `AbstractServerLoginModule` należy przesłonić następujące metody:

- `void initialize(Subject,CallbackHandler,Map,Map)` — metoda ta jest przesłaniana, jeśli trzeba przetwarzać niestandardowe opcje.
- `boolean login()` — tą metodę przesłania się w celu zaimplementowania niestandardowego sposobu uwierzytelniania. Koniecznie należy pamiętać, aby w przypadku pomyślnego zakończenia operacji logowania zmiennej `loginOk` przypisać wartość `true`, a w przeciwnym przypadku — wartość `false`.
- `Principal getIdentity()` — ta metoda jest przesłaniana w celu zwrócenia obiektu `Principal` reprezentującego uwierzytelnionego użytkownika.
- `Group[] getRoleSets()` — metoda ta jest przesłaniana, aby zwracać przynajmniej jeden obiekt `Gropu` skojarzony z nazwą `Roles` i zawierający role przypisane użytkownikowi uwierzytelnionemu podczas wykonywania metody `login()`. Drugim obiektem `Group`, który często jest zwracany przez tę metodę, jest obiekt skojarzony z nazwą `CallerPrincipal`; udostępnia on tożsamość użytkownika w aplikacji, a nie tożsamość dziedziny zabezpieczeń.

W przypadku tworzenia modułu logowania dziedziczącego po `UsernamePasswordLoginModule` przesłaniane są następujące metody:

- `void initialize(Subject, CallbackHandler, Map, Map)` — metoda ta jest przesłaniana, jeśli trzeba przetwarzać niestandardowe opcje.
- `Group[] getRoleSets()` — metoda ta jest przesłaniana, aby zwracać przynajmniej jeden obiekt `Group` skojarzony z nazwą `Roles` i zawierający role przypisane użytkownikowi uwierzytelnionemu podczas wykonywania metody `login()`. Drugim obiektem `Group`, który często jest zwracany przez tę metodę, jest obiekt skojarzony z nazwą `CallerPrincipal`; udostępnia on tożsamość użytkownika w aplikacji, a nie tożsamość dziedziny zabezpieczeń.
- `String getUserPassword()` — metoda ta jest przesłaniana w celu zwrócenia oczekiwanego hasła dla bieżącej nazwy użytkownika, którą można określić przy użyciu metody `getUsername()`. Metoda `getUserPassword()` jest wywoływana wewnątrz metody `login()` po tym, jak metoda zwrotna `callbackhandler` zwróci nazwę użytkownika oraz podane przez niego hasło.

Przykład niestandardowego modułu logowania

W tej części rozdziału zostanie stworzony przykładowy, niestandardowy moduł logowania, który dziedziczy po klasie `UsernamePasswordLoginModule` i pobiera niezbędne dane przy użyciu wyszukiwania JNDI. Moduł ten zakłada istnienie kontekstu JNDI,

którego przeszukanie zwraca hasło użytkownika. Kontekst jest przeszukiwany przy użyciu nazwy zapisanej w postaci `password/<nazwaUżytkownika>`, gdzie `<nazwaUżytkownika>` określa aktualnie uwierzytelnianego użytkownika. Drugie przeszukanie, o postaci `roles/<nazwaUżytkownika>`, zwraca role podanego użytkownika.

Kod źródłowy tego przykładu można znaleźć w materiałach dołączonych do książki w katalogu *src/main/org/jboss/chap8/ex2*. Kod źródłowy modułu `JndiUserAndPass` został przedstawiony na listingu 8.11. Należy zwrócić uwagę, iż klasa przedstawionego modułu dziedziczy po `UsernamePasswordLoginModule`, dlatego też jedynymi operacjami, jakie musi wykonać, jest pobranie hasła użytkownika oraz jego ról przy użyciu JNDI. Przedstawiony moduł logowania nie zwraca uwagi na wszelkie operacje, jakie muszą wykonywać moduły `LoginModule`.

Listing 8.11. *Niestandardowy moduł logowania JndiUserAndPass*

```
1: package org.jboss.chap8.ex2;
2:
3: import java.security.acl.Group;
4: import java.util.Map;
5: import javax.naming.InitialContext;
6: import javax.naming.NamingException;
7: import javax.security.auth.Subject;
8: import javax.security.auth.callback.CallbackHandler;
9: import javax.security.auth.login.LoginException;
10: import org.jboss.security.SimpleGroup;
11: import org.jboss.security.SimplePrincipal;
12: import org.jboss.security.auth.spi.UsernamePasswordLoginModule;
13:
14: /**
15:  * Przykład niestandardowego modułu logowania, który pobiera
16:  * hasło oraz role użytkownika przy wykorzystaniu wyszukiwania JNDI.
17:  *
18:  * @author Scott.Stark@jboss.org
19:  * @version $Revision: 1.5 $
20:  */
21: public class JndiUserAndPass
22:     extends UsernamePasswordLoginModule
23: {
24:     /** Nazwa JNDI określająca kontekst obsługujący wyszukiwanie password/username 25: */
26:     private String userPathPrefix;
27:     /** Nazwa JNDI określająca kontekst obsługujący wyszukiwanie roles/username */
28:     private String rolesPathPrefix;
29:
30:     /**
31:      * Tę metodę należy przesłonić w celu pobrania wartości opcji userPathPrefix
32:      * oraz rolesPathPrefix
33:      */
34:     public void initialize(Subject subject, CallbackHandler callbackHandler,
35:                            Map sharedState, Map options)
36:     {
37:         super.initialize(subject, callbackHandler, sharedState, options);
38:         userPathPrefix = (String) options.get("userPathPrefix");
39:         rolesPathPrefix = (String) options.get("rolesPathPrefix");
40:     }
41:
```

```
42:     /**
43:      * Metoda pobiera role, do jakich należy bieżący użytkownik, poprzez
44:      * wyszukanie w kontekście JNDI łańcucha
45:      * rolesPathPrefix + '/' + super.getUsername().
46:      */
47:     protected Group[] getRoleSets() throws LoginException
48:     {
49:         try {
50:             InitialContext ctx = new InitialContext();
51:             String rolesPath = rolesPathPrefix + '/' + super.getUsername();
52:             String[] roles = (String[]) ctx.lookup(rolesPath);
53:             Group[] groups = {new SimpleGroup("Roles")};
54:             log.info("Pobieranie ról dla użytkownika="+super.getUsername());
55:             for(int r = 0; r < roles.length; r ++) {
56:                 SimplePrincipal role = new SimplePrincipal(roles[r]);
57:                 log.info("Odnaleziono rolę="+roles[r]);
58:                 groups[0].addMember(role);
59:             }
60:             return groups;
61:         } catch(NamingException e) {
62:             log.error("Nie udało się pobrać grup dla użytkownika="+super.
➥getUsername(), e);
63:             throw new LoginException(e.toString(true));
64:         }
65:     }
66:
67:     /**
68:      * Metoda pobiera hasło bieżącego użytkownika poprzez
69:      * wyszukanie w kontekście JNDI łańcucha
70:      * userPathPrefix + '/' + super.getUsername().
71:      */
72:
73:     protected String getUsersPassword()
74:         throws LoginException
75:     {
76:         try {
77:             InitialContext ctx = new InitialContext();
78:             String userPath = userPathPrefix + '/' + super.getUsername();
79:             log.info("Pobieranie hasła użytkownika="+super.getUsername());
80:             String passwd = (String) ctx.lookup(userPath);
81:             log.info("Odnaleziono hasło="+passwd);
82:             return passwd;
83:         } catch(NamingException e) {
84:             log.error("Nie udało się odczytać hasła dla użytkownika="+super.
➥getUsername(), e);
85:             throw new LoginException(e.toString(true));
86:         }
87:     }
88: }
```

Szczegółowe informacje na temat magazynu JNDI można znaleźć w komponencie MBean `org.jboss.chap8.ex2.service.JndiStore`. Ta usługa wiąże obiekt `Object-Factory` zwracający pośrednika `javax.naming.Context` z JNDI. Pośrednik ten obsługuje przeszukiwanie, dodając do poszukiwanej nazwy użytkownika słowa `password` lub `roles`. Jeśli łańcuch używany podczas przeszukiwania rozpoczyna się od słowa

password, to poszukiwane będzie hasło użytkownika; z kolei, jeśli zaczyna się on od słowa roles — poszukiwane będą role, do jakich należy użytkownik. Przedstawiona przykładowa implementacja modułu logowania zawsze, niezależnie od podanej nazwy użytkownika, zwraca hasło theduke oraz tablicę nazw ról o postaci {"TheDuke", "Echo"}. Oczywiście Czytelnik może wypróbować także inne implementacje modułu.

Kod przykładu zawiera także prosty komponent sesyjny służący do testowania przedstawionego powyżej modułu logowania. Aby przygotować, wdrożyć i uruchomić ten przykład, należy przejść do katalogu zawierającego materiały dołączone do książki i wykonać następujące polecenie:

```
C:\JBoss\przyklady>ant -Dchap=chap8 -Dex=2 run-example
...
run-example2:
     [copy] Copying 1 file to /tmp/jboss-4.0.1/server/default/deploy
     [echo] Waiting for 5 seconds for deploy...
     [java] [INFO,ExClient] Logowanie z użyciem danych: nazwa=jduke, hasło=theduke
     [java] [INFO,ExClient] Poszukiwanie EchoBean2
     [java] [INFO,ExClient] Utworzono Echo
     [java] [INFO,ExClient] Echo.echo('Witamy') = Witamy


19:06:13,266 INFO  [EjbModule] Deploying EchoBean2
19:06:13,482 INFO  [JndiStore] Start, bound security/store
19:06:13,486 INFO  [SecurityConfig] Using JAAS AuthConfig:
jar:file:/private/tmp/jboss-4.0.1/
server/default/tmp/deploy/tmp23012chap8-ex2.jar-contents/chap8-ex2.sar!/META-INF/
➥login-config.xml
19:06:13,654 INFO  [EJBDeployer] Deployed: file:/private/tmp/jboss-4.0.1/server/
➥default/deploy/chap8-ex2.jar
```

To, czy do uwierzytelniania użytkowników po stronie serwera będzie wykorzystywany niestandardowy moduł logowania JndiUserAndPass, jest określane przez konfigurację logowania przykładowej domeny bezpieczeństwa. Domena bezpieczeństwa jest definiowana w pliku JAR *META-INF/jboss.xml* w następujący sposób:

```
<?xml version="1.0"?>
<jboss>
    <security-domain>java:/jaas/chap8-ex2</security-domain>
</jboss>
```

Deskryptor pliku SER *META-INF/login-config.xml* definiuje konfigurację modułu logowania w następujący sposób:

```
<application-policy name = "chap8-ex2">
    <authentication>
        <login-module code="org.jboss.chap8.ex2.JndiUserAndPass"
                      flag="required">
            <module-option name = "userPathPrefix">/security/store/password</
➥module-option>
            <module-option name = "rolesPathPrefix">/security/store/roles</module-
➥option>
        </login-module>
    </authentication>
</application-policy>
```

Usługa DynamicLoginConfig

Domeny bezpieczeństwa zdefiniowane w pliku *login-config.xml* są w zasadzie statyczne. Informacje o nich są odczytywane podczas uruchamiania serwera JBoss, jednak nie ma żadnej możliwości późniejszego dodania nowej domeny bezpieczeństwa lub zmodyfikowania informacji o domenie już zdefiniowanej. Na dynamiczne wdrażanie domen bezpieczeństwa pozwala jednak usługa `DynamicLoginConfig`. Dzięki temu możliwe jest określenie konfiguracji logowania JAAS jako jednego z elementów wdrożenia (bądź też jako niezależnej usługi) i nie trzeba już ręcznie wprowadzać modyfikacji w pliku *login-config.xml*.

Usługa `DynamicLoginConfig` udostępnia następujące atrybuty:

- `AuthConfig` — ścieżka określająca plik konfiguracyjny logowania, który ma być używany. Domyślnie będzie to plik *login-config.xml*.
- `LoginConfigService` — nazwa usługi `XMLLoginConfig`, której należy używać do logowania. Usługa ta musi obsługiwać operację `String loadConfig(URL)` służącą do wczytywania konfiguracji.
- `SecurityManagerService` — nazwa usługi `SecurityMangerService` używanej do usuwania zarejestrowanych domen bezpieczeństwa. Usługa ta musi udostępniać operację `flushAuthenticationCache(String)` służącą do usunięcia informacji o domenie bezpieczeństwa o podanej nazwie. Wykonanie tej operacji spowoduje, że po zatrzymaniu usługi pamięci podręczne używane do uwierzytelniania zostaną wyczyszczone.

Poniżej przedstawiony został przykład definicji komponentu MBean korzystającego z usługi `DynamicLoginConfig`:

```
<server>
    <mbean code="org.jboss.security.auth.login.DynamicLoginConfig" name="...">
        <attribute name="AuthConfig">login-config.xml</attribute>

        <!-- Usługa obsługująca dynamiczne przetwarzanie plików
             konfiguracyjnych login-config.xml
        -->
        <depends optional-attribute-name="LoginConfigService">
            jboss.security:service=XMLLoginConfig </depends>

        <!-- Opcjonalnie można także określić usługę menedżera bezpieczeństwa,
         która zostanie użyta po zatrzymaniu tej usługi do opróżniania
         pamięci podręcznych przechowujących informacje uwierzytelniające
         w domentach zarejestrowanych w tej usłudze
        -->
        <depends optional-attribute-name="SecurityManagerService">
            jboss.security:service=JaasSecurityManager </depends>
    </mbean>
</server>
```

Powyższa definicja spowoduje wczytanie zasobu określonego w atrybucie `AuthConfig` przy użyciu określonego komponentu MBean `LoginConfigService`, który wywoła metodę `loadConfig()` i przekaże do niej odpowiedni adres URL zasobu. Po zatrzymaniu usługi, konfiguracja zostanie usunięta. Zasób określany w atrybucie `AuthConfig` może być bądź to plikiem XML, bądź też konfiguracją logowania JAAS.

Protokół Secure Remote Password (SRP)

Protokół SRP stanowi implementację sposobu wymiany kluczy publicznych opisanego w pliku RFC 2945. Poniżej przedstawiony został wyciąg z tego dokumentu:

> Ten dokument opisuje mocny pod względem kryptograficznym mechanizm uwierzytelniania sieciowego nazywany protokołem Secure Remote Password (SRP). Mechanizm ten nadaje się do negocjowania bezpiecznych połączeń sieciowych przy wykorzystaniu hasła podanego przez użytkownika, a jednocześnie eliminuje tradycyjne problemy związane ze stosowaniem haseł używanych wielokrotnie. Prezentowany system w ramach procesu uwierzytelniania przeprowadza także bezpieczną wymianę kluczy, dzięki czemu warstwy bezpieczeństwa (zabezpieczenia prywatności oraz integralności) mogą być włączone w czasie trwania sesji. Korzystanie z prezentowanego mechanizmu nie stwarza konieczności wykorzystywania zaufanych serwerów kluczy ani infrastruktury zarządzającej certyfikatami, a klienty nie muszą ani przechowywać, ani zarządzać kluczami używanymi przez dłuższy czas. W porównaniu z istniejącymi technikami typu wyzwanie-odpowiedź protokół SRP wykazuje wiele zalet, i to zarówno pod względem bezpieczeństwa, jak i stosowania. Dzięki temu doskonale się on nadaje do zastosowania w rozwiązaniach wymagających bezpiecznego uwierzytelniania przy użyciu hasła.

Kompletną specyfikację zawartą w pliku RFC 2945 można znaleźć na witrynie *www.rfc-editor.org/rfc.html*. Dodatkowe informacje dotyczące algorytmu SRP oraz jego historii są dostępne na witrynie *http://srp.stanford.edu/*.

Pod względem ogólnego rozwiązania oraz zapewnianego bezpieczeństwa protokół SRP przypomina inne algorytmy wymiany kluczy publicznych, takie jak algorytm Diffiego-Hellmana lub RSA. Protokół SRP bazuje na prostych hasłach tekstowych, jednak nie stwarza konieczności przechowywania na serwerze haseł zapisanych w niezaszyfrowanej postaci tekstowej. Pod tym względem różni się on od innych algorytmów wymiany klucza publicznego, które wymagają zastosowania certyfikatów klienta oraz niezbędnej infrastruktury zarządzania tymi certyfikatami.

Algorytmy Diffiego-Hellmana lub RSA nazywane są *algorytmami wymiany klucza publicznego*. Pomysł działania takich algorytmów opiera się założeniu, że istnieją dwa klucze, z których jeden jest publiczny i dostępny dla wszystkich, a drugi prywatny

i dostępny jedynie dla nas. Kiedy ktoś chce wysłać do drugiej osoby zaszyfrowaną informację, szyfruje ją przy użyciu klucza publicznego adresata. Tylko adresat będzie w stanie odszyfrować tę informację, posługując się swoim kluczem prywatnym. Rozwiązanie to różni się od stosowanych wcześniej algorytmów szyfrowania wykorzystujących klucz wspólny, które wymagały, by nadawca wiadomości oraz jej odbiorca posługiwali się tym samym, wspólnym kluczem. Algorytmy klucza publicznego eliminują konieczność stosowania wspólnych haseł.

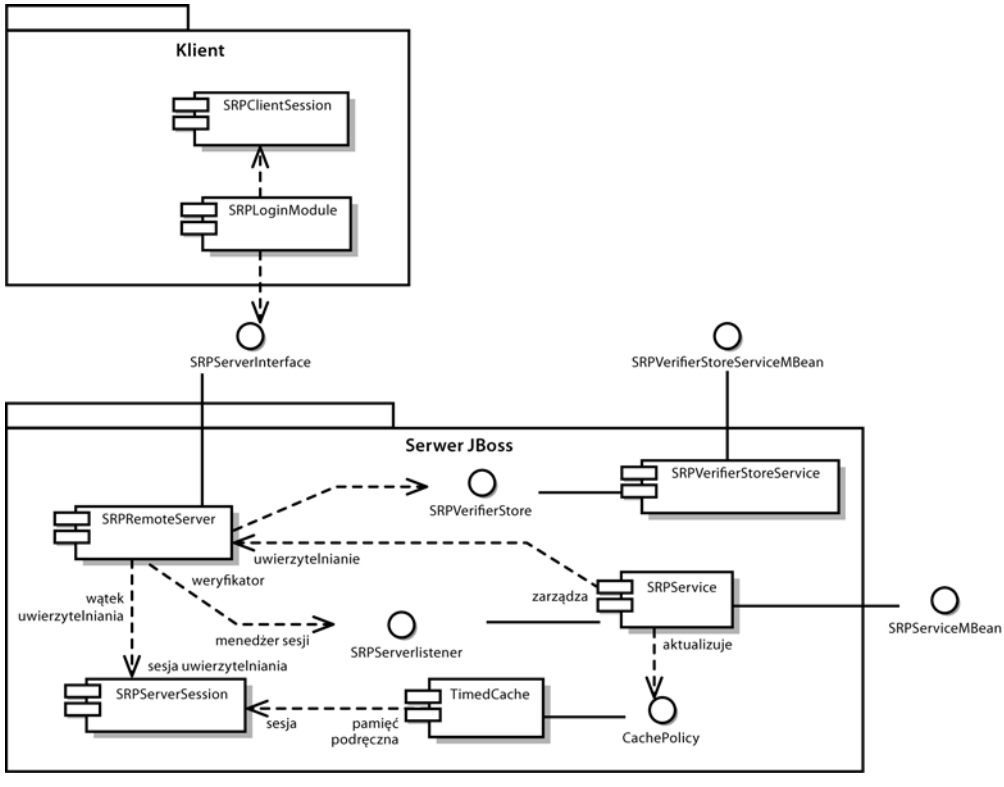
Podsystem JBossSX zawiera implementacje protokołu SRP składającą się z następujących elementów:

- Implementacji protokołu SRP z potwierdzeniem, która jest niezależna od jakichkolwiek innych protokołów klient-serwer.
- Implementacji protokołu z potwierdzeniem wykorzystującej technologię RMI i stanowiącej domyślną implementację protokołu SRM.
- Modułu logowania JAAS `LoginModule` wykorzystującego implementację RMI protokołu i służącego do bezpiecznego uwierzytelniania klientów.
- Komponentu JMX MBean służącego do zarządzania implementacją serwera RMI. Komponent ten pozwala, by obiekt stanowiący implementację serwera RMI został włączony do szkieletu JMX i umożliwia zewnętrzną konfigurację magazynu informacji używanych podczas weryfikacji. Dodatkowo komponent ten tworzy pamięć podręczną przeznaczoną na informacje uwierzytelniające powiązaną z przestrzenią nazw JNDI JBossa.
- Działającej po stronie serwera implementacji modułu logowania JAAS `LoginModule`, która używa pamięci podręcznej z danymi uwierzytelniającymi zarządzanymi przez komponent SRP JMX MBean.

Rysunek 8.14 przedstawia diagram kluczowych komponentów należących do implementacji protokołu SRP w podsystemie JBossSX.

Po stronie klienta SRP jest widoczny jako implementacja niestandardowego modułu logowania JAAS `LoginModule`, która komunikuje się z serwerem uwierzytelniającym przy użyciu pośrednika `org.jboss.security.srp.SRPServerInterface`. Klient uaktywnia uwierzytelnianie przy wykorzystaniu protokołu SRP poprzez utworzenie w konfiguracji wpisu zawierającego `org.jboss.security.srp.jaas.SRPLoginModule`. Moduł ten obsługuje następujące opcje konfiguracyjne:

- `principalClassName` — ta opcja nie jest już obsługiwana. Obecnie zawsze jest używana klasa `org.jboss.security.srp.jaas.SRPPrincipal`.
- `srpServerJndiName` — ta opcja określa nazwę JNDI obiektu `SRPServerInterface`, który ma być używany do prowadzenia komunikacji z serwerem uwierzytelniającym SRP. Jeśli podano zarówno opcję `srpServerJndiName`, jak i opcję `srpServerRmiUrl`, to w pierwszej kolejności zostanie wypróbowana pierwsza z nich.
- `srpServerRmiUrl` — ta opcja określa adres URL protokołu RMI określający położenie pośrednika `SRPServerInterface`, którego należy używać podczas prowadzenia komunikacji z serwerem uwierzytelniającym SRP.



Rysunek 8.14. *Komponenty implementacji protokołu SRP stosowane po stronie klienta oraz serwera*

- `externalRandomA` — ta opcja jest flagą przyjmującą wartości `true` lub `false`, która określa, czy metoda zwrotna klienta powinna zwracać losowo wybrany fragment klucza publicznego A klienta. Metody tej można używać do podawania mocnej pod względem kryptograficznym liczby losowej odczytywanej, na przykład, z jakiegoś urządzenia.
- `hasAuxChallenge` — ta opcja jest flagą przyjmująca wartości `true` lub `false`, która określa, czy na serwer będzie przesyłane dodatkowe wyzwanie zawierające łańcuch znaków, które ten powinien zweryfikować. Jeśli sesja klienta obsługuje szyfrowanie, to na podstawie prywatnego klucza sesji oraz łańcucha przesłanego jako obiekt `javax.crypto.SealedObject` zostanie utworzony szyfr tymczasowy.
- `multipleSessions` — ta opcja jest flagą przyjmującą wartości `true` lub `false`, która wskazuje, czy dany klient może dysponować kilkoma równocześnie otwartymi sesjami logowani SRP.

Wszelkie dodatkowe przekazane opcje, które nie odpowiadają powyższym, będą traktowane jako opcje JNDI używane przez środowisko przekazywane do konstruktora klasy `InitialContext`. Możliwość ta jest przydatna w sytuacjach, gdy domyślny obiekt `InitialContext` nie jest w stanie zapewnić dostępu do interfejsu serwera SRP.

Aby informacje uwierzytelniające używane przez protokół SRP mogły być zastosowane w celu określenia możliwości dostępu do komponentów J2EE związanych z systemem bezpieczeństwa, moduł `SRPLoginModule` musi być używany razem ze standardowym modułem logowania `ClientLoginModule`. Poniżej przedstawiony został fragment wpisu konfigurującego logowanie, który zapewnia wspólne wykorzystanie obu tych modułów logowania:

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

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

Po stronie serwera JBoss istnieją dwa komponenty MBean zarządzające obiektami, które wspólnie stanowią serwer SRP. Są to komponenty: `org.jboss.security.srp.SRPService` oraz `org.jboss.security.srp.SRPVerifierStoreService`. Podstawową usługą jest pierwszy z tych komponentów i to właśnie on odpowiada za udostępnienie implementacji interfejsu `SRPServerInterface`, z której można korzystać za pośrednictwem RMI, oraz za aktualizację pamięci podręcznej sesji uwierzytelniania SRP. Poniżej przedstawione zostały atrybuty komponentu `SRPService`, których wartości można określać:

- `JndiName` — określa nazwę, pod którą powinien być dostępny pośrednik `SRPServerInterface`. To właśnie w tym miejscu dynamiczny pośrednik jest wiązany z obiektem implementującym interfejs `SRPServerInterface`. W razie braku jawnego określenia wartości tego atrybutu zostanie użyta wartość domyślna `srp/SRPServerVerifierSource`.
- `VerifierSourceJndiName` — określa nazwę JNDI implementacji `SRPVerifierSource`, która powinna być używana przez obiekt `SRPService`. Jeśli nie zostanie jawnie podany, atrybut ten przyjmie wartość `srp/DefaultVerifierSource`.
- `AuthenticationCacheJndiName` — określa nazwę JNDI, pod jaką został powiązany obiekt implementujący interfejs `org.jboss.util.CachePolicy`, który ma być używanych do przechowywania informacji uwierzytelniających. Pamięć podręczna sesji SRP będzie dostępna przy użyciu właśnie tej nazwy JNDI. Jeśli atrybut ten nie został podany jawnie, przyjmie on domyślną wartość `srp/AuthenticationCache`.
- `ServerPort` — ten atrybut określa numer portu, którego ma używać obiekt implementujący interfejs `SRPRemoteServerInterface`. Jeśli nie został on podany jawnie, domyślnie zostanie użyty port `10099`.
- `ClientSocketFactory` — ten atrybut określa nazwę opcjonalnej, niestandardowej klasy implementującej interfejs `java.rmi.server.RMIClientSocketFactory`, która ma być używana podczas eksportowania obiektu typu `SRPServerInterface`. Jeśli atrybut ten nie został podany jawnie, przyjmie domyślną wartość `RMIClientSocketFactory`.

- `ServerSocketFactory` — ten atrybut określa nazwę opcjonalnej, niestandardowej klasy implementującej interfejs `java.rmi.server.RMIServerSocketFactory`, która ma być używana podczas eksportowania obiektu typu `SRPServerInterface`. Jeśli atrybut ten nie został podany jawnie, przyjmie domyślną wartość `RMIServerSocketFactory`.
- `AuthenticationCacheTimeout` — atrybut określa wyrażony w sekundach czas przechowywania danych w buforze. Jego domyślną wartością jest `1800`.
- `AuthenticationCacheResolution` — atrybut ten określa, jak często należy przeglądać bufor w poszukiwaniu danych, których okres składowania już minął. Czas tej jest określany w sekundach, a domyślna wartość atrybutu wynosi `60`.
- `RequireAuxChallenge` — jeśli ten atrybut zostanie podany, to klient, w trakcie fazy weryfikacji, będzie przekazywać dodatkowe wyzwanie. Zapewnia on kontrolę nad tym, czy w konfiguracji modułu `SRPLoginModule` została uaktywniona opcja `useAuxChallenge`.
- `OverwriteSessions` — ten atrybut stanowi flagę, która określa, czy w przypadku istniejącej sesji udane uwierzytelnienie użytkownika powinno spowodować nadpisanie bieżącej sesji. Atrybut ten kontroluje działanie pamięci podręcznej serwera SRP w przypadkach, gdy klient nie uaktywnił obsługi wielu sesji. Domyślna wartość tego atrybutu wynosi `false`, co oznacza, że kolejna próba uwierzytelnienia użytkownika zakończy się pomyślnie, lecz nowa sesja SRP nie nadpisze stanu poprzedniej sesji.

Jedynym ustawieniem wejściowym jest atrybut `VerifierSourceJndiName`. Określa on położenie implementacji magazynu informacji o hasłach SRP, która ma być udostępniona za pośrednictwem JNDI. Przykładem zastosowania implementacji interfejsu `SRPVerifierStore` jest komponent MBean `org.jboss.security.srp.SRPVerifierStoreService`, w której role trwałego magazynu pełni plik obiektów serializowanych. Choć zastosowanie tego komponentu w środowisku produkcyjnym raczej nie jest prawdopodobne, to jednak pozwala on na testowanie protokołu SRP, jak również stanowi przykład wymagań, jakie musi spełniać usługa `SRPVerifierStore`. Poniżej przedstawione zostały jego atrybuty:

- `JndiName` — nazwa JNDI, pod którą jest dostępny obiekt implementujący interfejs `SRPVerifierStore`. Jeśli nie zostanie określony, atrybut ten przyjmuje wartość domyślną `srp/DefaultVerifierSource`.
- `StoreFile` — położenie pliku obiektów serializowanych pełniącego rolę trwałego magazynu haseł używanych podczas weryfikacji. Może to być zarówno adres URL, jak i nazwa zasobu. Jeśli nie zostanie podany, atrybut ten przyjmuje domyślną wartość `SRPVerifierStore.ser`.

Komponent `SRPVerfierStoreService` obsługuje także operacje o nazwach `addUser` oraz `delUser` służące odpowiednio do dodawania nowych użytkowników oraz usuwania użytkowników już zapisanych. Poniżej przedstawione zostały sygnatury tych metod:

```
public void addUser(String nazwaUzytkownika, String haslo)
     throws IOException;
public void delUser(String nazwaUzytkownika)
     throws IOException;
```

Udostępnianie informacji o hasłach

Najprawdopodobniej domyślna implementacja interfejsu `SRPVerfierStore` nie będzie się nadawać do zastosowania w produkcyjnym środowisku bezpieczeństwa, gdyż wymaga, by wszystkie informacje o hasłach użytkowników były dostępne w postaci pliku serializowanych obiektów. A zatem konieczne będzie stworzenie usługi MBean udostępniającej implementację interfejsu `SRPVerifierStore`, która będzie w stanie współpracować z istniejącymi magazynami informacji używanych przez system bezpieczeństwa. Kod źródłowy interfejsu `SRPVerifierStore` został przedstawiony na listingu 8.12.

Listing 8.12. *Interfejs SRPVerifierStore*

```
1: package org.jboss.security.srp;
2:
3: import java.io.IOException;
4: import java.io.Serializable;
5: import java.security.KeyException;
6:
7: public interface SRPVerifierStore
8: {
9:     public static class VerifierInfo implements Serializable
10:     {
11:         /**
12:          * Nazwa użytkownika, którego dotyczą informacje. Zapewne jest
13:          * to informacja nadmiarowa, jednak sprawia, że obiekt jest
14:          * spójną całością niezależną od informacji zewnętrznych.
15:          */
16:         public String username;
17:
18:         /** Skrót wykorzystywany przez SRP do weryfikacji hasła */
19:         public byte[] verifier;
20:         /** Losowa dana inicjalizacyjna użyta początkowo do weryfikacji hasła */
21:         public byte[] salt;
22:         /** Tablica wartości podstawowych algorytmu SRP */
23:         public byte[] g;
24:         /** Moduł z bezpiecznej liczby pierwszej */
25:         public byte[] N;
26:     }
27:
28:     /**
29:      * Metoda zwraca informacje używane do weryfikacji hasła
30:      * dla podanego użytkownika.
31:      */
32:     public VerifierInfo getUserVerifier(String username)
33:         throws KeyException, IOException;
34:     /**
35:      * Metoda zapisuje informacje używane podczas weryfikacji hasła
36:      * dla wskazanego użytkownika. Operacja ta odpowiada zmianie hasła
37:      * użytkownika i zazwyczaj powinna spowodować unieważnienie
```

```
38:       *  wszystkich istniejących sesji SRP tego użytkownika oraz
39:       *  dotyczących go informacji przechowywanych w pamięci podręcznej.
40:       */
41:      public void setUserVerifier(String username, VerifierInfo info)
42:          throws IOException;
43:
44:      /**
45:       * Metoda weryfikuje opcjonalne dodatkowe wyzwanie przesłane przez
46:       * klienta na serwer. Jeśli obiekt auxCallenge zostanie przesłany
47:       * przez klienta w formie zaszyfrowanej, to po odebraniu go na
48:       * serwerze zostanie on odszyfrowany. Przykładem takiego dodatkowego
49:       * wyzwania byłaby weryfikacja urządzenia sprzętowego (SafeWord, SecureID
50:       * lub iButton), które serwer weryfikuje w celu jeszcze większego
51:       * podniesienia bezpieczeństwa wymiany kluczy realizowanej przez
52:       * protokół SRP.
53:       */
54:       public void verifyUserChallenge(String username, Object auxChallenge)
55:           throws SecurityException;
56: }
```

Podstawowym zadaniem klasy implementującej interfejs `SRPVerifierStore` jest zapewnienie dostępu do obiektu `SRPVerifierStore.VerifierInfo` odpowiadającego podanej nazwie użytkownika. Na początku sesji SRP obiekt `SRPService` wywołuje metodę `getUserVerifier(String)` w celu pobrania parametrów niezbędnych do działania algorytmu wykorzystywanego przez protokół SRP. Poniżej opisane zostały elementy składowe `VerifierInfo`:

- `username` — nazwa użytkownika lub identyfikator używany podczas logowania.
- `verifier` — skrót (niemożliwy do odszyfrowania) hasła lub numeru identyfikacyjnego, który użytkownik podaje w celu potwierdzenia swojej tożsamości. Klasa `org.jboss.security.Util` dysponuje metodą `calculateVerifier`, która obsługuje wyznaczenie tego skrótu. Zwracane hasło jest wyznaczane zgodnie ze wzorem `H(salt | H(username | ':' | password))` opisanym w dokumencie RFC 2945. W tym przypadku `H` jest funkcją wyznaczającą bezpieczny skrót SHA. Nazwa użytkownika jest przekształcana z łańcucha znaków na tablicę bajtów przy wykorzystaniu kodowania UTF-8.
- `salt` — liczba losowa służąca do zwiększenia trudności przeprowadzenia ataku słownikowego metodą siły brutalnej. Atak taki można przeprowadzić w celu złamania bezpiecznych haseł w przypadku udanego włamania do bazy danych. Wartość ta powinna być generowana w momencie wyznaczania skrótu hasła podanego przez użytkownika przy użyciu algorytmu generującego liczby pseudolosowe w sposób mocny pod względem kryptograficznym.
- `g` — algorytm SRP pełniący rolę generatora wartości podstawowych. Ogólnie rzecz biorąc, wartość tej składowej nie musi być określana niezależnie dla każdego użytkownika, lecz być powszechnie znana. Klasa pomocnicza `org.jboss.security.srp.SRPConf` posiada kilka ustawień tej składowej, w tym także dobrą wartość domyślną, którą można pobrać przy użyciu metody `SRPConf.getDefaultParams().g()`.

- `N` — moduł bezpiecznej liczby pierwszej używany przez algorytm SRP. Ogólnie rzecz biorąc, wartość tej składowej nie musi być określana niezależnie dla każdego użytkownika, lecz być powszechnie znana. Klasa pomocnicza `org.jboss.security.srp.SRPConf` posiada kilka ustawień tej składowej, w tym także dobrą wartość domyślną, którą można pobrać przy użyciu metody `SRPConf.getDefaultParams().N()`.

Poniżej przedstawione zostały czynności, jakie należy wykonać w celu zintegrowania protokołu SRP z istniejącym magazynem haseł:

1. Utworzyć skróty haseł. Jeśli hasła już są przechowywane w formie skrótów, których nie można doprowadzić do postaci oryginalnej, to czynność tę można wykonać jedynie dla wybranych użytkowników (na przykład jako etap procedury aktualizacji danych). Należy zauważyć, że metoda `setUserVerfier(String, VerifierInfo)` nie jest używana przez aktualną wersję klasy `SRPService` i może być zaimplementowana jako metoda, która nie wykonuje żadnych operacji, a nawet może zgłaszać wyjątek informujący o tym, iż magazyn haseł jest przeznaczony wyłącznie do odczytu.
2. Stworzyć własną implementację interfejsu `SRPVerifierStore`, która będzie w stanie pobierać informacje z magazynu utworzonego w kroku 1. Metoda `verifyUserChallenge(String, Object)` tego interfejsu jest wywoływana wyłącznie w przypadku, gdy w konfiguracji modułu `SRPLoginModule` działającego po stronie klienta został podany atrybut `hasAuxChallenge`. Dzięki tej implementacji możliwe jest zintegrowanie z algorytmem SRP istniejących już sposobów uwierzytelniania bazujących na urządzeniach sprzętowych.
3. Stworzyć komponent MBean, który zapewni, że zarówno obiekt klasy napisanej w poprzednim kroku, jak i wszelkie konfigurowalne parametry będą dostępne za pośrednictwem JNDI. Oprócz standardowego przykładu klasy `org.jboss.security.SRPVerifierStoreService` przykład wykorzystania protokołu SRP przedstawiony w dalszej części rozdziału pokazuje implementację klasy `SRPVerifierStore` korzystającej z pliku właściwości. Oba te przykłady powinny dostarczyć Czytelnikowi dostatecznie dużo informacji, aby był w stanie zintegrować protokół SRP z istniejącym magazynem bezpieczeństwa.

Szczegóły działania algorytmu SRP

Atrakcyjność algorytmu SRP polega na tym, iż pozwala on na wzajemne uwierzytelnienie klienta i serwera przy użyciu zwyczajnych haseł tekstowych i to bez konieczności stosowania bezpiecznych kanałów komunikacyjnych. Można się zastanawiać, jak to jest możliwe. Jeśli Czytelnik jest zainteresowany szczegółami działania oraz teorią, na jakiej bazuje ten algorytm, to wszelkie informacje na ten temat można znaleźć na witrynie *http://srp.stanford.edu*. Uwierzytelnianie jest realizowane w sześciu krokach:

1. Działający po stronie klienta moduł logowania `SRPLoginModule` pobiera, przy użyciu usługi nazewniczej, instancję implementującą interfejs `SRPServerInterface` i reprezentującą zdalny serwer obsługujący uwierzytelnianie.

2. Moduł logowania działający po stronie klienta prosi o przekazanie parametrów SRP skojarzonych z użytkownikiem, którego próbuje uwierzytelnić. Za pierwszym razem, gdy użytkownik poda hasło, gdy jest ono przekształcane do sprawdzonej postaci używanej przez algorytm SRP, konieczne jest podanie kilku parametrów wykorzystywanych podczas jego działania. Parametrów tych nie trzeba podawać na stałe (co można by zrobić, i to bez narażania się na poważne obniżenie bezpieczeństwa protokołu) — implementacja podsystemu JBossSX pozwala na ich pobieranie w ramach protokołu wymiany informacji; służy do tego metoda `getSRPParameters(nazwaUzytkownika)`.

3. Moduł `SRPLoginModule` działający po stronie klienta rozpoczyna sesję SRP. W tym celu tworzony jest obiekt `SRPClientSession`, przy czym podczas jego tworzenia zostaje użyta nazwa użytkownika, jego hasło oraz parametry SRP pobrane w poprzednim kroku. Następnie klient tworzy losową liczbę A, która zostanie użyta do utworzenia klucza prywatnego sesji SRP. Kolejną czynnością, jaką wykonuje klient, jest zainicjowanie operacji wykonywanych po stronie serwera. W tym celu wywoływana jest metoda `SRPServerInterface.init`, do której zostaje przekazana nazwa użytkownika oraz wygenerowana przez klienta losowa liczba A. Następnie serwer generuje swoją liczbę losową — B — i przekazuje ją klientowi. Czynności te odpowiadają wymianie kluczy publicznych.

4. Moduł logowania działający po stronie klienta pobiera prywatny klucz sesji SRP, który został wygenerowany w wyniku wykonania czynności opisanych w poprzednim kroku. Zostaje on zapisany w obiekcie `Subject` jako prywatne informacje uwierzytelniające. Odpowiedź `M2` serwera wygenerowana w kroku 4. jest weryfikowana poprzez wywołanie metody `SRPClientSession.verify`. Jeśli jej wywołanie zakończy się pomyślnie, będzie to oznaczać, że wzajemne uwierzytelnienie klienta przez serwer oraz serwera przez klienta zostało zakończone. Następnie moduł `SRPLoginModule` działający po stronie serwera tworzy wyzwanie `M1`, wywołując w tym celu metodę `SRPClientSession.response` i przekazując do niej losową liczbę B (wygenerowaną przez serwer). Wyzwanie to jest przekazywane na serwer przy użyciu metody `SRPServerInterface.verify`, a odpowiedź przesłana z serwera jest zapisywana jako `M2`. Ten krok odpowiada operacji przekazania wyzwań. W tym momencie serwer ma już pewność, że użytkownik jest tym, za kogo się podaje.

5. Moduł logowania `SRPLoginModule` działający po stronie klienta zapisuje nazwę użytkownika oraz wyzwanie `M1` w mapie `sharedState` (określanej w wywołaniu metody `LoginModule.initialize`). Standardowa implementacja klasy `ClientLoginModule` serwera JBoss używa tej informacji jako nazwy tożsamości oraz danych uwierzytelniających. Wyzwanie to jest używane zamiast hasła jako potwierdzenie tożsamości podczas wywoływania wszelkich metod komponentów J2EE. `M1` jest mocnym pod względem kryptograficznym skrótem skojarzonym z sesją SRP. Jego przechwycenie nie może być wykorzystane do odczytania hasła użytkownika.

6. Na samym końcu procesu uwierzytelniania obiekt `SRPServerSession` jest umieszczany w pamięci podręcznej obiektu `SRPService` w celu późniejszego wykorzystania przez moduł logowania `SRPCacheLoginModule`.

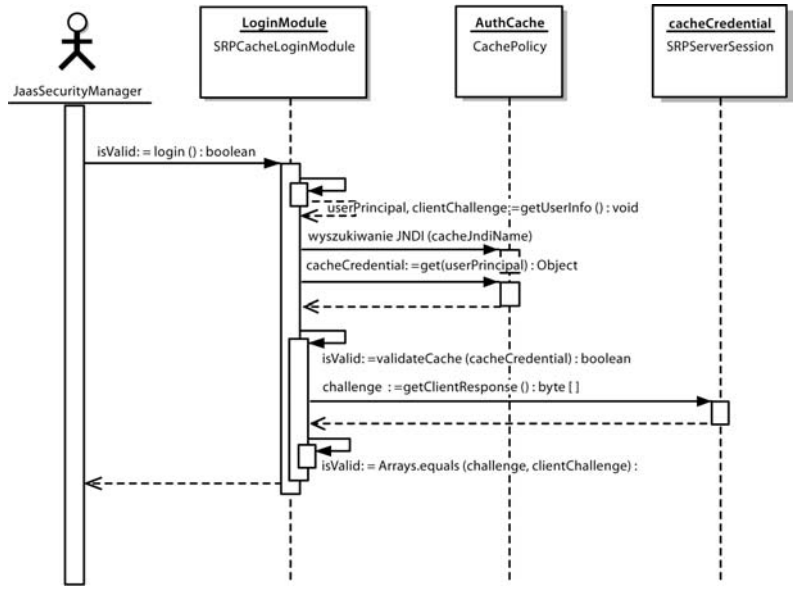
Choć protokół SRP ma wiele bardzo interesujących cech, niemniej jednak wciąż stanowi on rozwijający się element infrastruktury JBossSX i posiada kilka ograniczeń, o których należy wiedzieć. Oto one:

- Ze względu na sposób, w jaki JBoss odłącza protokół transportu metod od kontenera komponentu, który dokonuje uwierzytelniania, nieuwierzytelniony użytkownik może przechwycić wyzwanie `M1` i z powodzeniem zastosować je do przesyłania żądań, podszywając się pod użytkownika o podanej nazwie. Problem ten można rozwiązać, tworząc niestandardowe obiekty przechwytujące, które będą szyfrować wyzwanie przy użyciu klucza sesji SRP.
- Obiekt `SRPService` przechowuje informacje o sesjach SRP w pamięci podręcznej, której ważność wygasa po upłynięciu określonego czasu. Kiedy ważność sesji upłynie, każde zgłoszone potem odwołanie do komponentu J2EE zakończy się niepowodzeniem, gdyż obecnie nie ma jeszcze żadnego, niezauważalnego dla użytkownika, sposobu ponownego negocjowania informacji uwierzytelniających używanych przez protokół SRP. A zatem konieczne jest ustawienie bardzo długiego czasu ważności pamięci podręcznej (maksymalna wartość wynosi 2 147 483 647 sekund, czyli około 68 lat), bądź to, w razie niepowodzenia, obsługiwanie uwierzytelniania we własnym kodzie.
- Domyślnie dla każdej nazwy użytkownika może istnieć tylko jedna sesja SRP. Ponieważ uzgodniona sesja SRP generuje klucz prywatny, który może być używany do szyfrowania i deszyfrowania danych przekazywanych pomiędzy klientem a serwerem, zatem sesja w zasadzie zachowuje stan. JBoss obsługuje wiele jednoczesnych sesji tego samego klienta z serwerem, jednak nie można szyfrować danych przy użyciu klucza jednej sesji i deszyfrować ich przy użyciu klucza innej sesji.

Aby stosować pełne uwierzytelnianie wywołań komponentów J2EE, należy skonfigurować domenę bezpieczeństwa odpowiadającą za ochronę komponentów w taki sposób, by korzystała z modułu logowania `org.jboss.security.srp.jaas.SRPCacheLoginModule`. Moduł ten posiada tylko jedną opcję konfiguracyjną — `cacheJndiName` — określającą nazwę JNDI obiektu implementującego interfejs `CachePolicy`. Nazwa ta musi odpowiadać wartości atrybutu `AuthenticationCacheJndiName` komponentu MBean `SRPService`. Moduł `SRPCacheLoginModule` uwierzytelnia dane użytkowników, pobierając z obiektu `SRPServerSession` (przechowywanego w pamięci podręcznej) wyzwanie klienta i porównując je z wyzwaniem przesłanym jako element informacji uwierzytelniających użytkownika. Operacje wykonywane przez metodę `SRPCacheLoginModule.login` zostały przedstawione na rysunku 8.15.

Przykład SRP

W niniejszym rozdziale przedstawionych zostało stosunkowo dużo informacji na temat protokołu SRP, nadszedł więc czas, by przedstawić przykład demonstrujący jego zastosowanie. Zamieszczony przykład prezentuje autoryzację użytkowników przeprowadzaną po stronie klienta przy wykorzystaniu SRP, jak również późniejsze korzystanie z prostego komponentu EJB, podczas którego rolę informacji uwierzytelniających użytkownika pełni wyzwanie sesji SRP. Kod testowy realizuje wdrożenie pliku JAR komponentu EJB zawierającego archiwum SAR służące do skonfigurowania modułu



Rysunek 8.15. *Diagram sekwencji ilustrujący interakcję pomiędzy obiektem SRPCacheLoginModule oraz pamięcią podręczną sesji SRP*

logowania działającego po stronie serwera oraz usług SRP. Podobnie jak w poprzednich przykładach także i w tym konfiguracja modułu logowania działającego po stronie serwera zostanie zainstalowana dynamicznie, przy użyciu komponentu MBean `SecurityConfig`. Przedstawiony przykład wykorzystuje także niestandardową implementację interfejsu `SRPVerifierStore`, która posługuje się magazynem przechowywanym w pamięci i inicjowanym przez dane zapisane w pliku właściwości, a nie w magazynie obiektów serializowanych (jakiego używa klasa `SRPVerifierStoreService`). Ta niestandardowa klasa nosi nazwę `org.jboss.chap8.ex3.service.PropertiesVerifierStore`. Poniższa lista przedstawia zawartość pliku JAR, w którym został umieszczony przykładowy komponent EJB oraz usługi SRP:

```
C:\JBoss\przyklady>java -cp output/classes ListJar output/chap8/chap8-ex3.jar
output/chap8/chap8-ex3.jar
+- META-INF/MANIFEST.MF
+- META-INF/ejb-jar.xml
+- META-INF/jboss.xml
+- org/jboss/chap8/ex3/Echo.class
+- org/jboss/chap8/ex3/EchoBean.class
+- org/jboss/chap8/ex3/EchoHome.class
+- roles.properties
+- users.properties
+- chap8-ex3.sar (archive)
| +- META-INF/MANIFEST.MF
| +- META-INF/jboss-service.xml
| +- META-INF/login-config.xml
| +- org/jboss/chap8/ex3/service/PropertiesVerifierStore$1.class
| +- org/jboss/chap8/ex3/service/PropertiesVerifierStore.class
| +- org/jboss/chap8/ex3/service/PropertiesVerifierStoreMBean.class
| +- org/jboss/chap8/service/SecurityConfig.class
| +- org/jboss/chap8/service/SecurityConfigMBean.class
```

Kluczowymi elementami tego przykładu, związanymi z zastosowaniem protokołu SRP, są konfiguracja komponentu MBean usługi SRP oraz konfiguracja modułu logowania SRP. Listing 8.13 przedstawia plik *jboss-service.xml* zawierający deskryptor *chap8-ex3.sar*, natomiast listingi 8.14 oraz 18.15 prezentują odpowiednio konfiguracje modułów logowania działających po stronie klienta oraz po stronie serwera.

Listing 8.13. *Deskryptor usług SRP umieszczony w pliku jboss-service.xml*

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

<server>
   <!--
       Konfiguracja niestandardowego logowania instalująca
       obiekt Configuration pozwalający na dynamiczną aktualizację
       ustawień konfiguracyjnych.
   -->
   <mbean code="org.jboss.chap8.service.SecurityConfig"
          name="jboss.docs.chap8:service=LoginConfig-EX3">
      <attribute name="AuthConfig">META-INF/login-config.xml</attribute>
      <attribute name="SecurityConfigName">jboss.security:service=
➥XMLLoginConfig</attribute>
   </mbean>

  <!-- Usługa SRP udostępniająca serwer SRP RMI oraz pamięć podręczną
       służącą do przechowywania danych uwierzytelniających działająca
       po stronie serwera. -->
  <mbean code="org.jboss.security.srp.SRPService"
 name="jboss.docs.chap8:service=SRPService">
    <attribute name="VerifierSourceJndiName">srp-test/chap8-ex3</attribute>
    <attribute name="JndiName">srp-test/SRPServerInterface</attribute>
    <attribute name="AuthenticationCacheJndiName">srp-
test/AuthenticationCache</attribute>
    <attribute name="AuthenticationCacheTimeout">10</attribute>
    <attribute name="AuthenticationCacheResolution">5</attribute>
    <attribute name="ServerPort">0</attribute>
    <depends>jboss.docs.chap8:service=PropertiesVerifierStore</depends>
  </mbean>

  <!-- Usługa obsługująca magazyn SRP, która udostępnia informacje
       weryfikatora haseł użytkowników. -->
  <mbean code="org.jboss.chap8.ex3.service.PropertiesVerifierStore"
   name="jboss.docs.chap8:service=PropertiesVerifierStore">
    <attribute name="JndiName">srp-test/chap8-ex3</attribute>
  </mbean>
</server>
```

Listing 8.14. *Standardowa konfiguracja modułu logowania działającego po stronie klienta*

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

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

Listing 8.15. *Konfiguracja modułu logowania XMLLoginConfig działającego po stronie serwera*

```
<application-policy name="chap8-ex3">
    <authentication>
        <login-module code="org.jboss.security.srp.jaas.SRPCacheLoginModule"
                      flag = "required">
            <module-option name="cacheJndiName">srp-test/AuthenticationCache</
➥module-option>
        </login-module>
        <login-module code="org.jboss.security.auth.spi.UsersRolesLoginModule"
                      flag = "required">
            <module-option name="password-stacking">useFirstPass</module-option>
        </login-module>
    </authentication>
</application-policy>
```

Przykładowymi usługami używanymi w tym przykładzie są komponenty MBean: `ServiceConfig`, `PropertiesVerifierStore` oraz `SRPService`. Warto zwrócić uwagę, iż atrybut `JndiName` komponentu `PropertiesVerifierStore` ma taką samą wartość jak ten sam atrybut komponentu `VerifierSourceJndiName` oraz że komponent `SRPService` jest zależny od komponentu `PropertiesVerifierStore`. Zależność tak jest niezbędna, gdyż komponent `SRPService` potrzebuje implementacji interfejsu `SRPVerifierStore` w celu uzyskania dostępu do informacji używanych podczas weryfikacji haseł użytkowników.

Konfiguracja modułu logowania działającego po stronie klienta używa modułu `SRPLoginModule` oraz wartości opcji `srpServerJndiName`, która odpowiada wartości atrybutu `SRPServiceJndiName` (`srptest/SRPServerInterface`) komponentu serwera. Niezbędny jest także moduł `ClientLoginModule`, w którego konfiguracji został umieszczony atrybut `password-stacking="useFirstPass"`, dzięki czemu informacje uwierzytelniające użytkowników generowane przez moduł `SRPLoginModule` będą przekazywane do warstwy wywołującej EJB.

Jeśli chodzi o konfigurację modułu logowania działającego po stronie serwera, to należy w niej zwrócić uwagę na dwa zagadnienia. Przede wszystkim opcja konfiguracyjna `cacheJndiName=srp-test/AuthenticationCache` przekazuje modułowi `SRPCacheLoginModule` informację o lokalizacji obiektu `CachePolicy` zawierającego obiekty `SRPServerSession` reprezentujące użytkowników, którzy zostali już uwierzytelnieni przez `SRPService`. Wartość ta odpowiada wartości atrybutu `AuthenticationCacheJndiName` usługi `SRPService`. Poza tym konfiguracja zawiera dane modułu `UserRolesLoginModule`, wśród których została umieszczona opcja `password-stacking=useFirstPass`. Zastosowanie drugiego modułu logowania wraz z modułem `SRPCacheLoginModule` jest konieczne, gdyż protokół SRP służy jedynie do uwierzytelniania. Drugi moduł logowania musi być skonfigurowany tak, by akceptował informacje uwierzytelniające zweryfikowane przez moduł `SRPCacheLoginModule` oraz by określał role przypisane danej tożsamości w celu określenia jej uprawnień. Moduł logowania `UserRolesLoginModule` uzupełnia zatem proces uwierzytelniania realizowany przez protokół SRP o możliwości autoryzacji wykorzystującej informacje zapisane w pliku właściwości. Role użytkowników są odczytywane z pliku *roles.properties* dołączonego do archiwuj JAR zawierającego komponent EJB.

Teraz można już uruchomić klienta, przechodząc do katalogu zawierającego przykład i wykonując następujące polecenie:

```
C:\JBoss\przyklady>ant -Dchap=chap8 -Dex=3 run-example
...
run-example3:
     [copy] Copying 1 file to /tmp/jboss-4.0.1/server/default/deploy
     [echo] Waiting for 5 seconds for deploy...
     [java] Rejestracja przy użyciu konfiguracji 'srp'
     [java] Utworzono Echo
     [java] Echo.echo()#1 = To jest wywołanie nr. 1
     [java] Echo.echo()#2 = To jest wywołanie nr. 2
```

W katalogu *przyklady/logs* można znaleźć plik o nazwie *ex3-trace.log*. Zawiera on szczegółowy dziennik operacji wykonywanych przez część algorytmu SRP działającą po stronie klienta. Dzienniki tego typu dokumentują krok po kroku proces tworzenia kluczy publicznych, wyzwań, klucza sesji oraz weryfikacji.

Czytelnik na pewno zwróci uwagę, iż w porównaniu z innymi prostymi przykładami wykonanie tego klienta zabiera stosunkowo dużo czasu. Wynika to z faktu, iż klient musi wygenerować swój klucz prywatny, a to z kolei wymaga wygenerowania mocnej pod względem kryptologicznym liczby losowej. To właśnie wykonanie tego procesu zabiera tak wiele czasu. Na szczęście generacja klucza jest wykonywana tylko za pierwszym razem — gdyby pojawiła się konieczność ponownego zalogowania klienta na tej samej wirtualnej maszynie Javy, cały proces zabrałby znacznie mniej czasu. Należy także zwrócić uwagę, iż drugie wywołanie metody `Echo.echo()` (`Echo.echo()#2`) zgłasza wyjątek informujący o niepowodzeniu uwierzytelniania. Po wykonaniu pierwszego wywołania metody `echo()` klient czeka 15 sekund i próbuje ponownie wykonać tę metodę; w ten sposób przykład demonstruje skutki działania czasu ważności informacji przechowywanych w pamięci podręcznej usługi `SRPService`. Aby umożliwić takie działanie przykładu, czas ważności pamięci podręcznej usługi `SRPService` został ustawiony na 10 sekund. Zgodnie z informacjami podanymi we wcześniejszej części rozdziału zazwyczaj czas ważności pamięci podręcznej powinien być bardzo długi, bądź też należy obsługiwać ponowne uwierzytelnianie.

Uruchamianie serwera JBoss z użyciem menedżera bezpieczeństwa Java 2

Domyślnie serwer JBoss nie używa menedżera bezpieczeństwa udostępnianego przez platformę Java 2. Jeśli chcemy, by przywileje kodu były określane na podstawie informacji o uprawnieniach, jakimi dysponuje platforma Java 2, to konieczne będzie skonfigurowanie serwera JBoss w taki sposób, by używał menedżera bezpieczeństwa Java 2. W tym celu należy odpowiednio skonfigurować opcje wirtualnej maszyny Javy podane w plikach *run.bat* lub *run.sh* umieszczonych w podkatalogu *bin* katalogu instalacyjnego JBossa. Chodzi konkretnie o dwie opisane poniżej opcje:

- `java.security.manager` — jeśli wartość tej opcji nie zostanie jawnie określona, będzie to oznaczać, że należy użyć domyślnego menedżera bezpieczeństwa. Stosowanie menedżera domyślnego jest rozwiązaniem preferowanym. Jednak można także określić wartość tej opcji, aby zastosować niestandardową implementację menedżera bezpieczeństwa. Wartość ta musi zawierać pełną nazwę klasy dziedziczącej po klasie `java.lang.SecurityManager`. Będzie to oznaczać, że plik reguł bezpieczeństwa powinien rozszerzać domyślne zasady zabezpieczeń skonfigurowane w instalacji wirtualnej maszyny Javy.
- `java.security.policy` — ta opcja pozwala na określenie pliku reguł bezpieczeństwa, który będzie rozszerzał domyślne informacje dotyczące reguł bezpieczeństwa wirtualnej maszyny Javy. Wartość tej opcji można podawać w dwóch postaciach: `java.security.policy=URL_pliku_reguł` bądź `java.security.policy==URL_pliku_reguł`. Użycie pierwszego sposobu zapisu oznacza, że wskazany plik reguł powinien zmodyfikować i rozszerzyć domyślny zestaw reguł bezpieczeństwa określony podczas instalacji wirtualnej maszyny Javy. Z kolei drugi zapis mówi, że należy zastosować zasady bezpieczeństwa określone wyłącznie we wskazanym pliku. Wartość opcji `URL_pliku_reguł` może zawierać dowolny adres URL dla którego zdefiniowano metodę obsługi protokołu; ewentualnie może też określać ścieżkę dostępu do pliku.

Oba skrypty służące do uruchamiania serwera JBoss, zarówno *run.bat*, jak i *run.sh*, odwołują się do zmiennej środowiskowej `JAVA_OPTS`, którą można wykorzystać do określenia właściwości menedżera bezpieczeństwa.

Uaktywnienie menedżera bezpieczeństwa Java 2 jest prostą częścią całego zagadnienia. Znacznie trudniejszym zadaniem jest określenie przydzielonych uprawnień. Jeśli zajrzymy do pliku *server.policy* umieszczonego w domyślnym zbiorze plików konfiguracyjnych, to okaże się, że zawiera on następujący wpis:

```
grant {
   // Allow everything for now[3]
   permission java.security.AllPermission;
};
```

Powyższy wpis w praktyce oznacza wyłączenie sprawdzania wszystkich uprawnień niezależnie od wykonywanego kodu; innymi słowy, powyższy zapis stwierdza, iż dowolny kod może wykonać dowolną operację, co oczywiście nie jest zbyt rozsądne. Określenie sensownego zbioru uprawnień pozostaje w gestii autorów aplikacji.

Tabela 8.1 przedstawia zbiór aktualnie dostępnych uprawnień `java.lang.RuntimePermissions` (odnoszących się do serwera JBoss), których określenie jest wymagane.

W ramach zakończenia tej dyskusji poniżej podana została stosunkowo mało znana informacja dotycząca ustawień reguł bezpieczeństwa związanych z testowaniem. Otóż istnieje możliwość stosowania flag określających, w jaki sposób menedżer będzie używać pliku reguł bezpieczeństwa oraz jaki plik zawiera informacje dotyczące uprawnień. Przedstawione poniżej wywołanie programu *java* wyświetla wszystkie możliwe wartości flagi testującej:

[3] Zezwalamy na wszystko

Tabela 8.1. *Wymagane uprawnienia java.lang.RuntimePermissions odnoszące się do JBossa*

Nazwa	Na co zezwala to uprawnienie	Zagrożenia
`org.jboss.security.SecurityAssociation.getPrincipalInfo`	Dostęp do metod `org.jboss.security.SecurityAssociation.getPrincipal()` oraz `getPrincipals()`.	Możliwość zdobycia informacji o obiekcie, który spowodował uruchomienie bieżącego wątku, oraz jego informacji uwierzytelniających.
`org.jboss.security.SecurityAssociation.setPrincipalInfo`	Dostęp do metod `org.jboss.security.SecurityAssociation.setPrincipal()` oraz `setPrincipals()`.	Możliwość określenia obiektu, który spowodował uruchomienie bieżącego wątku, oraz jego informacji uwierzytelniających.
`org.jboss.security.SecurityAssociation.setServer`	Dostęp do metody `org.jboss.security.SecurityAssociation.setServer()`.	Możliwość uaktywnienia lub dezaktywacji wielowątkowego przechowywania tożsamości oraz danych uwierzytelniających obiektu, który spowodował uruchomienie bieżącego wątku.
`org.jboss.security.SecurityAssociation.setRunAsRole`	Dostęp do metod `org.jboss.security.SecurityAssociation.pushRunAsRole()` oraz `popRunAsRole`.	Możliwość zmiany tożsamości roli, jakiej używa bieżący wątek.

```
C:\JBoss\przyklady>java -Djava.security.debug=help

all           turn on all debugging
access        print all checkPermission results
combiner      SubjectDomainCombiner debugging
jar           jar verification
logincontext  login context results
policy        loading and granting
provider      security provider debugging
scl           permissions SecureClassLoader assigns

The following can be used with access:

stack     include stack trace
domain    dumps all domains in context
failure   before throwing exception, dump stack
          and domain that didn't have permission

Note: Separate multiple options with a comma
```

Zastosowanie opcji `-Djava.security.debug=all` powoduje wygenerowanie największej ilości informacji i rzeczywiście jest ich bardzo dużo. Być może to właśnie od tej opcji należy zacząć, jeśli w ogóle nie rozumiemy powodów, z jakich menedżer bezpieczeństwa nie pozwala na wykonanie zamierzonych operacji. Kolejnymi opcjami przydatnymi podczas określania przyczyn problemów związanych z uprawnieniami, które jednak generują nieco mniej informacji, są: `-Djava.security.debug=access,failure`. Także w tym przypadku informacje są obszerne, jednak sytuacja nie jest aż tak zła jak w przypadku zastosowania opcji `all`, gdyż informacje związane z domeną bezpieczeństwa są wyświetlane wyłącznie w przypadku niepowodzeń.

Zastosowanie protokołu SSL przy użyciu JSSE

Do obsługi protokołu SSL JBoss wykorzystuje rozszerzenie JSSE (ang. *Java Secure Socket Extension*). Stanowi ono jeden z elementów platformy JDK 1.4. Aby móc rozpocząć korzystanie z niego, należy uzyskać parę kluczy — prywatny i publiczny — zapisanych w postaci certyfikatu X509. Klucze te będą wykorzystywane przez gniazda SSL serwera. Na potrzeby przykładów zamieszczonych w niniejszej książce wygenerowano certyfikat przy użyciu programu *keytool* (wchodzącego w skład JDK), a wynikowy plik umieszczono w katalogu *chap8* pod nazwą *chap8.keystore*. Plik ten został wygenerowany przy użyciu następującego polecenia:

```
keytool -genkey -keystore chap8.keystore -storepass rmi+ssl -keypass rmi+ssl
➥-keyalg RSA -alias chapter8 -validity 3650 -dname "cn=chapter8 example,ou=admin
book,dc=jboss,dc=org"
```

Wykonanie powyższego polecenia powoduje utworzenie pliku magazynu kluczy (ang. *keystore*) o nazwie *chap8.keystore* (*magazyn kluczy* to baza danych zawierająca klucze używane przez systemy bezpieczeństwa). W magazynie kluczy umieszczane są dwa rodzaje wpisów:

- **Klucze** — każdy wpis tego typu zawiera bardzo wrażliwe, poufne informacje dotyczące kluczy kryptograficznych, które, w celu uniemożliwienia dostępu osób nieupoważnionych, są przechowywane w zabezpieczonym formacie. Zazwyczaj kluczem przechowywanym we wpisach tego typu jest klucz tajny lub klucz prywatny wraz z dołączonym do niego łańcuchem certyfikatów odpowiedniego klucza publicznego. Programy *keytool* oraz *jarsigner* są w stanie obsługiwać jedynie wpisy tego drugiego rodzaju, czyli te, w których są umieszczane klucze prywatne wraz z łańcuchem certyfikatów.
- **Zaufane certyfikaty** — każdy wpis tego typu zawiera jeden certyfikat klucza publicznego należącego do kogoś innego. Nosi on nazwę *zaufanego certyfikatu*, gdyż właściciel magazynu kluczy ufa, iż klucz publiczny zapisany w certyfikacie faktycznie należy do osoby będącej podmiotem (właścicielem) certyfikatu. Wystawca certyfikatu, podpisując go, ręczy, że tożsamość jego właściciela jest poprawna.

Poniżej przedstawione zostały wyniki wyświetlenie zawartości pliku *src/main/org/jboss/chap8/chap8.keystore* przy użyciu programu *keytool*:

```
C:\JBoss\przyklady>keytool -list -v -keystore src/main/org/jboss/chap8/chap8.keystore
Enter keystore password:  rmi+ssl

Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry

Alias name: chapter8
Creation date: 2004-12-17
Entry type: keyEntry
```

```
Certificate chain length: 1
Certificate[1]:
Owner: CN=chapter8 example, OU=admin book, DC=jboss, DC=org
Issuer: CN=chapter8 example, OU=admin book, DC=jboss, DC=org
Serial number: 41c23d6c
Valid from: Thu Dec 16 19:59:08 CST 2004 until: Sun Dec 14 19:59:08 CST 2014
Certificate fingerprints:
         MD5:  36:29:FD:1C:78:44:14:5E:5A:C7:EB:E5:E8:ED:06:86
         SHA1: 37:FE:BB:8A:A5:CF:D9:3D:B9:61:8C:53:CE:19:1E:4D:BC:C9:18:F2


*****************************************
*****************************************
```

Kiedy rozszerzenie JSSE będzie poprawnie działać, a magazyn kluczy zawierający niezbędny certyfikat został już wygenerowany, można przystąpić do konfiguracji serwera JBoss tak, by dostęp do komponentów EJB odbywał się przy użyciu protokołu SSL. W tym celu należy odpowiednio skonfigurować fabryki gniazd RMI obiektu wywołującego EJB. Podsystem JBossSX zawiera implementacje interfejsów `java.rmi.server.RMIServerSocketFactory` oraz `java.rmi.server.RMIClientSocketFactory`, które pozwalają na stosowanie RMI wraz z gniazdami obsługującymi komunikacje szyfrowaną przy użyciu protokołu SSL. Interfejsy te są odpowiednio implementowane przez klasy: `org.jboss.security.ssl.RMISSLServerSocketFactory` oraz `org.jboss.security.ssl.RMISSLClientSocketFactory`. Rozpoczęcie wykorzystania protokołu SSL w celu odwoływania się do komponentów EJB wymaga wykonania dwóch operacji. Pierwszą z nich jest włączenie wykorzystania magazynu kluczy jako bazy danych zawierającej certyfikaty SSL serwera. Operacja ta wymaga odpowiedniego skonfigurowania komponentu MBean `org.jboss.security.plugins.JaasSecurityDomain`. Stosowny deskryptor umieszczony w pliku *jboss-service.xml* w katalogu *chap8/ex4* został przedstawiony na listingu 8.16.

Listing 8.16. *Przykładowa konfiguracja JaasSecurityDomain w przypadku zastosowania RMI i SSL*

```
<!-- Konfiguracja domeny SSL -->
<mbean code="org.jboss.security.plugins.JaasSecurityDomain"
       name="jboss.security:service=JaasSecurityDomain,domain=RMI+SSL">
    <constructor>
        <arg type="java.lang.String" value="RMI+SSL"/>
    </constructor>
    <attribute name="KeyStoreURL">chap8.keystore</attribute>
    <attribute name="KeyStorePass">rmi+ssl</attribute>
</mbean>
```

Klasa `JaasSecurityDomain` dziedziczy po „standardowej" klasie `JaasSecurityManager` i dodaje do niej obsługę magazynów kluczy oraz możliwości dostępu do obiektów typu `KeyManagerFactory` oraz `TrustManagerFactory`. Rozbudowuje ona możliwości standardowego menadżera bezpieczeństwa o obsługę protokołu SSL oraz innych operacji kryptograficznych wymagających użycia kluczy. Przedstawiona powyżej konfiguracja powoduje po prostu wczytanie magazynu kluczy *chap8.keystore* (z pliku SAR tworzonego w przykładzie 4.) przy wykorzystaniu wskazanego hasła.

Drugim krokiem jest zdefiniowanie konfiguracji obiektu wywołującego EJB, który będzie korzystać z obiektów fabrykujących gniazda RMI obsługujące protokół SSL. W tym celu należy zdefiniować własną konfiguracje obiektu `JRMPInvoker` zgodnie z informacjami zamieszczonymi w rozdziale 5., „Komponenty EJB w JBossie", jak również przygotować konfiguracje komponentu EJB, który będzie używać tego obiektu wywołującego. Poniższy fragment pliku *jboss-service.xml* przedstawia deskryptor definiujący niestandardowy obiekt `JRMPInvoker`:

```
<mbean code="org.jboss.invocation.jrmp.server.JRMPInvoker"
       name="jboss:service=invoker,type=jrmp,socketType=SSL">
    <attribute name="RMIObjectPort">14445</attribute>
    <attribute name="RMIClientSocketFactory">
        org.jboss.security.ssl.RMISSLClientSocketFactory
    </attribute>
    <attribute name="RMIServerSocketFactory">
        org.jboss.security.ssl.RMISSLServerSocketFactory
    </attribute>
    <attribute name="SecurityDomain">java:/jaas/RMI+SSL</attribute>
    <depends>jboss.security:service=JaasSecurityDomain,domain=RMI+SSL</depends>
</mbean>
```

W celu skonfigurowania obiektu wywołującego używającego protokołu SSL należy stworzyć powiązanie o nazwie `stateless-ssl-invoker` używające niestandardowego obiektu `JRMPInvoker`. Poniższy plik *jboss.xml* pokazuje, w jaki sposób można utworzyć takie powiązanie i połączyć je z komponentem `EchoBean4`:

```
<?xml version="1.0"?>
<jboss>
    <enterprise-beans>
        <session>
            <ejb-name>EchoBean4</ejb-name>
            <configuration-name>Standardowy bezstanowy komponent sesyjny</
➥configuration-name>
            <invoker-bindings>
                <invoker>
                    <invoker-proxy-binding-name>
                        stateless-ssl-invoker
                    </invoker-proxy-binding-name>
                </invoker>
            </invoker-bindings>
        </session>
    </enterprise-beans>

    <invoker-proxy-bindings>
        <invoker-proxy-binding>
            <name>stateless-ssl-invoker</name>
            <invoker-mbean>jboss:service=invoker,type=jrmp,socketType=SSL</invoker-
➥mbean>
            <proxy-factory>org.jboss.proxy.ejb.ProxyFactory</proxy-factory>
            <proxy-factory-config>
            <client-interceptors>
                <home>
                    <interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
                    <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
                    <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
                    <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
```

```
                </home>
                <bean>
                    <interceptor>org.jboss.proxy.ejb.StatelessSessionInterceptor
➥</interceptor>
                    <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
                    <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
                    <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
                </bean>
            </client-interceptors>
            </proxy-factory-config>
        </invoker-proxy-binding>
    </invoker-proxy-bindings>
</jboss>
```

Kod przykładu 4. umieszczony jest w przykładach dołączonych do niniejszej książki w katalogu *src/main/org/jboss/chap8/ex4*. Przykład ten zawiera kolejny prosty bezstanowy komponent sesyjny udostępniający metodę `echo`, która zwraca przekazany argument. Jeśli nie pojawią się żadne problemy z wykorzystaniem protokołu SSL, to bardzo trudno jest określić, kiedy jest on stosowany; dlatego też, aby się upewnić, że podczas odwoływania się do komponentu EJB faktycznie jest wykorzystywany protokół SSL, przedstawiony przykład można uruchamiać na dwa różne sposoby. Najpierw należy uruchomić serwer JBoss, używając przy tym standardowej konfiguracji, a następnie wykonać przykład 4b w sposób następujący:

```
C:\JBoss\przyklady>ant -Dchap=chap8 -Dex=4b run-example
...
run-example4b:
     [copy] Copying 1 file to /tmp/jboss-4.0.1/server/default/deploy
     [echo] Waiting for 15 seconds for deploy...
...
     [java] Exception in thread "main" java.rmi.ConnectIOException: error during JRMP
connection establishment; nested exception is:
     [java]     javax.net.ssl.SSLHandshakeException: sun.security.validator.
➥ValidatorException: No trusted certificate found
...
```

Pojawienie się wyjątku, jaki zostanie w tym przypadku zgłoszony, jest całkowicie zgodne z oczekiwaniami; taki był cel przygotowania tego przykładu. Warto zwrócić uwagę, iż zamieszczone powyżej informacje o zgłoszonych wyjątkach zostały ręcznie sformatowane w celu poprawienia ich przejrzystości w tekście książki; z tego względu wyniki uzyskane przez Czytelnika podczas uruchamiania tego przykładu mogą mieć nieco inną postać. Kluczowym aspektem tego przykładu jest fakt, że zgłoszony wyjątek jasno pokazuje, iż do komunikacji z kontenerem EJB serwera JBoss używane są klasy JSSE firmy Sun. Wyjątek ten informuje, iż weryfikacja certyfikatu, który jest używany w przykładzie jako certyfikat serwera JBoss, wykazała, że jest został on podpisany przez żadną z domyślnych instytucji certyfikujących. Takiego wyniku można się było spodziewać, gdyż domyślny magazyn kluczy instytucji certyfikujących dostarczany wraz z pakietem JSSE zawiera jedynie powszechnie znane instytucje, takie jak VeriSign, Thawte oraz RSA Data Security. Aby klient EJB uznał, iż samodzielnie podpisany certyfikat jest ważny, należy poinstruować klasy JSSE, by korzystały z zaufanego magazynu (ang. *truststore*) *chap8.keystore* (*magazyn zaufany* to magazyn kluczy zawierający certyfikaty klucza publicznego użytego do podpisywania innych certyfikatów).

W tym celu należy uruchomić przykład 4., używając przy tym opcji `-Dex=4`, a nie opcji `-Dex=4b`, aby przekazać położenie odpowiedniego zaufanego magazynu przy użyciu właściwości systemowej `javax.net.ssl.trustStore`:

```
C:\JBoss\przyklady>ant -Dchap=chap8 -Dex=4 run-example
...
run-example4:
     [copy] Copying 1 file to /tmp/jboss-4.0.1/server/default/deploy
     [echo] Waiting for 5 seconds for deploy...
...
     [java] Created Echo
     [java] Echo.echo()#1 = To jest wywołanie nr. 1
```

Tym razem jedyną informacją o tym, iż wykorzystywane są gniazda SSL, jest komunikat `SSL handshakeCompleted`. Jest to komunikat poziomu testowania rejestrowany w dzienniku, a generuje go klasa `RMISSLClientSocketFactory`. Jeśli jednak klient nie został skonfigurowany w taki sposób, by komunikaty poziomu testowania generowane przez bibliotekę *log4j* były wyświetlane, to nie pojawi się żadna informacja o wykorzystaniu protokołu SSL. Poważną różnicę zauważymy jednak, jeśli przyjrzymy się czasom działania oraz obciążeniu procesora. Protokół SSL, podobnie jak SRP, wymaga zastosowania liczb losowych mocnych pod względem kryptograficznym, których zainicjowanie podczas pierwszego użycia zabiera wiele czasu. To właśnie generacja liczb losowych jest tak wyraźnie widoczna w użyciu procesora i czasie uruchamiania programu.

W konsekwencji, jeśli Czytelnik będzie uruchamiać opisywane przykłady, choćby przykład 4b, na komputerze wolniejszym od tego, na jakim były one tworzone, to może wystąpić wyjątek podobny do poniższego:

```
javax.naming.NameNotFoundException: EchoBean4 not bound
   at sun.rmi.transport.StreamRemoteCall.exceptionReceivedFromServer
...
```

Powyższy problem wynika z faktu, iż serwer JBoss nie zakończył wdrażania przykładowego komponentu w czasie, na jaki pozwalał klient. Problem ten jest spowodowany długim czasem inicjalizacji generatora liczb losowych używanego przez gniazda SSL serwera. W razie wystąpienia tego problemu wystarczy ponownie uruchomić przykład lub zwiększyć czas oczekiwania podany w pliku *build.xml* w katalogu zawierającego przykłady do rozdziału 8.

Konfiguracja serwera JBoss działającego za firewallem

JBoss posiada wiele usług korzystających z gniazd, które wymagają utworzenia portów serwera. W tym podrozdziale przedstawione zostały usługi używające portów, które najprawdopodobniej trzeba będzie skonfigurować w przypadku korzystania z serwera JBoss działającego za firewallem. W tabeli 8.2 przedstawione zostały numery

portów, ich typ oraz używające ich usługi, przy czym zamieszczone informacje dotyczą standardowej konfiguracji JBoss. Z kolei tabela 8.3 przedstawia analogiczne informacje dla usług podanych w konfiguracji `all`.

Tabela 8.2. *Porty używane w konfiguracji domyślnej*

Numer	Typ	Usługa
1099	TCP	`org.jboss.naming.NamingService`
1098	TCP	`org.jboss.naming.NamingService`
4444	TCP	`org.jboss.invocation.jrmp.server.JRMPInvoker`
4445	TCP	`org.jboss.invocation.pooled.server.PooledInvoker`
8009	TCP	`org.jboss.web.tomcat.tc4.EnbeddedTomcatService`
8080	TCP	`org.jboss.web.tomcat.tc4.EnbeddedTomcatService`
8083	TCP	`org.jboss.web.WebService`
8093	TCP	`org.jboss.mq.il.uil2.UILServerILService`

Tabela 8.3. *Porty używane w konfiguracji o nazwie „all”*

Numer	Typ	Usługa
1100	TCP	`org.jboss.ha.jndi.HANamingService`
0[a]	TCP	`org.jboss.ha.jndi.HANamingService`
1102	UDP	`org.jboss.ha.jndi.HANamingService`
1161	UDP	`org.jboss.jmx.adaptor.snmp.agent.SnmpAgentService`
1162	UDP	`org.jboss.jmx.adaptor.snmp.trapd.TrapdService`
3528	TCP	`org.jboss.invocation.iiop.IIOPInvoker`
45566[b]	UDP	`org.jboss.ha.framework.server.ClusterPartition`

a — aktualnie jest to port anonimowy, lecz można go podać przy użyciu atrybutu `RmiPort`.

b — istnieją dwa dodatkowe porty anonimowe UDP, określić jednak można tylko jeden z nich; służy do tego celu atrybut `rcv_port`.

Zabezpieczanie serwera JBoss

JBoss posiada kilka punktów dostępu administracyjnego, które należy odpowiednio zabezpieczyć lub usunąć, by uniemożliwić nieautoryzowany dostęp do serwera produkcyjnego. W kolejnych punktach zostały opisane różne usługi administracyjne oraz sposoby ich ochrony.

Usługa jmx-console.war

Usługa `jmx-console.war`, którą można znaleźć w głównym katalogu wdrożeniowym serwera, generuje strony HTML dostarczające informacji o stanie jądra serwera. A zatem usługa ta zapewnia dostęp do wszelkich operacji administracyjnych, takich jak zamykanie serwera, zatrzymywanie usług, wdrażanie nowych usług i tak dalej. Usługę tę należy zabezpieczyć jak każdą aplikację sieciową bądź też całkowicie wyłączyć.

Usługa web-console.war

Usługa `web-console.war`, którą można znaleźć w katalogu *deploy/management*, jest kolejną aplikacją sieciową zapewniającą dostęp do jądra serwera JMX. Wykorzystuje ona kombinację apletów oraz stron HTML i zapewnia porównywalne możliwości administracyjne co usługa `jmx-console.war`. Właśnie z tego względu należy ją zabezpieczyć lub usunąć. W archiwum *web-console.war* można znaleźć zapisane w komentarzach szablony podstawowych zabezpieczeń (w pliku *WEB-INF/web.xml*) oraz konfigurację dziedziny bezpieczeństwa (w pliku *WEB-INF/jboss-web.xml*).

Usługa http-invoker.sar

Plik *http-invoker.sar* umieszczony w katalogu wdrożeniowym zawiera usługę zapewniającą dostęp do komponentów EJB i usługi `Naming` JNDI za pomocą RMI i HTTP. Usługa ta zawiera serwlet, który przetwarza wiadomości generowane przez szeregowane obiekty `org.jboss.invocation.Invocation` reprezentujące wywołania, który powinny zostać przekazane do `MBeanServer`. W efekcie daje ona możliwość dostępu za pośrednictwem protokołu HTTP do komponentów MBean wspomagających działanie odłączonej usługi wywołującej RMI/JRMP, gdyż istnieje możliwość określenia postaci odpowiedniego żądania HTTP POST. Aby uchronić się przed tą możliwością dostępu, należałoby zabezpieczyć serwlet `JMXInvokerServlet`, zapisując niezbędne informacje konfiguracyjne w pliku *http-invokier.sar/invoker.war/WEB-INF/web.xml*. Ponieważ domyślnie jest już zdefiniowane bezpieczne odwzorowanie dla ścieżki */restricted/JMXInvokerServlet*, a zatem, aby go zastosować, wystarczy usunąć inne ścieżki oraz skonfigurować domenę bezpieczeństwa `http-invokier` w pliku *http-invoker.sar/invoker.war/WEB-INF.xml*.

Usługa jmx-invoker-adaptor-server.sar

Plik *jmx-invoker-adaptor-server.sar* zawiera usługę, która używając wydzielonej usługi wywołującej RMI/JRMP zapewnia dostęp do interfejsu `MBeanServer` przy użyciu interfejsu zgodnego z RMI. Obecnie jedynym sposobem zabezpieczenia tej usługi mogłoby być zmiana używanych protokołów na RMI i HTTP oraz zabezpieczenie usługi *http-invoker.sar* w sposób opisany w poprzednim punkcie. W przyszłości usługa ta zostanie uruchomiona jako komponent XMBean dysponujący odpowiedzialnym za bezpieczeństwo obiektem wywołującym obsługującym kontrolę dostępu bazującą na rolach.