JBoss Enterprise Application Platform Common Criteria Certification 5

JBoss Security Integration Guide

for use with JBoss Enterprise Application Platform 5 Common Criteria Certification
Edition 5.1.0

Last Updated: 2017-11-20

JBoss Enterprise Application Platform Common Criteria Certification 5
JBoss Security Integration Guide

for use with JBoss Enterprise Application Platform 5 Common Criteria Certification
Edition 5.1.0

Red Hat Documentation Group

Abstract

A reference guide aimed at developers. This guide contains the main classes JBoss Enterprise Application Platform uses to implement security functionality. Refer to this guide when writing applications for the JBoss Enterprise Application Platform 5.1.0.

Table of Contents

CHAPTER 1. SECURITY CONTEXT

The Security Context combines the Authentication, Authorization, Mapping and Auditing aspects of a security-conscious system. The following code is an example interface for the Security Context (**`SecurityContext`**):

```
package org.jboss.security;

/**
 *  Encapsulation of Authentication, Authorization, Mapping and other
 *  security aspects at the level of a security domain
 *  @author <a href="mailto:Anil.Saldhana@jboss.org">Anil Saldhana</a>
 */
public interface SecurityContext extends Serializable,Cloneable
{
   /**
    * Authentication Manager for the security domain
    */
   public AuthenticationManager getAuthenticationManager();
   /**
    * Authorization Manager for the security domain
    */
   public AuthorizationManager getAuthorizationManager();

   /**
    * Mapping manager configured with providers
    */
   public MappingManager getMappingManager();

   /**
    * AuditManager configured for the security domain
    */
   public AuditManager getAuditManager();

   /**
    * Context Map
    */
   public Map<String,Object> getData();

   /**
    * Return the Security Domain
    */
   public String getSecurityDomain();

   /**
    * Subject Info
    *
    * @see SecurityContextUtil#getSubject()
    * @see SecurityContextUtil#createSubjectInfo(Principal, Object,
Subject)
    */
   SubjectInfo getSubjectInfo();

   /**
    * Subject Info
    *
```

```
     * @see SecurityContextUtil#getSubject()
     * @see SecurityContextUtil#createSubjectInfo(Principal, Object, 
Subject)
     */
    void setSubjectInfo(SubjectInfo si);

    /**
     * RunAs Representation
     * 
     * @see #setRunAs(RunAs)
     */
    public RunAs getRunAs();

    /**
     * Set the current RunAs for the security context that will be
     * propagated out to other security context.
     * 
     * RunAs coming into this security context needs to be done
     * from SecurityContextUtil.getCallerRunAs/setCallerRunAs
     * 
     * @see SecurityContextUtil#getCallerRunAs()
     * @see SecurityContextUtil#setCallerRunAs(RunAs)
     * 
     * @param runAs
     */
    public void setRunAs(RunAs runAs);

    /**
     * Return a utility that is a facade to the internal 
     * storage mechanism of the Security Context
     * 
     * This utility can be used to store information like
     * roles etc in an implementation specific way
     * @return
     */
    public SecurityContextUtil getUtil();
}
```

The **SecurityUtil** is associated with the **SecurityContext**. It provides some utility methods to shield the details of any vendor implementation of the **SecurityContext** from the implementation. The following code is an example of the **SecurityUtil** abstract class:

```
package org.jboss.security;

import java.security.Principal;

import javax.security.auth.Subject;

/**
 *  General Utility methods for dealing with the SecurityContext
 */
public abstract class SecurityContextUtil
{
    protected SecurityContext securityContext = null;

    public void setSecurityContext(SecurityContext sc)
    {
       this.securityContext = sc;
    }

    /**
     * Get the username from the security context
     * @return username
     */
    public abstract String getUserName();

    /**
     * Get the user principal the security context
     * @return user principal
     */
    public abstract Principal getUserPrincipal();

    /**
     * Get the credential
     * @return
     */
    public abstract Object getCredential();

    /**
     * Get the subject the security context
     * @return
     */
    public abstract Subject getSubject();

    /**
     * Get the RunAs that was passed into the current security context
     * The security context RunAs is the RunAs that will be propagated out
of it
     * @return
     */
    public abstract RunAs getCallerRunAs();

    /**
     * Set the Caller RunAs in the security context
     * Security Context implementations are free to store
     * the caller runas in any manner
     * @param runAs
     */
    public abstract void setCallerRunAs(RunAs runAs);

    /**
     * Get a holder of subject, runAs and caller RunAs
     * @return
     */
    public abstract SecurityIdentity getSecurityIdentity();

    /**
     * Inject subject, runAs and callerRunAs into the security context
     * Mainly used by integration code base to cache the security identity
     * and put back to the security context
     * @param si The SecurityIdentity Object
     */
    public abstract void setSecurityIdentity(SecurityIdentity si);

    /**
     * Get the Roles associated with the user for the
     * current security context
     * @param <T>
     * @return
     */
    public abstract <T>  T getRoles();

    /**
     * Set the roles for the user for the current security context
     * @param <T>
     * @param roles
     */
    public abstract <T>  void setRoles(T roles);

    /**
     * Create SubjectInfo and set it in the current security context
     * @param principal
     * @param credential
     * @param subject
     */
    public void createSubjectInfo(Principal principal, Object
credential,Subject subject)
    {
       SubjectInfo si = new SubjectInfo(principal, credential, subject);
       this.securityContext.setSubjectInfo(si);
    }

    /**
     * Set an object on the Security Context
     * The context implementation may place the object in its internal
     * data structures (like the Data Map)
     * @param <T> Generic Type
     * @param sc Security Context Object
     * @param key Key representing the object being set
     * @param obj
     */
    public abstract <T> void set(String key, T obj);

    /**
     * Return an object from the Security Context
     * @param <T>
     * @param sc Security Context Object
     * @param key key identifies the type of object we are requesting
     * @return
     */
    public abstract <T> T get(String key);

    /**
     * Remove an object represented by the key from the security context
     * @param <T>
     * @param sc Security Context Object
     * @param key key identifies the type of object we are requesting
     * @return the removed object
     */
    public abstract <T> T remove(String key);
}
```

The **SecurityContextUtil** provides methods that retrieve the user **principal** and **credential**, handle the **RunAs** operation and allow you to use the **SecurityContext** to store objects by using key pairs.

The **SecurityContextUtil** also uses a **SecurityIdentity** component, which represents the identity of the agent that is interfacing with the security system. It contains the subject and several run-as options, such as **RunAs** and **CallerRunAs**:

```
package org.jboss.security;

import java.security.Principal;
import javax.security.auth.Subject;

//$Id$

/**
 *  Represents an Identity of an agent interacting with the
 *  security service. It can be an user or a process. It
 *  consists of a subject and various run-as
 */
public class SecurityIdentity
{
   SubjectInfo theSubject = null;
   RunAs runAs = null;
   RunAs callerRunAs = null;

   public SecurityIdentity(SubjectInfo subject, RunAs runAs, RunAs
callerRunAs)
   {
      this.theSubject = subject;
      this.runAs = runAs;
      this.callerRunAs = callerRunAs;
   }

   public Principal getPrincipal()
   {
      return theSubject != null ? theSubject.getAuthenticationPrincipal()
: null;
   }

   public Object getCredential()
   {
      return theSubject != null ?
theSubject.getAuthenticationCredential(): null;
   }

   public Subject getSubject()
   {
      return theSubject != null ? theSubject.getAuthenticatedSubject() :
null;
   }
```

```
    public RunAs getRunAs()
    {
       return runAs;
    }

    public RunAs getCallerRunAs()
    {
       return callerRunAs;
    }
}
```

RunAs differs from **CallerRunAs** in that it represents the run-as that is leaving the current context. The **CallerRunAs** represents the run-as method that is entering the security context. The folllowing is a **RunAs** interface:

```
package org.jboss.security;

import java.security.Principal;
/**
 *  Represent an entity X with a proof of identity Y
 */
public interface RunAs extends Principal
{
   /**
    * Return the identity represented
    * @return
    */
   public <T> T getIdentity();

   /**
    * Return the proof of identity
    * @return
    */
   public <T> T getProof();
}
```

CHAPTER 2. AUTHENTICATION MANAGER

The Authentication Manager provides authentication support to a security-conscious subsystem, and can be obtained from the **`SecurityContext`**.

```
package org.jboss.security;

import java.security.Principal;
import java.util.Map;
import javax.security.auth.Subject;
import javax.security.auth.message.MessageInfo;

/** The AuthenticationManager is responsible for validating credentials
 * associated with principals.
 */
public interface AuthenticationManager
{
   /** Retrieves the security domain that the security manager is from.
Every
       security manager belongs to a named domain. The meaning of the
security
       domain name depends on the implementation. For example, names may
be
       fine-grained and refer to EJB names, or refer to coarse-grained
objects such
       as J2EE applications and DNS domain names. @return the security
domain
       name. If null, the security manager belongs to the logical default
domain.
   */
   String getSecurityDomain();

   /** The isValid method is invoked to check whether a user ID (and
associated
   credentials) as known to the operational environment are sufficient and
valid
   proof of the user's identity. This is typically implemented as a call
to isValid
   with a null subject.

    @see #isValid(Principal, Object, Subject)

    @param principal - the user identity in the operation environment
    @param credential - the proof of user identity as known in the
    operation environment
    @return true if the principal, credential pair is valid, false
otherwise.
   */
   public boolean isValid(Principal principal, Object credential);

   /** The isValid method is invoked to check whether a user ID (and
associated
   credentials) as known to the operational environment are sufficient and
valid
   proof of the user's identity. This also extends the
AuthenticationManager to
```

```
   provide a copy of the resulting authenticated Subject. This allows a
caller to
   authenticate a user and obtain a Subject whose state cannot be modified
by
   other threads associated with the same principal.
   @param principal - the user identity in the operation environment
   @param credential - the proof of user identity as it is known in the
    operation environment
    @param activeSubject - the Subject which should be populated with the
      validated Subject contents. A JAAS based implementation would
typically
      populate the activeSubject with the LoginContext.login result.
    @return true if the principal, credential pair is valid, false
otherwise.
   */
   boolean isValid(Principal principal, Object credential,
      Subject activeSubject);

  /**
    * Authenticate a Subject given the request response JSR-196(JASPI)
messages
    * @param requestMessage
    * @param clientSubject Pre-created or null subject
    * @param layer Message Layer for the JASPI (Optional):  Default: HTTP
    * @return true if client subject is valid, false otherwise
    */
   boolean isValid(MessageInfo requestMessage, Subject clientSubject,
String layer);

   /** Retrieve the currently authenticated subject. Previously,
implementing the
   AuthenticationManager isValid method could set the active Subject,
which
   caused problems in multi-threaded use-cases where the Subject instance
was
   shared between multiple threads. This has been deprecated in favour of
the
   JACC PolicyContextHandler#getContext(String, Object)

    @return the previously authenticated Subject (if isValid succeeded),
    null if isValid failed or has not been called for the active thread.
    */
   Subject getActiveSubject();

   /**
    * Trust-related use-cases may need their principal translated from
another domain
    * to the current domain. This interface may need to contact the
external trust
    * provider to derive the target principal.
    * @param anotherDomainPrincipal
    *                  Principal that is applicable in the other domain
    *                  (Can be null - in which case the contextMap is
used
    *                  solely to derive the target principal)
    * @param contextMap
    *                   Any context information (including information on
the other domain
    *                   that may be relevant in deriving the target
principal). Any SAML
    *                   assertions that may be relevant can be passed
here.
    * @return principal from a target security domain
    */
   Principal getTargetPrincipal(Principal anotherDomainPrincipal,
Map<String,Object> contextMap);
}
```

`getActiveSubject` is a deprecated API which was used to determine the **`subject`**.

The **`isValid`** method takes a **`MessageInfo`** object, and lets you validate the message according to the Java Authentication Service Provider Interface for Containers (JSR-196) specification.

CHAPTER 3. AUTHORIZATION MANAGER

`AuthorizationManager` is an interface that provides the fine-grained aspects of authorization to a security-conscious subsystem. It is obtained from the **`SecurityContext`**.

```
package org.jboss.security;

import java.security.Principal;
import java.security.acl.Group;
import java.util.Map;
import java.util.Set;

import org.jboss.security.authorization.AuthorizationException;
import org.jboss.security.authorization.Resource;
/**
 *  Generalized Authorization Manager Interface.
 */
public interface AuthorizationManager
{
   /**
    * Authorize a resource
    * @param resource
    * @return
    * @throws AuthorizationException
    */
   public int authorize(Resource resource) throws AuthorizationException;


   /** Validates the application domain roles to which the operational
   environment Principal belongs.
   @param principal - the caller principal as known in the operation
environment.
   @param roles - the Set<Principal> for the application domain roles that
the
    principal will be validated against.
   @return true if the principal has at least one of the roles in the
roles set,
       otherwise false.
    */
   public boolean doesUserHaveRole(Principal principal, Set roles);


   /** Returns the set of domain roles assigned to the principal.
   @return The Set<Principal> for the application domain roles that the
    principal has been assigned.
    */
   public Set getUserRoles(Principal principal);

   /**
    * Trust-based use-cases may need to determine the roles of the target
    * principal, which are derived by the Authentication Manager via a
principal
    * from another domain.
    * An implementation of this interface may need to contact a trust
provider
    * for additional information about the principal
```

```
     * @param targetPrincipal - the principal applicable in current domain
     * @param contextMap - read-only contextual information that can assist
the
     *                                    implementation when
determining roles
     * @return roles from the target domain
     */
   public Group getTargetRoles(Principal targetPrincipal, Map contextMap);
 }
```

The Resource interface looks like this:

```
package org.jboss.security.authorization;

import java.util.Map;

/**
 *  Resource that is subject to Authorization Decisions
 */
public interface Resource
{
   //Get the Layer (Web/EJB etc)
   public ResourceType getLayer();

   //Return the contextual map
   public Map getMap();
}
```

An authorization module interface looks like this:

```
package org.jboss.security.authorization;

import java.util.Map;

import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;

/**
 *  Represents a Policy Decision Module that is used by the
 *  Authorization Context
 */
public interface AuthorizationModule
{
   /**
    * Abort the Authorization Process
    * @return true - abort passed, false-otherwise
    */
   boolean abort() throws AuthorizationException;

   /**
    * Overall authorization process has succeeded.
    * The module can commit any decisions it has made, with
    * third party systems like a database.
    * @return
    */
   boolean commit() throws AuthorizationException;

   /**
    * Initialize the module
    *
    * @param subject the authenticated subject
    * @param handler CallbackHandler
    * @param sharedState state shared with other configured modules
    * @param options options specified in the Configuration
    *                for this particular module
    */
   void initialize(Subject subject, CallbackHandler handler,
         Map sharedState, Map options);

   /**
    * Authorize the resource
    * @param resource
    * @return AuthorizationContext.PERMIT or AuthorizationContext.DENY
    */
   int authorize(Resource resource);

   /**
    * A final cleanup opportunity offered
    * @return cleanup by the module passed or not
    */
   boolean destroy();
}
```

There is a **`PolicyRegistration`** interface that can provide a mechanism for registering policies (such as the XACML Policy), which looks like this:

```
package org.jboss.security.authorization;

/**
 *  Interface to register policies
 */
public interface PolicyRegistration
{
    /**
     * Registers a policy given the location and a context ID
     * @param contextID
     * @param location - location of the Policy File
     */
    void registerPolicy(String contextID, URL location);

    /**
     *
     * Registers a policy given an XML-based stream and a context ID
     *
     * @param contextID
     * @param stream - InputStream that is an XML stream
     */
    void registerPolicy(String contextID, InputStream stream);

    /**
     * Unregister a policy
     * @param contextID Context ID
     */
    void deRegisterPolicy(String contextID);

    /**
     * Obtain the registered policy for the context ID
     * @param contextID - Context ID
     * @param contextMap - A map that can be used by the implementation
     *                               to determine the policy choice
(typically null)
     */
    Object getPolicy(String contextID, Map contextMap);
}
```

The Resource interface identifies resources that require authorization:

```
  * JBoss, Home of Professional Open Source
  * Copyright 2005, JBoss Inc., and individual contributors as indicated
  * by the @authors tag. See the copyright.txt in the distribution for a
  * full listing of individual contributors.
  *
  * This is free software; you can redistribute it and/or modify it
  * under the terms of the GNU Lesser General Public License as
  * published by the Free Software Foundation; either version 2.1 of
  * the License, or (at your option) any later version.
  *
  * This software is distributed in the hope that it will be useful,
  * but WITHOUT ANY WARRANTY; without even the implied warranty of
  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  * Lesser General Public License for more details.
  *
  * You should have received a copy of the GNU Lesser General Public
  * License along with this software; if not, write to the Free
  * Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA
  * 02110-1301 USA, or see the FSF site: http://www.fsf.org.
  */
package org.jboss.security.authorization;

import java.util.Map;

/**
 *  Resource that is subject to Authorization Decisions
 *  @author <a href="mailto:Anil.Saldhana@jboss.org">Anil Saldhana</a>
 *  @version $Revision$
 */
public interface Resource
{
   //Get the Layer (Web/EJB etc)
   public ResourceType getLayer();

   //Return the contextual map
   public Map getMap();
}
```

For example, the EJB Container authorization uses a resource called **`EJBResource`**, and the Web Container uses the **`WebResource`**.

CHAPTER 4. MAPPING MANAGER

The Mapping Manager is an interface that obtains preconfigured **`MappingContext`**s for particular Mapping Class types, such as the **`java.security.acl.Group`** used in role mapping. Implementations of the Service Provider Interface (SPI) can define their own Mapping Class types.

The **`MappingManager`** interface is found in the following package:

```
 package org.jboss.security.mapping;

/**
 *  Manager used to map various types
 */
public interface MappingManager
{
   MappingContext getMappingContext(Class mappingType);
}
```

The **`MappingContext`** is a set of preconfigured **`MappingProvider`** instances for a particular class type and security domain. The **`MappingContext`** interface looks like this:

```
package org.jboss.security.mapping;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;

/**
 *  Generic Context used by the Mapping Framework
 */
public class MappingContext
{
   private List modules = new ArrayList();

   public MappingContext(List mod)
   {
      this.modules = mod;
   }

   /**
    * Get the set of mapping modules
    * @return
    */
   public List getModules()
   {
      return this.modules;
   }

   /**
    * Apply mapping semantics on the passed object
    * @param obj Read-only Contextual Map
    * @param mappedObject an object on which mapping will be applied
    */
   public <T> void performMapping(Map obj, T mappedObject)
   {
      int len = modules.size();

      for(int i = 0 ; i < len; i++)
      {
         MappingProvider<T> mp = (MappingProvider<T>)modules.get(i);
         mp.performMapping(obj, mappedObject);
      }
   }
}
```

The **`MappingProvider`** interface looks like the following:

```
package org.jboss.security.mapping;

import java.util.Map;

/**
 *  A provider with mapping functionality
 */
public interface MappingProvider<T>
{
   /**
    * Initializes the provider with the configured module options
    * @param options
    */
   void init(Map options);

   /**
    * Maps the passed object
    * @param map A read-only contextual map that can provide information
to the provider
    * @param mappedObject an Object on which the mapping will be applied
    * @throws IllegalArgumentException if the mappedObject is not
understood by the
    * provider.
    */
    void performMapping(Map map, T mappedObject);
}
```

CHAPTER 5. AUDIT MANAGER

The Audit Manager interface audits security information. The interface is as follows:

```
package org.jboss.security.audit;

/**
 *  An interface that defines the Security Audit Service
 */
public interface AuditManager
{
   /**
    * Audits the information available in the audit event
    * @param ae the Audit Event
    * @see AuditEvent
    */
   public void audit(AuditEvent ae);
}
```

The **`AuditEvent`** is the container object for the audit information. The interface looks like this:

```
package org.jboss.security.audit;

/**
 *  Holds audit information
 */
public class AuditEvent
{
   private String auditLevel = AuditLevel.INFO;

   private Map<String,Object> contextMap = new HashMap<String,Object>();

   private Exception underlyingException = null;

   public AuditEvent(String level)
   {
      this.auditLevel = level;
   }

   public AuditEvent(String level, Map<String,Object> map)
   {
      this(level);
      this.contextMap = map;
   }

   public AuditEvent(String level, Map<String,Object> map, Exception ex)
   {
      this(level,map);
      this.underlyingException = ex;
   }

   /**
    * Return the Audit Level
    * @return
    */
   public String getAuditLevel()
   {
      return this.auditLevel;
   }

   /**
    * Get the Contextual Map
    * @return Map that is final
    */
   public Map getContextMap()
   {
      return contextMap;
   }

   /**
    * Set a non-modifiable Context Map
    * @param cmap Map that is final
    */
   public void setContextMap(final Map<String,Object> cmap)
   {
      this.contextMap = cmap;
   }

   /**
    * Get the Exception part of the audit
    * @return
    */
   public Exception getUnderlyingException()
   {
      return underlyingException;
   }

   /**
    * Set the exception on which an audit is happening
    * @param underlyingException
    */
   public void setUnderlyingException(Exception underlyingException)
   {
      this.underlyingException = underlyingException;
   }

   public String toString()
   {
      StringBuilder sbu  = new StringBuilder();
      sbu.append("[").append(auditLevel).append("]");
      sbu.append(dissectContextMap());
      return sbu.toString();
   }
}
```

The **`AuditEvent`** contains a *context map* and an optional *exception*. This information should be set by the process that uses the auditing framework. The **`AuditLevel`** defines the level of severity.

```
package org.jboss.security.audit;

/**
```

```
 *  Defines the Audit Levels of Severity
 */
public interface AuditLevel
{
   /** Denotes situations where there has been a server exception */
  String ERROR = "Error";

  /** Denotes situations when there has been a failed attempt */
  String FAILURE = "Failure";

  String SUCCESS = "Success";

  /** Information is passed into the audit logs */
  String INFO = "Info";
}
```

The **AuditContext** is a set of **AuditProvider**s. The interface for an **AuditProvider** looks like this:

```
package org.jboss.security.audit;

/**
 *  Audit Provider that can log audit events to an external
 *  log file
 */
public interface AuditProvider
{
   /**
    * Performs an audit of the event passed
    * A provider can log the audit as required.
    * @param ae audit event that holds information on the audit
    * @see AuditEvent
    */
  public void audit(AuditEvent ae);
}
```

CHAPTER 6. IDENTITY TRUST MANAGER

The Identity Trust Manager is an interface that enables pluggable identity trust validation. The **`IdentityTrustManager`** interface is obtained from the **`SecurityContext`**:

```
/*
  * JBoss, Home of Professional Open Source
  * Copyright 2007, JBoss Inc., and individual contributors as indicated
  * by the @authors tag. See the copyright.txt in the distribution for a
  * full listing of individual contributors.
  *
  * This is free software; you can redistribute it and/or modify it
  * under the terms of the GNU Lesser General Public License as
  * published by the Free Software Foundation; either version 2.1 of
  * the License, or (at your option) any later version.
  *
  * This software is distributed in the hope that it will be useful,
  * but WITHOUT ANY WARRANTY; without even the implied warranty of
  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  * Lesser General Public License for more details.
  *
  * You should have received a copy of the GNU Lesser General Public
  * License along with this software; if not, write to the Free
  * Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA
  * 02110-1301 USA, or see the FSF site: http://www.fsf.org.
  */
package org.jboss.security.identitytrust;

import org.jboss.security.BaseSecurityManager;
import org.jboss.security.SecurityContext;

//$Id$

/**
 *  Interface representing the trust manager
 *  for determining Identity
 *  @author Anil.Saldhana@redhat.com
 *  @version $Revision$
 */
public interface IdentityTrustManager extends BaseSecurityManager
{
   public enum TrustDecision{Permit,Deny,NotApplicable};

   /**
    * Make a Trust Decision on a security context
    * @param securityContext Security Context on which to act on
    * @return
    */
   TrustDecision isTrusted(SecurityContext securityContext);
}
```

The **`IdentityTrustManager`** interface can use the information in a security context to make a trust-based decision. The possible outcomes of a trust-based decision are:

- **`PERMIT`** — the framework trusts the identity completely, and bypasses authentication

- **`DENY`** — access is denied. This outcome acts as a virtual firewall.
- **`NOTAPPLICABLE`** — the framework cannot process the security context provided, and the process must be passed further (to the authentication phase).

Implementations can use **`IdentityTrustModule`**s to make trust-based decisions.

CHAPTER 7. IDENTITY TRUST CONTEXT

The **`IdentityTrust`** Context contains the **`IdentityTrustModule`**s.

```
package org.jboss.security.identitytrust;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import javax.security.auth.callback.CallbackHandler;

import org.jboss.security.SecurityContext;
import
org.jboss.security.identitytrust.IdentityTrustManager.TrustDecision;

//$Id$

/**
 *  Identity Trust Context that encloses multiple
 *  IdentityTrustModules, which make trust decisions
 *  @author Anil.Saldhana@redhat.com
 *  @since  Aug 2, 2007
 *  @version $Revision$
 */
public abstract class IdentityTrustContext
{
   protected TrustDecision DENY = TrustDecision.Deny;
   protected TrustDecision PERMIT = TrustDecision.Permit;
   protected TrustDecision NOTAPPLICABLE = TrustDecision.NotApplicable;

   /**
    * Provide a Security Context that the Trust Context
    * should come to a decision about. The security domain
    * driving the Security Context need not match the security
    * domain of the Trust Context.
    */
   protected SecurityContext securityContext;

   /**
    * Any Callback Handler
    */
   protected CallbackHandler callbackHandler;

   /**
    * Shared State between trust modules
    */
   protected Map sharedState = new HashMap();

   /**
    * Security Domain of the Identity Trust Context
    */
   protected String securityDomain;

   /**
```

```
     * List of Identity Trust Modules
     */
   protected List<IdentityTrustModule> modules = new
ArrayList<IdentityTrustModule>();

   /**
     * Control Flags for the individual modules
     */
   protected ArrayList controlFlags = new ArrayList();

   /**
     * Make a trust decision
     * @return {@link TrustDecision#Deny},{@link
TrustDecision#NotApplicable},
     *         {@link TrustDecision#Permit}
     * @throws IdentityTrustException
     */
   public abstract TrustDecision isTrusted() throws
IdentityTrustException;
}
```

The context contains a list of modules that handle trust-based decisions.

CHAPTER 8. SECURITY CLIENT

The **Security Client** is a generic client, which can perform plain username and password, Java Authentication and Authorization Security (JAAS), or Simple Authentication and Security Layer (SASL) services.

```
package org.jboss.security.client;

import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.login.LoginException;

/**
 *  Generic Security Client class <br/>
 *  <b>Basic Users:</b><br/>
 *  <p>Basic users will just use the methods that set the username and
credential
 *  @see {@link #setUserName(String)} and @see {@link
#setCredential(Object)} </p>
 *  <b>Intermediate Users:</b></br/>
 *  <p>You can specify usage of JAAS as the framework in the client
implementation.
 *  In this case, you will @see {@link #setLoginConfigName(String)} and
 *  @see #setCallbackHandler(CallbackHandler)</p>
 *  <b>Advanced Users:</b>
 *  <p>You will use the @see {@link #setSASLMechanism(String)} method</p>
 */
public abstract class SecurityClient
{
   protected Object userPrincipal = null;
   protected Object credential = null;
   protected CallbackHandler callbackHandler = null;
   protected String loginConfigName = null;
   protected String saslMechanism = null;
   protected String saslAuthorizationId = null;

   protected boolean jaasDesired = false;
   protected boolean saslDesired = false;

   /**
    * Login with the desired method
    * @throws LoginException
    */
   public void login() throws LoginException
   {
      if(jaasDesired)
         performJAASLogin();
      else
         if(saslDesired)
            peformSASLLogin();
         else
            performSimpleLogin();
   }

   /**
    * Log Out
    */
   public void logout()
   {
      setSimple(null,null);
      setJAAS(null,null);
      setSASL(null,null,null);
      cleanUp();
   }

   /**
    * Set the user name and credential for simple login (non-jaas, non-
sasl)
    * @param username (Can be null)
    * @param credential (Can be null)
    */
   public void setSimple(Object username, Object credential)
   {
      this.userPrincipal = username;
      this.credential = credential;
   }

   /**
    * Set the JAAS Login Configuration Name and Callback handler
    * @param configName can be null
    * @param cbh can be null
    */
   public void setJAAS(String configName, CallbackHandler cbh)
   {
      this.loginConfigName = configName;
      this.callbackHandler = cbh;
      clearUpDesires();
      this.jaasDesired = true;
   }

   /**
    * Set the mechanism and other parameters for SASL Client
    * @param mechanism
    * @param authorizationId
    * @param cbh
    */
   public void setSASL(String mechanism, String authorizationId,
         CallbackHandler cbh)
   {
      this.saslMechanism = mechanism;
      this.saslAuthorizationId = authorizationId;
      this.callbackHandler = cbh;
      clearUpDesires();
      this.saslDesired = true;
   }

   protected abstract void performJAASLogin() throws LoginException;
   protected abstract void peformSASLLogin();
   protected abstract void performSimpleLogin();

   /**
    * Provide an opportunity for client implementations to clean up
    */
   protected abstract void cleanUp();

   private void clearUpDesires()
   {
      jaasDesired = false;
      saslDesired = false;
   }
}
```

CHAPTER 9. JBOSSSX (DEFAULT IMPLEMENTATION)

The JBoss Security distribution contains JBossSX, the default implementation of the Security Service Provider Interface (SPI). This is driven by the **`jbosssx.jar`** JAR file. The chapters that follow describe the default implementation in detail.

CHAPTER 10. SECURITY CONFIGURATION

This chapter discusses the **SecurityConfiguration** class, which configures various managers in the Security Context. **SecurityConfiguration** has the following static methods:

```
package org.jboss.security.config;

import java.security.Key;
import java.security.spec.AlgorithmParameterSpec;
import java.util.HashMap;

/**
 *  Class that provides the Configuration for authentication,
 *  authorization, mapping information, etc.
 *  It also holds the information like JSSE keystores, keytypes and
 *  other crypto configuration
 */
public class SecurityConfiguration
{
   /**
    * Map of Application Policies keyed in by name
    */
   private static HashMap appPolicies = new HashMap();
   private static String cipherAlgorithm;
   private static int iterationCount;
   private static String salt;
   private static String keyStoreType;
   private static String keyStoreURL;
   private static String keyStorePass;
   private static String trustStoreType;
   private static String trustStorePass;
   private static String trustStoreURL;
   private static Key cipherKey;
   private static AlgorithmParameterSpec cipherSpec;

   public static void addApplicationPolicy(ApplicationPolicy aP)
   {
      if(aP == null)
         throw new IllegalArgumentException("application policy is null");
      appPolicies.put(aP.getName(), aP);
   }

   public static ApplicationPolicy getApplicationPolicy(String policyName)
   {
      return (ApplicationPolicy)appPolicies.get(policyName);
   }

   public static String getCipherAlgorithm()
   {
      return cipherAlgorithm;
   }

   public static void setCipherAlgorithm(String ca)
   {
      cipherAlgorithm = ca;
   }
```

```
    public static Key getCipherKey()
    {
       return cipherKey;
    }

    public static void setCipherKey(Key ca)
    {
       cipherKey = ca;
    }

    public static AlgorithmParameterSpec getCipherSpec()
    {
       return cipherSpec;
    }

    public static void setCipherSpec(AlgorithmParameterSpec aps)
    {
       cipherSpec = aps;
    }

    public static int getIterationCount()
    {
       return iterationCount;
    }

    /** Set the iteration count used with PBE based on the keystore
password.
     * @param count - an iteration count randomization value
     */
    public static void setIterationCount(int count)
    {
       iterationCount = count;
    }


    public static String getSalt()
    {
       return salt;
    }
    /** Set the salt used with PBE based on the keystore password.
     * @param salt - an 8 char randomization string
     */
    public static void setSalt(String s)
    {
       salt = s;
    }


    /** KeyStore implementation type being used.
    @return the KeyStore implementation type being used.
    */
    public static String getKeyStoreType()
    {
       return keyStoreType;
    }
    /** Set the type of KeyStore implementation to use. This is
    passed to the KeyStore.getInstance() factory method.
    */
    public static void setKeyStoreType(String type)
    {
       keyStoreType = type;
    }
    /** Get the KeyStore database URL string.
    */
    public static String getKeyStoreURL()
    {
       return keyStoreURL;
    }
    /** Set the KeyStore database URL string. This is used to obtain
    an InputStream to initialize the KeyStore.
    */
    public static void setKeyStoreURL(String storeURL)
    {
       keyStoreURL = storeURL;
    }

    /** Get the credential string for the KeyStore.
     */
     public static String getKeyStorePass()
     {
        return keyStorePass ;
     }

    /** Set the credential string for the KeyStore.
    */
    public static void setKeyStorePass(String password)
    {
       keyStorePass = password;
    }

   /** Get the type of the trust store
    * @return the type of the trust store
    */
   public static String getTrustStoreType()
   {
      return trustStoreType;
   }

   /** Set the type of the trust store
    * @param type - the trust store implementation type
    */
   public static void setTrustStoreType(String type)
   {
      trustStoreType = type;
   }

   /** Set the credential string for the trust store.
    */
    public static String getTrustStorePass()
    {
       return trustStorePass;
    }

   /** Set the credential string for the trust store.
   */
   public static void setTrustStorePass(String password)
   {
      trustStorePass = password;
   }

   /** Get the trust store database URL string.
    */
   public static String getTrustStoreURL()
   {
      return trustStoreURL;
   }

   /** Set the trust store database URL string. This is used to obtain
    an InputStream to initialize the trust store.
    */
   public static void setTrustStoreURL(String storeURL)
   {
      trustStoreURL = storeURL;
   }
 }
```

The **SecurityConfiguration** can hold a map of **ApplicationPolicy** objects that are identified with names associated with that of the Security Domain. The **SecurityConfiguration** class also provides commonly-used Java Cryptography Architecture (JCA) information where required.

The **ApplicationPolicy** class combines the **AuthenticationInfo**, **AuthorizationInfo**, **MappingInfo** and **AuditInfo** classes, which drive the configuration of individual context managers in the Security Context.

```
package org.jboss.security.config;

import org.jboss.security.auth.login.BaseAuthenticationInfo;

/**
 *  Application Policy Information Holder
 *  - Authentication
 *  - Authorization
 *  - Audit
 *  - Mapping
 */
public class ApplicationPolicy
{
   private String name;
   private BaseAuthenticationInfo authenticationInfo;
   private AuthorizationInfo authorizationInfo;
   private AuditInfo auditInfo;
   private MappingInfo roleMappingInfo;

   //Parent PolicyConfig
   private PolicyConfig policyConfig;

   public ApplicationPolicy(String theName)
   {
      if(theName == null)
         throw new IllegalArgumentException("name is null");
      this.name = theName;
   }

   public ApplicationPolicy(String theName,BaseAuthenticationInfo info)
   {
      this(theName);
      authenticationInfo = info;
   }

   public ApplicationPolicy(String theName,AuthorizationInfo info)
   {  :p
      this(theName);
      authorizationInfo = info;
   }

   public ApplicationPolicy(String theName,
        BaseAuthenticationInfo info, AuthorizationInfo info2)
   {
      this(theName);
      authenticationInfo = info;
      authorizationInfo = info2;
   }

   public BaseAuthenticationInfo getAuthenticationInfo()
   {
      return authenticationInfo;
   }

   public void setAuthenticationInfo(BaseAuthenticationInfo
authenticationInfo)
   {
      this.authenticationInfo = authenticationInfo;
   }

   public AuthorizationInfo getAuthorizationInfo()
   {
      return authorizationInfo;
   }

   public void setAuthorizationInfo(AuthorizationInfo authorizationInfo)
   {
      this.authorizationInfo = authorizationInfo;
   }

   public MappingInfo getRoleMappingInfo()
   {
      return roleMappingInfo;
   }

   public void setRoleMappingInfo(MappingInfo roleMappingInfo)
   {
      this.roleMappingInfo = roleMappingInfo;
   }
```

```
    public AuditInfo getAuditInfo()
    {
       return auditInfo;
    }

    public void setAuditInfo(AuditInfo auditInfo)
    {
       this.auditInfo = auditInfo;
    }

    public String getName()
    {
       return name;
    }

    public PolicyConfig getPolicyConfig()
    {
       return policyConfig;
    }

    public void setPolicyConfig(PolicyConfig policyConfig)
    {
       this.policyConfig = policyConfig;
    }
 }
```

`ApplicationPolicy` objects must be generated and established in the **`SecurityConfiguration`** by the system integrators, using JBossXB, JAXB, or other preferred mechanism.

CHAPTER 11. SECURITY CONTEXT FACTORY

The default implementation of the Security Service Provider Interface (SPI) inludes the **`SecurityContextFactory`**, which is used to construct both the **`SecurityContext`** and the **`SecurityContextUtil`** associated with it.

The **`SecurityContextFactory`** interface is as follows:

```
package org.jboss.security.plugins;

import java.security.Principal;

import javax.security.auth.Subject;

import org.jboss.security.SecurityContext;
import org.jboss.security.SecurityContextUtil;

//$Id$

/**
 *  Factory class used to create Security Context instances
 */
public class SecurityContextFactory
{
   /**
    * Create a security context
    * @param securityDomain Security Domain driving the context
    * @return
    */
   public static SecurityContext createSecurityContext(String
securityDomain)
   {
      JBossSecurityContext jsc = new JBossSecurityContext(securityDomain);
      return jsc;
   }

   /**
    * Create a security context
    * @param p Principal
    * @param cred Credential
    * @param s Subject
    * @param securityDomain SecurityDomain
    * @return
    * @see #createSecurityContext(String)
    */
   public static SecurityContext createSecurityContext(Principal p,
         Object cred,Subject s, String securityDomain)
   {
      JBossSecurityContext jsc = new JBossSecurityContext(securityDomain);
      jsc.getUtil().createSubjectInfo(p,cred,s);
      return jsc;
   }


   /**
    * Return an instance of the SecurityContextUtil
```

```
      * @return
      */
     public static SecurityContextUtil createUtil(SecurityContext sc)
     {
        return new JBossSecurityContextUtil(sc);
     }
  }
```

CHAPTER 12. SECURITY CONTEXT ASSOCIATION

The default implementation of the Security Service Provider Interface (SPI) includes the **`SecurityContextAssociation`** class, which includes a **`thread-local`** variable in which Security Context objects can be stored. System integrators are responsible for pushing and popping the Security Context to and from the reference in the call request path.

The **`SecurityContextAssociation`** interface resembles the following:

```
package org.jboss.security.plugins;

import org.jboss.security.SecurityContext;

/**
 *  Security Context association in a threadlocal
 */
public class SecurityContextAssociation
{
   private static ThreadLocal<SecurityContext> securityContextLocal
               = new ThreadLocal<SecurityContext>();

   public static void setSecurityContext(SecurityContext sc)
   {
      securityContextLocal.set(sc);
   }

   public static SecurityContext getSecurityContext()
   {
      return securityContextLocal.get();
   }

   public static void clearSecurityContext()
   {
      securityContextLocal.set(null);
   }
}
```

CHAPTER 13. JBOSS AUTHENTICATION MANAGER

The default implementation includes an authentication manager interface, **JBossAuthenticationManager**, which is driven by the Java Authentication and Authorization Service (JAAS). The interface is as follows:

```
package org.jboss.security.plugins;

import java.security.Principal;
import java.util.HashMap;
import java.util.Map;

import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;

import org.jboss.logging.Logger;
import org.jboss.security.AuthenticationManager;
import org.jboss.security.SecurityConstants;
import org.jboss.security.cache.JBossAuthenticationCache;
import org.jboss.security.cache.SecurityCache;
import org.jboss.security.cache.SecurityCacheException;


/**
 *  Default Implementation of the AuthenticationManager Interface
 */
public class JBossAuthenticationManager implements AuthenticationManager
{
   private static Logger log =
Logger.getLogger(JBossAuthenticationManager.class);

   protected String securityDomain =
SecurityConstants.DEFAULT_APPLICATION_POLICY;

   protected CallbackHandler callbackHandler = null;

   private ThreadLocal<Subject> subjectLocal = new ThreadLocal<Subject>();

   private SecurityCache<Principal> sCache = null;

   private boolean cacheValidation = false;

   public JBossAuthenticationManager(String sdomain, CallbackHandler cbh)
   {
   }

   /**
    * Create JBossAuthenticationManager
    * @param sdomain SecurityDomain
    * @param cbh CallbackHandler
    * @param initCapacity Initial Capacity for the internal Security Cache
    * @param loadFactor Load Factor for the internal Security Cache
    * @param level Concurrency Level for the internal Security Cach
    */
   public JBossAuthenticationManager(String sdomain, CallbackHandler cbh,
         int initCapacity, float loadFactor, int level)
   {
   }

   public void setSecurityCache(String className)
   {
   }

   /**
    * @see AuthenticationManager#getActiveSubject()
    */
   public Subject getActiveSubject()
   {
   }

   /**
    * @see AuthenticationManager#getSecurityDomain()
    */
   public String getSecurityDomain()
   {
   }

   /**
    * @see AuthenticationManager#getTargetPrincipal(Principal, Map)
    */
   public Principal getTargetPrincipal(Principal principal,
Map<String,Object> map)
   {
   }

   /**
    * @see AuthenticationManager#isValid(Principal, Object)
    */
   public boolean isValid(Principal principal, Object credential)
   {
   }

   /**
    * @see AuthenticationManager#isValid(Principal, Object, Subject)
    */
   public boolean isValid(Principal principal, Object credential, Subject
subject)
   {
   }
}
```

CHAPTER 14. JBOSS AUTHORIZATION MANAGER

The default implementation includes an authorization manager interface, **`JBossAuthorizationManager`**, which obtains fine-grained and pluggable authorization via authorization modules. This interface also provides support for the **`PolicyRegistration`** interface.

The interface resembles the following:

```
package org.jboss.security.plugins;
...
import static org.jboss.security.SecurityConstants.ROLES_IDENTIFIER;

/**
 *  Authorization Manager implementation
 */
public class JBossAuthorizationManager
implements AuthorizationManager,PolicyRegistration
{
   private String securityDomain;

   private Map contextIdToPolicy = new HashMap();
   protected boolean trace = log.isTraceEnabled();

   private CallbackHandler callbackHandler = null;

   public JBossAuthorizationManager(String securityDomainName)
   {
   }

   public JBossAuthorizationManager(String securityDomainName,
CallbackHandler cbh)
   {
   }

   /**
    * @see AuthorizationManager#authorize(Resource)
    */
   public int authorize(Resource resource) throws AuthorizationException
   {
      String SUBJECT_CONTEXT_KEY = SecurityConstants.SUBJECT_CONTEXT_KEY;
      Subject subject = null;
      try
      {
         subject = (Subject)
PolicyContext.getContext(SUBJECT_CONTEXT_KEY);
      }
      catch (PolicyContextException e)
      {
         log.error("Error obtaining AuthenticatedSubject:",e);
      }
      AuthorizationContext ac = new
JBossAuthorizationContext(this.securityDomain,subject,
            this.callbackHandler );
      return ac.authorize(resource);
   }
   /** Determines whether the Subject has a role (Principal) that matches
    *  a defined role name. This method obtains the "Roles" Group from the
    *  principal set of the currently-authenticated Subject (as determined
    *  by the SecurityAssociation.getSubject() method). It then creates a
    *  SimplePrincipal for each name in roleNames. If the Subject's role
    *  can be matched to a role in the "Roles" Group, then the user has
that
    *  role. The caller must therefore establish the correct
    *  SecurityAssociation Subject before this method is called.
    *  (This is no longer a side-effect of the isValid() call.)
    *
    *  @param principal - ignored. The current authenticated Subject
determines
    *  the active user and assigned user roles.
    *
    *  @param rolePrincipals - a Set of Principals for the roles to check.
    *
    *  @see java.security.acl.Group;
    *  @see Subject#getPrincipals()
    */
   public boolean doesUserHaveRole(Principal principal, Set
rolePrincipals)
   {
   }

   /** Determines whether the current Subject has a role (Principal) that
    *  matches one of the role names.
    *
    *  @see #doesUserHaveRole(Principal, Set)
    *
    *  @param principal - ignored. The currently-authenticated Subject
    *  determines the active user and assigned user roles.
    *  @param role - the application domain role that the principal is
    *  validated against.
    *  @return true if the active principal has the role, false otherwise.
    */
   public boolean doesUserHaveRole(Principal principal, Principal role)
   {
   }

   /** Returns the set of domain roles that were found by the current
active
    *  Subject "Roles" Group, in the Subject Principals Set.
    *
    *  @param principal - ignored. The current authenticated Subject
determines
    *  the active user and assigned user roles.
    *  @return The Set<Principal> for the application domain roles that
the
    *  principal has been assigned.
    */
   public Set getUserRoles(Principal principal)
   {
   }
   /** Checks that the indicated application domain role belongs to the
    *  group of roles assigned to the user. This handles the special
    *  AnybodyPrincipal and NobodyPrincipal independent of the Group
    *  implementation.
    *
    *  @param role - the application domain role required for access
    *  @param userRoles - the set of roles assigned to the user
    *  @return true - if role is in userRoles or an AnybodyPrincipal
instance, false
    *  if role is a NobodyPrincipal or no a member of userRoles
    */
   protected boolean doesRoleGroupHaveRole(Principal role, Group
userRoles)
   {
   }

   /**
    * @see PolicyRegistration#registerPolicy(String, URL)
    */
   public void registerPolicy(String contextID, URL location)
   {
   }

   /**
    * @see PolicyRegistration#registerPolicy(String, InputStream)
    */
   public void registerPolicy(String contextID, InputStream stream)
   {
   }

   /**
    * @see PolicyRegistration#deRegisterPolicy(String)
    */
   public void deRegisterPolicy(String contextID)
   {
   }

   /**
    * @see PolicyRegistration#getPolicy(String, Map)
    */
   public Object getPolicy(String contextID, Map contextMap)
   {
   }

   /**
    * @see AuthorizationManager#getTargetRoles(Principal, Map)
    */
   public Group getTargetRoles(Principal targetPrincipal, Map contextMap)
   {
      throw new RuntimeException("Not implemented");
   }
}
```

CHAPTER 15. EXAMPLE OF AUTHENTICATION

The following is a test case for the default implementation's **`JBossAuthenticationManager`**.

```
package org.jboss.test.authentication;

import java.security.Principal;
import java.util.HashMap;

import javax.security.auth.login.AppConfigurationEntry;
import javax.security.auth.login.Configuration;
import
javax.security.auth.login.AppConfigurationEntry.LoginModuleControlFlag;

import org.jboss.security.AuthenticationManager;
import org.jboss.security.SimplePrincipal;
import org.jboss.security.auth.callback.AppCallbackHandler;
import org.jboss.security.plugins.JBossAuthenticationManager;

import junit.framework.TestCase;

//$Id$

/**
 *  Unit tests for the JBossAuthenticationManager
 */
public class JBossAuthenticationManagerUnitTestCase extends TestCase
{
   @Override
   protected void setUp() throws Exception
   {
      super.setUp();
      establishSecurityConfiguration();
   }

   public void testSecurityDomain() throws Exception
   {
      AuthenticationManager am = new JBossAuthenticationManager("test1",
            new AppCallbackHandler("a","b".toCharArray()));
      assertEquals("test1", am.getSecurityDomain());
   }

   public void testLogin() throws Exception
   {
      Principal p = new SimplePrincipal("jduke");
      AppCallbackHandler acbh = new
AppCallbackHandler("jduke","theduke".toCharArray());
      AuthenticationManager am = new
JBossAuthenticationManager("test",acbh);
      assertTrue(am.isValid(p, "theduke"));
      assertNotNull("Subject is valid",am.getActiveSubject());
      assertTrue("Principal is present",
            am.getActiveSubject().getPrincipals().contains(p));
   }

   public void testUnsuccessfulLogin() throws Exception
   {
      Principal p = new SimplePrincipal("jduke");
      AppCallbackHandler acbh = new
AppCallbackHandler("jduke","bad".toCharArray());
      AuthenticationManager am = new
JBossAuthenticationManager("test",acbh);
      assertFalse(am.isValid(p, "bad"));
   }

   public void testSecurityCache() throws Exception
   {
      Principal p = new SimplePrincipal("jduke");
      AppCallbackHandler acbh = new
AppCallbackHandler("jduke","theduke".toCharArray());
      JBossAuthenticationManager am = new
JBossAuthenticationManager("test",acbh);
      assertFalse("Cache Validation is false", am.fromCache());
      assertTrue(am.isValid(p, "theduke"));
      assertNotNull("Subject is valid",am.getActiveSubject());
      assertTrue("Principal is present",
            am.getActiveSubject().getPrincipals().contains(p));
      assertFalse("Cache Validation is false", am.fromCache());
      assertTrue(am.isValid(p, "theduke"));
      assertTrue("Cache Validation", am.fromCache());
      assertTrue(am.isValid(p, "theduke"));
      assertTrue("Cache Validation", am.fromCache());

      acbh = new AppCallbackHandler("jduke","dummy".toCharArray());
      am = new JBossAuthenticationManager("test",acbh);
      assertFalse(am.isValid(p, "dummy"));
      assertFalse("Cache Validation is false", am.fromCache());
   }

   public void testSecurityCacheInjection() throws Exception
   {
      Principal p = new SimplePrincipal("jduke");
      AppCallbackHandler acbh = new
AppCallbackHandler("jduke","theduke".toCharArray());
      JBossAuthenticationManager am = new
JBossAuthenticationManager("test",acbh);
      am.setSecurityCache(TestSecurityCache.class.getName());
      assertFalse("Cache Validation is false", am.fromCache());
      assertTrue(am.isValid(p, "theduke"));
      assertNotNull("Subject is valid",am.getActiveSubject());
      assertTrue("Principal is present",
            am.getActiveSubject().getPrincipals().contains(p));
      assertFalse("Cache Validation is false", am.fromCache());
      assertTrue(am.isValid(p, "theduke"));
      assertTrue("Cache Validation", am.fromCache());
      assertTrue(am.isValid(p, "theduke"));
      assertTrue("Cache Validation", am.fromCache());

      acbh = new AppCallbackHandler("jduke","dummy".toCharArray());
      am = new JBossAuthenticationManager("test",acbh);
      assertFalse(am.isValid(p, "dummy"));
      assertFalse("Cache Validation is false", am.fromCache());
```

```
    }

    private void establishSecurityConfiguration()
    {
       Configuration.setConfiguration(new TestConfig());
    }

    public class TestConfig extends Configuration
    {
       @Override
       public AppConfigurationEntry[] getAppConfigurationEntry(String name)
       {
          HashMap map = new HashMap();
          map.put("usersProperties", "users.properties");
          map.put("rolesProperties", "roles.properties");
          String moduleName =
"org.jboss.security.auth.spi.UsersRolesLoginModule";
          AppConfigurationEntry ace = new AppConfigurationEntry(moduleName,
                LoginModuleControlFlag.REQUIRED, map);

          return new AppConfigurationEntry[]{ace};
       }

       @Override
       public void refresh()
       {
       }
    }
}
```

This test case should be used with the Java Authentication and Authorization Service (JAAS) configuration.

CHAPTER 16. EXAMPLE OF AUTHORIZATION

The following is a test case for the default implementation's **JBossAuthorizationManager**. It tests the web layer's authorization module. The default web layer module permits all actions, because decisions are handled by the Tomcat RealmBase. Note that **AuthorizationModule** is injected into the **AuthorizationInfo** class, which is part of the **ApplicationPolicy** object set in the **SecurityConfiguration**.

```
package org.jboss.test.authorization;

import java.security.Principal;
import java.security.acl.Group;
import java.util.HashMap;

import javax.security.auth.Subject;
import javax.security.jacc.PolicyContext;

import org.jboss.security.AuthorizationManager;
import org.jboss.security.SecurityConstants;
import org.jboss.security.SecurityContext;
import org.jboss.security.SimpleGroup;
import org.jboss.security.SimplePrincipal;
import org.jboss.security.authorization.ResourceKeys;
import org.jboss.security.authorization.config.AuthorizationModuleEntry;
import org.jboss.security.authorization.resources.WebResource;
import org.jboss.security.config.ApplicationPolicy;
import org.jboss.security.config.AuthorizationInfo;
import org.jboss.security.config.SecurityConfiguration;
import org.jboss.security.jacc.SubjectPolicyContextHandler;
import org.jboss.security.plugins.JBossAuthorizationManager;
import org.jboss.security.plugins.SecurityContextAssociation;
import org.jboss.security.plugins.SecurityContextFactory;
import org.jboss.test.authorization.xacml.TestHttpServletRequest;

import junit.framework.TestCase;

/**
 *  Unit test the JBossAuthorizationManager
 */
public class JBossAuthorizationManagerUnitTestCase extends TestCase
{
   private Principal p = new SimplePrincipal("jduke");
   private String contextID = "web.jar";
   private String uri = "/xacml-subjectrole/test";

   protected void setUp() throws Exception
   {
      super.setUp();
      setSecurityContext();
      setUpPolicyContext();
      setSecurityConfiguration();
   }

   public void testAuthorization() throws Exception
   {
      HashMap cmap = new HashMap();
      cmap.put(ResourceKeys.WEB_REQUEST, new
TestHttpServletRequest(p,"test", "get"));
      WebResource wr = new WebResource(cmap);
      AuthorizationManager am = new JBossAuthorizationManager("other");
      am.authorize(wr);//This should just pass as the default module
PERMITS all
   }

   private Group getRoleGroup()
   {
      Group gp = new SimpleGroup(SecurityConstants.ROLES_IDENTIFIER);
      gp.addMember(new SimplePrincipal("ServletUserRole"));
      return gp;
   }

   private void setSecurityContext()
   {
      Subject subj = new Subject();
      subj.getPrincipals().add(p);
      SecurityContext sc =
SecurityContextFactory.createSecurityContext("other");
      sc.getUtil().createSubjectInfo(p, "cred", subj);
      sc.getUtil().setRoles(getRoleGroup());
      SecurityContextAssociation.setSecurityContext(sc);
   }

   private void setUpPolicyContext() throws Exception
   {
      PolicyContext.setContextID(contextID);
      PolicyContext.registerHandler(SecurityConstants.SUBJECT_CONTEXT_KEY,
            new SubjectPolicyContextHandler(), true);
   }

   private void setSecurityConfiguration() throws Exception
   {
      String name =
"org.jboss.security.authorization.modules.web.WebAuthorizationModule";
      ApplicationPolicy ap = new ApplicationPolicy("other");
      AuthorizationInfo ai = new AuthorizationInfo("other");
      AuthorizationModuleEntry ame = new AuthorizationModuleEntry(name);
      ai.add(ame);
      ap.setAuthorizationInfo(ai);
      SecurityConfiguration.addApplicationPolicy(ap);
   }
}
```

CHAPTER 17. EXAMPLE OF AUDITING

The following is a test case for the default implementation's **`JBossAuditManager`**.

```
 package org.jboss.test.audit;

import org.jboss.security.SecurityContext;
import org.jboss.security.audit.AuditEvent;
import org.jboss.security.audit.AuditLevel;
import org.jboss.security.audit.AuditManager;
import org.jboss.security.audit.config.AuditProviderEntry;
import org.jboss.security.config.ApplicationPolicy;
import org.jboss.security.config.AuditInfo;
import org.jboss.security.config.SecurityConfiguration;
import org.jboss.security.plugins.SecurityContextFactory;

import junit.framework.TestCase;

/**
 *  Tests for the Auditing Layer
 */
public class AuditUnitTestCase extends TestCase
{
   @Override
   protected void setUp() throws Exception
   {
      super.setUp();
      setUpSecurityConfiguration();
   }

   /**
    * We invoke the AuditManager on the security context to audit
    * a particular AuditEvent. The AuditManager is configured with a
    * test logging provider, which places the event on a thread-local
    * of a static class. The test then checks the thread-local for
    * the audit event.
    */
   public void testAuditConfiguration()
   {
      SecurityContext sc =
SecurityContextFactory.createSecurityContext("test");
      AuditManager am = sc.getAuditManager();
      AuditEvent ae = new AuditEvent(AuditLevel.ERROR);
      am.audit(ae);

      //Now check that the Audit Event has been placed on the thread local
      //by our TestAuditProvider
      AuditEvent aev = (AuditEvent)
AuditTestAssociation.auditEventLocal.get();
      assertEquals("Audit events are the same", ae, aev);
   }


   private void setUpSecurityConfiguration()
   {
      String p = TestAuditProvider.class.getName();
```

```
        ApplicationPolicy ap = new ApplicationPolicy("test");
        AuditInfo auditInfo = new AuditInfo("test");
        AuditProviderEntry ape = new AuditProviderEntry(p);
        auditInfo.add(ape);
        ap.setAuditInfo(auditInfo);
        SecurityConfiguration.addApplicationPolicy(ap);
     }
  }
```

The **TestAuditProvider** class interface looks like this:

```
package org.jboss.test.audit;

import org.jboss.security.audit.AbstractAuditProvider;
import org.jboss.security.audit.AuditEvent;

//$Id$

/**
 *  Test Audit Provider that places the Audit Event on the
 *  thread local of AuditTestAssociation
 */
public class TestAuditProvider extends AbstractAuditProvider
{
   public TestAuditProvider()
   {
   }

   @Override
   public void audit(AuditEvent ae)
   {
      AuditTestAssociation.auditEventLocal.set(ae);
   }
}
```

The **AuditTestAssociation** class has a thread-local.

```
package org.jboss.test.audit;

/**
 *  A test class that stores a static thread-local
 */
public class AuditTestAssociation
{
   public static ThreadLocal auditEventLocal = new ThreadLocal();
}
```

APPENDIX A. REVISION HISTORY

Revision 5.1.0-110.33.400	**2013-10-31**	**Rüdiger Landmann**

Rebuild with publican 4.0.0

Revision 5.1.0-110.33	**July 24 2012**	**Ruediger Landmann**

Rebuild for Publican 3.0

Revision 5.1.0-110	**Wed 29 Sep 2010**	**Laura Bailey**

Changed version number in line with new versioning requirements.

Revised for JBoss Enterprise Application Platform 5.1.0.GA.