Red Hat JBoss Enterprise Application Platform 7.4

Security Architecture

Descriptions of high-level Red Hat JBoss Enterprise Application Platform security concepts and capabilities, along with the components that support them.

Last Updated: 2024-02-08

Red Hat JBoss Enterprise Application Platform 7.4 Security Architecture

Descriptions of high-level Red Hat JBoss Enterprise Application Platform security concepts and capabilities, along with the components that support them.

Abstract

This document focuses on the high-level concepts of security within JBoss EAP and what components exist to implement those concepts. This document focuses on what and why and much less on how, meaning specifics on how to configure a specific scenario will be housed in other documents. When completing this document, readers should have a solid conceptual understanding of the components of security within JBoss EAP, as well as how those components fit together.

Table of Contents

PROVIDING FEEDBACK ON JBOSS EAP DOCUMENTATION

To report an error or to improve our documentation, log in to your Red Hat Jira account and submit an issue. If you do not have a Red Hat Jira account, then you will be prompted to create an account.

Procedure

1. Click the following link to **create a ticket**.
2. Please include the **Document URL**, the **section number** and **describe the issue**.
3. Enter a brief description of the issue in the **Summary**.
4. Provide a detailed description of the issue or enhancement in the **Description**. Include a URL to where the issue occurs in the documentation.
5. Clicking **Submit** creates and routes the issue to the appropriate documentation team.

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message .

CHAPTER 1. OVERVIEW OF GENERAL SECURITY CONCEPTS

Before digging into how JBoss EAP handles security, it is important to understand a few basic security concepts.

1.1. AUTHENTICATION

Authentication refers to identifying a subject and verifying the authenticity of the identification. The most common authentication mechanism is a username and password combination, but other mechanisms, such as shared keys, smart cards or fingerprints, are also used for authentication. When in the context of Jakarta EE declarative security, the result of a successful authentication is called a principal.

1.2. AUTHORIZATION

Authorization refers to a way of specifying access rights or defining access policies. A system can then implement a mechanism to use those policies to permit or deny access to resources for the requester. In many cases, this is implemented by matching a principal with a set of actions or places they are allowed to access, sometimes referred to as a role.

1.3. AUTHENTICATION AND AUTHORIZATION IN PRACTICE

Although authentication and authorization are distinct concepts, they are often linked. Many modules written to handle authentication also handle authorization and vice versa.

Example

The application **MyPersonalSoapbox** provides the ability to post and view messages. Principals with the **Talk** role can post messages and view other posted messages. Users who have not logged in have the **Listen** role and can view posted messages. Suzy, Adam, and Bob use the application. Suzy and Bob can authenticate with their username and password, but Adam does not have a username and password yet. Suzy has the **Talk** role, but Bob does not have any roles, neither **Talk** nor **Listen**. When Suzy authenticates, she may post and view messages. When Adam uses **MyPersonalSoapbox**, he cannot log in, but he can see posted messages. When Bob logs in, he cannot post any messages, nor can he view any other posted messages.

Suzy is both authenticated and authorized. Adam has not authenticated, but he is authorized, with the **Listen** role, to view messages. Bob is authenticated but has no authorization and no roles.

1.4. ENCRYPTION

Encryption refers to encoding sensitive information by applying mathematical algorithms to it. Data is secured by converting, or encrypting, it to an encoded format. To read the data again, the encoded format must be converted, or decrypted, to the original format. Encryption can be applied to simple string data in files or databases or even on data sent across communications streams.

Examples of encryption include the following scenarios.

- LUKS can be used to encrypt Linux file system disks.
- The blowfish or AES algorithms can be used to encrypt data stored in Postgres databases.
- The HTTPS protocol encrypts all data via Secure Sockets Layer/Transport Layer Security, SSL/TLS, before transferring it from one party to another.

- When a user connects from one server to another using the Secure Shell, SSH protocol, all of the communication is sent in an encrypted tunnel.

1.5. SSL/TLS AND CERTIFICATES

SSL/TLS encrypts network traffic between two systems by using a symmetric key that is exchanged between and only known by those two systems. To ensure a secure exchange of the symmetric key, SSL/TLS uses Public Key Infrastructure (PKI), a method of encryption that uses a key pair. A key pair consists of two separate but matching cryptographic keys: a public key and a private key. The public key is shared with any party and is used to encrypt data; the private key is kept secret and is used to decrypt data that has been encrypted using the public key.

When a client requests a secure connection to exchange symmetric keys, a handshake phase occurs before secure communication can begin. During the SSL/TLS handshake, the server passes its public key to the client in the form of a certificate. The certificate contains the identity of the server, its URL, the public key of the server, and a digital signature that validates the certificate. The client validates the certificate and decides whether the certificate is trusted. If the certificate is trusted, the client generates the symmetric key for the SSL/TLS connection, encrypts it using the public key of the server, and sends it back to the server. The server uses its private key to decrypt the symmetric key. Further communication between the two machines over this connection is encrypted using the symmetric key.

There are two kinds of certificates: self-signed certificates and authority-signed certificates. A self-signed certificate uses its private key to sign itself; that signature is unverified because it is not connected to a chain of trust. An authority-signed certificate is a certificate that is issued to a party by a certificate authority, CA, and is signed by that CA, for example, VeriSign, CAcert, or RSA. The CA verifies the authenticity of the certificate holder.

Self-signed certificates can be faster and easier to generate and require less infrastructure to manage, but they can be difficult for clients to verify their authenticity because a third party has not confirmed their authenticity. This inherently makes the self-signed certificate less secure. Authority-signed certificates can take more effort to set up, but they are easier for clients to verify their authenticity. A chain of trust has been created because a third party has confirmed the authenticity of the certificate holder.

WARNING

Red Hat recommends that SSLv2, SSLv3, and TLSv1.0 be explicitly disabled in favor of TLSv1.1 or TLSv1.2 in all affected packages.

1.6. SINGLE SIGN-ON

Single sign-on (SSO) allows principals authenticated to one resource to implicitly authorize access to other resources. If a set of distinct resources is secured by SSO, a user is only required to authenticate the first time they access any of the secured resources. Upon successful authentication, the roles associated with the user are stored and used for authorization of all other associated resources. This allows the user to access any additional authorized resources without reauthenticating.

If the user logs out of a resource or a resource invalidates the session programmatically, all persisted authorization data is removed and the process starts over. In the case of a resource session timeout, the SSO session is not invalidated if there are other valid resource sessions associated with that user. SSO may be used for authentication and authorization on web applications and desktop applications. In some cases, an SSO implementation can integrate with both.

Within SSO, there are a few common terms used to describe different concepts and parts of the system.

Identity Management

Identity management (IDM) refers to the idea of managing principals and their associated authentication, authorization, and privileges across one or more systems or domains. The term identity and access management (IAM) is sometimes used to describe this same concept.

Identity Provider

An identity provider (IDP) is the authoritative entity responsible for authenticating an end user and asserting the identity for that user in a trusted fashion to trusted partners.

Identity Store

An identity provider needs an identity store to retrieve users' information to use during the authentication and authorization process. Identity stores can be any type of repository: a database, Lightweight Directory Access Protocol (LDAP), properties file, and so on.

Service Provider

A service provider (SP) relies on an identity provider to assert information about a user via an electronic user credential, leaving the service provider to manage access control and dissemination based on a trusted set of user credential assertions.

Clustered and Non-Clustered SSO

Non-clustered SSO limits the sharing of authorization information to applications on the same virtual host. There is also no resiliency in the event of a host failure. In a clustered SSO scenario, data can be shared between applications on multiple virtual hosts, which makes it resilient to failures. In addition, a clustered SSO is able to receive requests from a load balancer.

1.6.1. Third-Party SSO Implementations

Kerberos

Kerberos is a network authentication protocol for client-server applications. It uses secret-key symmetric cryptography to allow secure authentication across a non-secure network.

Kerberos uses security tokens called tickets. To use a secured service, users need to obtain a ticket from the ticket granting service (TGS) which is a service that runs on a server in their network. After obtaining the ticket, users request a Service Ticket (ST) from an authentication service (AS) which is another service running in the same network. Users then use the ST to authenticate to the desired service. The TGS and the AS run inside an enclosing service called the key distribution center (KDC).

Kerberos is designed to be used in a client-server desktop environment and is not usually used in web applications or thin client environments. However, many organizations use a Kerberos system for desktop authentication and prefer to reuse their existing system rather than create a second one for their web applications. Kerberos is an integral part of Microsoft’s Active Directory and is used in many Red Hat Enterprise Linux environments.

SPNEGO

Simple and protected GSS_API negotiation mechanism (SPNEGO) provides a mechanism for extending a Kerberos-based SSO environment for use in web applications.

When an application on a client computer, such as a web browser, attempts to access a protected page on a web server, the server responds that authorization is required. The application then requests an ST from the KDC. The application wraps the ticket in a request formatted for SPNEGO and sends it back to the web application via the browser. The web container running the deployed web application unpacks the request and authenticates the ticket. Access is granted upon successful authentication.

SPNEGO works with all types of Kerberos providers, including the Kerberos service within Red Hat Enterprise Linux and the Kerberos server, which is an integral part of Microsoft's Active Directory.

Microsoft's Active Directory

Active Directory (AD) is a directory service developed by Microsoft to authenticate users and computers in a Microsoft Windows domain. It comes as part of Windows Server. The computer running Windows Server controlling the domain is referred to as the domain controller. Red Hat Enterprise Linux can integrate with Active Directory domains as can Red Hat Identity Management, Red Hat JBoss Enterprise Application Platform, and other Red Hat products.

Active Directory relies on three core technologies that work together:

1. LDAP to store information about users, computers, passwords, and other resources
2. Kerberos to provide secure authentication over the network
3. Domain name service (DNS) to provide mappings between IP addresses and host names of computers and other devices in the network

1.6.2. Claims-Based Identity

One way to implement SSO is to use a claims-based identity system. A claims-based identity system allows systems to pass identity information but abstracts that information into two components: a claim and an issuer or authority. A claim is statement that one subject, such as a user, group, application, or organization, makes about another. That claim or set of claims is packaged into a token or set of tokens and issued by a provider. Claims-based identity allows individual secured resources to implement SSO without having to know everything about a user.

Security Token Service

A security token service (STS) is an authentication service that issues security tokens to clients for use when authenticating and authorizing users for secured applications, web services or Jakarta Enterprise Beans. A client attempting to authenticate against an application secured with STS, known as a service provider, will be redirected to a centralized STS authenticator and issued a token. If successful, that client will reattempt to access the service provider, providing their token along with the original request. That service provider will validate the token from the client with the STS and proceed accordingly. This same token may be reused by the client against other web services or Jakarta Enterprise Beans that are connected to the STS. The concept of a centralized STS that can issue, cancel, renew, and validate security tokens and specifies the format of security token request and response messages is known as **WS-Trust**.

Browser-Based SSO

In browser-based SSO, one or more web applications, known as service providers, connect to a centralized identity provider in a hub and spoke architecture. The IDP acts as the central source, or hub, for identity and role information by issuing claim statements in SAML tokens to service providers, or spokes. Requests may be issued when a user attempts to access a service provider or if a user attempts to authenticate directly with the identity provider. These are known as SP-initiated and IDP-initiated flows, respectively, and will both issue the same claim statements.

SAML

Security Assertion Markup Language (SAML) is a data format that allows two parties, usually an identity provider and a service provider, to exchange authentication and authorization information. A SAML token is a type of token issued by an STS or IDP; it can be used to enable SSO. A resource secured by SAML, SAML service provider, redirects users to the SAML identity provider, a type of STS or IDP, to obtain a valid SAML token before authenticating and authorizing that user.

Desktop-Based SSO

Desktop-based SSO enables service providers and desktop domains, for example Active Directory or Kerberos, to share a principal. In practice, this allows users to log in on their computer using their domain credentials and then have service providers reuse that principal during authentication, without having to reauthenticate, thus providing SSO.

1.7. LDAP

Lightweight Directory Access Protocol (LDAP) is a protocol for storing and distributing directory information across a network. This directory information includes information about users, hardware devices, access roles and restrictions, and other information.

In LDAP, the distinguished name (DN), uniquely identifies an object in a directory. Each distinguished name must have a unique name and location from all other objects, which is achieved using a number of attribute-value pairs (AVPs). The AVPs define information such as common names and organization unit. The combination of these values results in a unique string required by the LDAP.

Some common implementations of LDAP include Red Hat Directory Server, OpenLDAP, Active Directory, IBM Tivoli Directory Server, Oracle Internet Directory, and 389 Directory Server.

CHAPTER 2. HOW RED HAT JBOSS ENTERPRISE APPLICATION PLATFORM HANDLES SECURITY OUT OF THE BOX

There are three components that ship with JBoss EAP that relate to security:

- The Elytron Subsystem, introduced in JBoss EAP 7.1
- Core Management Authentication
- The Security Subsystem

These components are based on the general security concepts discussed in the Overview of General Security Concepts, but they also incorporate some JBoss EAP-specific concepts in their implementation.

2.1. CORE SERVICES, SUBSYSTEMS, AND PROFILES

JBoss EAP is built on the concept of modular class loading. Each API or service provided by JBoss EAP is implemented as a module, which is loaded and unloaded on demand. The core services are services that are always loaded on server startup and are required to be running prior to starting an additional subsystem.

A subsystem is a set of capabilities added to the core server by an extension. For example, different subsystems handle servlet processing, manage the Jakarta Enterprise Beans container, and provide Jakarta Transactions support.

A profile is a named list of subsystems, along with the details of each subsystem's configuration. A profile with a large number of subsystems results in a server with a large set of capabilities. A profile with a small, focused set of subsystems will have fewer capabilities but a smaller footprint. By default, JBoss EAP comes with several predefined profiles, for example **default**, **full**, **ha**, **full-ha**. In these profiles, the management interfaces and the associated security realms are loaded as core services.

2.2. MANAGEMENT INTERFACES

JBoss EAP offers two main management interfaces for interacting with and editing its configuration: the management console and the management CLI. Both interfaces expose the functionality of the core management of JBoss EAP. These interfaces offer two ways to access the same core management system.

The management console is a web-based administration tool for JBoss EAP. It may be used to start and stop servers, deploy and undeploy applications, tune system settings, and make persistent modifications to the server configuration. The management console also has the ability to perform administrative tasks, with live notifications when any changes require the server instance to be restarted or reloaded. In a managed domain, server instances and server groups in the same domain can be centrally managed from the management console of the domain controller.

The management CLI is a command-line administration tool for JBoss EAP. The management CLI may be used to start and stop servers, deploy and undeploy applications, configure system settings, and perform other administrative tasks. Operations can be performed in batch mode, allowing multiple tasks to be run as a group. The management CLI may also connect to the domain controller in a managed domain to execute management operations on the domain. The management CLI can perform all tasks that the web-based administration tool can perform as well as many other lower-level operations that are unavailable to the web-based administration tool.

NOTE

In addition to the clients that ship with JBoss EAP, other clients can be written to invoke the management interfaces over either the HTTP or native interfaces using the APIs included with JBoss EAP.

2.3. JAKARTA MANAGEMENT

Jakarta Management provides a way to remotely trigger JDK and application management operations. The management API of JBoss EAP is exposed as Jakarta Management managed beans. These managed beans are referred to as core MBeans, and access to them is controlled and filtered exactly the same as the underlying management API itself.

In addition to the management CLI and management console, Jakarta Management-exposed beans are an alternative mechanism to access and perform management operations.

2.4. ROLE-BASED ACCESS CONTROL

Role-Based Access Control (RBAC) is a mechanism for specifying a set of permissions for management users. It allows multiple users to share responsibility for managing JBoss EAP servers without each of them requiring unrestricted access. By providing a separation of duties for management users, JBoss EAP makes it easy for an organization to spread responsibility between individuals or groups without granting unnecessary privileges. This ensures the maximum possible security of your servers and data while still providing flexibility for configuration, deployment, and management.

RBAC in JBoss EAP works through a combination of role permissions and constraints. Seven predefined roles are provided, with every role having different fixed permissions. Each management user is assigned one or more roles that specify what the user is permitted to do when managing the server.

RBAC is disabled by default for JBoss EAP.

Standard Roles

JBoss EAP provides seven predefined user roles: **Monitor**, **Operator**, **Maintainer**, **Deployer**, **Auditor**, **Administrator**, and **SuperUser**. Each role has a different set of permissions and is designed for specific use cases. The **Monitor**, **Operator**, **Maintainer**, **Administrator**, and **SuperUser** roles build successively upon each other, with each having more permissions than the previous. The **Auditor** and **Deployer** roles are similar to the **Monitor** and **Maintainer** roles, respectively, but they have some special permissions and restrictions.

Monitor

Users of the **Monitor** role have the fewest permissions and can only read the current configuration and state of the server. This role is intended for users who need to track and report on the performance of the server. **Monitor** cannot modify server configuration, nor can they access sensitive data or operations.

Operator

The **Operator** role extends the **Monitor** role by adding the ability to modify the runtime state of the server. This means that **Operator** can reload and shutdown the server as well as pause and resume Jakarta Messaging destinations. The **Operator** role is ideal for users who are responsible for the physical or virtual hosts of the application server so they can ensure that servers can be shutdown and restarted correctly when need be. **Operator** cannot modify server configuration or access sensitive data or operations.

Maintainer

The **Maintainer** role has access to view and modify the runtime state and all configurations except sensitive data and operations. The **Maintainer** role is the general purpose role that does not have access to sensitive data and operation. The **Maintainer** role allows users to be granted almost complete access to administer the server without giving those users access to passwords and other sensitive information. **Maintainer** cannot access sensitive data or operations.

Administrator

The **Administrator** role has unrestricted access to all resources and operations on the server except the audit logging system. The **Administrator** role has access to sensitive data and operations. This role can also configure the access control system. The **Administrator** role is only required when handling sensitive data or configuring users and roles. **Administrator** cannot access the audit logging system and cannot change themselves to the **Auditor** or **SuperUser** role.

SuperUser

The **SuperUser** role does not have any restrictions, and it has complete access to all resources and operations of the server, including the audit logging system. If RBAC is disabled, all management users have permissions equivalent to the **SuperUser** role.

Deployer

The **Deployer** role has the same permissions as the **Monitor**, but it can modify the configuration and state for deployments and any other resource type enabled as an application resource.

Auditor

The **Auditor** role has all the permissions of the **Monitor** role and can also view, but not modify, sensitive data. It has full access to the audit logging system. The **Auditor** role is the only role besides **SuperUser** that can access the audit logging system. **Auditor** cannot modify sensitive data or resources. Only read access is permitted.

Permissions

Permissions determine what each role can do; not every role has every permission. Notably, **SuperUser** has every permission and **Monitor** has the least. Each permission can grant read and/or write access for a single category of resources. The categories are runtime state, server configuration, sensitive data, the audit log, and the access control system.

Table 2.1. Permissions of Each Role for Monitor, Operator, Maintainer and Deployer

	Monitor	Operator	Maintainer	Deployer
Read Config and State	✘	✘	✘	✘
Read Sensitive Data [2]				
Modify Sensitive Data [2]				
Read/Modify Audit Log				
Modify Runtime State		✘	✘	✘[1]
Modify Persistent Config			✘	✘[1]
Read/Modify Access Control				

[1] Permissions are restricted to application resources.

Table 2.2. Permissions of Each Role for Auditor, Administration and SuperUser

	Auditor	Administrator	SuperUser
Read Config and State	✘	✘	✘
Read Sensitive Data [2]	✘	✘	✘
Modify Sensitive Data [2]		✘	✘
Read/Modify Audit Log	✘		✘
Modify Runtime State		✘	✘
Modify Persistent Config		✘	✘
Read/Modify Access Control		✘	✘

[2] Which resources are considered to be sensitive data are configured using sensitivity.

Constraints

Constraints are named sets of access-control configuration for a specified list of resources. The RBAC system uses the combination of constraints and role permissions to determine if any specific user can perform a management action.

Constraints are divided into three classifications.

Application Constraints

Application constraints define sets of resources and attributes that can be accessed by Deployer users. By default, the only enabled application constraint is core, which includes deployments and deployment overlays. Application constraints are also included, but not enabled by default, for data sources, logging, mail, messaging, naming, resource adapters, and security. These constraints allow Deployer users to not only deploy applications, but also configure and maintain the resources that are required by those applications.

Sensitivity Constraints

Sensitivity constraints define sets of resources that are considered sensitive. A sensitive resource is generally one that is either secret, like a password, or one that will have serious impact on the operation of the server, like networking, JVM configuration, or system properties. The access control system itself is also considered sensitive. The only roles permitted to write to sensitive resources are Administrator and SuperUser. The Auditor role is only able to read sensitive resources. No other roles have access.

Vault Expression Constraint

The vault expression constraint defines if reading and writing vault expressions are considered sensitive operations. By default, reading and writing vault expressions are sensitive operations.

2.4.1. Configuring RBAC

If RBAC is already enabled, you must have the **SuperUser** or **Administrator** role assigned to make configuration changes at the user or group level.

Procedure

1. Enable RBAC using the following command:

```
/core-service=management/access=authorization:write-attribute(name=provider,value=rbac)
```

2. As a **SuperUser** or an **Administrator** of JBoss EAP, configure RBAC:

   - To add one of the supported roles, such as the **Monitor** role that has read-only access, use the following command:

     ```
     /core-service=management/access=authorization/role-mapping=Monitor:add()
     ```

     NOTE

     For more information about the **Monitor** role and other supported roles that you can add, see Role-Based Access Control.

   - To add a user to a specific role, such as the **Monitor** role, use the following command:

     ```
     /core-service=management/access=authorization/role-mapping=Monitor/include=user-timRO:add(name=timRO,type=USER)
     ```

   - To add a group to a specific role, such as the **Monitor** role, use the following command:

     ```
     /core-service=management/access=authorization/role-mapping=Monitor/include=group-LDAP_MONITORS:add(name=LDAP_MONITORS, type=GROUP)
     ```

   - To exclude users or groups from a specific role, use the following command:

     ```
     /core-service=management/access=authorization/role-mapping=Monitor/exclude=group-LDAP_MONITORS:add(name=LDAP_, type=GROUP)
     ```

3. Restart the server or the host to enable it to operate with RBAC configuration:

   - To restart the host machine, use the following command:

     ```
     reload --host=master
     ```

   - To restart the server in the standalone mode, use the following command:

     ```
     reload
     ```

2.5. DECLARATIVE SECURITY AND JAKARTA AUTHENTICATION

Declarative security is a method to separate security concerns from application code by using the container to manage security. The container provides an authorization system based on either file permissions or users, groups, and roles. This approach is usually superior to programmatic security, which gives the application itself all of the responsibility for security. JBoss EAP provides declarative security by using security domains in the **security** subsystem.

Jakarta Authentication is a declarative security API comprising a set of Java packages designed for user authentication and authorization. The API is a Java implementation of the standard Pluggable Authentication Modules (PAM) framework. It extends the Jakarta EE access control architecture to support user-based authorization. The JBoss EAP **security** subsystem is actually based on the Jakarta Authentication API.

Because Jakarta Authentication is the foundation for the **security** subsystem, authentication is performed in a pluggable fashion. This permits Java applications to remain independent from underlying authentication technologies, such as Kerberos or LDAP, and allows the security manager to work in different security infrastructures. Integration with a security infrastructure is achievable without changing the security manager implementation. Only the configuration of the authentication stack that Jakarta Authentication uses needs to be changed.

2.6. ELYTRON SUBSYSTEM

The **elytron** subsystem was introduced in JBoss EAP 7.1. It is based on the WildFly Elytron project, which is a security framework used to unify security across the entire application server. The **elytron** subsystem enables a single point of configuration for securing both applications and the management interfaces. WildFly Elytron also provides a set of APIs and SPIs for providing custom implementations of functionality and integrating with the **elytron** subsystem.

In addition, there are several other important features of WildFly Elytron:

- Stronger authentication mechanisms for HTTP and SASL authentication.
- Improved architecture that allows for **SecurityIdentities** to be propagated across security domains. This ensures transparent transformation that is ready to be used for authorization. This transformation takes place using configurable role decoders, role mappers, and permission mappers.
- Centralized point for SSL/TLS configuration including cipher suites and protocols.
- SSL/TLS optimizations such as eager **SecureIdentity** construction and closely tying authorization to establishing an SSL/TLS connection. Eager **SecureIdentity** construction eliminates the need for a **SecureIdentity** to be constructed on a per-request basis. Closely tying authentication to establishing an SSL/TLS connection enables permission checks to happen *BEFORE* the first request is received.
- A secure credential store that replaces the previous vault implementation to store plain text strings.

The new **elytron** subsystem exists in parallel to the legacy **security** subsystem and legacy core management authentication. Both the legacy and Elytron methods can be used for securing the management interfaces as well as providing security for applications.

IMPORTANT

The architectures of Elytron and the legacy security subsystem that is based on PicketBox are very different. With Elytron, an attempt was made to create a solution that allows you to operate in the same security environments in which you currently operate; however, this does *not* mean that every PicketBox configuration option has an equivalent configuration option in Elytron.

If you are not able to find information in the documentation to help you achieve similar functionality using Elytron that you had when using the legacy security implementation, you can find help in one of the following ways.

- If you have a Red Hat Development subscription , you have access to Support Cases, Solutions, and Knowledge Articles on the Red Hat Customer Portal. You can also open a case with Technical Support and get help from the WildFly community as described below.
- If you do not have a Red Hat Development subscription, you can still access Knowledge Articles on the Red Hat Customer Portal. You can also join the user forums and live chat to ask questions of the WildFly community. The WildFly community offerings are actively monitored by the Elytron engineering team.

2.6.1. Core Concepts and Components

The concept behind the architecture and design of the **elytron** subsystem is using smaller components to assemble a full security policy. By default, JBoss EAP provides implementations for many components, but the **elytron** subsystem also allows you to provide specialized, custom implementations.

Each implementation of a component in the 'elytron' subsystem is handled as an individual capability. This means that different implementations can be mixed, matched and modeled using distinct resources.

2.6.1.1. Capabilities and Requirements

A capability is a piece of functionality used in JBoss EAP and is exposed using the management layer. One capability can depend on other capabilities and this dependency is mediated by the management layer. Some capabilities are provided automatically by JBoss EAP, but the full set of available capabilities available at runtime are determined using the JBoss EAP configuration. The management layer validates that all capabilities required by other capabilities are present during server startup as well as when any configuration changes are made. Capabilities integrate with JBoss Modules and extensions, but they are all distinct concepts.

In addition to registering other capabilities it depends on, a capability must also register a set of requirements related to those capabilities. A capability can specify the following types of requirements:

Hard requirements

A capability is depended on another capability to function, therefore it must always be present.

Optional requirements

An optional aspect of a capability depends on another capability, which can or can not be enabled. Therefore the requirement cannot be determined or known until the configuration is analyzed.

Runtime-only requirements

A capability will check if the required capability exists at runtime. If the required capability is present it will be used. If the required capability is not present, it will not be used.

You can find more information on capabilities and requirements in the WildFly documentation.

2.6.1.2. APIs, SPIs and Custom Implementations

Elytron provides a set of security APIs and SPIs so that other subsystems and consumers can use them directly, which reduces integration overhead. While the majority of users will use the provided functionality of JBoss EAP, the Elytron APIs and SPIs can also be used by custom implementations to replace or extend Elytron functionality.

2.6.1.3. Security Domains

A security domain is the representation of a security policy which is backed by one or more security realms and a set of resources that perform transformations. A security domain produces a **SecurityIdentity**. The **SecurityIdentity** is used by other resources that perform authorizations, such as an application. A **SecurityIdentity** is the representation of the current user, which is based on the raw **AuthorizationIdentity** and its associated roles and permissions.

You can also configure a security domain to allow *inflow* of a **SecurityIdentity** from another security domain. When an identity is *inflowed*, it retains its original raw **AuthorizationIdentity**, and a new set of roles and permissions are assigned to it, creating a new **SecurityIdentity**.

IMPORTANT

Deployments are limited to using one Elytron security domain per deployment. Scenarios that may have required multiple legacy security domains can now be accomplished using a single Elytron security domain.

2.6.1.4. Security Realms

Security realms provide access to an identity store and are used to obtain credentials. These credentials allow authentication mechanisms to obtain the raw **AuthorizationIdentity** for performing authentication. They also allow authentication mechanisms to perform verification when doing validation of evidence.

You can associate one or more security realms with a security domain. Some security realm implementations also expose an API for modifications, meaning the security realm can make updates to the underlying identity store.

2.6.1.5. Role Decoders

A role decoder is associated with a security domain and is used to decode the current user's roles. The role decoder takes the raw **AuthorizationIdentity** returned from the security realm and converts its attributes into roles.

2.6.1.6. Role Mappers

A role mapper applies a role modification to an identity. This can range from normalizing the format of the roles to adding or removing specific roles. A role mapper can be associated with both security realms as well as security domains. In cases where a role mapper is associated with a security realm, the role mapping will be applied at the security realm level before any transformations, such as role decoding or additional role mapping, occur at the security domain level. If a role mapper and another transformation, such as a role decoder, are both configured in a security domain, all other transformations are performed before the role mapper is applied.

2.6.1.7. Permission Mappers

A permission mapper is associated with a security domain and assigns a set of permissions to a **SecurityIdentity**.

2.6.1.8. Principal Transformers

A principal transformer can be used in multiple locations within the **elytron** subsystem. A principal transformer can transform or map a name to another name.

2.6.1.9. Principal Decoders

A principal decoder can be used in multiple locations within the **elytron** subsystem. A principal decoder converts an identity from a **Principal** to a string representation of the name. For example, the **X500PrincipalDecoder** allows you to convert an **X500Principal** from a certificate's distinguished name to a string representation.

2.6.1.10. Realm Mappers

A realm mapper is associated with a security domain and is used in cases where a security domain has multiple security realms configured. The realm mappers can be also associated with **mechanism** or **mechanism-realm** of **http-authentication-factory** and **sasl-authentication-factory**. The realm mapper uses the name provided during authentication to choose a security realm for authentication and obtaining the raw **AuthorizationIdentity**.

2.6.1.11. Authentication Factories

An authentication factory is a representation of an authentication policy. An authentication is associated with security domain, mechanism factory, and a mechanism selector. The security domain provides the **SecurityIdentity** to be authenticated, the mechanism factory provides the server-side authentication mechanisms, and the mechanism selector is used to obtain configuration specific to the mechanism selected. The mechanism selector can include information about realm names a mechanism should present to a remote client plus additional principal transformers and realm mappers to use during the authentication process.

2.6.1.12. KeyStores

A **key-store** is the definition of a keystore or truststore including the type of keystore, its location, and credential for accessing it.

2.6.1.13. Key Managers

A **key-manager** references a **key-store** and is used in conjunction with an SSL context.

2.6.1.14. Trust Managers

A **trust-manager** references as truststore, which is defined in a **key-store**, and is used in conjunction with an SSL context, usually for two-way SSL/TLS.

2.6.1.15. SSL Context

The SSL context defined within the **elytron** subsystem is a **javax.net.ssl.SSLContext** meaning it can be used by anything that uses an SSL context directly. In addition to the usual configuration for an SSL context it is possible to configure additional items such as cipher suites and protocols. The SSL context will wrap any additional items that are configured.

2.6.1.16. Secure Credential Store

The previous vault implementation used for plain text string encryption has been replaced with a newly designed credential store. In addition to the protection for the credentials it stores, the credential store is used to store plain text strings.

2.6.2. Elytron Authentication Process

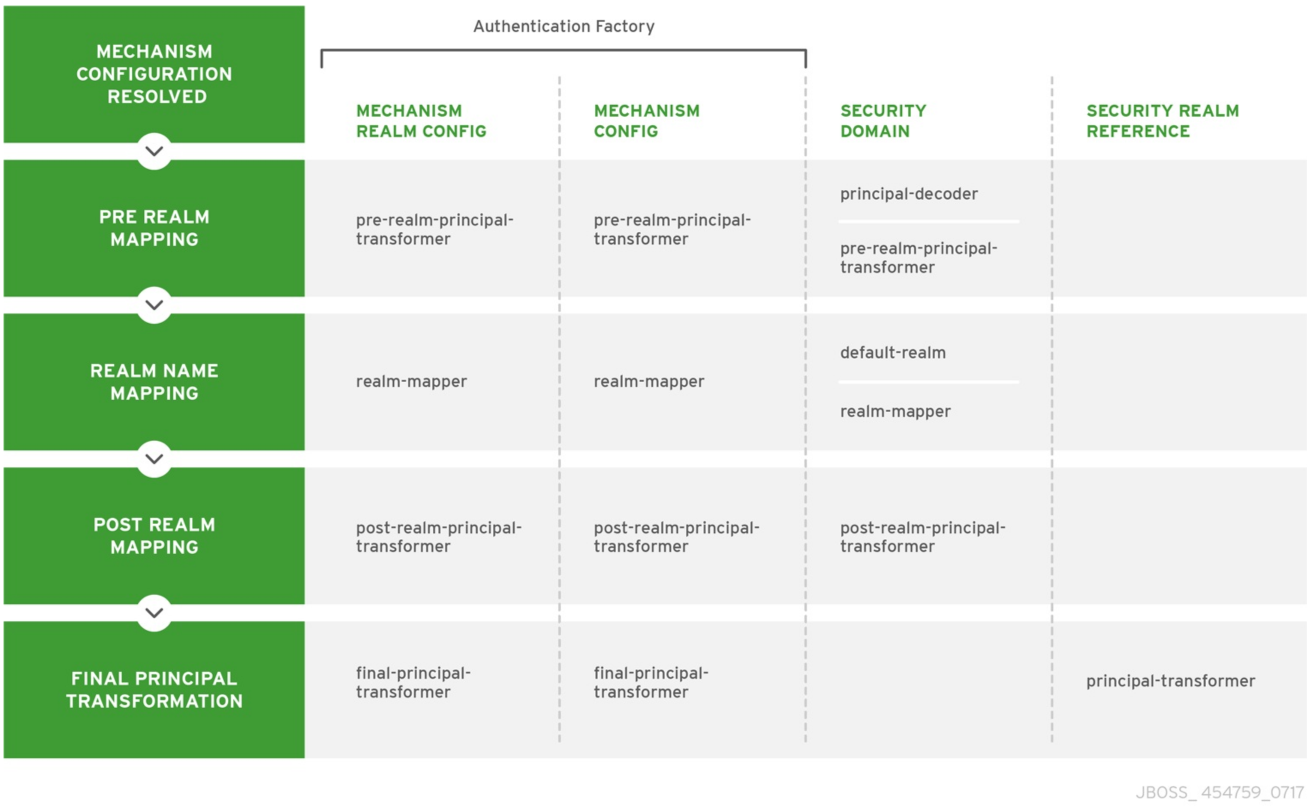
Multiple principal transformers, realm mappers, and a principal decoder can be defined within the **elytron** subsystem. The following sections discuss how these components function during the authentication process, and how principals are mapped to the appropriate security realm.

When a principal is authenticated it performs the following steps, in order:

1. The appropriate mechanism configuration is determined and configured.
2. The incoming principal is mapped into a **SecurityIdentity**.
3. This **SecurityIdentity** is used to determine the appropriate security realm.
4. After the security realm has been identified the principal is transformed again.
5. One final transformation occurs to allow for mechanism-specific transformations.

The following image demonstrates these steps, highlighted in the left column, along with showing the components used in each phase.

Figure 2.1. Elytron Authentication Process


Authentication Factory
MECHANISM CONFIGURATION RESOLVED
MECHANISM REALM CONFIG
MECHANISM CONFIG
SECURITY DOMAIN
SECURITY REALM REFERENCE
PRE REALM MAPPING
pre-realm-principal-transformer
pre-realm-principal-transformer
principal-decoder
pre-realm-principal-transformer
REALM NAME MAPPING
realm-mapper
realm-mapper
default-realm
realm-mapper
POST REALM MAPPING
post-realm-principal-transformer
post-realm-principal-transformer
post-realm-principal-transformer
FINAL PRINCIPAL TRANSFORMATION
final-principal-transformer
final-principal-transformer
principal-transformer
JBOSS_454759_0717

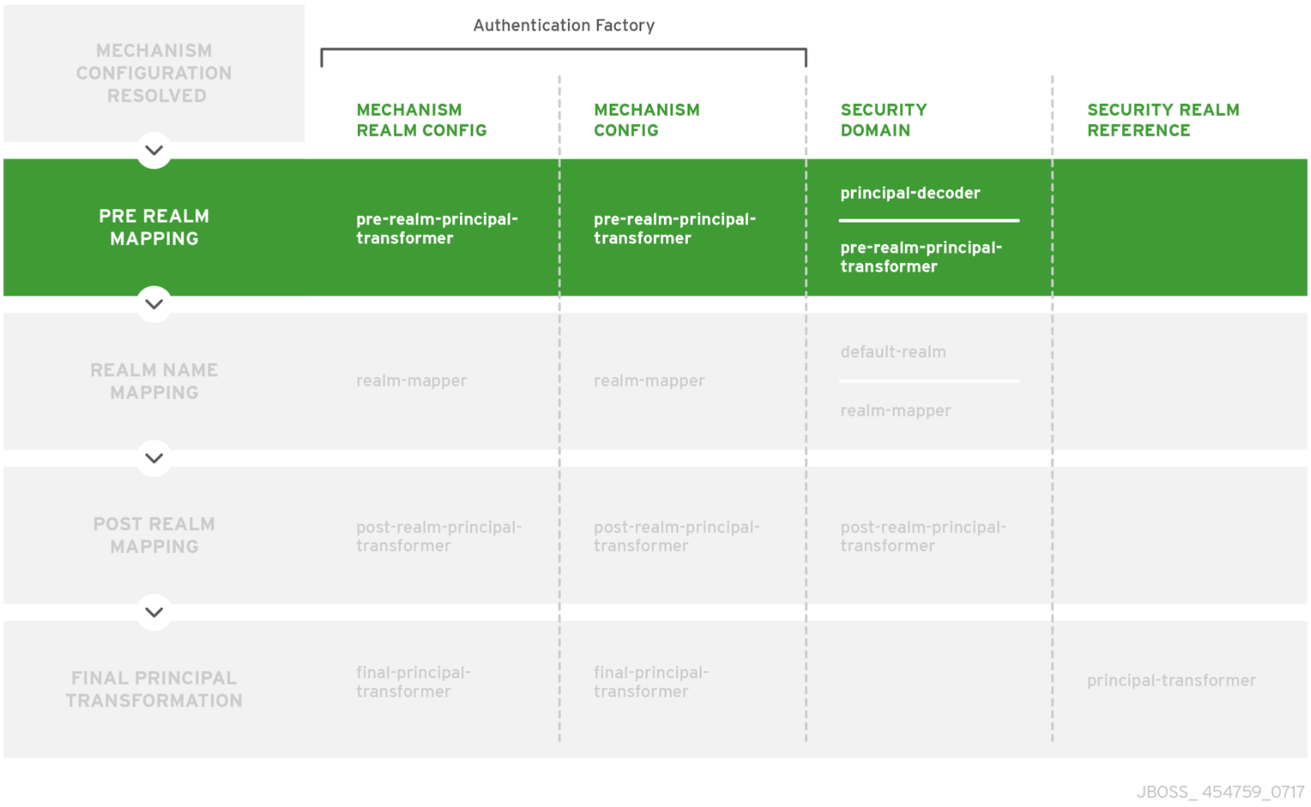
Pre-realm Mapping

During pre-realm mapping the authenticated principal is mapped to a **SecurityIdentity**, a form that can identify which security realm should be used, and will contain a single **Principal** that represents the authenticated information. Principal transformers and principal decoders are called in the following order:

1. Mechanism Realm - **pre-realm-principal-transformer**
2. Mechanism Configuration - **pre-realm-principal-transformer**
3. Security Domain - **principal-decoder** and **pre-realm-principal-transformer**

If this procedure results in a null principal, then an error will be thrown and authentication will terminate.

Figure 2.2. Pre-realm Mapping



MECHANISM CONFIGURATION RESOLVED
Authentication Factory
MECHANISM REALM CONFIG
MECHANISM CONFIG
SECURITY DOMAIN
SECURITY REALM REFERENCE
PRE REALM MAPPING
pre-realm-principal-transformer
pre-realm-principal-transformer
principal-decoder
pre-realm-principal-transformer
REALM NAME MAPPING
realm-mapper
realm-mapper
default-realm
realm-mapper
POST REALM MAPPING
post-realm-principal-transformer
post-realm-principal-transformer
post-realm-principal-transformer
FINAL PRINCIPAL TRANSFORMATION
final-principal-transformer
final-principal-transformer
principal-transformer
JBOSS_ 454759_0717

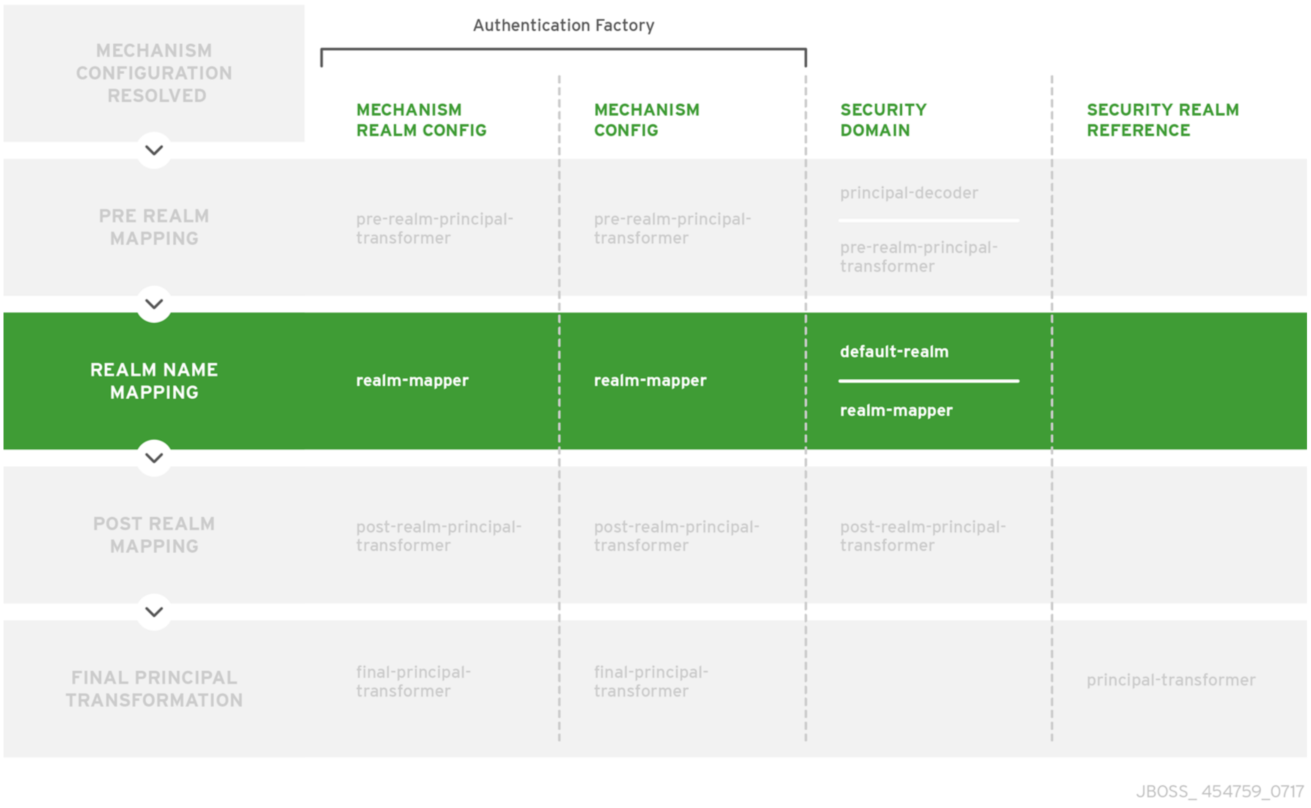
Realm Name Mapping

Once a mapped principal has been obtained, a security realm is identified which will be used to load the identity. At this point the realm name is the name defined by the security realm as referenced by the security domain, and is not yet the mechanism realm name. The configuration will look for a security realm name in the following order:

1. Mechanism Realm - **realm-mapper**
2. Mechanism Configuration - **realm-mapper**
3. Security Domain - **realm-mapper**

If the **RealmMapper** returns null, or if no mapper is available, then the **default-realm** on the security domain will be used.

Figure 2.3. Realm Name Mapping


Authentication Factory
MECHANISM CONFIGURATION RESOLVED
MECHANISM REALM CONFIG
MECHANISM CONFIG
SECURITY DOMAIN
SECURITY REALM REFERENCE
PRE REALM MAPPING
pre-realm-principal-transformer
pre-realm-principal-transformer
principal-decoder
pre-realm-principal-transformer
REALM NAME MAPPING
realm-mapper
realm-mapper
default-realm
realm-mapper
POST REALM MAPPING
post-realm-principal-transformer
post-realm-principal-transformer
post-realm-principal-transformer
FINAL PRINCIPAL TRANSFORMATION
final-principal-transformer
final-principal-transformer
principal-transformer
JBOSS_ 454759_0717

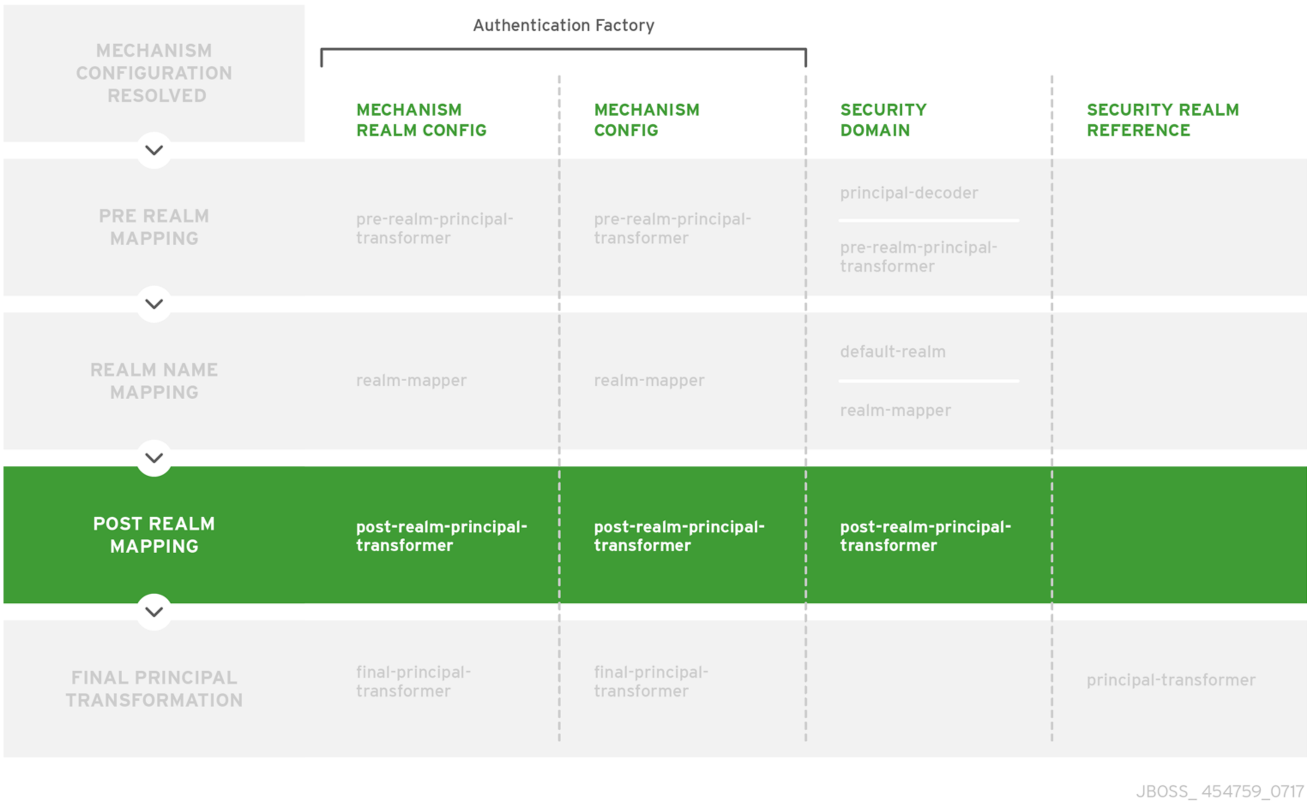
Post-realm Mapping

After a realm has been identified, the principal goes through another round of transformation. Transformers are called in the following order:

1. Mechanism Realm - **post-realm-principal-transformer**
2. Mechanism Configuration - **post-realm-principal-transformer**
3. Security Domain - **post-realm-principal-transformer**

If this procedure results in a null principal, then an error will be thrown and authentication will terminate.

Figure 2.4. Post-realm Mapping


MECHANISM CONFIGURATION RESOLVED
Authentication Factory
MECHANISM REALM CONFIG
MECHANISM CONFIG
SECURITY DOMAIN
SECURITY REALM REFERENCE
PRE REALM MAPPING
pre-realm-principal-transformer
pre-realm-principal-transformer
principal-decoder
pre-realm-principal-transformer
REALM NAME MAPPING
realm-mapper
realm-mapper
default-realm
realm-mapper
POST REALM MAPPING
post-realm-principal-transformer
post-realm-principal-transformer
post-realm-principal-transformer
FINAL PRINCIPAL TRANSFORMATION
final-principal-transformer
final-principal-transformer
principal-transformer
JBOSS_ 454759_0717

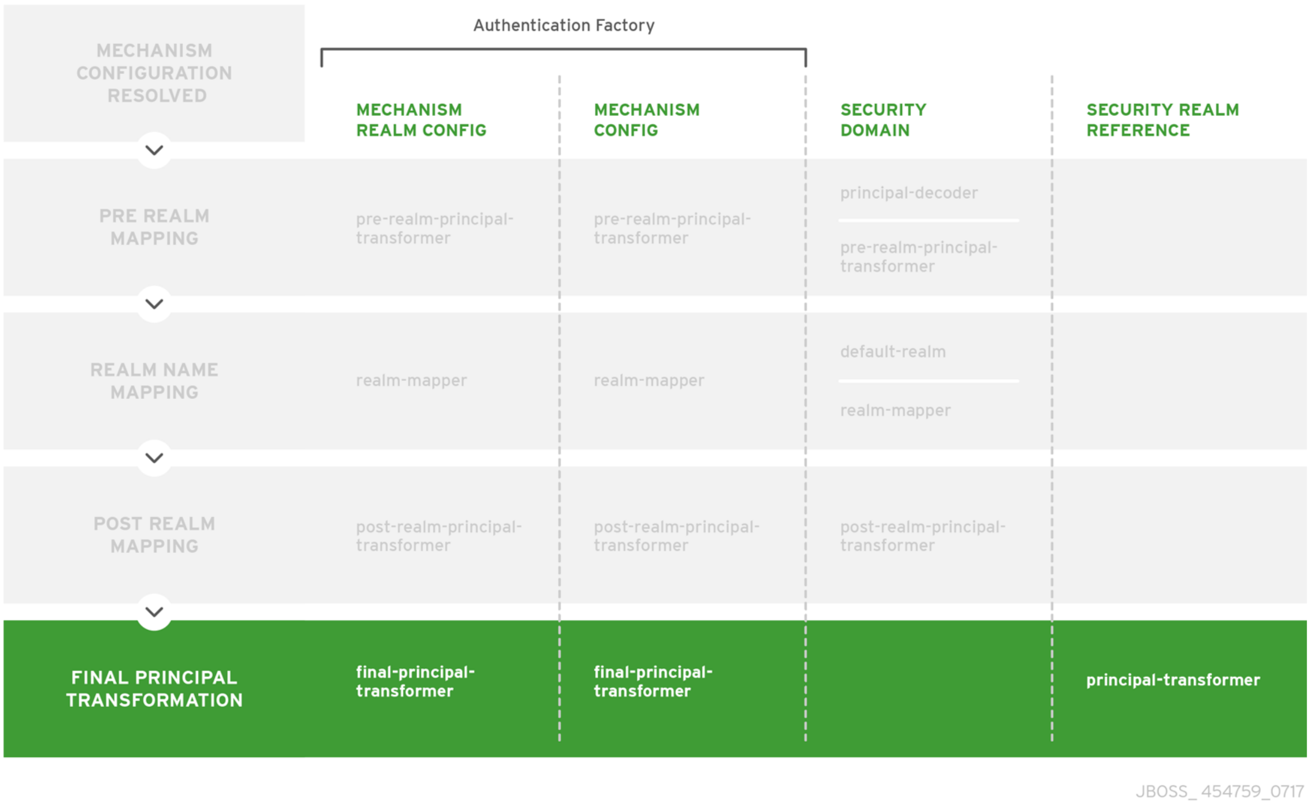
Final Principal Transformation

Finally, one last round of principal transformation occurs to allow for mechanism-specific transformations to be applied both before and after domain-specific transformations. If this stage is not required, then the same results can be obtained during the post-realm mapping stage. Transformers are called in the following order:

1. Mechanism Realm - **final-principal-transformer**
2. Mechanism Configuration - **final-principal-transformer**
3. Realm Mapping - **principal-transformer**

If this procedure results in a null principal, then an error will be thrown and authentication will terminate.

Figure 2.5. Final Principal Transformation


Authentication Factory
MECHANISM CONFIGURATION RESOLVED
MECHANISM REALM CONFIG
MECHANISM CONFIG
SECURITY DOMAIN
SECURITY REALM REFERENCE
PRE REALM MAPPING
pre-realm-principal-transformer
pre-realm-principal-transformer
principal-decoder
pre-realm-principal-transformer
REALM NAME MAPPING
realm-mapper
realm-mapper
default-realm
realm-mapper
POST REALM MAPPING
post-realm-principal-transformer
post-realm-principal-transformer
post-realm-principal-transformer
FINAL PRINCIPAL TRANSFORMATION
final-principal-transformer
final-principal-transformer
principal-transformer
JBOSS_ 454759_0717

Obtain the Realm Identity

After the final round of principal transformation, the security realm identified in realm name mapping is called to obtain a realm identity used to continue authentication.

2.6.3. HTTP Authentication

Elytron provides a complete set of HTTP authentication mechanisms including **BASIC**, **FORM**, **DIGEST**, **SPNEGO**, and **CLIENT_CERT**. HTTP authentication is handled using the **HttpAuthenticationFactory**, which is both an authentication policy for using HTTP authentication mechanisms as well as factory for configured authentication mechanisms.

The **HttpAuthenticationFactory** references the following:

SecurityDomain
: The security domain that any mechanism authentication will be performed against.

HttpServerAuthenticationMechanismFactory
: The general factory for server-side HTTP authentication mechanisms.

MechanismConfigurationSelector
: You can use this to supply additional configuration for the authentication mechanisms. The purpose of the **MechanismConfigurationSelector** is to obtain configuration specific for the mechanism selected. This can include information about realm names a mechanism should present to a remote client, additional principal transformers, and realm mappers to use during the authentication process.

2.6.4. SASL Authentication

SASL is a framework for authentication that separate the authentication mechanism itself from the protocol it uses. It also allows for additional authentication mechanisms such as **DIGEST-MD5**, **GSSAPI**, **OTP**, and **SCRAM**. SASL authentication is not part of the Jakarta EE specification. SASL authentication is handled using the **SaslAuthenticationFactory**, which is both an authentication policy for using SASL authentication mechanisms as well as a factory for configured authentication mechanisms.

The **SaslAuthenticationFactory** references the following:

SecurityDomain
: The security domain that any mechanism authentication will be performed against.

SaslServerFactory
: The general factory for server-side SASL authentication mechanisms.

MechanismConfigurationSelector
: You can use this to supply additional configuration for the authentication mechanisms. The purpose of the **MechanismConfigurationSelector** is to obtain configuration specific for the mechanism selected. This can include information about realm names a mechanism should present to a remote client, additional principal transformers, and realm mappers to use during the authentication process.

2.6.5. Interaction between the Elytron Subsystem and Legacy Systems

You can map some of the major components of both the legacy **security** subsystem components as well as the legacy core management authentication to Elytron capabilities. This allows those legacy components to be used in an Elytron based configuration and allow for an incremental migration from legacy components.

2.6.6. Resources in the Elytron Subsystem

JBoss EAP provides a set of resources in the **elytron** subsystem:

- Factories
- Principal Transformers
- Principal Decoders
- Realm Mappers
- Realms
- Permission Mappers
- Role Decoders
- Role Mappers
- SSL Components
- Other

Factories

aggregate-http-server-mechanism-factory
: An HTTP server factory definition where the HTTP server factory is an aggregation of other HTTP server factories.

aggregate-sasl-server-factory
: A SASL server factory definition where the SASL server factory is an aggregation of other SASL server factories.

configurable-http-server-mechanism-factory

An HTTP server factory definition that wraps another HTTP server factory and applies the specified configuration and filtering.

configurable-sasl-server-factory

A SASL server factory definition that wraps another SASL server factory and applies the specified configuration and filtering.

custom-credential-security-factory

A custom credential **SecurityFactory** definition.

http-authentication-factory

Resource containing the association of a security domain with a **HttpServerAuthenticationMechanismFactory**.
For more information, see Configure Authentication with Certificates in *How to Configure Identity Management* for JBoss EAP.

kerberos-security-factory

A security factory for obtaining a **GSSCredential** for use during authentication.
For more information, see Configure the Elytron Subsystem in *How to Set Up SSO with Kerberos* for JBoss EAP.

mechanism-provider-filtering-sasl-server-factory

A SASL server factory definition that enables filtering by provider where the factory was loaded using a provider.

provider-http-server-mechanism-factory

An HTTP server factory definition where the HTTP server factory is an aggregation of factories from the provider list.

provider-sasl-server-factory

A SASL server factory definition where the SASL server factory is an aggregation of factories from the provider list.

sasl-authentication-factory

Resource containing the association of a security domain with a SASL server factory.
For more information, see Secure the Management Interfaces with a New Identity Store in *How to Configure Server Security* for JBoss EAP.

service-loader-http-server-mechanism-factory

An HTTP server factory definition where the HTTP server factory is an aggregation of factories identified using a **ServiceLoader**.

service-loader-sasl-server-factory

A SASL server factory definition where the SASL server factory is an aggregation of factories identified using a **ServiceLoader**.

Principal Transformers

aggregate-principal-transformer

Individual transformers attempt to transform the original principal until one returns a non-null principal.

chained-principal-transformer

A principal transformer definition where the principal transformer is a chaining of other principal transformers.

constant-principal-transformer

A principal transformer definition where the principal transformer always returns the same constant.

custom-principal-transformer

A custom principal transformer definition.

regex-principal-transformer

A regular expression based principal transformer.

regex-validating-principal-transformer

A regular expression based principal transformer which uses the regular expression to validate the name.

Principal Decoders

aggregate-principal-decoder

A principal decoder definition where the principal decoder is an aggregation of other principal decoders.

concatenating-principal-decoder

A principal decoder definition where the principal decoder is a concatenation of other principal decoders.

constant-principal-decoder

Definition of a principal decoder that always returns the same constant.

custom-principal-decoder

Definition of a custom principal decoder.

x500-attribute-principal-decoder

Definition of an X500 attribute based principal decoder.
For more information, see Configure Authentication with Certificates in *How to Configure Identity Management* for JBoss EAP.

x509-subject-alternative-name-evidence-decoder

Evidence decoder to use a subject alternative name extension in an X.509 certificate as the principal.
For more information, see Configuring Evidence Decoder for X.509 Certificate with Subject Alternative Name Extension in *How to Configure Server Security* for JBoss EAP.

Realm Mappers

constant-realm-mapper

Definition of a constant realm mapper that always returns the same value.

custom-realm-mapper

Definition of a custom realm mapper.

mapped-regex-realm-mapper

Definition of a realm mapper implementation that first uses a regular expression to extract the realm name, this is then converted using the configured mapping of realm names.

simple-regex-realm-mapper

Definition of a simple realm mapper that attempts to extract the realm name using the capture group from the regular expression, if that does not provide a match then the delegate realm mapper is used instead.

Realms

aggregate-realm

A realm definition that is an aggregation of two realms, one for the authentication steps and one for loading the identity for the authorization steps.

> **NOTE**
>
> The exported legacy security domain cannot be used as Elytron security realm for the authorization step of the **aggregate-realm**.

caching-realm

A realm definition that enables caching to another security realm. The caching strategy is *Least Recently Used*, where the least accessed entries are discarded when the maximum number of entries is reached.
For more information, see Set Up Caching for Security Realms in *How to Configure Identity Management* for JBoss EAP.

custom-modifiable-realm

Custom realm configured as being modifiable will be expected to implement the **ModifiableSecurityRealm** interface. By configuring a realm as being modifiable management operations will be made available to manipulate the realm.

custom-realm

A custom realm definitions can implement either the s **SecurityRealm** interface or the **ModifiableSecurityRealm** interface. Regardless of which interface is implemented management operations will not be exposed to manage the realm. However other services that depend on the realm will still be able to perform a type check and cast to gain access to the modification API.

filesystem-realm

A simple security realm definition backed by the file system.
For more information, see Configure Authentication with a Filesystem-Based Identity Store in *How to Configure Identity Management* for JBoss EAP.

identity-realm

A security realm definition where identities are represented in the management model.

jdbc-realm

A security realm definition backed by database using JDBC.
For more information, see Configure Authentication with a Database-Based Identity Store in *How to Configure Identity Management* for JBoss EAP.

key-store-realm

A security realm definition backed by a keystore.
For more information, see Configure Authentication with Certificates in *How to Configure Identity Management* for JBoss EAP.

ldap-realm

A security realm definition backed by LDAP.
For more information, see Configure Authentication with a LDAP-Based Identity Store in *How to Configure Identity Management* for JBoss EAP.

properties-realm

A security realm definition backed by properties files.

Configure Authentication with a Properties File-Based Identity Store in *How to Configure Identity Management* for JBoss EAP.

token-realm

A security realm definition capable of validating and extracting identities from security tokens.

Permission Mappers

custom-permission-mapper

Definition of a custom permission mapper.

logical-permission-mapper

Definition of a logical permission mapper.

simple-permission-mapper

Definition of a simple configured permission mapper.

constant-permission-mapper

Definition of a permission mapper that always returns the same constant.

Role Decoders

custom-role-decoder

Definition of a custom RoleDecoder.

simple-role-decoder

Definition of a simple RoleDecoder that takes a single attribute and maps it directly to roles.

source-address-role-decoder

Definition of a **source-address-role-decoder** that assigns roles to an identity based on the IP address of the client.

aggregate-role-decoder

Definition of an **aggregate-role-decoder** that aggregates the roles returned by two or more role decoders.
For more information, see Configure Authentication with a Filesystem-Based Identity Store in *How to Configure Identity Management* for JBoss EAP.

Role Mappers

add-prefix-role-mapper

A role mapper definition for a role mapper that adds a prefix to each provided.

add-suffix-role-mapper

A role mapper definition for a role mapper that adds a suffix to each provided.

aggregate-role-mapper

A role mapper definition where the role mapper is an aggregation of other role mappers.

constant-role-mapper

A role mapper definition where a constant set of roles is always returned.
For more information, see Configure Authentication with Certificates in *How to Configure Identity Management* for JBoss EAP.

custom-role-mapper

Definition of a custom role mapper.

logical-role-mapper

A role mapper definition for a role mapper that performs a logical operation using two referenced role mappers.

mapped-role-mapper

A role mapper definition for a role mapper that uses preconfigured mapping of role names.

regex-role-mapper

A role mapper definition for a role mapper that uses a regular expression to translate roles. For example, you can map "app-admin", "app-operator" to "admin" and "operator" respectively.
For more information, see regex-role-mapper Attributes.

SSL Components

client-ssl-context

An SSLContext for use on the client side of a connection.
For more information, see Using a client-ssl-context in *How to Configure Server Security* for JBoss EAP.

filtering-key-store

A filtering keystore definition, which provides a keystore by filtering a **key-store**.
For more information, see Using a filtering-key-store in *How to Configure Server Security* for JBoss EAP.

key-manager

A key manager definition for creating the key manager list as used to create an SSL context.
For more information, see Enable One-way SSL/TLS for the Management Interfaces Using the Elytron Subsystem in *How to Configure Server Security* for JBoss EAP.

key-store

A keystore definition.
For more information, see Enable One-way SSL/TLS for the Management Interfaces Using the Elytron Subsystem in *How to Configure Server Security* for JBoss EAP.

ldap-key-store

An LDAP keystore definition, which loads a keystore from an LDAP server.
For more information, see Using an ldap-key-store in *How to Configure Server Security* for JBoss EAP.

server-ssl-context

An SSL context for use on the server side of a connection.
For more information, see Enable One-way SSL/TLS for the Management Interfaces Using the Elytron Subsystem in *How to Configure Server Security* for JBoss EAP.

trust-manager

A trust manager definition for creating the **TrustManager** list as used to create an SSL context.
For more information, see Enable Two-way SSL/TLS for the Management Interfaces using the Elytron Subsystem in *How to Configure Server Security* for JBoss EAP.

Other

aggregate-providers

An aggregation of two or more **provider-loader** resources.

authentication-configuration

An individual authentication configuration definition, which is used by clients deployed to JBoss EAP and other resources for authenticating when making a remote connection.

authentication-context

An individual authentication context definition, which is used to supply an **ssl-context** and **authentication-configuration** when clients deployed to JBoss EAP and other resources make a remoting connection.

credential-store

Credential store to keep alias for sensitive information such as passwords for external services.
For more information, see Create a Credential Store in *How to Configure Server Security* for JBoss EAP.

dir-context

The configuration to connect to a directory (LDAP) server.
For more information, see Using an ldap-key-store in *How to Configure Server Security* for JBoss EAP.

provider-loader

A definition for a provider loader.

security-domain

A security domain definition.
For more information, see Configure Authentication with Certificates in *How to Configure Identity Management* for JBoss EAP.

security-property

A definition of a security property to be set.

2.7. CORE MANAGEMENT AUTHENTICATION

Core management authentication is responsible for securing the management interfaces, HTTP and native, for the core management functionality using the **ManagementRealm**. It is built into the core management and is enabled and configured as a core service by default. It is only responsible for securing the management interfaces.

2.7.1. Security Realms

A security realm is an identity store of usernames, passwords, and group membership information that can be used when authenticating users in Jakarta Enterprise Beans, web applications, and the management interface. Initially, JBoss EAP comes preconfigured with two security realms by default: **ManagementRealm** and **ApplicationRealm**. Both security realms use the file system to store mappings between users and passwords and users and group membership information. They both use a digest mechanism by default when authenticating.

A digest mechanism is an authentication mechanism that authenticates the user by making use of one-time, one-way hashes comprising various pieces of information, including information stored in the usernames and passwords mapping property file. This allows JBoss EAP to authenticate users without sending any passwords in plain text over the network.

The JBoss EAP installation contains a script that enables administrators to add users to both realms. When users are added in this way, the username and hashed password are stored in the corresponding usernames and passwords properties file. When a user attempts to authenticate, JBoss EAP sends back a one-time use number, nonce, to the client. The client then generates a one-way hash using its username, password, nonce, and a few other fields, and sends the username, nonce, and one-way hash to JBoss EAP. JBoss EAP looks up the user's prehashed password and uses it, along with the provided username, nonce, and a few other fields, to generate another one-way hash in the same manner. If all the same information is used on both sides, including the correct password, hashes will match and the user is authenticated.

Although security realms use the digest mechanism by default, they may be reconfigured to use other authentication mechanisms. On startup, the management interfaces determine which authentication mechanisms will be enabled based on what authentication mechanisms are configured in **ManagementRealm**.

Security realms are not involved in any authorization decisions; however, they can be configured to load a user's group membership information, which can subsequently be used to make authorization decisions. After a user has been authenticated, a second step occurs to load the group membership information based on the username.

By default, the **ManagementRealm** is used during authentication and authorization for the management interfaces. The **ApplicationRealm** is a default realm made available for web applications and Jakarta Enterprise Beans to use when authenticating and authorizing users.

2.7.2. Default Security

By default, the core management authentication secures each of the management interfaces, HTTP and native, in two different forms: local clients and remote clients, both of which are configured using the **ManagementRealm** security realm by default. These defaults may be configured differently or replaced entirely.

NOTE

Out of the box, the management interfaces are configured to use simple access controls, which does not use roles. As a result, all users by default, when using simple access controls, have the same privileges as the SuperUser role, which essentially has access to everything.

2.7.2.1. Local and Remote Client Authentication with the Native Interface

The native interface, or management CLI, can be invoked locally on the same host as the running JBoss EAP instance or remotely from another machine. When attempting to connect using the native interface, JBoss EAP presents the client with a list of available SASL authentication mechanisms, for example, **local jboss user**, BASIC, etc. The client chooses its desired authentication mechanism and attempts to authenticate with the JBoss EAP instance. If it fails, it retries with any remaining mechanisms or stops attempting to connect. Local clients have the option to use the **local jboss user** authentication mechanism. This security mechanism is based on the client's ability to access the local file system. It validates that the user attempting to log in actually has access to the local file system on the same host as the JBoss EAP instance.

This authentication mechanism happens in four steps:

1. The client sends a message to the server that includes a request to authenticate using **local jboss user**.

2. The server generates a one-time token, writes it to a unique file, and sends a message to the client with the full path of the file.

3. The client reads the token from the file and sends it to the server, verifying that it has local access to the file system.

4. The server verifies the token and then deletes the file.

This form of authentication is based on the principle that if physical access to the file system is achieved, other security mechanisms are superfluous. The reasoning is that if a user has local file system access, that user has enough access to create a new user or otherwise thwart other security mechanisms put in place. This is sometimes referred to as silent authentication because it allows the local user to access the management CLI without username or password authentication.

This functionality is enabled as a convenience and to assist local users running management CLI scripts without requiring additional authentication. It is considered a useful feature given that access to the local configuration typically also gives the user the ability to add their own user details or otherwise disable security checks.

The native interface can also be accessed from other servers, or even the same server, as a remote client. When accessing the native interface as a remote client, clients will not be able to authenticate using **local jboss user** and will be forced to use another authentication mechanism, for example, DIGEST. If a local client fails to authenticate by using **local jboss user**, it will automatically fall back and attempt to use the other mechanisms as a remote client.

NOTE

The management CLI may be invoked from other servers, or even the same server, using the HTTP interface as opposed to the native interface. All HTTP connections, CLI or otherwise, are considered to be remote and **NOT** covered by local interface authentication.

IMPORTANT

By default, the native interface is not configured, and all management CLI traffic is handled by the HTTP interface. JBoss EAP 7 supports HTTP upgrade, which allows a client to make an initial connection over HTTP but then send a request to upgrade that connection to another protocol. In the case of the management CLI, an initial request over HTTP to the HTTP interface is made, but then the connection is upgraded to the native protocol. This connection is still handled over the HTTP interface, but it is using the native protocol for communication rather than HTTP. Alternatively, the native interface may still be enabled and used if desired.

2.7.2.2. Local and Remote Client Authentication with the HTTP Interface

The HTTP interface can be invoked locally by clients on the same host as the running JBoss EAP instance or remotely by clients from another machine. Despite allowing local and remote clients to access the HTTP interface, all clients accessing the HTTP interface are treated as remote connections.

When a client attempts to connect to the HTTP management interfaces, JBoss EAP sends back an HTTP response with a status code of **401 Unauthorized**, and a set of headers that list the supported authentication mechanisms, for example, Digest, GSSAPI, and so on. The header for Digest also includes the nonce generated by JBoss EAP. The client looks at the headers and chooses which authentication method to use and sends an appropriate response. In the case where the client chooses Digest, it prompts the user for their username and password. The client uses the supplied fields such as username and password, the nonce, and a few other pieces of information to generate a one-way hash.

The client then sends the one-way hash, username, and nonce back to JBoss EAP as a response. JBoss EAP takes that information, generates another one-way hash, compares the two, and authenticates the user based on the result.

2.7.3. Advanced Security

There are a number of ways to change the default configuration of management interfaces as well as the authentication and authorization mechanisms to affect how it is secured.

2.7.3.1. Updating the Management Interfaces

In addition to modifying the Authentication and Authorization mechanisms, JBoss EAP allows administrators to update the configuration of the management interface itself. There are a number of options.

Configuring the Management Interfaces to Use One-way SSL/TLS

Configuring the JBoss EAP management console for communication only using one-way SSL/TLS provides increased security. All network traffic between the client and management console is encrypted, which reduces the risk of security attacks, such as a man-in-the-middle attack. Anyone administering a JBoss EAP instance has greater permissions on that instance than non-privileged users, and using one-way SSL/TLS helps protect the integrity and availability of that instance. When configuring one-way SSL/TLS with JBoss EAP, authority-signed certificates are preferred over self-signed certificates because they provide a chain of trust. Self-signed certificates are permitted but are not recommended.

Using Two-way SSL/TLS

Two-way SSL/TLS authentication, also known as client authentication, authenticates the client and the server using SSL/TLS certificates. This provides assurance that not only is the server what it says it is, but the client is also what it says it is.

Updating or Creating a New Security Realm

The default security realm can be updated or replaced with a new security realm.

2.7.3.2. Adding Outbound Connections

Some security realms connect to external interfaces, such as an LDAP server. An outbound connection defines how to make this connection. A predefined connection type, **ldap-connection**, sets all of the required and optional attributes to connect to the LDAP server and verify the credential.

2.7.3.3. Adding RBAC to the Management Interfaces

By default the RBAC system is disabled. It is enabled by changing the **provider** attribute from **simple** to **rbac**. This can be done using the management CLI. When RBAC is disabled or enabled on a running server, the server configuration must be reloaded before it takes effect.

When RBAC is enabled for the management interfaces, the role assigned to a user determines the resources to which they have access and what operations they can conduct with a resource's attributes. Only users of the **Administrator** or **SuperUser** role can view and make changes to the access control system.

WARNING

Enabling RBAC without having users and roles properly configured could result in administrators being unable to log in to the management interfaces.

RBAC's Effect on the Management Console

In the management console, some controls and views are disabled, which show up as grayed out, or not visible at all, depending on the permissions of the role the user has been assigned.

If the user does not have read permissions to a resource attribute, that attribute will appear blank in the console. For example, most roles cannot read the username and password fields for data sources.

If the user has read permissions but does not have write permissions to a resource attribute, that attribute will be disabled in the edit form for the resource. If the user does not have write permissions to the resource, the edit button for the resource will not appear.

If a user does not have permissions to access a resource or attribute, meaning it is unaddressable for that role, it will not appear in the console for that user. An example of that is the access control system itself, which is only visible to a few roles by default.

The management console also provides an interface for the following common RBAC tasks:

- View and configure what roles are assigned to, or excluded from, each user.
- View and configure what roles are assigned to, or excluded from, each group.
- View group and user membership per role.
- Configure default membership per role.
- Create a scoped role.

NOTE

Constraints cannot be configured in the management console at this time.

RBAC's Effect on the Management CLI or Management API

Users of the management CLI or management API will encounter slightly different behavior when RBAC is enabled.

Resources and attributes that cannot be read are filtered from results. If the filtered items are addressable by the role, their names are listed as **filtered-attributes** in the **response-headers** section of the result. If a resource or attribute is not addressable by the role, it is not listed.

Attempting to access a resource that is not addressable will result in a **Resource Not Found** error.

If a user attempts to write or read a resource that they can address but lacks the appropriate write or read permissions, a **Permission Denied** error is returned.

[illegible]

The management CLI can perform all of same RBAC tasks as the management console as well as a few additional tasks:

- Enable and disable RBAC
- Change permission combination policy
- Configuring application resource and resource sensitivity constraints

RBAC's Effect on Jakarta Management Managed Beans

Role-Based Access Control applies to Jakarta Management in three ways:

1. The management API of JBoss EAP is exposed as Jakarta Management managed beans. These managed beans are referred to as **core mbeans**, and access to them is controlled and filtered exactly the same as the underlying management API itself.
2. The **jmx** subsystem is configured with write permissions being sensitive. This means only users of the **Administrator** and **SuperUser** roles can make changes to that subsystem. Users of the **Auditor** role can also read this subsystem configuration.
3. By default, managed beans registered by deployed applications and services, or non-core MBeans, can be accessed by all management users, but only users of the **Maintainer**, **Operator**, **Administrator**, and **SuperUser** roles can write to them.

RBAC Authentication

RBAC works with the standard authentication providers that are included with JBoss EAP:

Username/Password

Users are authenticated using a username and password combination that is verified according to the settings of the **ManagementRealm**, which has the ability to use a local properties file or LDAP.

Client Certificate

The truststore provides authentication information for client certificates.

local jboss user

The **jboss-cli** script authenticates automatically as **local jboss user** if the server is running on the same machine. By default **local jboss user** is a member of the **SuperUser** group.

Regardless of which provider is used, JBoss EAP is responsible for assigning roles to users. When authenticating with the **ManagementRealm** or an LDAP server, those systems can supply user group information. This information can also be used by JBoss EAP to assign roles to users.

2.7.3.4. Using LDAP with the Management Interfaces

JBoss EAP includes several authentication and authorization modules that allow an LDAP server to be used as the authentication and authorization authority for web and Jakarta Enterprise Beans applications.

To use an LDAP directory server as the authentication source for the management console, management CLI, or management API, the following tasks must be performed:

1. Create an outbound connection to the LDAP server.
2. Create an LDAP-enabled security realm or update an existing security realm to use LDAP.
3. Reference the new security realm in the management interface.

The LDAP authenticator operates by first establishing a connection to the remote directory server. It then performs a search using the username, which the user passed to the authentication system, to find the fully qualified distinguished name (DN) of the LDAP record. A new connection to the LDAP server is established, using the DN of the user as the credential and password supplied by the user. If this authentication to the LDAP server is successful, the DN is verified as valid.

Once an LDAP-enabled security realm is created, it can be referenced by the management interface. The management interface will use the security realm for authentication. JBoss EAP can also be configured to use an outbound connection to an LDAP server using two-way SSL/TLS for authentication in the management interface and management CLI.

2.7.3.5. Jakarta Authentication and the Management Interfaces

Jakarta Authentication can be used to secure the management interfaces. When using Jakarta Authentication for the management interfaces, the security realm must be configured to use a security domain. This introduces a dependency between core services and the subsystems. While SSL/TLS is not required to use Jakarta Authentication to secure the management interfaces, it is recommended that administrators enable SSL/TLS to avoid accidentally transmitting sensitive information in an unsecured manner.

NOTE

When JBoss EAP instances are running in **admin-only** mode, using Jakarta Authentication to secure the management interfaces is not supported. For more information on **admin-only** mode, see Running JBoss EAP in Admin-only Mode in the JBoss EAP *Configuration Guide*.

2.8. SECURITY SUBSYSTEM

The **security** subsystem provides security infrastructure for applications and is based on the Jakarta Authentication API. The subsystem uses a security context associated with the current request to expose the capabilities of the authentication manager, authorization manager, audit manager, and mapping manager to the relevant container.

The authentication and authorization managers handle authentication and authorization. The mapping manager handles adding, modifying, or deleting information from a principal, role, or attribute before passing the information to the application. The auditing manager allows users to configure provider modules to control the way that security events are reported.

In most cases, administrators should need to focus only on setting up and configuring security domains in regards to updating the configuration of the **security** subsystem. Outside of security domains, the only security element that may need to be changed is **deep-copy-subject-mode**. See the Security Management section for more information on deep copy subject mode.

2.8.1. Security Domains

A security domain is a set of Jakarta Authentication declarative security configurations that one or more applications use to control authentication, authorization, auditing, and mapping. Four security domains are included by default: **jboss-ejb-policy**, **jboss-web-policy**, **other**, and **jaspitest**. The **jboss-ejb-policy** and **jboss-web-policy** security domains are the default authorization mechanisms for the JBoss EAP instance. They are used if an application’s configured security domain does not define any authentication mechanisms. Those security domains, along with **other**, are also used internally within JBoss EAP for authorization and are required for correct operation. The **jaspitest** security domain is a simple Jakarta Authentication security domain included for development purposes.

A security domain comprises configurations for authentication, authorization, security mapping, and auditing. Security domains are part of the JBoss EAP **security** subsystem and are managed centrally by the domain controller or standalone server. Users can create as many security domains as needed to accommodate application requirements.

You can also configure the type of authentication cache to be used by a security domain, using the **cache-type** attribute. If this attribute is removed, no cache will be used. The allowed values for this property are **default** or **infinispan**.

Comparison Between Elytron and PicketBox Security Domains
A deployment should be associated with either a single Elytron security domain or one or more legacy PicketBox security domain. A deployment should not be associated with both. That is an invalid configuration.

An exception occurs if a deployment is associated with more than one Elytron security domain, whereas a deployment can be associated with multiple legacy security domains.

NOTE

When working with PicketBox, the security domain encapsulates both access to the underlying identity store and the mapping for authorization decisions. Thus, users of PicketBox with different stores are required to use different security domains for different sources.

In Elytron, these two functions are separated. Access to the stores is handled by security realms and mapping for authorization is handled by security domains.

So, a deployment requiring independent PicketBox security domains does not necessarily require independent Elytron security domains.

2.8.2. Using Security Realms and Security Domains

Security realms and security domains can be used to secure web applications deployed to JBoss EAP. When deciding if either should be used, it is important to understand the difference between the two.

Web applications and Jakarta Enterprise Beans deployments can only use security domains directly. They perform the actual authentication and authorization using login modules using the identity information passed from an identity store. Security domains can be configured to use security realms for identity information; for example, **other** allows applications to specify a security realm to use for authentication and getting authorization information. They can also be configured to use external identity stores. Web applications and Jakarta Enterprise Beans deployments cannot be configured to directly use security realms for authentication. The security domains are also part of the **security** subsystem and are loaded after core services.

Only the core management, for example the management interfaces and the Jakarta Enterprise Beans remoting endpoints, can use the security realms directly. They are identity stores that provide authentication as well as authorization information. They are also a core service and are loaded before any subsystems are started. The out-of-the-box security realms, **ManagementRealm** and **ApplicationRealm**, use a simple file-based authentication mechanism, but they can be configured to use other mechanisms.

2.8.3. Security Auditing

Security auditing refers to triggering events, such as writing to a log, in response to an event that happens within the **security** subsystem. Auditing mechanisms are configured as part of a security domain, along with authentication, authorization, and security mapping details. Auditing uses provider modules to control the way that security events are reported. JBoss EAP ships with several security auditing providers, but custom ones may be used. The core management of JBoss EAP also has its own security auditing and logging functionality, which is configured separately and is not part of the **security** subsystem.

2.8.4. Security Mapping

Security mapping adds the ability to combine authentication and authorization information after the authentication or authorization happens but before the information is passed to your application. Roles for authorization, principals for authentication, or credentials which are attributes that are not principals or roles, may all be mapped. Role mapping is used to add, replace, or remove roles to the subject after authentication. Principal mapping is used to modify a principal after authentication. You can use credential mapping to convert attributes from an external system to be used by your application. You can also use credential mapping conversely, to convert attributes from your application for use by an external system.

2.8.5. Password Vault System

JBoss EAP has a password vault to encrypt sensitive strings, store them in an encrypted keystore, and decrypt them for applications and verification systems. In plain text configuration files, such as XML deployment descriptors, it is sometimes necessary to specify passwords and other sensitive information. The JBoss EAP password vault can be used to securely store sensitive strings for use in plain text files.

2.8.6. Security Domain Configuration

Security domains are configured centrally either at the domain controller or on the standalone server. When security domains are used, an application may be configured to use a security domain in lieu of individually configuring security. This allows users and administrators to leverage Declarative Security.

Example

One common scenario that benefits from this type of configuration structure is the process of moving applications between testing and production environments. If an application has its security individually configured, it may need to be updated every time it is promoted to a new environment, for example, from a testing environment to a production environment. If that application used a security domain instead, the JBoss EAP instances in the individual environments would have their security domains properly configured for the current environment, allowing the application to rely on the container to provide the proper security configuration, using the security domain.

2.8.6.1. Login Modules

JBoss EAP includes several bundled login modules suitable for most user management roles that are configured within a security domain. The **security** subsystem offers some core login modules that can read user information from a relational database, an LDAP server, or flat files. In addition to these core login modules, JBoss EAP provides other login modules that provide user information and functionality for customized needs.

Summary of Commonly Used Login Modules

Ldap Login Module

The Ldap login module is a login module implementation that authenticates against an LDAP server. The **security** subsystem connects to the LDAP server using connection information, that is, a **bindDN** that has permissions to search the **baseCtxDN** and **rolesCtxDN** trees for the user and roles, provided using a Java Naming and Directory Interface initial context. When a user attempts to authenticate, the LDAP login module connects to the LDAP server and passes the user's credentials to the LDAP server. Upon successful authentication, an **InitialLDAPContext** is created for that user within JBoss EAP, populated with the user's roles.

LdapExtended Login Module

The LdapExtended login module searches for the user as well as the associated roles to bind for authentication. The roles query recursively and follow DNs to navigate a hierarchical role structure. The login module options include whatever options are supported by the chosen LDAP Java Naming and Directory Interface provider supports.

UsersRoles Login Module

The UsersRoles login module is a simple login module that supports multiple users and user roles loaded from Java properties files. The primary purpose of this login module is to easily test the security settings of multiple users and roles using properties files deployed with the application.

Database Login Module

The Database login module is a JDBC login module that supports authentication and role mapping. This login module is used if username, password, and role information are stored in a relational database. This works by providing a reference to logical tables containing principals and roles in the expected format.

Certificate Login Module

The Certificate login module authenticates users based on X509 certificates. A typical use case for this login module is CLIENT-CERT authentication in the web tier. This login module only performs authentication and must be combined with another login module capable of acquiring authorization roles to completely define access to secured web or Jakarta Enterprise Beans components. Two subclasses of this login module, **CertRolesLoginModule** and **DatabaseCertLoginModule**, extend the behavior to obtain the authorization roles from either a properties file or database.

Identity Login Module

The Identity login module is a simple login module that associates a hard-coded username to any subject authenticated against the module. It creates a **SimplePrincipal** instance using the name specified by the principal option. This login module is useful when a fixed identity is required to be provided to a service. This can also be used in development environments for testing the security associated with a given principal and associated roles.

RunAs Login Module

The RunAs login module is a helper module that pushes a run-as role onto the stack for the duration of the login phase of authentication; it then pops the run-as role from the stack in either the commit or abort phase. The purpose of this login module is to provide a role for other login modules that must access secured resources to perform their authentication, for example, a login module that accesses a secured Jakarta Enterprise Beans. The RunAs login module must be configured ahead of the login modules that require a run as role established.

Client Login Module

The Client login module is an implementation of a login module for use by JBoss clients when establishing caller identity and credentials. This creates a new **SecurityContext**, assigns it a principal and a credential, and sets the **SecurityContext** to the **ThreadLocal** security context. The Client login module is the only supported mechanism for a client to establish the current thread's caller. Both standalone client applications and server environments, acting as JBoss Jakarta Enterprise Beans clients where the security environment has not been configured to use the JBoss EAP **security** subsystem transparently, must use the Client login module.

> **WARNING**
>
> This login module does not perform any authentication. It merely copies the login information provided to it into the server Jakarta Enterprise Beans invocation layer for subsequent authentication on the server. Within JBoss EAP, this is only supported for switching a user's identity for in-JVM calls. This is *not* supported for remote clients to establish an identity.

SPNEGO Login Module

The SPNEGO login module is an implementation of a login module that establishes caller identity and credentials with a KDC. The module implements SPNEGO and is a part of the JBoss Negotiation project. This authentication can be used in the chained configuration with the AdvancedLdap login module to allow cooperation with an LDAP server. Web applications must also enable the **NegotiationAuthenticator** within the application to use this login module.

RoleMapping Login Module

The RoleMapping login module supports mapping roles that are the end result of the authentication process to one or more declarative roles. For example, if the authentication process has determined that the user John has the roles **ldapAdmin** and **testAdmin**, and the declarative role defined in the **web.xml** or **ejb-jar.xml** file for access is **admin**, then this login module maps the **ldapAdmin** and **testAdmin** roles to John. The RoleMapping login module must be defined as an optional module to a login module configuration as it alters mapping of the previously mapped roles.

Remoting Login Module

The Remoting login module checks if the request that is currently being authenticated was received over the remoting connection. In cases where the request was received using the remoting interface, that request is associated with the identity created during the authentication process.

RealmDirect Login Module

The RealmDirect login module allows an existing security realm to be used in making authentication and authorization decisions. When configured, this module will look up identity information using the referenced realm for making authentication decisions and mapping user roles. For example, the preconfigured **other** security domain that ships with JBoss EAP has a RealmDirect login module. If no realm is referenced in this module, the **ApplicationRealm** security realm is used by default.

Custom Modules

In cases where the login modules bundled with the JBoss EAP security framework do not meet the needs of the security environment, a custom login module implementation may be written. The AuthenticationManager requires a particular usage pattern of the Subject principals set. A full understanding of the Jakarta Authentication Subject class's information storage features and the expected usage of these features are required to write a login module that works with the AuthenticationManager.

The UnauthenticatedIdentity login module option is also commonly used. There are certain cases when requests are not received in an authenticated format. The Unauthenticated Identity is a login module configuration option that assigns a specific identity, for example, **guest**, to requests that are made with no associated authentication information. This can be used to allow unprotected servlets to invoke methods on Jakarta Enterprise Beans that do not require a specific role. Such a principal has no associated roles and can only access either unsecured Jakarta Enterprise Beans methods that are associated with the unchecked permission constraint.

2.8.6.2. Password Stacking

Multiple login modules can be chained together in a stack, with each login module providing the credentials verification and role assignment during authentication. This works for many use cases, but sometimes credentials verification and role assignment are split across multiple user management stores.

Consider the case where users are managed in a central LDAP server and application-specific roles are stored in the application's relational database. The **password-stacking** module option captures this relationship.

To use password stacking, each login module should set the **password-stacking** attribute to **useFirstPass**, which is located in the **<module-option>** section. If a previous module configured for password stacking has authenticated the user, all the other stacking modules will consider the user authenticated and only attempt to provide a set of roles for the authorization step.

When the **password-stacking** option is set to **useFirstPass**, this module first looks for a shared username and password under the property names **javax.security.auth.login.name** and **javax.security.auth.login.password**, respectively, in the login module shared state map.

If found, these properties are used as the principal name and password. If not found, the principal name and password are set by this login module and stored under the property names **javax.security.auth.login.name** and **javax.security.auth.login.password**, respectively.

2.8.6.3. Password Hashing

Most login modules must compare a client-supplied password to a password stored in a user management system. These modules generally work with plain text passwords, but they can be configured to support hashed passwords to prevent plain text passwords from being stored on the server side. JBoss EAP supports the ability to configure the hashing algorithm, encoding, and character set. It also defines when the user password and store password are hashed.

IMPORTANT

Red Hat JBoss Enterprise Application Platform Common Criteria certified configuration does not support hash algorithms weaker than SHA-256.

2.8.7. Security Management

The security management portion of the **security** subsystem is used to override the high-level behaviors of the **security** subsystem. Each setting is optional. It is unusual to change any of these settings except for the deep copy subject mode.

2.8.7.1. Deep Copy Mode

If the deep copy subject mode is disabled, which it is by default, copying a security data structure makes a reference to the original rather than copying the entire data structure. This behavior is more efficient, but it is prone to data corruption if multiple threads with the same identity clear the subject by means of a flush or logout operation.

If the deep copy subject mode is enabled, a complete copy of the data structure, along with all its associated data as long as they are marked cloneable, is made. This is more thread-safe but less efficient.

2.8.8. Additional Components

2.8.8.1. Jakarta Authentication

Jakarta Authentication is a pluggable interface for Java applications and is defined in Jakarta Authentication specification. In addition to Jakarta Authentication authentication, JBoss EAP allows for Jakarta Authentication to be used. Jakarta Authentication authentication is configured using login modules in a security domain, and those modules may be stacked. The **jaspitest** security domain is a simple Jakarta Authentication security domain that is included by default for development purposes.

The web-based administration console provides the following operations to configure the Jakarta Authentication module:

- add
- edit
- remove
- reset

Applications deployed to JBoss EAP require a special authenticator to be configured in their deployment descriptors to use the Jakarta Authentication security domains.

2.8.8.2. Jakarta Authorization

Jakarta Authorization is a standard that defines a contract between containers and authorization service providers, which results in the implementation of providers for use by containers. For details about the specifications, see Jakarta Authorization 1.1 Specification.

JBoss EAP implements support for Jakarta Authorization within the security functionality of the **security** subsystem.

2.8.8.3. Jakarta Security

Jakarta Security defines portable plug-in interfaces for authentication and identity stores, and a new injectable-type **SecurityContext** interface that provides an access point for programmatic security. You can use the built-in implementations of these APIs, or define custom implementations. For details about the specifications, see Jakarta Security Specification.

The Jakarta Security API is available in the **elytron** subsystem and can be enabled from the management CLI. For more information, see About Jakarta Security API in the Development Guide.

2.8.8.4. About Fine-Grained Authorization and XACML

Fine-grained authorization allows administrators to adapt to the changing requirements and multiple variables involved in the decision making process for granting access to a module. As a result, fine-grained authorization can become complex.

NOTE

The XACML bindings, for web or Jakarta Enterprise Beans, are not supported in JBoss EAP.

JBoss EAP uses XACML as a medium to achieve fine-grained authorization. XACML provides standards-based solution to the complex nature of achieving fine-grained authorization. XACML defines a policy language and an architecture for decision making. The XACML architecture includes a

Policy Enforcement Point (PEP) which intercepts any requests in a normal program flow and asks a Policy Decision Point (PDP) to make an access decision based on the policies associated with the PDP. The PDP evaluates the XACML request created by the PEP and runs through the policies to make one of the following access decisions.

PERMIT

The access is approved.

DENY

The access is denied.

INDETERMINATE

There is an error at the PDP.

NOTAPPLICABLE

There is some attribute missing in the request or there is no policy match.

XACML also has the following features:

- Oasis XACML v2.0 library
- JAXB v2.0 based object model
- ExistDB integration for storing and retrieving XACML policies and attributes

2.8.8.5. SSO

JBoss EAP provides out-of-the-box support for clustered and non-clustered SSO using the **undertow** and **infinispan** subsystems. This requires:

- A configured security domain that handles authentication and authorization.
- The **SSO** infinispan replication cache. It is present in the **ha** and **full-ha** profiles for a managed domain, or by using the **standalone-ha.xml** or **standalone-full-ha.xml** configuration files for a standalone server.
- The **web** cache-container and **SSO** replication cache within it must be present.
- The **undertow** subsystem needs to be configured to use SSO.
- Each application that will share the SSO information must be configured to use the same security domain.

CHAPTER 3. ADDITIONAL USE CASES FOR SSO WITH RED HAT JBOSS ENTERPRISE APPLICATION PLATFORM

In addition to the out-of-the-box functionality, JBoss EAP supports additional use cases for SSO, including SAML for browser-based SSO, desktop-based SSO, and SSO via a secure token service.

3.1. BROWSER-BASED SSO USING SAML

In a browser-based SSO scenario, one or more web applications, known as service providers, are connected to a centralized identity provider in a hub-and-spoke architecture. The identity provider (IDP) acts as the central source, or hub, for identity and role information to all the service providers, or spokes. When an unauthenticated user attempts to access one of the service providers, that user is redirected to an IDP to perform the authentication. The IDP authenticates the user, issues a SAML token specifying the role of the principal, and redirects them to the requested service provider. From there, that SAML token is used across all of the associated service providers to determine the user's identity and access. This specific method of using SSO starting at the service providers is known as a service provider-initiated flow.

Like many SSO systems, JBoss EAP uses IDPs and SPs. Both of these components are enabled to be run within JBoss EAP instances and work in conjunction with the JBoss EAP **security** subsystem. SAML v2, FORM-based web application security, and HTTP/POST and HTTP/Redirect Bindings are also used to implement SSO.

To create an identity provider, create a security domain, for example **idp-domain**, in a JBoss EAP instance that defines an authentication and authorization mechanism, for example LDAP or a database, to serve as the identity store. A web application, for example **IDP.war**, is configured to use additional modules, **org.picketlink**, required for running an IDP in conjunction with **idp-domain** and is deployed to that same JBoss EAP instance. IDP.war will serve as an identity provider. To create a service provider, a security domain is created, for example **sp-domain**, that uses **SAML2LoginModule** as an authentication mechanism. A web application, for example **SP.war**, is configured to use additional modules, **org.picketlink**, and contains a service provider valve that uses **sp-domain**. **SP.war** is deployed to an JBoss EAP instance where **sp-domain** is configured and is now a service provider. This process can be replicated for one or more SPs, for example **SP2.war**, **SP3.war**, and so on, and across one or more JBoss EAP instances.

3.1.1. Identity Provider Initiated Flow

In most SSO scenarios, the SP starts the flow by sending an authentication request to the IDP, which sends a SAML response to SP with a valid assertion. This is known as a SP-initiated flow. The SAML 2.0 specifications define another flow, one called IDP-initiated or Unsolicited Response flow. In this scenario, the service provider does not initiate the authentication flow to receive a SAML response from the IDP. The flow starts on the IDP side. Once authenticated, the user can choose a specific SP from a list and get redirected to the SP's URL. No special configuration is necessary to enable this flow.

Walkthrough

1. User accesses the IDP.

2. The IDP, seeing that there is neither a SAML request nor a response, assumes an IDP-initiated flow scenario using SAML.

3. The IDP challenges the user to authenticate.

4. Upon authentication, the IDP shows the hosted section where the user gets a page that links to all the SP applications.
5. The user chooses an SP application.
6. The IDP redirects the user to the SP with a SAML assertion in the query parameter, SAML response.
7. The SP checks the SAML assertion and provides access.

3.1.2. Global Logout

A global logout initiated at one SP logs out the user from the IDP and all the SPs. If a user attempts to access secured portions of any SP or IDP after performing a global logout, they must reauthenticate.

3.2. DESKTOP-BASED SSO

A desktop-based SSO scenario enables a principal to be shared across the desktop, usually governed by an Active Directory or Kerberos server, and a set of web applications which are the SPs. In this case, the desktop IDP serves as the IDP for the web applications.

In a typical setup, the user logs in to a desktop governed by the Active Directory domain. The user accesses a web application via a web browser configured with JBoss Negotiation and hosted on the JBoss EAP. The web browser transfers the sign-on information from the local machine of the user to the web application. JBoss EAP uses background GSS messages with the Active Directory or any Kerberos Server to validate the user. This enables the user to achieve a seamless SSO into the web application.

To set up a desktop-based SSO as an IDP for a web application, a security domain is created that connects to the IDP server. A NegotiationAuthenticator is added as a valve to the desired web application, and JBoss Negotiation is added to the SP container's class path. Alternatively, an IDP can be set up similarly to the browser-based SSO scenario but using the desktop-based SSO provider as an identity store.

3.3. SSO USING STS

JBoss EAP offers several login modules for SPs to connect to an STS. It can also run an STS (**PicketLinkSTS**). More specifically, the **PicketLinkSTS** defines several interfaces to other security token services and provides extension points. Implementations can be plugged in by using configuration, and the default values can be specified for some properties via configuration. This means that the **PicketLinkSTS** generates and manages the security tokens but does not issue tokens of a specific type. Instead, it defines generic interfaces that allow multiple token providers to be plugged in. As a result, it can be configured to deal with various types of token, as long as a token provider exists for each token type. It also specifies the format of the security token request and response messages.

The following steps are the order that the security token requests are processed when using the JBoss EAP STS.

1. A client sends a security token request to **PicketLinkSTS**.
2. **PicketLinkSTS** parses the request message and generates a Jakarta XML Binding object model.
3. **PicketLinkSTS** reads the configuration file and creates the **STSConfiguration** object, if needed. It obtains a reference to the **WSTrustRequestHandler** from the configuration and delegates the request processing to the handler instance.

4. The request handler uses the **STSConfiguration** to set default values when needed, for example when the request does not specify a token lifetime value.

5. The **WSTrustRequestHandler** creates the **WSTrustRequestContext** and sets the Jakarta XML Binding request object and the caller principal it received from **PicketLinkSTS**.

6. The **WSTrustRequestHandler** uses the **STSConfiguration** to get the **SecurityTokenProvider** that must be used to process the request based on the type of the token that is being requested. It invokes the provider and passes the constructed **WSTrustRequestContext** as a parameter.

7. The **SecurityTokenProvider** instance processes the token request and stores the issued token in the request context.

8. The **WSTrustRequestHandler** obtains the token from the context, encrypts it, if needed, and constructs the WS-Trust response object containing the security token.

9. **PicketLinkSTS** dictates the response generated by the request handler and returns it to the client.

An STS login module, for example STSIssuingLoginModule, STSValidatingLoginModule, SAML2STSLoginModule, and so on, is typically configured as part of the security setup of a JEE container to use an STS for authenticating users. The STS may be collocated on the same container as the login module or be accessed remotely through web service calls or another technology.

CHAPTER 4. ELYTRON SUBSYSTEM EXAMPLE SCENARIOS

4.1. OUT OF THE BOX

By default, JBoss EAP provides the following components preconfigured:

ApplicationDomain

The **ApplicationDomain** security domain uses **ApplicationRealm** and **groups-to-roles** for authentication. It also uses **default-permission-mapper** to assign the login permission.

ApplicationRealm

The **ApplicationRealm** security realm is a properties realm that authenticates principals using **application-users.properties** and assigns roles using **application-roles.properties**. These files are located under **jboss.server.config.dir**, which by default, maps to ***EAP_HOME*/standalone/configuration**. They are also the same files used by the legacy security default configuration.

application-http-authentication

The **application-http-authentication** http-authentication-factory can be used for doing authentication over HTTP. It uses the **global** provider-http-server-mechanism-factory to filter authentication mechanism and uses **ApplicationDomain** for authenticating principals. It accepts **BASIC** and **FORM** authentication mechanisms and exposes **BASIC** as **Application Realm** to applications.

application-sasl-authentication

The **application-sasl-authentication** sasl-authentication-factory can be used for authentication using SASL. It uses the **configured** sasl-server-factory to filter authentication mechanisms, which also uses the **global** provider-sasl-server-factory to filter by provider names. **application-sasl-authentication** uses the **ApplicationDomain** security domain for authentication of principals.

configured (configurable-sasl-server-factory)

This is used to filter **sasl-authentication-factory** is used based on the mechanism name. In this case, **configured** will match on **JBOSS-LOCAL-USER** and **DIGEST-MD5**. It also sets the **wildfly.sasl.local-user.default-user** to **$local**.

default-permission-mapper

The **default-permission-mapper** is a simple permission mapper that uses the **default-permissions** permission set to assign the full set of permissions that an identity requires to access any services on the server.
For example, the **default-permission-mapper** uses **org.wildfly.extension.batch.jberet.deployment.BatchPermission** specified by the **default-permissions** permission set to assign permissions for batch jobs. The batch permissions are start, stop, restart, abandon, and read which aligns with **javax.batch.operations.JobOperator**.

The **default-permission-mapper** also uses **org.wildfly.security.auth.permission.LoginPermission** specified by the **login-permission** permission set to assign the login permission.

elytron (mechanism-provider-filtering-sasl-server-factor)

This is used to filter which **sasl-authentication-factory** is used based on the provider. In this case, **elytron** will match on the **WildFlyElytron** provider name.

global (provider-http-server-mechanism-factory)

This is the HTTP server factory mechanism definition used to list the supported authentication mechanisms when creating an HTTP authentication factory.

global (provider-sasl-server-factory)

This is the SASL server factory definition used to create SASL authentication factories.

groups-to-roles

The **groups-to-roles** mapper is a simple-role-decoder that will decode the **groups** information of a principal and use it for the **role** information.

local (mapper)

The **local** mapper is a constant role mapper that maps to the **local** security realm. This is used to map authentication to the **local** security realm.

local (security realm)

The **local** security realm does no authentication and sets the identity of principals to **$local**.

ManagementDomain

The **ManagementDomain** security domain uses two security realms for authentication: **ManagementRealm** with **groups-to-roles** and **local** with **super-user-mapper**. It also uses **default-permission-mapper** to assign the login permission.

ManagementRealm

The **ManagementRealm** security realm is a properties realm that authenticates principals using **mgmt-users.properties** and assigns roles using **mgmt-roles.properties**. These files are located under **jboss.server.config.dir**, which by default, maps to ***EAP_HOME*/standalone/configuration**. They are also the same files used by the legacy security default configuration.

management-http-authentication

The **management-http-authentication** http-authentication-factory can be used for doing authentication over HTTP. It uses the **global** provider-http-server-mechanism-factory to filter authentication mechanism and uses **ManagementDomain** for authenticating principals. It accepts the **DIGEST** authentication mechanisms and exposes it as **ManagementRealm** to applications.

management-sasl-authentication

The **management-sasl-authentication** sasl-authentication-factory can be used for authentication using SASL. It uses the **configured** sasl-server-factory to filter authentication mechanisms, which also uses the **global** provider-sasl-server-factory to filter by provider names. **management-sasl-authentication** uses the **ManagementDomain** security domain for authentication of principals. It also maps authentication using **JBOSS-LOCAL-USER** mechanisms using the **local** realm mapper and authentication using **DIGEST-MD5** to **ManagementRealm**.

super-user-mapper

The **super-user-mapper** mapper is a constant role mapper that maps the **SuperUser** role to a principal.

4.1.1. Security

For securing applications, JBoss EAP comes preconfigured with **application-http-authentication** for using HTTP and **application-sasl-authentication** for using SASL. The **application-http-authentication** http-authentication-factory uses **ApplicationDomain** which uses **ApplicationRealm** and **groups-to-roles** for authentication. **ApplicationRealm** is a properties-realm backed by **application-users.properties** and **application-roles.properties** for username, password, and role information.

For securing the management interfaces, JBoss EAP comes preconfigured with **management-http-authentication** for HTTP and **management-sasl-authentication** for SASL. The **management-http-authentication** http-authentication-factory uses **ManagementDomain** which uses **ManagementRealm** and **groups-to-roles** for authentication. **ManagementRealm** is a properties-realm backed by **mgmt-users.properties** and **mgmt-roles.properties** for username, password, and role information. The

management-sasl-authentication sasl-authentication-factory uses **ManagementDomain** which uses **local** for **JBOSS-LOCAL-USER** authentication and **ManagementRealm** for **DIGEST-MD5** authentication.

4.1.2. How It Works

By default, there are no users for JBoss EAP, but for the purposes of this example the following users have been added:

Table 4.1. Users

Username	Password	Roles	Security Realm
Susan	Testing123!		ManagementRealm
Sarah	Testing123!	sample	ApplicationRealm
Frank	Testing123!		ApplicationRealm

On startup, the JBoss EAP instance loads all four authentication factories and their associated security domains, security realms, and other configured components.

If anyone attempts to access the management interface with the management CLI using **JBOSS-LOCAL-USER**, in other words running the management CLI from the same host as the JBoss EAP instance, the user will be directed to the **management-sasl-authentication** sasl-authentication-factory which will attempt to authenticate the user with **ManagementDomain** using the **local** security realm.

If Susan attempts to access the management interface using the management CLI from a different host, she will be using the **DIGEST-MD5** authentication mechanism with SASL. She will be directed to the **management-sasl-authentication** sasl-authentication-factory which will attempt to authenticate the user with **ManagementDomain** using the **ManagementRealm** security realm.

If Susan attempts to access the management interface using the web-based management console, she will be using the **DIGEST** authentication mechanism with HTTP. She will be directed to the **management-http-authentication** http-authentication-factory which will attempt to authenticate the user with **ManagementDomain** using the **ManagementRealm** security realm.

The application **sampleApp1.war** has two HTML files, **/hello.html** and **/secure/hello.html**, and uses **BASIC** HTTP authentication to secure the path **/secure/***. It uses the **Application Realm** and requires the role of **sample**. When a user attempts to access **sampleApp1.war**, they are directed to the **application-http-authentication** http-authentication-factory. It will attempt to authenticate the user with **ApplicationDomain** using the **ApplicationRealm** security realm.

When Sarah requests **/hello.html**, she is able to view the page without authenticating. When Sarah tries to request **/secure/hello.html**, she is prompted to enter her username and password. She is able to view **/secure/hello.html** after successfully logging in. Frank and Susan, or any user, can access **/hello.html**, but neither can access **/secure/hello.html**. Frank does not have the **sample** role, and Susan is not in the **ApplicationRealm** security realm.

4.2. USING SSL/TLS TO SECURE THE MANAGEMENT INTERFACES AND APPLICATIONS

This scenario shows how JBoss EAP is secured when using Elytron for SSL/TLS with both management interfaces and applications.

4.2.1. Security

JBoss EAP provides the ability to secure both the management interfaces as well as applications with SSL/TLS. With Elytron, this configuration is now unified so you now have the option to secure both the management interfaces and applications with the same SSL/TLS configuration. SSL/TLS is configured in Elytron by creating a **key-store**, **key-manager**, and **server-ssl-context**. SSL/TLS is enabled for the management interfaces by setting **secure-socket-binding** on the **http-interface** and by assigning the **server-ssl-context** to the management interfaces. This enables the management interfaces to use SSL/TLS for HTTP traffic. SSL/TLS is enabled for applications by assigning the **server-ssl-context** to the **https-listener** in the **undertow** subsystem. For more background information on SSL/TLS see the Advanced Security section.

4.2.2. How It Works

On startup, JBoss EAP loads the management interfaces as part of the core services, which also starts the **http-interface**, configured for SSL/TLS for the management interfaces. It also starts the **undertow** subsystem, which is configured for SSL/TLS for applications, and the **elytron** subsystem which provides the SSL/TLS configuration through the **server-ssl-context**. Both the management interfaces and applications can then be accessed over the secure ports with SSL/TLS enabled.

4.3. SECURING THE MANAGEMENT INTERFACES AND APPLICATIONS WITH A NEW IDENTITY STORE

This scenario shows how both the management interfaces and applications in JBoss EAP are secured with a new identity store in Elytron. An application, **sampleApp2.war**, is deployed to JBoss EAP and is configured to use **basicExampleDomain**.

4.3.1. Security

JBoss EAP provides the ability to secure both the management interfaces as well as applications with identity stores beyond **ManagementRealm** and **ApplicationRealm**. With Elytron, the same identity store can be used to secure the management interfaces as well as applications, but you also still have the option to set up separate identity stores for each. An identity store is represented by a security realm, for example a **filesystem-realm**, **jdbc-realm**, or **ldap-realm**. For the purposes of this example, a **filesystem-realm** named **exampleRealm** has been created. A security domain named **exampleDomain** has also been created which uses **exampleRealm** as an identity store, the **groups-to-roles** role mapper to decode the group information provided by **exampleRealm** into roles, and **default-permission-mapper** for mapping permissions.

For HTTP authentication, an **http-authentication-factory** called **exampleHttpAuthFactory** has been created. It uses the **global** HTTP server factory mechanism and **exampleDomain** for authentication. It also has two mechanism configurations. One that uses the **BASIC** authentication method exposed as **basicExampleDomain**, and one that uses the **DIGEST** authentication method exposed as **digestExampleDomain**. The HTTP management interface has been configured to use **exampleHttpAuthFactory**. The **undertow** subsystem has also been configured with a new **application-security-domain** which also uses **exampleHttpAuthFactory**. The application **sampleApp2.war** is configured to use **basicExampleDomain** with **BASIC** authentication.

For SASL authentication a **sasl-authentication-factory** called **exampleSaslAuthFactory** has been created. It uses the **configured** SASL server factory and **exampleDomain** for authentication. It also has a **DIGEST-MD5** authentication mechanism configured which is exposed as **digestMD5ExampleDomain**.

The SASL configuration for the management interfaces has been set to use **exampleSaslAuthFactory**.

4.3.2. How It Works

The following users have been added to **exampleRealm**:

Table 4.2. exampleRealm Users

Username	Password	Roles
Vincent	samplePass	sample
Issac	samplePass	guest

On startup, JBoss EAP loads the core services and starts the **undertow** and **elytron** subsystems. This secures the management interfaces and exposes **basicExampleDomain**, **digestExampleDomain**, and **digestMD5ExampleDomain** for applications deployed to JBoss EAP.

When **sampleApp2.war** is loaded, it looks for **basicExampleDomain** to provide authentication and authorization for its secured URLs. It has two HTML files, **/hello.html** and **/secure/hello.html**, and uses BASIC authentication to secure the path **/secure/***. It requires the role of **sample** to access any secure URLs.

When users authenticate, their credentials are submitted using a specific mechanism depending on how they are accessing JBoss EAP. For instance, if a user attempts to access the management console using HTTP with DIGEST authentication, they will authenticate using the DIGEST authentication method exposed as **digestExampleDomain**. If they attempt to access **sampleApp2.war** using HTTP with BASIC authentication, they will authenticate using the BASIC authentication method exposed as **basicExampleDomain**. If they attempt to access the management CLI using SASL with DIGEST authentication, they will authenticate using the DIGEST-MD5 exposed as **digestMD5ExampleDomain**. The HTTP authentication mechanisms use **exampleHttpAuthFactory**, and the SASL authentication mechanism uses **exampleSaslAuthFactory**. Both authentication factories handle authentication and role mapping with **exampleDomain**.

If Vincent or Issac attempt to access the management interfaces, they are prompted for their username and password. After successfully logging in, they are each able to perform management operations.

When Vincent or Issac requests **/hello.html**, they are able to view the page without authenticating. When Vincent or Issac tries to request **/secure/hello.html**, they are prompted to enter a username and password. After successfully logging in, Vincent is able to view **/secure/hello.html** since he has the **sample** role but Issac will not be able to view **/secure/hello.html** since he has the **guest** role. All other users can access **/hello.html** without logging in, but none can access **/secure/hello.html** because Vincent and Issac are the only users in **exampleRealm**.

4.4. USING RBAC TO SECURE THE MANAGEMENT INTERFACES

This scenario shows how the JBoss EAP management interfaces are secured with RBAC and an identity store in the **elytron** subsystem.

4.4.1. Security

JBoss EAP offers the ability to use RBAC on the management interfaces. The concepts behind RBAC are covered in the RBAC section. This example uses a security realm called **exampleRealm**. The remainder of the security configuration, including role decoder, security domain, and authentication factories, are the same as in the Securing the Management Interfaces and Applications with a New Identity Store section. RBAC is enabled by setting the **provider** attribute to **rbac** for the management interface and updating the **exampleRealm** with the desired users and roles.

4.4.2. How It Works

For this scenario, the following users have been added to the existing security realms:

Table 4.3. exampleRealm Users

Username	Password
Suzy	Testing123!
Tim	Testing123!
Emily	Testing123!
Zach	Testing123!
Natalie	Testing123!

Based on group membership, the users have also been mapped to the following RBAC roles:

Table 4.4. RBAC Roles

Username	RBAC Role
Suzy	SuperUser
Tim	Administrator
Emily	Maintainer
Zach	Deployer
Natalie	Monitor

On startup, JBoss EAP loads the core services and **elytron** subsystem, which loads the security configuration and the RBAC configuration. If RBAC was not enabled, any user in the **exampleRealm** is considered a **SuperUser** and has unlimited access. Because RBAC has been enabled, each user is now restricted based on the roles they have. Suzy, Tim, Emily, Zach, and Natalie have different roles, which are shown in the table above. For example, only Suzy and Tim can read and modify access control information. Suzy, Tim, and Emily can modify runtime state and other persistent configuration information. **Zach** can also modify runtime state and other persistent configuration information but only related to application resources. Suzy, Tim, Emily, Zach, and Natalie can read configuration and state information, but Natalie cannot update anything. For more details on each of the roles and how JBoss EAP handles RBAC, see the Role-Based Access Control and Adding RBAC to the Management Interfaces sections.

4.5. USING KERBEROS TO PROVIDE SSO FOR WEB APPLICATIONS

This scenario shows how Kerberos can be used with JBoss EAP to provide SSO for web applications. A JBoss EAP instance has been created, **EAP1**, and is running as a standalone server. Two web applications, **sampleAppA** and **sampleAppB**, have been deployed to **EAP1**. Both the web applications and **EAP1** have been configured to authenticate using desktop-based SSO with Kerberos.

4.5.1. Security

JBoss EAP provides authentication with Kerberos using the **SPNEGO** authentication method. For more information on the specifics of Kerberos and SPNEGO, please see the Third-Party SSO Implementations section. To enable JBoss EAP and deployed web applications to use Kerberos for authentication, a **kerberos-security-factory** is created to connect to the Kerberos server. A security realm, role mapper, and security domain are also created for assigning roles to users from the Kerberos server. An **http-authentication-factory** is created that uses the **kerberos-security-factory** and uses the security domain for authentication and role assignment. An authentication mechanism is exposed as **exampleSpnegoDomain** using a **SPNEGO** authentication mechanism. The **undertow** subsystem is also configured to use the **http-authentication-factory** for authentication.

Both **sampleAppA** and **sampleAppB** are configured to use **exampleSpnegoDomain** to perform authentication and get a user's roles for authorization. Both applications can also be configured with **FORM** authentication as a fallback authentication mechanism in case the security tokens cannot be passed from the operating system to the browser. If **FORM** authentication is configured as a fallback, an additional authentication mechanism, along with a supporting security domain, must be configured. This authentication mechanism is independent of Kerberos and **SPNEGO** and only has to support **FORM** authentication. In this case, an additional mechanism and supporting security domain have been configured for **FORM** authentication and exposed as **exampleFormDomain**. Each application is configured to use a **exampleFormDomain** and to provide **FORM** authentication as a fallback. Each application is also configured to secure the path **/secure/*** and supplies its own list of roles for handling authorization.

4.5.2. How It Works

The following users have been created in the Kerberos server:

Table 4.5. Kerberos Users

Username	Password
Sande	samplePass
Andrea	samplePass
Betty	samplePass
Chuck	samplePass

The following roles are mapped to the users using the security domain:

Table 4.6. User Roles

Username	Roles
Sande	all
Andrea	A
Betty	B
Chuck	

The following roles have also been configured in each application:

Table 4.7. Application Roles

Application/SP	Allowed Roles
sampleAppA	all, A
sampleAppB	all, B

On startup, **EAP1** loads the core services, followed by the **elytron** and other subsystems. The **kerberos-security-factory** establishes a connection to the Kerberos server. Both **sampleAppA** and **sampleAppB** are deployed and connect to **exampleSpnegoDomain** and **exampleFormDomain** for authentication.

Sande has logged in to a computer that is secured with Kerberos. She opens a browser and attempts to access **sampleAppA/secure/hello.html**. Because that is secured, authentication is required. **EAP1** directs the browser to send a request for a key to the Kerberos server, specifically the Kerberos Key Distribution Center that is configured on her computer. After the browser obtains a key, it is sent to **sampleAppA**. **sampleAppA** sends the ticket using **exampleSpnegoDomain** to JBoss EAP where it is unpacked and authentication is performed in conjunction with the configured Kerberos server in the **kerberos-security-factory**. Once the ticket is authenticated, Sande's role is passed back to **sampleAppA** to perform authorization. Because Sande has the **all** role, she will be able to access **sampleAppA/secure/hello.html**. If Sande tries to access **sampleAppB/secure/hello.html**, the same process will occur. She will be granted access, due to her having the **all** role. Andrea and Betty would follow the same process, but with Andrea only having access to **sampleAppA/secure/hello.html** and not **sampleAppB/secure/hello.html**. Betty would be the opposite, having access to **sampleAppB/secure/hello.html** and not **sampleAppA/secure/hello.html**. Chuck would pass authentication to either **sampleAppA/secure/hello.html** or **sampleAppB/secure/hello.html** but would not be granted access to either because he does not have any roles.

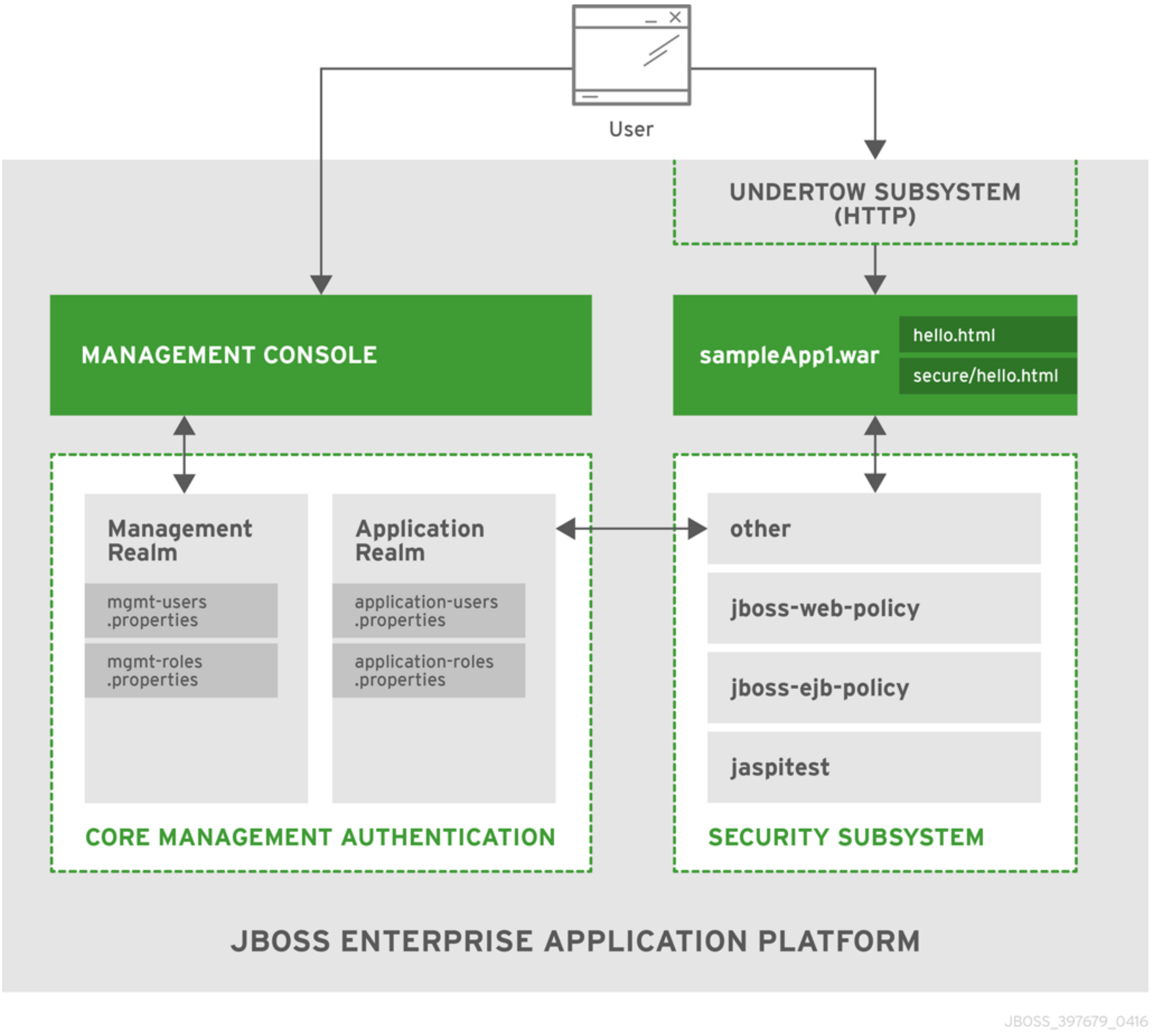
If Sande were to attempt to access **sampleAppA/secure/hello.html** from a computer not secured by Kerberos, for example a personal laptop connected to the office network, she would be directed to the **FORM** login page as a fallback. Her credentials would then be authenticated using the fallback authentication mechanism and the process would continue with authorization.

CHAPTER 5. LEGACY CORE MANAGEMENT AND SECURITY SUBSYSTEM EXAMPLE SCENARIOS

One way of understanding how JBoss EAP security and its components work together is to review real scenarios. The following sections cover several generalized but realistic situations that involve the various JBoss EAP security components and configuration options. They focus on how an application or set of applications is secured as well as how the management interfaces are secured.

5.1. RED HAT JBOSS ENTERPRISE APPLICATION PLATFORM OUT OF THE BOX

This scenario shows how JBoss EAP and a sample application are secured when no configuration changes are made after the initial installation. An application, **sampleApp1.war**, is deployed and configured to use container-based security.



5.1.1. Core Management Authentication out of the Box

5.1.1.1. Security

The Core Management Authentication offers two preconfigured security realms by default:

ManagementRealm and **ApplicationRealm**. These realms use property files to store the usernames, passwords, and roles. The **ManagementRealm**, which is used to store the authentication information and the management APIs, also defines and enables local authentication for users using the CLI on the same host as the JBoss EAP instance. The **ApplicationRealm** is preconfigured to store authentication and authorization information but for use with other applications besides the management APIs. In addition, **ApplicationRealm** comes preconfigured with local authentication enabled, but it is not commonly used.

5.1.1.2. How It Works

For this scenario, the following users have been added to a default installation of JBoss EAP:

Table 5.1. Users

Username	Password	Roles	Security Realm
Susan	Testing123!		ManagementRealm
Sarah	Testing123!	sample	ApplicationRealm
Frank	Testing123!		ApplicationRealm

On startup, the JBoss EAP instance loads the **ManagementRealm** and **ApplicationRealm** security realms.

If Susan attempts to access either of the management interfaces, HTTP or CLI, she is required to authenticate. Her username, password, and roles must match an entry in the **ManagementRealm** security realm. By default, no roles are required to access the management APIs. Sarah and Frank would not be able to access the management APIs because they are not in the **ManagementRealm** security realm.

5.1.2. Security Subsystem out of the Box

5.1.2.1. Security

The **security** subsystem comes preconfigured with four security domains: **other**, **jboss-web-policy**, **jboss-ejb-policy**, and **jaspitest**. The **other** security domain performs authentication and authorization by delegating to the realm specified in the login module, which uses **ApplicationRealm** by default.

Additional information about the default security realms and the default security domains can be found in the Security Realms and Security Domains sections.

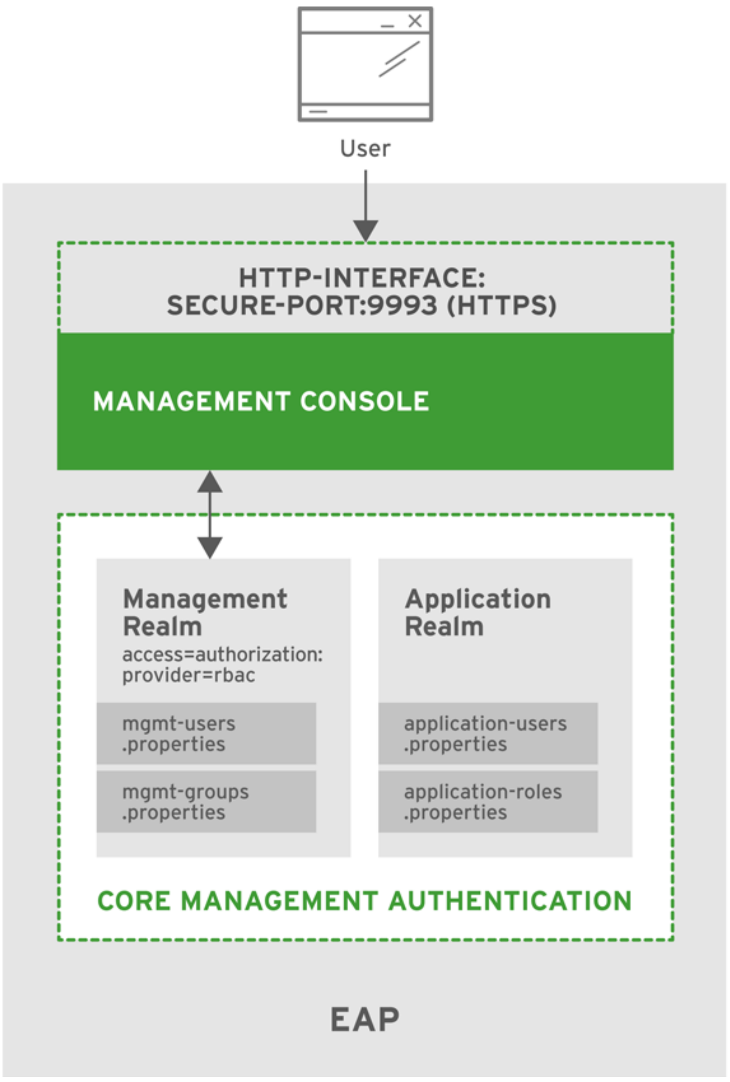
5.1.2.2. How It Works

The application **sampleApp1.war** has two HTML files, **/hello.html** and **/secure/hello.html**, and uses basic HTTP authentication to secure the path **/secure/***. It uses the **other** security domain and requires the role of **sample**. Because the **other** security domain defers to **ApplicationRealm** for its authentication and authorization information, refer to the users in the **Users** table from the previous section.

When Sarah requests **/hello.html**, she is able to view the page without authenticating. When Sarah tries to request **/secure/hello.html**, she is prompted to enter her username and password. She is able to view **/secure/hello.html** after successfully logging in. Frank and Susan, or any user, can access **/hello.html**, but neither can access /**secure/hello.html**. Frank does not have the **sample** role, and Susan is not in the **ApplicationRealm** security realm.

5.2. RED HAT JBOSS ENTERPRISE APPLICATION PLATFORM WITH HTTPS AND RBAC ADDED TO THE MANAGEMENT INTERFACES

This scenario shows how JBoss EAP is secured when HTTPS is enabled for the management interfaces and RBAC is added to the **ManagementRealm** security realm.



5.2.1. Security

JBoss EAP provides support for using HTTPS with the management interfaces, which includes the management console. HTTPS can be enabled by configuring the **secure-interface** element, adding a **secure-port** to the **http-interface** section of the management interface, and configuring an existing or new security realm with the desired settings, for example, server-identities, protocol, keystore, alias, and so on. This enables the management interfaces to use SSL/TLS for all HTTP traffic. For more background information on HTTPS and securing the management interfaces in general, see the Advanced Security section.

JBoss EAP also offers the ability to enable RBAC on the management interfaces using a couple of different authentication schemes. This example uses username and password authentication with the existing property files used in the **ManagementRealm**. RBAC is enabled by setting the **provider** attribute to **rbac** for the management interface and updating the **ManagementRealm** with the desired users and roles.

5.2.2. How It Works

For this scenario, the following users have been added to the existing security realms:

Table 5.2. Management Users

Username	Password	Security Realm
Suzy	Testing123!	ManagementRealm
Tim	Testing123!	ManagementRealm
Emily	Testing123!	ManagementRealm
Zach	Testing123!	ManagementRealm
Natalie	Testing123!	ManagementRealm

Based on group membership, the users have also been mapped to the following RBAC roles:

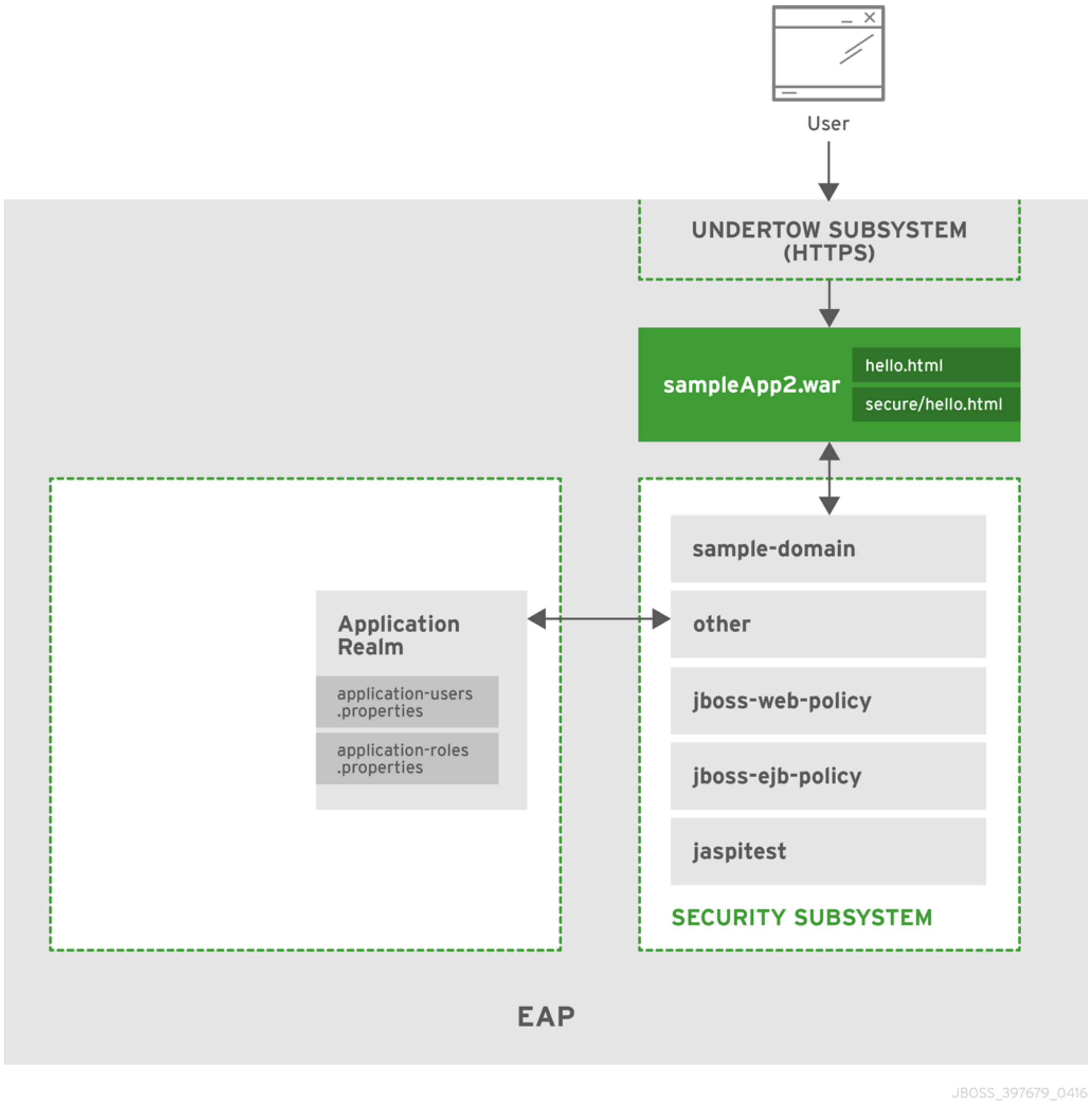
Table 5.3. RBAC Roles

Username	RBAC Role
Suzy	SuperUser
Tim	Administrator
Emily	Maintainer
Zach	Deployer
Natalie	Monitor

On startup, JBoss EAP loads the **ManagementRealm**, with the RBAC configuration, and management interfaces as part of the core services, which also starts the **http-interface**, configured for HTTPS, for the management interfaces. Users now access the management interfaces via HTTPS, and RBAC has also been enabled and configured. If RBAC is not enabled, any user in the **ManagementRealm** security realm is considered a **SuperUser** and has unlimited access. Because RBAC has been enabled, each user is now restricted based on the roles they have. Suzy, Tim, Emily, Zach, and Natalie have different roles, which are shown in the table above. For example, only Suzy and Tim can read and modify access control information. Suzy, Tim, and Emily can modify runtime state and other persistent configuration information. **Zach** can also modify runtime state and other persistent configuration information but only related to application resources. Suzy, Tim, Emily, Zach, and Natalie can read configuration and state information, but Natalie cannot update anything. For more details on each of the roles and how JBoss EAP handles RBAC, see the Role-Based Access Control and Adding RBAC to the Management Interfaces sections.

5.3. RED HAT JBOSS ENTERPRISE APPLICATION PLATFORM WITH AN UPDATED SECURITY SUBSYSTEM INCLUDING HTTPS

This scenario shows how a sample application running on JBoss EAP is secured when a new security domain is added and HTTPS is enabled. An application, **sampleApp2.war**, is deployed that is configured to use container-based security, **sample-domain**, and HTTPS.



5.3.1. Security

JBoss EAP provides support for using HTTPS for use with applications, which is handled using the **undertow** subsystem. A new connector for HTTPS is added to the **undertow** subsystem and is configured with the desired settings, for example, protocol, port, keystore, and so on. Once the configuration is saved, web applications can start accepting HTTPS traffic on the configured port. A new security domain called **sample-domain** has been added, and it uses the **IdentityLoginModule** for authentication. **sampleApp2.war** is configured to use **sample-domain** to authenticate users.

5.3.2. How It Works

A security domain, **sample-domain**, has been created and configured to use the **IdentityLoginModule**. The following credentials have been configured in the login module:

Table 5.4. sample-domain Users

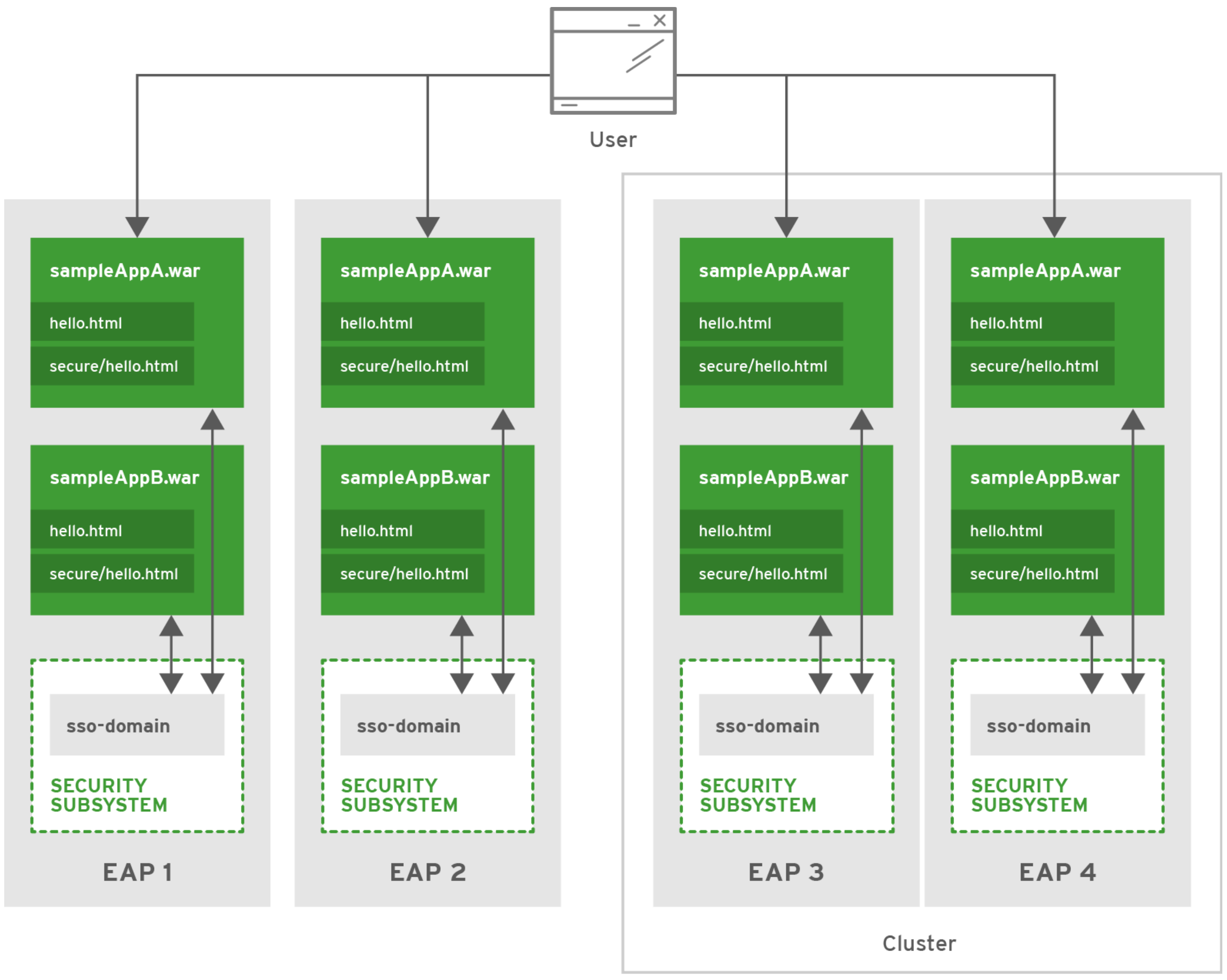
Username	Password	Roles
Vincent	samplePass	sample

On startup, JBoss EAP loads the core services and starts the **undertow** and **security** subsystems that manage the HTTPS connections for all web applications and the **sample-domain**, respectively. **sampleApp2.war** is loaded and looks for **sample-domain** for providing authentication and authorization for its secured URLs. **sampleApp2.war** has two HTML files, **/hello.html** and **/secure/hello.html**, and uses basic HTTP authentication to secure the path **/secure/***. It uses the **sample-domain** security domain and requires the role of **sample**.

When Vincent requests **/hello.html**, he is able to view the page without authenticating. When Vincent tries to request **/secure/hello.html**, he is prompted to enter his username and password. After successfully logging in, he is able to view **/secure/hello.html**. All other users can access **/hello.html** without logging in, but none can access **/secure/hello.html** because Vincent is the only user in **sample-domain**. This also applies to all traffic handled over HTTPS.

5.4. SSO FOR WEB APPLICATIONS ON RED HAT JBOSS ENTERPRISE APPLICATION PLATFORM

This scenario shows how web applications make use of clustered and non-clustered SSO on JBoss EAP. Four JBoss EAP instances are created: **EAP1**, **EAP2**, **EAP3**, and **EAP4**. **EAP1** and **EAP2** operate as standalone servers, and **EAP3** and **EAP4** operate as a two-node cluster. Two web applications, **sampleAppA** and **sampleAppB**, have been deployed to each of the four JBoss EAP instances.



5.4.1. Security

JBoss EAP provides support for clustered and non-clustered SSO with web applications by using a combination of the **security**, **undertow**, and **infinispan** subsystems. The **security** subsystem provides a security domain for performing authentication and authorization, while the **infinispan** and **undertow** subsystems help with caching and distributing the SSO information between all the web applications on an JBoss EAP instance or across a JBoss EAP cluster. All four EAP instances have a security domain, **sso-domain**, configured to use the **IdentityLoginModule**. **sampleAppA** and **sampleAppB** have been configured to use the **sso-domain** security domain to secure the path **/secure/*** and require the role of **sample** to access it. The following credentials have been configured in the **sso-domain** login module:

Table 5.5. sso-domain Users

Username	Password	Roles
Jane	samplePass	sample

All four JBoss EAP instances have also been configured to start up with either the **standalone-full-ha** or **full-ha** profile, which adds the **infinispan** subsystem and other functionality needed for enabling SSO in this scenario. The **web** cache-container and **SSO** replicated-cache have also been added, and the **undertow** subsystem has been configured to use them both. **EAP1** and **EAP2** have configured their **undertow** subsystems for non-clustered SSO, while the cluster containing **EAP3** and **EAP4** has been configured to use clustered SSO.

APPLICATION CLUSTERING VS. CLUSTERED SSO

There is a distinct difference between a clustered web application and clustered SSO. A clustered web application is one which is distributed across the nodes of a cluster to spread the load of hosting that application. In clustered applications marked as distributable, all new sessions and changes to existing sessions are replicated to other members of the cluster. Clustered SSO allows for replication of security context and identity information, regardless of whether the applications are themselves clustered. Although these technologies may be used together, they are mutually exclusive and may be used independently.

5.4.2. How It Works

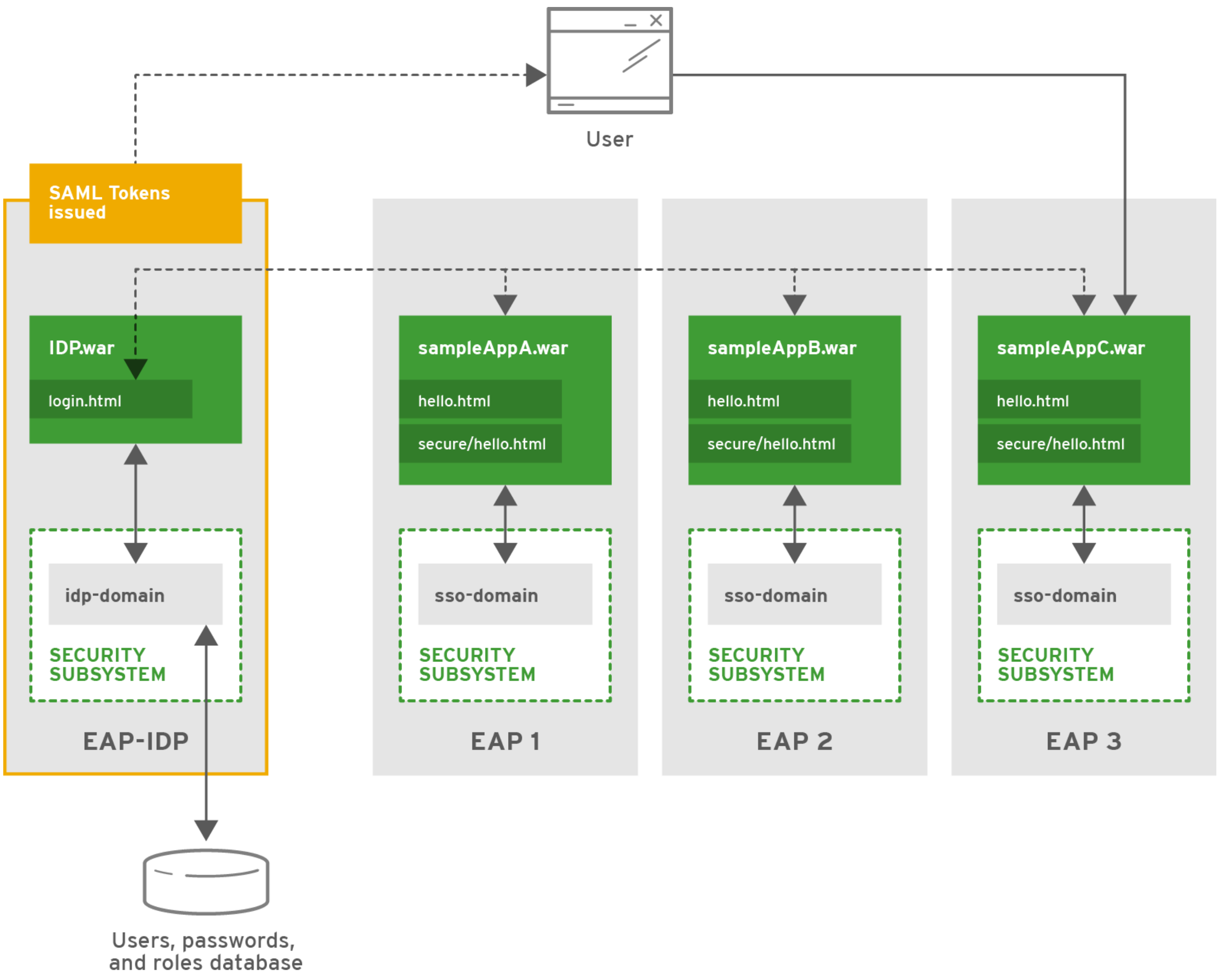
On startup, JBoss EAP loads the core services and starts the **security**, **undertow**, and **infinispan** subsystems that manage **sso-domain** and the associated caching for SSO information. **sampleAppA.war** and **sampleAppB.war** are loaded on all four JBoss EAP instances, each of which look for **sso-domain** for providing authentication and authorization.

If Jane attempts to access **EAP1/sampleAppA/secure/hello.html**, she will be asked to authenticate. After providing the correct information, she will be allowed to see **EAP1/sampleAppA/secure/hello.html**. Jane’s session will be added in the SSO caches used by the **undertow** and **infinispan** subsystems. If she attempts to access **EAP1/sampleAppB/secure/hello.html**, she will not be asked to reauthenticate. **sampleAppB** running on **EAP1** will find her session using the **undertow** subsystem caches, along with the **infinispan** subsystem, and grant her access because she is already authenticated. If Jane attempts to access either **EAP2/sampleAppA/secure/hello.html** or **EAP2/sampleAppB/secure/hello.html**, she will be asked to authenticate again because **EAP1** and **EAP2** do not share a cache.

If Jane attempts to access **EAP3/sampleAppA/secure/hello.html**, she will be asked to authenticate and her session will be stored in the SSO caches. These caches are stored across the entire cluster, so if Jane attempts to log into **EAP3/sampleAppB/secure/hello.html**, **EAP4/sampleAppA/secure/hello.html**, or **EAP4/sampleAppB/secure/hello.html**, she will not have to reauthenticate. If either **EAP3** or **EAP4** are restarted, Jane’s SSO information will remain in the cache because the other JBoss EAP instance and the cluster remain running, thus preserving the cache. Similarly, if Jane’s session is invalidated, it ripples across the cache, and she will be asked to reauthenticate regardless of which application or server she tries to access in the cluster.

5.5. MULTIPLE RED HAT JBOSS ENTERPRISE APPLICATION PLATFORM INSTANCES AND MULTIPLE APPLICATIONS USING BROWSER-BASED SSO WITH SAML

This scenario shows how multiple instances of JBoss EAP are secured and browser-based SSO is added. Three separate, non-clustered JBoss EAP instances are configured: **EAP1**, **EAP2**, and **EAP3**. Three sample applications: **sampleAppA.war**, **sampleAppB.war**, and **sampleAppC.war**, are configured to use browser-based SSO for authentication.



5.5.1. Security

JBoss EAP provides support for doing browser-based SSO via SAML with web applications as well as hosting an identity provider. To host an identity provider, a security domain, in this case named **idp-domain**, must be configured with an authentication mechanism defined. In this case, **idp-domain** is configured to use the **DatabaseLoginModule** as the authentication mechanism. The IDP application, for example **IDP.war**, is deployed to that JBoss EAP instance, in this example **EAP-IDP**, and configured to use **idp-domain** as its identity store. **IDP.war** uses the identity store in combination with functionality in the application to authenticate users as well as issue SAML tokens containing the user's identity and role information. Three additional JBoss EAP instances: **EAP1**, **EAP2**, and **EAP3**, each host one distinct application that will serve as an individual SP, **sampleAppA.war**, **sampleAppB.war**, and **sampleAppC** respectively. Each JBoss EAP instance has a security domain, **sso-domain**, configured to use the **SAML2LoginModule** as the authentication mechanism. Each application contains functionality to support SSO, use **IDP.war** as an identity provider, and use HTTP/POST binding for authentication. Each application is also configured to use **sso-domain** to secure the path **/secure/*** and supplies its own list of roles for handling authorization.

5.5.2. How It Works

For this scenario, the following users have been added to the database used by the **DatabaseLoginModule** in the **idp-domain** security domain:

Table 5.6. idp-domain Users

Username	Password	Roles
Eric	samplePass	all
Amar	samplePass	AB
Brian	samplePass	C
Alan	samplePass	

On startup of **EAP-IDP**, the management interfaces start, followed by the subsystems and deployed applications, including the **security** subsystem, which includes **idp-domain**, and **IDP.war**. **idp-domain** connects to the database and loads the usernames, passwords, and roles as configured in the **DatabaseLoginModule** login module. To prevent sensitive information from being stored in plain text in the configuration of the **DatabaseLoginModule** login module, password hashing is configured to obscure certain fields, for example, password for the database. **IDP.war** uses **idp-domain** for authentication and authorization. **IDP.war** is also configured, using **jboss-web.xml**, **web.xml**, **picketlink.xml**, and **jboss-deployment-structure.xml**, to issue SAML tokens to authenticated users and supplies a simple login form for users to authenticate against. This allows it to serve as an IDP.

On startup of **EAP1**, **EAP2** and **EAP3**, the management interfaces start, followed by the subsystems and deployed applications, including the **security** subsystem, which includes **sso-domain** on each instance, and **sampleAppA.war**, **sampleAppB.war**, and **sampleAppC.war**, respectively. **sso-domain** is configured to use the **SAML2LoginModule** login module but relies on the application to supply the proper SAML token. This means any application using **sso-domain** must properly connect to an IDP to obtain a SAML token. In this case, **sampleAppA**, **sampleAppB**, and **sampleAppC** are each configured, via **jboss-web.xml**, **web.xml**, **picketlink.xml**, and **jboss-deployment-structure.xml**, to obtain SAML tokens from an IDP, **IDP.war**, using that IDP's login form.

Each application is also configured with its own set of allowed roles:

Table 5.7. Application/SP roles

Application/SP	Allowed Roles
sampleAppA	all, AB
sampleAppB	all, AB
sampleAppC	all, C

When an unauthenticated user attempts to access the URLs secured by **sso-domain**, that is, **/secure/***, of any application, that user is redirected to the login page at **IDP.war**. The user then authenticates using the form and is issued a SAML token containing their identity and role information. The user is redirected to the original URL, and their SAML token is presented to the application, SP. The application ensures the SAML token is valid and then authorizes the user based on the roles provided by the SAML token and the ones configured in the application. Once a user is issued a SAML token, they use that token to be authenticated and authorized on the other applications secured by SSO using the same IDP. Once the SAML token expires or becomes invalidated, the user will be required to obtain a new SAML token from the IDP.

In this example, if Eric accesses **EAP1/sampleAppA/secure/hello.html**, he is redirected to **EAP-IDP/IDP/login.html** to log in. After successfully logging in, he is issued a SAML token containing his user information and the role **all** and is redirected to **EAP1/sampleAppA/secure/hello.html**. His SAML token is presented to **sampleAppA** to be checked and to authorize him based on his roles. Because **sampleAppA** allows the roles **all** and **AB** to access **/secure/*** and Eric has the role **all**, he is allowed to access **EAP1/sampleAppA/secure/hello.html**.

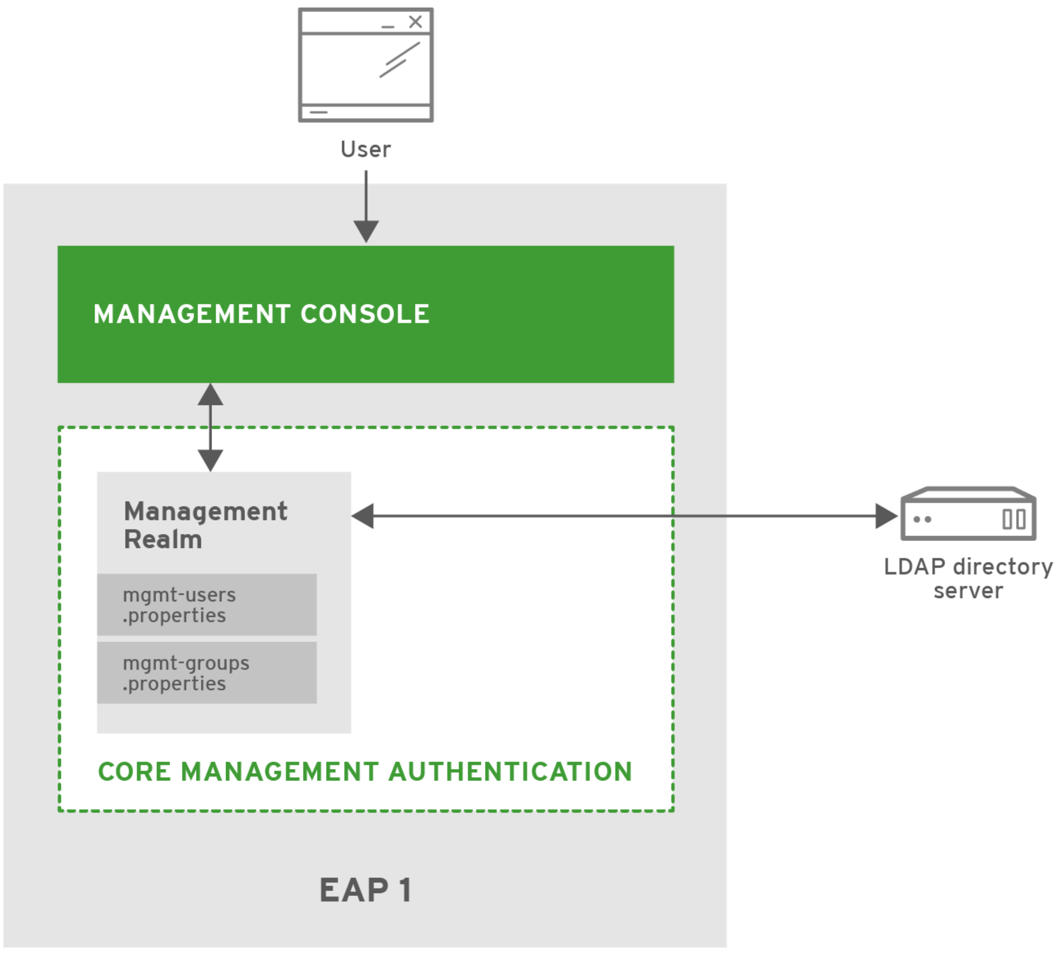
If Eric tries to access **EAP2/sampleAppB/secure/hello.html**, because he is not already authenticated against that SP, he is redirected to **IDP.war** with an authentication request. Because Eric has already authenticated against the IDP, he is redirected to **EAP2/sampleAppB/secure/hello.html** by **IDP.war** with a new SAML token for that SP without having to reauthenticate. **sampleAppB** checks his new SAML token and authorizes him based on his role. Because **sampleAppB** allows the roles **all** and **AB** to access **/secure/*** and Eric has the role **all**, he is allowed to access **EAP2/sampleAppB/secure/hello.html**. The same thing would apply if Eric were to try to access **EAP3/sampleAppC/secure/hello.html**.

If Eric were to return to **EAP1/sampleAppA/secure/hello.html**, or any URL under **EAP2/sampleAppB/secure/*** or **EAP3/sampleAppC/secure/***, after his SAML token became invalidated via global logout, he would be redirected to **EAP-IDP/IDP/login.html** to authenticate again and be issued a new SAML token.

If Amar attempted to access **EAP1/sampleAppA/secure/hello.html** or **EAP2/sampleAppB/secure/hello.html**, he would be directed through the same flow as Eric. If Amar attempted to access **EAP3/sampleAppC/secure/hello.html**, he would still be required to have or obtain a SAML token but would be restricted from viewing **EAP3/sampleAppC/secure/hello.html** because his role **AB** only allows him to access **EAP1/sampleAppA/secure/*** and **EAP2/sampleAppB/secure/***. Brian is in a similar situation, but he is only allowed to access **EAP3/sampleAppC/secure/***. If Alan tries to access any service provider’s secured area, he would still be required to have or obtain a SAML token but would be restricted from seeing anything because he has no role. Each SP also has an unsecured area which any user, authorized or not, can view without obtaining a SAML token.

5.6. USING LDAP WITH THE MANAGEMENTREALM

This scenario shows the **ManagementRealm** using LDAP for securing the management interfaces. A JBoss EAP instance has been created, **EAP1**, and is running as a standalone server. The **ManagementRealm** on **EAP1** has also been updated to use LDAP as the authentication and authorization mechanism.



5.6.1. Security

JBoss EAP supports using LDAP, as well as Kerberos, for authentication in security realms. This is accomplished by updating the existing **ManagementRealm** to use **ldap** as the authentication type and creating an outbound connection to the LDAP server. This changes the authentication mechanism from **digest** to **BASIC** / **Plain** and will transmit usernames and passwords in the clear over the network by default.

5.6.2. How It Works

The following users have been added to the LDAP directory:

Table 5.8. Management Users

Username	Password	Roles
Adam	samplePass	SuperUser
Sam	samplePass	

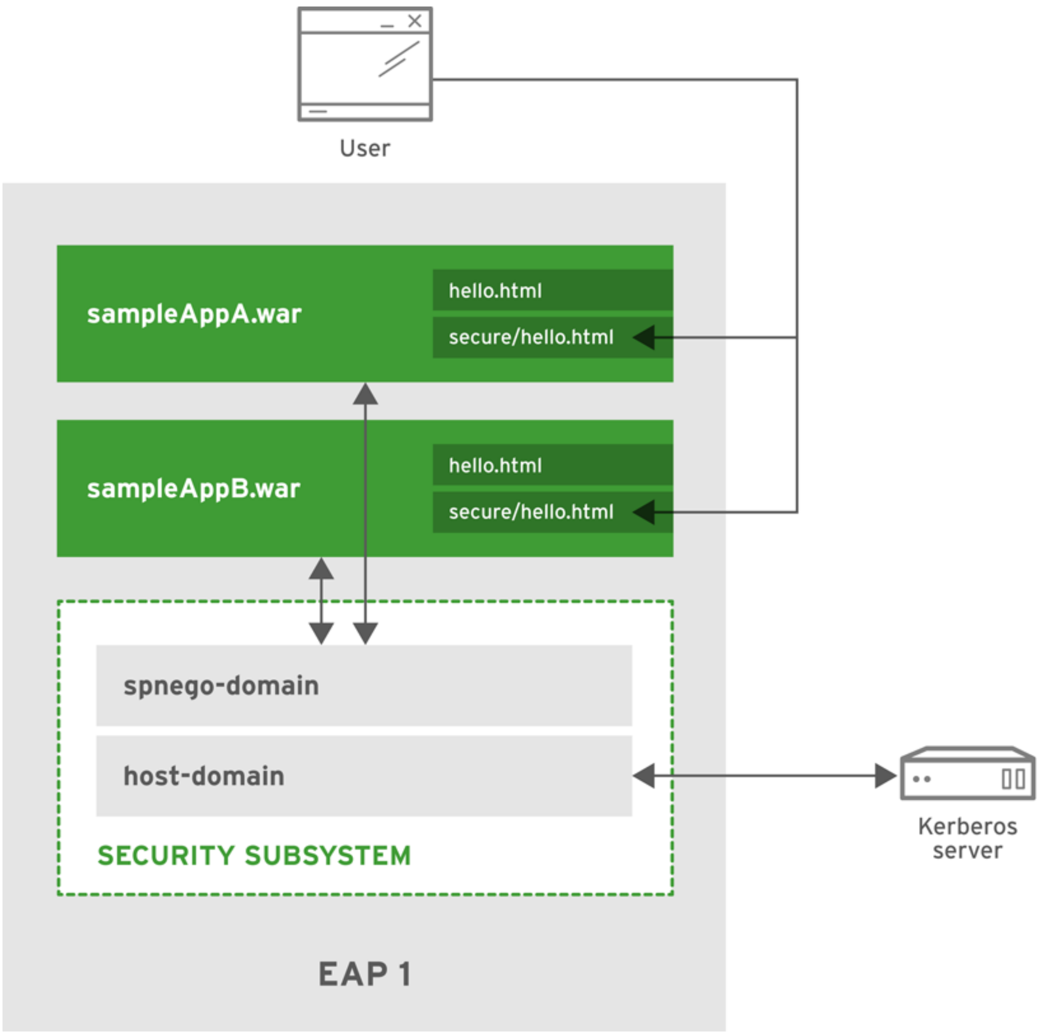
On startup, **EAP1** loads the core services, including **ManagementRealm** and the management interfaces. **ManagementRealm** connects to the LDAP directory server and provides the authentication for the management interfaces as needed.

If Adam` attempts to access a management interface, he will be forced to authenticate. His credentials will be passed to the **ManagementRealm** security realm, which will use the LDAP directory server for authentication. Since **EAP1** is using the default simple access controls, after authentication, Adam's roles will not be checked and he will be granted access. The same process will occur if Sam attempts to access the management interface.

If RBAC was enabled after Adam is authenticated, his roles would be passed back to the management interface for authorization. Because Adam has the **SuperUser** role, he would be granted access to the management interface. If Sam attempted to access a management interface with RBAC enabled, he would be authenticated via LDAP but would be denied access because he does not have a proper role.

5.7. USING DESKTOP SSO, USING KERBEROS, TO PROVIDE SSO FOR WEB APPLICATIONS

This scenario shows how Kerberos can be used with JBoss EAP to provide SSO for web applications. A JBoss EAP instance has been created, **EAP1**, and is running as a standalone server. Two web applications, **sampleAppA** and **sampleAppB**, have been deployed to **EAP1**. Both the web applications and **EAP1** have been configured to authenticate using desktop-based SSO via Kerberos.



5.7.1. Security

JBoss EAP offers support for using Kerberos for SSO in web applications via SPNEGO and JBoss Negotiation. For more information on the specifics of Kerberos and SPNEGO, see the Third-Party SSO Implementations section. To enable JBoss EAP and deployed web applications to use Kerberos for authentication, two security domains must be created. The first security domain, **host-domain**, is configured with the **kerberos** login module to connect to the Kerberos server. This allows JBoss EAP to authenticate at the container level. A second security domain, **spnego-domain**, is created with two login modules. One uses the **spnego** login module to connect to **host-domain** to authenticate users. The second can use any other login module to load role information for use in authorization decisions, for example, **UsersRoles** using properties files to map users to roles.

These two login modules also make use of **password-stacking** to map the users and roles in the authorization module with the users in the authentication login module. **EAP1** is configured with both **host-domain** and **spnego-domain**. Applications are configured, via **web.xml** and **jboss-web.xml**, to use **spnego-domain** to perform authentication and get a user's roles for authorization. Applications can also be configured with FORM authentication as a fallback authentication mechanism in case the security tokens cannot be passed from the operating system to the browser. If FORM authentication is configured as a fallback, an additional security domain must be configured to support it. This security domain is independent of Kerberos and SPNEGO and only has to support FORM authentication, that is, username and password validation and roles. Each application is configured to use **spnego-domain** and to provide FORM authentication as a fallback. Each application is also configured to secure the path **/secure/*** and supplies its own list of roles for handling authorization.

5.7.1.1. How It Works

The following users have been created in the Kerberos server:

Table 5.9. Kerberos Users

Username	Password
Brent	samplePass
Andy	samplePass
Bill	samplePass
Ron	samplePass

The following roles have mapped to the users via an additional module that is chained by setting the **password-stacking** option to **useFirstPass**:

Table 5.10. User Roles

Username	Roles
Brent	all
Andy	A
Bill	B
Ron	

The following roles have also been configured in each application:

Table 5.11. Application Roles

Application/SP	Allowed Roles
sampleAppA	all, A
sampleAppB	all, B

On startup, **EAP1** loads the core services, followed by the **security** and other subsystems. **host-domain** establishes a connection to the Kerberos server, and **spnego-domain** connects to **host-domain**. **sampleAppA** and **sampleAppB** are deployed and connect to **spnego-domain** for authentication.

Brent has logged in to a computer that is secured with Kerberos. He opens a browser and attempts to access **sampleAppA/secure/hello.html**. Because that is secured, authentication is required. **EAP1** directs the browser to send a request for a key to the Kerberos server, specifically the Kerberos Key Distribution Center that is configured on his computer. After the browser obtains a key, it is sent to **simpleAppA**. **simpleAppA** sends the ticket to **spnego-domain**, where it is unpacked and authentication is performed by **host-domain**, in conjunction with the configured Kerberos server. Once the ticket is authenticated, Brent's role is passed back to **simpleAppA** to perform authorization. Because Brent has the **all** role, he will be able to access **sampleAppA/secure/hello.html**. If Brent tries to access **sampleAppB/secure/hello.html**, the same process will occur. He will be granted access, due to him having the **all** role. Andy and Bill would follow the same process, but with Andy only having access to **sampleAppA/secure/hello.html** and not **sampleAppB/secure/hello.html**. Bill would be the opposite, having access to **sampleAppB/secure/hello.html** and not **sampleAppA/secure/hello.html**. Ron would pass authentication to either **sampleAppA/secure/hello.html** or **sampleAppB/secure/hello.html** but would not be granted access to either because he has no role.

If Andy were to attempt to access **sampleAppA/secure/hello.html** from a computer not secured by Kerberos, for example a personal laptop connected to the office network, he would be directed to the FORM login page as a fallback login mechanism. His credentials would then be authenticated via the fallback security domain and the process would continue with authorization.

Revised on 2024-02-08 08:09:08 UTC