JAKARTA EE

Jakarta Authentication

Jakarta Authentication Team,
https://projects.eclipse.org/projects/ee4j.authentication

3.0, February 28, 2022: Final

Table of Contents

```
Specification: Jakarta Authentication

Version: 3.0

Status: Final

Release: February 28, 2022
```

Copyright

Eclipse Foundation Specification License

By using and/or copying this document, or the Eclipse Foundation document from which this statement is linked, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions:

Permission to copy, and distribute the contents of this document, or the Eclipse Foundation document from which this statement is linked, in any medium for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the document, or portions thereof, that you use:

- link or URL to the original Eclipse Foundation document.
- All existing copyright notices, or if one does not exist, a notice (hypertext is preferred, but a textual representation is permitted) of the form: "Copyright © [$date-of-document] Eclipse Foundation, Inc. https://www.eclipse.org/legal/efsl.php"

Inclusion of the full text of this NOTICE must be provided. We request that authorship attribution be provided in any software, documents, or other items or products that you create pursuant to the implementation of the contents of this document, or any portion thereof.

No right to create modifications or derivatives of Eclipse Foundation documents is granted pursuant to this license, except anyone may prepare and distribute derivative works and portions of this document in software that implements the specification, in supporting materials accompanying such software, and in documentation of such software, PROVIDED that all such works include the notice below. HOWEVER, the publication of derivative works of this document for use as a technical specification is expressly prohibited.

The notice is:

"Copyright © 2018, 2020 Eclipse Foundation. This software or document includes material copied from or derived from Jakarta® Authentication https://jakarta.ee/specifications/authentication/2.0/"

Disclaimers

THIS DOCUMENT IS PROVIDED "AS IS," AND THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DOCUMENT ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE DOCUMENT OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of the copyright holders or the Eclipse Foundation may NOT be used in advertising or publicity pertaining to this document or its contents without specific, written prior permission. Title to copyright in this document will at all times remain with copyright holders.

Preface

This document is the Jakarta Authentication Specification, version 2.0.

Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119, "Key words for use in RFCs to Indicate Requirement Levels" [RFC2119].

Audience

This document is intended for developers of a Compatible Implementation and of the Technology Compatibility Kit and for those who will be delivering implementations of this technology in their products.

Specification Scope

Jakarta Authentication defines a general low-level SPI for authentication mechanisms, which are controllers that interact with a caller and a container's environment to obtain the caller's credentials, validate these, and pass an authenticated identity (such as name and groups) to the container.

Jakarta Authentication consists of several profiles, with each profile telling how a specific container (such as Jakarta Servlet) can integrate with- and adapt to this SPI.

Abstract

This specification defines a service provider interface (SPI) by which authentication providers implementing message authentication mechanisms may be integrated in client or server message processing containers or runtimes. Authentication providers integrated through this interface operate on network messages provided to them by their calling container. They transform outgoing messages such that the source of the message may be authenticated by the receiving container, and the recipient of the message may be authenticated by the message sender. They authenticate incoming messages and return to their calling container the identity established as a result of the message authentication. The SPI is applicable to diverse messaging protocols (including SOAP, Jakarta Messaging, and HTTP) and message processing runtimes (including Jakarta EE containers).

This specification extends the pluggable authentication concepts of the Java Authentication and Authorization Service (JAAS) to the authentication of network messages. This effect is achieved by evolving the JAAS login model to facilitate the integration of security functionality at differentiated points within a logical message processing model and by defining corresponding authentication interfaces that make the network messages available for processing by authentication modules.

Acknowledgments

The authors would like to thank the original JCP JSR-196 Expert Group and Contributors.

Expert Group under the JCP

Steven Bazyl RSA Security, Inc.	Shing Wai Chan Sun Microsystems	Herb Erickson Novell, Inc.
Johan Gellner Tmax Soft, Inc.	Steven Kinser Novell, Inc.	Boris Koberle Sap AG.
Mikko Kolehmainen Nokia Networks	Charlie Lai Sun Microsystems	Hal Lockart BEA Systems
Thomas Maslen Quest Software	Cameron Morris Novell, Inc.	Larry McCay Individual
Ron Monzillo Sun Microsystems	Anthony Nadalin IBM	Nataraj Nagaratnam IBM
Raymond K. Ng Oracle Corporation	Arvind Prabhakar Sun Microsystems	Anil Saldhana JBoss, Inc.
Rajiv Shivane Pramati Technologies	Neil Smithline BEA Systems	Jeppe Sommer Trifork

Contributors under the JCP

Venu Gopal Sun Microsystems	Will Hopkins Oracle America, Inc.	V. B. Kumar Jayanti Sun Microsystems
Manveen Kaur Sun Microsystems	Raja Perumal Sun Microsystems	Tim Quinn Oracle America, Inc.
Gursharan Singh Sun Microsystems	Anil Tappetla Sun Microsystems	Arjan Tijms

Chapter 1. Overview

This chapter introduces the message processing model facilitated by this specification and the interfaces defined to integrate message authentication facilities within this model.

1.1. Message Processing Model

A typical message interaction between a client and server begins with a request from the client to the server. The server receives the request and dispatches it to a service to perform the requested processing. When the service completes, it may create a response that is returned back to the client.

The SPI defined by the specification is structured such that message processing runtimes can inject security processing at four points in the typical message interaction scenario. A message processing runtime uses the SPI at these points to delegate the corresponding message security processing to authentication providers (that is, authentication modules) integrated into the runtime by way of the SPI.

The following diagram depicts the four interaction points. The names of the interaction points represent the methods of the corresponding `ClientAuthModule` (client authentication module) and `ServerAuthModule` (server authentication module) interfaces defined by the SPI.

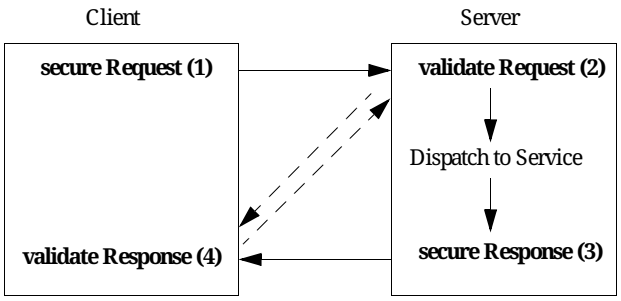

Figure 1-1 Message Processing Model [1]

1.1.1. Authentication Modules

As described above, there are two types of authentication modules. A client authentication module implements the `ClientAuthModule` interface and is invoked (indirectly) by a message processing runtime at points 1 and 4 (that is, `secureRequest` and `validateResponse`) in the message processing model. A server authentication module implements the ServerAuthModule interface and is invoked (indirectly) by a message processing runtime at points 2 and 3 (that is, `validateRequest` and `secureResponse`) in the message processing model.

When an authentication module is invoked at the identified message processing points, it is provided access to the request and response messages (as appropriate to the point in the interaction) and proceeds to secure or validate them as appropriate. For example, when `secureRequest` is invoked on a client authentication module, the module may attach a user name and password to the request

message. Similarly, when `validateRequest` is called, the server authentication module may extract a user name and password from the message and validate them against a user database. Note that authentication modules are responsible for securing or validating messages, while the message processing runtime remains responsible for transport of messages and invocation of the corresponding application level processing.

A message processing runtime invokes client authentication modules by interacting with a client authentication context object, and server authentication modules by interacting with a server authentication context object. An authentication context object is an implementation of either the `ClientAuthContext` or `ServerAuthContext` interface as defined by this specification. A message processing runtime may acquire the authentication context objects that it uses to invoke authentication modules by interacting with an authentication context configuration object. An authentication context configuration object is an implementation of either the ClientAuthConfig or ServerAuthConfig interface as defined by this specification.

1.1.2. Authentication Contexts

An authentication context is responsible for constructing, initializing, and coordinating the invocation of one or more encapsulated authentication modules. If the context implementation supports the configuration of multiple authentication modules within a context (for example, as sufficient alternatives), the context coordinates the invocation of the authentication modules on behalf of both the message processing runtime and the authentication modules.

A client message processing runtime interacts with an implementation of the `ClientAuthContext` interface to invoke the authentication modules of the context to perform message processing at points 1 and 4 (secureRequest and validateResponse) of the message processing model. Similarly, a server message processing runtime interacts with an implementation of the `ServerAuthContext` interface to invoke the modules of the context to perform message processing at points 2 and 3 (`validateRequest` and `secureResponse`) of the message processing model.

1.1.3. Authentication Context Configuration

An authentication context configuration object serves a message processing runtime as the source of authentication contexts pertaining to the messages of an application at a messaging layer. The context configuration implementation is responsible for returning authentication context objects that encapsulate authentication module invocations sufficient to satisfy the security policy configured for an application message. A message processing runtime may use a representation of the message being processed to obtain the corresponding authentication context from the appropriate authentication context configuration object.

A client authentication context configuration object implements the `ClientAuthConfig` interface and provides `ClientAuthContext` objects for use by a message processing runtime at points 1 and 4 (`secureRequest` and `validateResponse`) in the message processing model. A server authentication context configuration object implements the `ServerAuthConfig` interface and provides `ServerAuthContext` objects for use by a message processing runtime at points 2 and 3 (`validateRequest` and `secureResponse`) in the

message processing model.

A message processing runtime may acquire authentication context configuration objects by interacting with a provider of authentication context configuration objects.

1.1.4. Authentication Context Configuration Providers

An authentication context configuration provider is an implementation of the `AuthConfigProvider` interface. An authentication context configuration provider serves as a source of authentication context configuration objects, where as noted above, each configuration object serves as the source of authentication contexts pertaining to the messages of an application at a messaging layer.

An authentication context configuration provider embodies the implementation of a message authentication configuration mechanism. Each such configuration mechanism encapsulates the message authentication processing pertaining to applications in configuration objects that return context objects that coordinate the invocation of pluggable authentication modules to perform message authentication on behalf of the corresponding applications.

The `AuthConfigFactory` class serves as the catalog or registry of authentication context providers available for use by a runtime. A message processing runtime may interact with the factory to obtain or establish the provider registered for an application context and messaging layer.

1.1.5. Request and Response Messages

Request and response messages are Java representations of the corresponding protocol messages, and are passed to authentication modules through an implementation of the `MessageInfo` interface which provides common methods for accessing protocol specific messages.

Authentication Modules that operate on messages for a specific protocol (for example, SOAP messages) are expected to be configured for and called from an appropriate message processing runtime (for example, a SOAP message processing runtime).

1.1.6. Message Authentication Policy

When an authentication module is initialized within an authentication context, it is passed policy information that specifies what authentication guarantees the module is to enforce when securing or validating request and response messages within that context. Policy information is conveyed by the authentication context to the authentication module in the form of MessagePolicy objects. Two separate `MessagePolicy` objects are passed to the module through its `initialize method`: One defines the message authentication policy to be applied to the request message, and the other defines the message authentication policy to be applied to the response.

A message authentication policy can be targeted at specific parts of the related message or to the message as a whole, and conveys the high level authentication guarantees that must be enforced by the modules of a context. The policy may specify, for example, that the source of a request must be authenticated. The mechanisms by which a module enforces the guarantees, or, in other words, *how*

the module enforces the guarantees is up to the module.

1.1.7. Authentication Exchanges and State

Authentication modules should be implemented such that they may be invoked concurrently and such that they are able to apply and establish independent security identities for concurrent invocations. To this end, modules should rely on their invocation parameters and the callbacks supported by the `CallbackHandler` with which they were initialized to obtain any information required to establish the invocation context for which they were invoked.

In a multi-message authentication scenario, it is the responsibility of the authentication modules involved in the authentication to tie together or correlate the messages that comprise the authentication exchange. In addition to message correlation to tie together the messages required to complete an authentication, message correlation may also be employed post-authentication such that a prior authentication result or session may be applied to a subsequent invocation. Modules are expected to perform their message correlation function based on the parameters of their invocation and with the benefit of any additional facilities provided by the invoking runtime (for example, through their `CallbackHandler`).

To assist modules in performing their correlation function, calls made to `validateResponse` must be made with the same `messageInfo` object used in the call to `secureRequest` (or `validateResponse`) that elicited the response. Similarly, calls made to `secureResponse` must be made with the same `messagInfo` object that was passed to `validateRequest` (for the corresponding request message). Modules are also expected to avail themselves of persisted state management facilities (for example, jakarta.servlet.http.HttpSession facilities) provided by the invoking runtime. The use of such facilities prior to authentication may increase the system's susceptibility to a denial-of-service attack, and their use by authentication modules should be considered in that regard.

For security mechanisms or protocols where message correlation is dependent on the content of exchanged messages, it is the responsibility of the authentication modules to ensure that the required correlation information is inserted in the exchanged messages. For security mechanisms where message correlation is dependent on context external to the exchanged messages, such as the transport connection or session on which messages are received, the authentication modules will be dependent on correlation related facilities provided by the runtime.

This version of this specification does not define the interfaces by which runtimes present correlation facilities to authentication modules.

1.1.8. Callbacks for Information From the Runtime

Authentication modules may require security information from the message processing environment that invoked them. For example, a `ClientAuthModule` may require access to the client's key pair to sign requests made on behalf of the client. The client's keys would typically have been configured as part of the client application itself. Likewise, a `ServerAuthModule` may require access to the server's key pair to sign responses from the server. The server's keys would typically be configured as part of the server.

To access cryptographic keys or other external security credentials configured as part of the encompassing runtime, an authentication module is provided with a `CallbackHandler` (at initialization). The `CallbackHandler` is provided by the encompassing runtime and serves to provide the authentication module with access to facilities of the encompassing runtime.

The module can ask the `CallbackHandler` to handle requests for security information needed by the module to perform its message authentication processing.

1.1.9. Subjects

When an authentication module is invoked to validate a message, it is passed a `Subject` object to receive the credentials of the source of the message and a separate `Subject` object to represent the credentials of the recipient of the message (such that they are available to validate the message). When an authentication module is invoked to validate a message, it communicates the message source or caller authentication identity to its calling runtime (for example, container) through (that is, by modifying) the `Subject` associated with the source of the message.

Authentication modules may rely on the Subjects as well as the `CallbackHandler`, described in Section 1.1.8, to obtain the security information necessary to secure or validate messages. When an authentication module is invoked to secure a message, it is passed a `Subject` object that may convey the credentials of the source of the message (such that they are available to secure the request).

1.1.10. Status Values and Exceptions

Authentication modules and authentication contexts return `AuthStatus` values to characterize the outcome of their message processing. When an `AuthStatus` value is returned, its value represents the logical result of the module processing and indicates that the module has established a corresponding request or response message within the `MessageInfo` parameter exchanged with the runtime.

Authentication modules and authentication contexts throw exceptions when their processing was unsuccessful and when that processing did not establish a corresponding request or response message to convey the error.

The vocabulary of AuthStatus values and exceptions returned by authentication modules, and their mapping to the message processing points at which they may be returned, is represented in the following table.

Table 1-1 AuthStatus and AuthException to Message Processing Point Matrix

status or exception	secureRequest	validateRequest	secureResponse	validateResponse
SUCCESS		Yes		Yes
FAILURE	Yes			Yes
SEND_SUCCESS	Yes	Yes	Yes	
SEND_FAILURE		Yes	Yes	
SEND_CONTINUE	Yes	Yes	Yes	Yes

status or exception	secureRequest	validateRequest	secureResponse	validateResponse
AuthException	Yes	Yes	Yes	Yes

The following table describes the high level semantics associated with the status values and exceptions presented in the preceding table.

Table 1-2 AuthStatus and AuthException Semantics

status or exception	semantic
SUCCESS	Validation of a received message was successful and produced either the request (validateRequest) message to be dispatched to the service, or the response (validateResponse) message to be returned to the client application.
FAILURE	A failure occurred on the client-side (secureRequest or validateResponse) and produced a failure response message to be returned to the client application.
SEND_SUCCESS	Processing of a request (secureRequest or validateRequest) or response (secureResponse) message was successful and produced the request (secureRequest) or response (validateRequest, secureResponse) message to be sent to the peer.
SEND_FAILURE	A failure occurred on the service-side (validateRequest or secureResponse) and produced a failure response message to be sent to the client.
SEND_CONTINUE	Processing was incomplete. Additional message exchanges will be required to achieve successful completion. The processing produced the next request (secureRequest or validateResponse) or response (validateRequest or secureResponse) message to be sent to the peer.
AuthException	A failure occurred on the client-side (secureRequest or validateResponse) or service-side (validateRequest or secureResponse) without producing a failure response message.

The expected behavior of runtimes in response to AuthStatus return values and AuthException exceptions is described in See What the Runtime Must Do. These behaviors may be specialized in profiles of this specification.

1.2. Typical Runtime Use Model

In the typical use model, a runtime would perform the five steps defined in the following subsections to secure or validate a message. In many cases, some or all of steps 1-4 will be performed once, while step 5 would be repeated for each message to be processed.

1.2.1. Acquire AuthConfigProvider

The message processing runtime acquires a provider of authentication context configuration objects for the relevant messaging layer and application identifier. This step is typically done once for each application, and may be accomplished as follows:

```
AuthConfigFactory factory = AuthConfigFactory.getFactory();
AuthConfigProvider provider = factory.getConfigProvider(layer, appID, listener);
```

1.2.2. Acquire AuthConfig

The message processing runtime acquires the authentication context configuration object for the application from the provider. This step is typically done at application deployment, and may be accomplished as follows:

```
ClientAuthConfig clientConfig =
    provider.getClientAuthConfig(layer, appID, callbackHandler);
```

or:

```
ServerAuthConfig serverConfig =
    provider.getServerAuthConfig(layer, appID, callbackHandler);
```

The resulting authentication context configuration object encapsulates all authentication contexts for the application at the layer. Its internal state will be kept up to date by the configuration system, and from this point until the application is undeployed, the configuration object represents a stable point of interaction between the runtime and the integrated authentication mechanisms for the purpose of securing the messages of the application at the layer.

A callback handler is associated with the configuration object when it is obtained from the provider. This callback handler will be passed to the authentication modules within the authentication contexts acquired from the configuration object. The runtime provides the callback handler so that the authentication modules may employ facilities of the messaging runtime (such as keying infrastructure) in their processing of application messages.

1.2.3. Acquire AuthContext Identifier

At points (1) and (2) in the message processing model, a message processing runtime creates a MessageInfo object and sets within it the message or messages being processed. The runtime uses the MessageInfo to acquire the authentication context identifier corresponding to the message from the authentication configuration object. This step is typically performed for every different [2] request and may be accomplished by a runtime as follows:

```
String authContextID = clientConfig.getAuthContextID(messageInfo);
```

or:

```
String authContextID = serverConfig.getAuthContextID(messageInfo);
```

The authentication context identifier will be used to select the authentication context with which to perform the message processing. In cases where the configuration system cannot determine the context identifier [3], the value null will be returned.

1.2.4. Acquire Authentication Context

The authentication identifier is used to acquire an authentication context from the authentication context configuration object. The acquired authentication context encapsulates the one or more authentication modules that are to be invoked to process the identified messages. The authentication context is acquired from the authentication context configuration object as follows:

```
ClientAuthContext clientContext =
    clientConfig.getAuthContext(authContextID, clientSubject, properties);
```

or:

```
ServerAuthContext serverContext =
    serverConfig.getAuthContext(authContextID, serviceSubject, properties);
```

The properties argument is used to pass additional initialization time properties to the authentication modules encapsulated in the authentication context. Such properties might be used to convey values specific to this use of the context by a user or with a specific service.

The Subject argument is used to make the principals and credentials of the sending entity available during the acquisition of the authentication context. If the Subject is not null, additional principals or credentials (pertaining to the sending entity) may be added (to the Subject) during the context acquisition.

1.2.5. Process Messages

Appropriate to its point of processing in the messaging model, the messaging runtime uses the `MessageInfo` described in Step 3 to invoke a method of the authentication context obtained in Step 4.

At point (1) in the messaging model, the `clientSubject` may contain the credentials used to secure the request, or the modules of the context may collect the client credentials including by using the callback handler passed through to them by the context. `MessageInfo` would contain a request message about to

be sent. On successful return from the context, the runtime would extract the secured request message from `messageInfo` and send it.

```
(1) AuthStatus status = clientContext.secureRequest(messageInfo, clientSubject);
```

At point (2), the `clientSubject` receives any principals or credentials established as a result of message validation by the authentication modules of the context. The `serviceSubject` may contain the credentials of the service or the modules of the context may collect the service credentials, as necessary, by using the callback handler passed to them by the context. `MessageInfo` would contain a received request message. On successful return from the context, the runtime may use the `clientSubject` to authorize and dispatch the validated request message, as appropriate.

```
(2) AuthStatus status = serverContext.validateRequest(messageInfo, clientSubject,
serviceSubject);
```

At point (3), the `serviceSubject` may contain the credentials used to secure the response, or the modules of the context may collect the service credentials including by using the callback handler passed through to them by the context. The `MessageInfo` would contain a response message about to be sent and may also contain the corresponding request message received at point (2). On return from the context, the runtime would send the secured response message.

```
(3) AuthStatus status = serverContext.secureResponse(messageInfo, serviceSubject);
```

At point (4), the `serviceSubject` receives any principals or credentials established as a result of message validation by the authentication modules of the context. The `clientSubject` may contain the credentials of the receiving client or the modules of the context may collect the client credentials, as necessary, by using the callback handler passed to them by the context. `MessageInfo` would contain a received response message and may also contain the associated request message sent at point (1). On successful return from the context, the runtime may use the `serviceSubject` to authorize the response and would return the received message to the client, as appropriate.

```
(4) AuthStatus status =
        clientContext.validateResponse(messageInfo, clientSubject, serviceSubject);
```

1.3. Terminology

authentication context

A Java Object that implements the `ClientAuthContext` and/or `ServerAuthContext` interfaces and that is responsible for constructing, initializing, and coordinating the invocation of one or more encapsulated authentication modules. Authentication context objects are classified as client or server authentication contexts.

authentication context configuration

A Java Object that implements the `AuthConfig` Interface and that serves as the source of client or server authentication context objects pertaining to the processing of messages for an application at a messaging layer.

authentication context configuration provider

A Java Object that implements the `AuthConfigProvider` Interface and that serves as the source of authentication context configuration objects.

authentication module

A Java Object that implements the `ClientAuthModule` and/or `ServerAuthModule` message authentication interfaces defined by this specification.

authentication provider

A synonym for an authentication module.

client authentication context

An authentication context that implements the `ClientAuthContext` interface and that encapsulates client authentication modules.

client authentication context configuration

An authentication context configuration that implements the `ClientAuthConfig` interface and that returns client authentication contexts.

client authentication module

A Java Object that implements the `ClientAuthModule` interface defined by this specification.

message layer

The name associated within a message processing runtime with a messaging protocol or abstraction, and which may be used in the interfaces defined by this specification to cause the integration of security mechanisms at the corresponding points within the messaging runtime.

message processing runtime

The process or component (for example, container) responsible for sending and receiving, including establishing the transports used for such purposes, the application messages to be secured using the interfaces defined by this specification. Message processing runtimes are characterized as client, server, or as both client and server message processing runtimes. A client message processing runtime sends service requests and receives service responses. A server message processing runtime receives service requests and sends service responses.

message (layer) security

A network security mechanism that operates above the transport and below the application messaging layers, and that typically operates by encapsulating or associating application layer messages within a securing context that may be independent of the transport or connection over which the messages are communicated.

meta message

A mechanism specific message sent in addition to (for example, in an advance of) the application messages, typically for the purpose of establishing or modifying the context (such as security) in which application messages will be exchanged.

server authentication context

An authentication context that implements the `ServerAuthContext` interface and that encapsulates server authentication modules.

server authentication context configuration

An authentication context configuration that implements the `ServerAuthConfig` interface and that encapsulates client authentication context.

server authentication module

A Java Object that implements the `ServerAuthModule` interface defined by this specification.

1.4. Assumptions

The following assumptions apply to the interfaces defined by this specification:

1. This specification defines interfaces for integrating message layer security functionality in Java messaging runtimes. These interfaces are intended to be employed by Jakarta Enterprise Edition (Jakarta EE version 9 and beyond) messaging runtimes, and by any Java messaging runtime that chooses to use them to support integration of message layer security functionality.
2. The interfaces defined by this specification have been developed for use within the message processing runtimes of service consumers (for example, clients) and service providers (for example, servers).

3. Interoperability between a message processing runtime that employs the interfaces defined by this specification and any other system will depend on the formats of the exchanged messages, not on the interfaces used to process them.
4. This specification will define profiles to establish the requirements governing the use of its interfaces within specific messaging contexts or runtimes. Additional profiles may be defined in futures releases of this specification, or external to it.
5. This specification promotes authentication modules as the pluggable unit of message layer security functionality. In the typical integration scenario, a new message layer security mechanism is integrated in a message processing runtime as the result of the configuration of a new authentication module.
6. Mechanisms that feature or require more complex or specialized configuration functionality may depend on integration of a corresponding configuration provider which may encapsulate authentication module pluggability, including such that it occurs as the result of provider configuration.
7. A message processing runtime that uses the interfaces defined by this specification will remain responsible for sending and receiving, including establishing the transports used for such purposes, the application messages secured through these interfaces. The integrated security mechanism code is responsible for adding security constructs to messages to be sent, and for interpreting security constructs contained in received messages.
8. As needed to perform its primary function (that is, to add to and validate security constructs in messages provided to it by its messaging runtime), an authentication mechanism integrated through the interfaces defined in this specification may use its own facilities or those of its calling runtime to exchange additional messages with the same or with other parties.
9. Some multi-message authentication dialogs require that the sending runtime be able to delay or retry application message transmission until after a preliminary authentication dialog has completed. Where a sending runtime is unable to perform such functionality, effective integration of a dependent security mechanism may require that the integrated security facilities perform the required delay and retry functionality.
10. Authentication mechanisms integrated in a messaging runtime through the interfaces defined by this specification may require access to sensitive security information (for example, cryptographic keys) for which access may have otherwise been limited to the messaging runtime.
11. Independent of message transformations performed by one or more integrated security mechanisms, the client messaging runtime must remain capable of associating received responses with sent requests.

1.5. Requirements

The interfaces defined by this specification must comply with the following:

1. Be compatible with versions of Java beginning with 1.8.
2. Be compatible with a wide range of messaging protocols and runtimes.

3. Support the integration and configuration of message security mechanisms in Java message processing runtimes that retain responsibility for the transport of application layer messages.
4. Provide integrated authentication mechanisms with access to the application messages transported by the messaging runtime, especially for the purpose of adding or validating contained security credentials.
5. Define a means for an integrated security mechanism to establish (for example, application layer) response messages as necessary to implement security mechanisms.
6. Define a means for an integrated security mechanism to effect the destination address of outgoing messages.
7. Support the binding of received messages to configured security mechanisms at various levels of granularity such as per messaging runtime, per messaging layer, per application, and per message.
8. Support the integration of alternative security mechanism configuration facilities as required to support specific security mechanisms or to integrate into standard or existing configuration infrastructures.
9. Support the runtime binding of user or application client credentials to invocations of authentication modules.
10. Support the establishment of Subject based authorization identities by integrated authentication mechanisms.
11. Define a means for integrated security mechanisms to gain access to facilities (for example, key repositories, password databases, and subject or principal interpretation interfaces) of their calling messaging runtime.
12. Facilitate the correlation of the associated request and response processing performed by an authentication module.
13. Support runtime parameterization of security mechanism invocation such that a single mechanism configuration can be employed to secure commonly protected exchanges with different service entities.
14. Support the apportionment of responsibility for creation and maintenance of stateful security contexts among a messaging runtime and its integrated security mechanisms, especially such that context invalidation (including as a result of policy modification) by either party is appropriately detected by the other.
15. Support the portable implementation (including by third parties) of security mechanisms such that they may be integrated in any messaging runtime which is compatible with the corresponding interfaces of this specification.

1.5.1. Non Requirements

1. The standardization of specific principals or credentials to be added by authentication modules to subjects.
2. The standardization of additional interfaces or callbacks to allow JAAS login modules to secure the request and response messages exchanged by Jakarta EE containers.

3. The standardization of interfaces to interact with network authentication services, or to represent the security credentials acquired from such services.
4. The standardization of application programming interfaces for use in establishing or manipulating security contexts in Subjects.

[1] The dashed lines between validateRequest and validateResponse convey additional message exchanges that may occur when message validation requires a multi-message dialog, such as would occur in challenge-response protocols.

[2] A client runtime may be able to tell when a request is the same, based on the context (for example, stub) from which the request is made.

[3] For example, where the message content that defines the identifier is encrypted.

Chapter 2. Message Authentication

This chapter defines how message processing runtimes invoke authentication modules to secure or validate request and response messages. It describes the interactions that occur between message processing runtimes and authentication modules to cause security guarantees to be enforced on request and response messages.

The subsections of this chapter establish the common requirements that pertain to the use of this specification in a generic message processing context. Profiles are expected to be defined to establish the specific requirements pertaining to the use of this specification in a particular message processing context.

The API defined by this specification is intended to have more general applicability than the contexts of use defined in this specification. To that end, a runtime that provides compatible Java definitions of the interfaces defined by this specification and compatible Java implementations of the defined classes satisfies the baseline compatibility requirements of this specification.

2.1. Authentication

As defined in Section 1.2 a message processing runtime's interaction with the interfaces defined by this specification is divided into the following five phases:

1. Acquire AuthConfigProvider – Runtime acquires a provider of authentication context configuration objects for the relevant messaging layer and application identifier.
2. Acquire AuthConfig – Runtime acquires the authentication context configuration object for the application from the provider.
3. Acquire AuthContext Identifier – Runtime acquires the authentication context identifier corresponding to the messages to be processed.
4. Acquire Authentication Context – Runtime uses the context identifier to obtain the corresponding authentication context.
5. Process Message(s) – Runtime uses the authentication context to process the messages.

The remaining sections of this chapter define the requirements that must be satisfied by messaging runtimes and providers in support of each of the five interactions identified above.

2.1.1. Acquire AuthConfigProvider

2.1.1.1. What the Runtime Must Do

For a message processing runtime to be able to invoke authentication modules configured according to this specification, the JVM of the message processing runtime must have been configured or initialized such that it has loaded the abstract `AuthConfigFactory` class, and such that the `getFactory` method of the abstract class (loads, as necessary, and) returns a concrete implementation of `AuthConfigFactory`. When

called by the messaging runtime with `layer` and `appContext` arguments, the `getConfigProvider` method of the returned factory implementation must return the corresponding (as a result of configuration or registration) `AuthConfigProvider` object (or null if no provider is configured for the arguments).

This specification defines authorization protected configuration interfaces, and a message processing runtime must support the granting, to applications and administration utilities, of the permissions required to employ these configuration interfaces.

A message processing runtime that wishes to invoke authentication modules configured according to this specification must use the `AuthConfigFactory.getFactory` method to obtain a factory implementation. The runtime must invoke the `getConfigProvider` method of the factory to obtain the `AuthConfigProvider`. The runtime must specify appropriate (non-null) layer and application context identifiers in its call to `getConfigProvider`. The specified values must be as defined by the profile of this specification being followed by the messaging runtime.

A runtime may continue to reuse a provider for as long as it wishes. However, a runtime that wishes to be notified of changes to the factory that would cause the factory to return a different provider for the `layer` and `appContext` arguments should include a (non-null) `RegistrationListener` as an argument in the call used to acquire the provider. When a listener argument is included in the call to acquire a provider, the factory will invoke the `notify` method of the listener when the correspondence between the provider and the layer and application context for which it had been acquired is no longer in effect. When the `notify` method is invoked by the factory, the runtime should reacquire an `AuthConfigProvider` for the layer and application context.

2.1.1.2. What the Factory Must Do

The factory implementation must satisfy the requirements defined by the `AuthConfigFactory` class. In particular, it must offer a public, zero argument constructor that supports the construction and registration of `AuthConfigProvider` objects from a persistent declarative representation.

2.1.2. Acquire AuthConfig

2.1.2.1. What the Runtime Must Do

Once the runtime has obtained the appropriate (non-null) `AuthConfigProvider`, it must obtain from the provider the authentication context configuration object corresponding to the messaging layer, its role as client or server, and the application context for which it will be exchanging messages. It does this by invoking `getClientAuthConfig` or `getServerAuthConfig` as appropriate to the role of the runtime in the message exchange. A runtime operating at points 1 and 4 in the messaging model must invoke `getClientAuthConfig` to acquire its configuration object. A runtime operating at points 2 and 3 in the messaging model must invoke `getServerAuthConfig` to acquire its configuration object. The call to acquire the configuration object must specify the same values for layer and application context identifier that were used to acquire the provider. Depending on the profile of this specification being followed by the messaging runtime, a `CallbackHandler` may also be a required argument of the call to acquire the configuration object. When a profile requires a `CallbackHandler`, the profile must also specify the callbacks that must be supported by the handler.

A runtime may continue to reuse an acquired authentication context configuration object for as long as it is acting as client or server of the corresponding application. A runtime should reacquire an authentication context configuration object when it is notified (through a `RegistrationListener`) that it must reacquire the `AuthConfigProvider` from which the configuration object was acquired (and after having reacquired the provider).

2.1.2.2. What the Provider Must Do

The provider implementation must satisfy the requirements defined by the `AuthConfigProvider` interface. In particular, it must return non-null authentication configuration objects. Moreover, when the provider is a dynamic configuration provider, any change to the internal state of the provider occurring as the result of a call to its `refresh` method must be recognized by every authentication context configuration object obtained from the provider.

The provider implementation must provide a configuration facility that may be used to configure the information required to initialize authentication contexts for the (one or more) authentication context configuration scopes (defined by layer and application context) for which the provider is registered (at the factory).

To allow for delegation of session management to authentication contexts and their contained authentication modules, it must be possible for one or more of the authentication context configuration scopes handled by an `AuthConfigProvider` to be configured such that the `getAuthContext` method of the corresponding authentication context configuration objects will return a non-null authentication context for all authentication context identifier values, independent of whether or not the corresponding messages require protection. In this case, contexts returned for messages for which protection is NOT required must initialize their contained authentication modules with request and/or response MessagePolicy objects for which `isMandatory()` returns false (while allowing for the case where one of either request or response policy may be null).

A sample and perhaps typical context initialization model is described in Section 2.1.4.2. Providers must offer a configuration facility sufficient to sustain the typical context initialization model.

2.1.3. Acquire AuthContext Identifier

2.1.3.1. What the Runtime Must Do

At points (1) and (2) in the messaging model, the message processing runtime must obtain the authentication context identifier corresponding to the request message processing being performed by the runtime.

The identifier may be acquired by calling the `getAuthContextID` method of the authentication context configuration object (obtained in the preceding step). If the messaging runtime chooses to obtain the context identifier by this means, it must provide a `MessageInfo` object as argument to the `getAuthContextID` call, and the `MessageInfo` must have been initialized such that its `getRequestMessage` method will return the request message being processed by the runtime. The type of the returned request message must be as defined by the profile of this specification being followed by the messaging

runtime.

Alternatively and depending on the requirements relating to authentication context identifier inherent in the profile being followed by the messaging runtime, the runtime may obtain the identifier by other means. Where a profile defines or facilitates other means by which a messaging runtime may acquire the identifier, the identifier acquired by any such means must be equivalent to the identifier that would be acquired by calling `getAuthContextID` as described above.

2.1.3.2. What the Configuration Must Do

The configuration implementation must satisfy the requirements defined by the `AuthConfig` interface with respect to the `getAuthContextID` method.

2.1.4. Acquire Authentication Context

2.1.4.1. What the Runtime Must Do

At points (1) and (2) in the messaging model, the message processing runtime must invoke the `getAuthContext` method of the authentication context configuration object (obtained in step 2) to obtain the authentication context object corresponding to the message that is to be processed. This is accomplished by invoking `getAuthContext` with the authentication context identifier corresponding to the request message and obtained as described above. If required by the profile of this specification being followed by the runtime, the call to `getAuthContext` must pass a `Map` containing the required property elements. The value of the `Subject` argument provided by the runtime in its call to `getAuthContext` must correspond to the requirements of the profile of this specification being followed by the runtime.

Once an authentication context is acquired, it may be reused to process subsequent requests of the application for which an equivalent authentication context identifier, `Subject`, and properties `Map` (as used in the `getAuthContext`) applies. Runtimes that wish to be dynamic with respect to changes in context configuration should call `getAuthContext` for every request. An authentication context configuration object may return the same authentication context object for different authentication context identifiers for which the same module configuration and message protection policy applies.

At points (3) and (4) in the messaging model, the runtime may repeat the context acquisition performed at point (2) and (1) respectively, or it may reuse the previously acquired context.

2.1.4.2. What the Configuration Must Do

The configuration implementation must satisfy the requirements defined by the corresponding `ClientAuthConfig` or `ServerAuthConfig` interface with respect to the `getAuthContext` method. In this regard, the configuration implementation must determine the authentication modules that are to comprise the acquired context, and it must provide the context implementation with sufficient information to initialize the modules of the context. The `getAuthContext` method must return null when no authentication modules are to be invoked for an identified authentication context at the layer and application context represented by the configuration object.

The interfaces by which an authentication context configuration object obtains a properly configured or initialized authentication context object are implementation-specific. That said, it is expected that the typical context initialization will require the following information:

- The `CallbackHandler` (if any) to be passed to the modules of the context
- A list of one or more module configurations (one for each module of the context), and where each such configuration conveys (either directly or indirectly) the following information:
  - The implementation class for the authentication module (that is, an implementation of the `ClientAuthModule` or `ServerAuthModule` interface as appropriate to the type of the containing context)
  - The module specific initialization properties (in a form compatible with conveyance to the module by using a `Map`)
  - The request and response `MessagePolicy` objects for the module
  - The context-specific control attributes to be used by the context to coordinate the invocation of the module with respect to the other modules of the context

To sustain the above requirements, the `AuthConfigProvider` from which the authentication context configuration object was acquired must provide a configuration facility by which the information required to initialize authentication contexts may be configured and associated with one or more authentication context identifiers within the (one or more) layer and application context scopes for which the provider is registered (at the factory).

2.1.5. Process Messages

2.1.5.1. What the Context Must Do

Every context implementation must satisfy the requirements as defined by the corresponding `ClientAuthContext` or `ServerAuthContext` interface.

Every context is responsible for constructing and initializing the one or more authentication modules assigned to the context by the authentication context configuration object. The initialization step includes passing the relevant request and response MessagePolicy objects to the authentication modules. These policy objects may have been acquired by the authentication context configuration object and provided as arguments through the internal interfaces used by the configuration object to acquire the context.

Every context must delegate calls made to the methods of its corresponding `ClientAuth` or `ServerAuth` interface to the corresponding methods of its one or more authentication modules. If a context encapsulates multiple authentication modules, the context must embody the control logic to determine which modules of the context are to be invoked and in what order. Contexts which encapsulate alternative sufficient modules must ensure that the same message values are passed to each invoked alternative of the context. If a context invokes multiple authentication modules, the context must combine the `AuthStatus` values returned by the invoked authentication modules to establish the `AuthStatus` value returned by the context to the messaging runtime. The context implementation must

define the logic for combining the returned `AuthStatus` values.

2.1.5.2. What the Runtime Must Do

If a non-null authentication context object is returned by `getAuthContext`, the corresponding message processing runtime must invoke the methods of the acquired authentication context to process the corresponding request and response messages as defined below. Otherwise, the message processing runtime must proceed with its normal processing of the corresponding messages and without invoking the methods of an authentication context object.

At point (1) in the message processing model:

- The message processing runtime must call the secureRequest method of the ClientAuthContext.
- The messageInfo argument to the call must have been initialized such that its getRequestMessage method will return the request message being processed by the runtime. The type of the returned request message must be as defined by the profile being followed.
- If a non-null Subject was used to acquire the ClientAuthContext, the same Subject must be passed as the clientSubject in this call. If a non-null clientSubject is used in this call, it must not be read-only, and the same clientSubject argument must be passed in all calls to validateResponse made for the one or more responses processed to complete the message exchange.
- If the call to secureRequest returns:
  - `AuthStatus.SEND_SUCCESS` – The runtime should send (without calling `secureRequest`) the request message acquired by calling `messageInfo.getRequestMessage`. After sending the request, the runtime should proceed to point (4) in the message processing model (to receive and validate the response).
  - `AuthStatus.SEND_CONTINUE` – The module has returned, in messageInfo, an initial request message to be sent. Moreover, the module is informing the client runtime that it will be required to continue the message dialog by sending the message resulting from validation of the response to the initial message. If the runtime will be unable to continue the dialog by sending the message resulting from validation of the response, the runtime must not send the initial request and must convey its inability by returning an error to the client application. Otherwise, the runtime should send (without calling `secureRequest`) the request message acquired by calling `messageInfo.getRequestMessage`.
  - `AuthStatus.FAILURE` – The runtime should return an error to the client application. The runtime should derive the returned error from the response message acquired by calling `messageInfo.getResponseMessage`.
  - Throws an `AuthException` – The runtime should use the exception to convey to the client runtime that the request failed.

At point (4) in the message processing model:

- The message processing runtime must call the validateResponse method of the `ClientAuthContext`.
- In the call made to `validateResponse`, the runtime must pass the same `MessageInfo` instance that was

passed to `secureRequest` (at the start of the message exchange). The `messageInfo` argument must have been initialized such that its `getResponseMessage` method will return the response message being processed by the runtime.The type of the required return messages must be as defined by the profile being followed.

- The value of the `clientSubject` argument to the call must be the same as that passed in the call to secureRequest for the corresponding request.
- The `serviceSubject` argument to the call may be non-null, in which it must not be read-only and may be used by modules to store Principals and credentials determined to pertain to the source of the response.
- If the call to `validateResponse` returns:
  - `AuthStatus.SUCCESS` – The runtime should use the response message acquired by calling `messageInfo.getResponseMessage` to create the value to be returned to the client.
  - `AuthStatus.SEND_CONTINUE` – If the runtime is unable to process this status value, it must return an error to the client application indicating its inability to process this status value. To process this status value, the runtime must send (without calling `secureRequest`) the (continuation) request message obtained by calling `messageInfo.getRequestMessage`, and it must receive and process by using `validateResponse` (at least) the next corresponding response or error (before returning a value to the client).
  - `AuthStatus.FAILURE` – The runtime should return an error to the client application. The runtime should derive the returned error from the response message acquired by calling `messageInfo.getResponseMessage`.
  - Throws an `AuthException` – The runtime should use the exception to convey to the client runtime that the request failed.

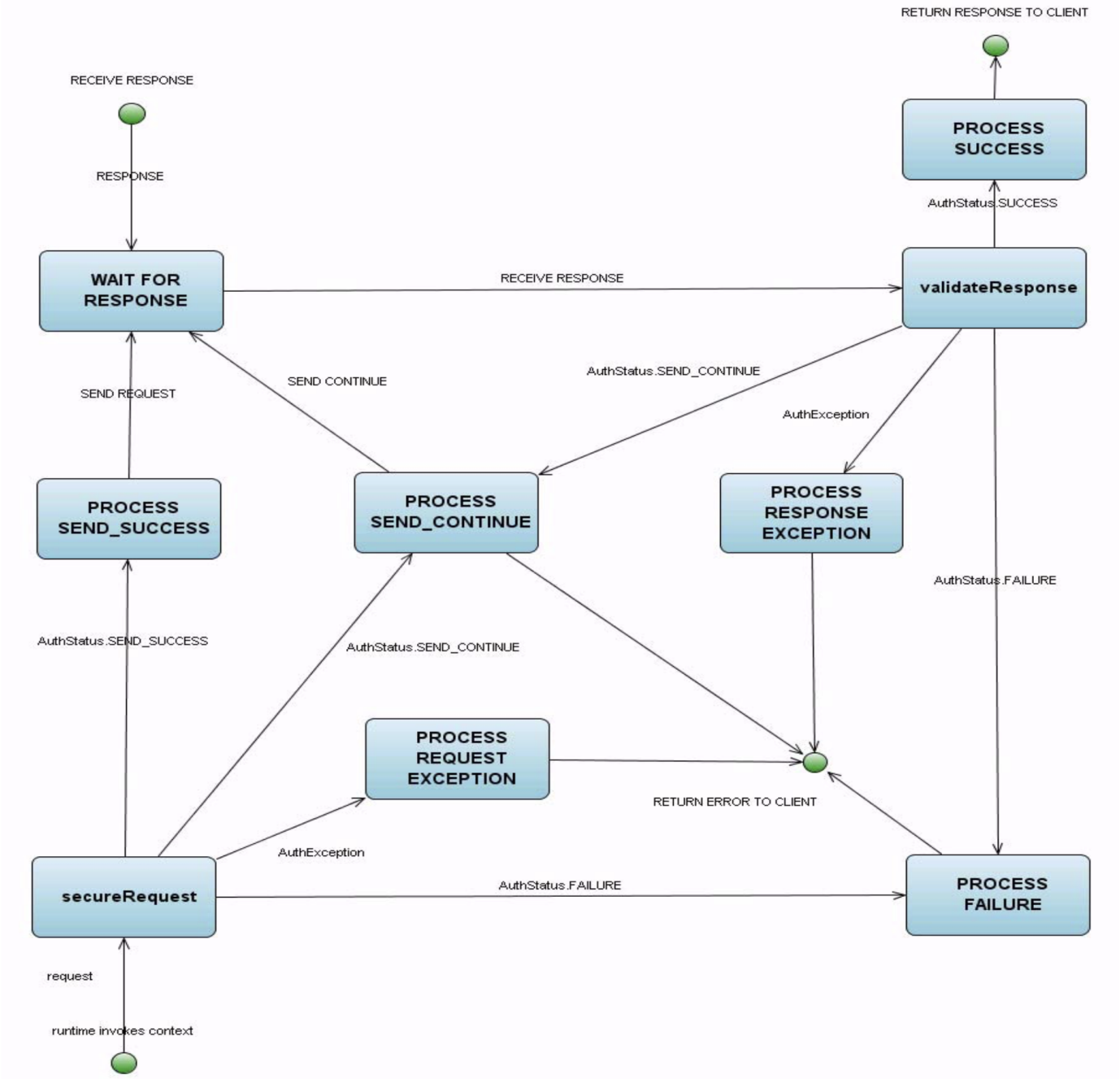

Figure 2-2 State Diagram of Client Message Processing Runtime

At point (2) in the message processing model:

- The message processing runtime must call the `validateRequest` method of the `ServerAuthContext`.
- The `messageInfo` argument to the call must have been initialized such that its getRequestMessage method will return the request message being processed by the runtime. For some profiles of this specification, the runtime must also initialize `messageInfo` such that its `getResponseMessage` method will return the response message being processed by the runtime. The type of the required return messages must be as defined by the profile being followed.
- The `clientSubject` argument must be non-null and it must not be read-only. It is expected that the modules of the authentication context will populate this `Subject` with principals and credentials

resulting from their processing of the request message.

- If a non-null `Subject` was used to acquire the `ServerAuthContext`, the same `Subject` must be passed as the `serviceSubject` in this call. If a non-null `serviceSubject` is used in this call, it must not be read-only, and the same `serviceSubject` must be passed in the call to `secureResponse` for the corresponding response (if there is one).
- If the call to `validateRequest` returns:
  - `AuthStatus.SUCCESS` – The runtime should proceed to authorize the request using the `clientSubject`, perform the application request processing (depending on the authorization result), and proceed to point (3) as appropriate [4]
  - `AuthStatus.SEND_SUCCESS` – The runtime should send (without calling `secureResponse`) the response message acquired by calling `messageInfo.getResponseMessage`, at which time the processing of the application request and its corresponding response will be complete. The runtime must NOT proceed to authorize the request or perform the application request processing.
  - `AuthStatus.SEND_CONTINUE` – The runtime should send (without calling `secureResponse`) the response message acquired by calling `messageInfo.getResponseMessage`. The runtime must NOT proceed to authorize the request or perform the application request processing. The processing of the application request is not finished, and as such, its outcome is not yet known.
  - `AuthStatus.SEND_FAILURE` – The runtime must NOT proceed to authorize the request or perform the application request processing. If the failure occurred after [5] the service invocation, the runtime must perform whatever processing it requires to complete the processing of a request that failed after a successful service invocation, and prior to communicating the invocation result to the client runtime. The runtime may send (without calling secureResponse) the response message acquired by calling `messageInfo.getResponseMessage`.
  - Throws an `AuthException` – The runtime must NOT proceed to authorize the request or perform the application request processing. If the failure occurred after the service invocation, the runtime must perform whatever processing it requires to complete the processing of a request that failed after the service invocation, and prior to communicating the invocation result to the client runtime. The runtime may send (without calling `secureResponse`) a failure message of its choice. If a failure message is returned, it should indicate whether the failure in request processing occurred before or after the service invocation.

At point (3) in the message processing model:

- The message processing runtime must call the `secureResponse` method of the `ServerAuthContext`. The call to `secureResponse` should be made independent of the result of the application request processing.
- In the call made to `secureResponse`, the runtime must pass the same `MessageInfo` instance that was passed to `validateRequest` (for the corresponding request message). The `messageInfo` argument must have been initialized such that its `getResponseMessage` method will return the response message being processed by the runtime. The type of the required return messages must be as defined by the profile being followed.

- The value of the `serviceSubject` argument to the call must be the same as that passed in the call to `validateRequest` for the corresponding request.
- If the call to `secureResponse` returns:
  - `AuthStatus.SEND_SUCCESS` – The runtime should send (without calling `secureResponse`) the response message acquired by calling `messageInfo.getResponseMessage` at which time the processing of the application request and its corresponding response will be complete.
  - `AuthStatus.SEND_CONTINUE` – The runtime should send (without calling `secureResponse`) the response message acquired by calling `messageInfo.getResponseMessage`. The processing of the response is not finished, and as such, its outcome is not yet known.
  - `AuthStatus.SEND_FAILURE` – The runtime must perform whatever processing it requires to complete the processing of a request that failed after (or during) service invocation, and prior to communicating the invocation result to the client runtime. This may include sending (without calling `secureResponse`) the response message acquired by calling `messageInfo.getResponseMessage`.
  - Throws an `AuthException` – The runtime must perform whatever processing it requires to complete the processing of a request that failed after (or during) service invocation, and prior to communicating the invocation result to the client runtime. The runtime may send (without calling secureResponse) an appropriate response message of its choice. If a failure message is returned, it should indicate that the failure in request processing occurred after the service invocation.

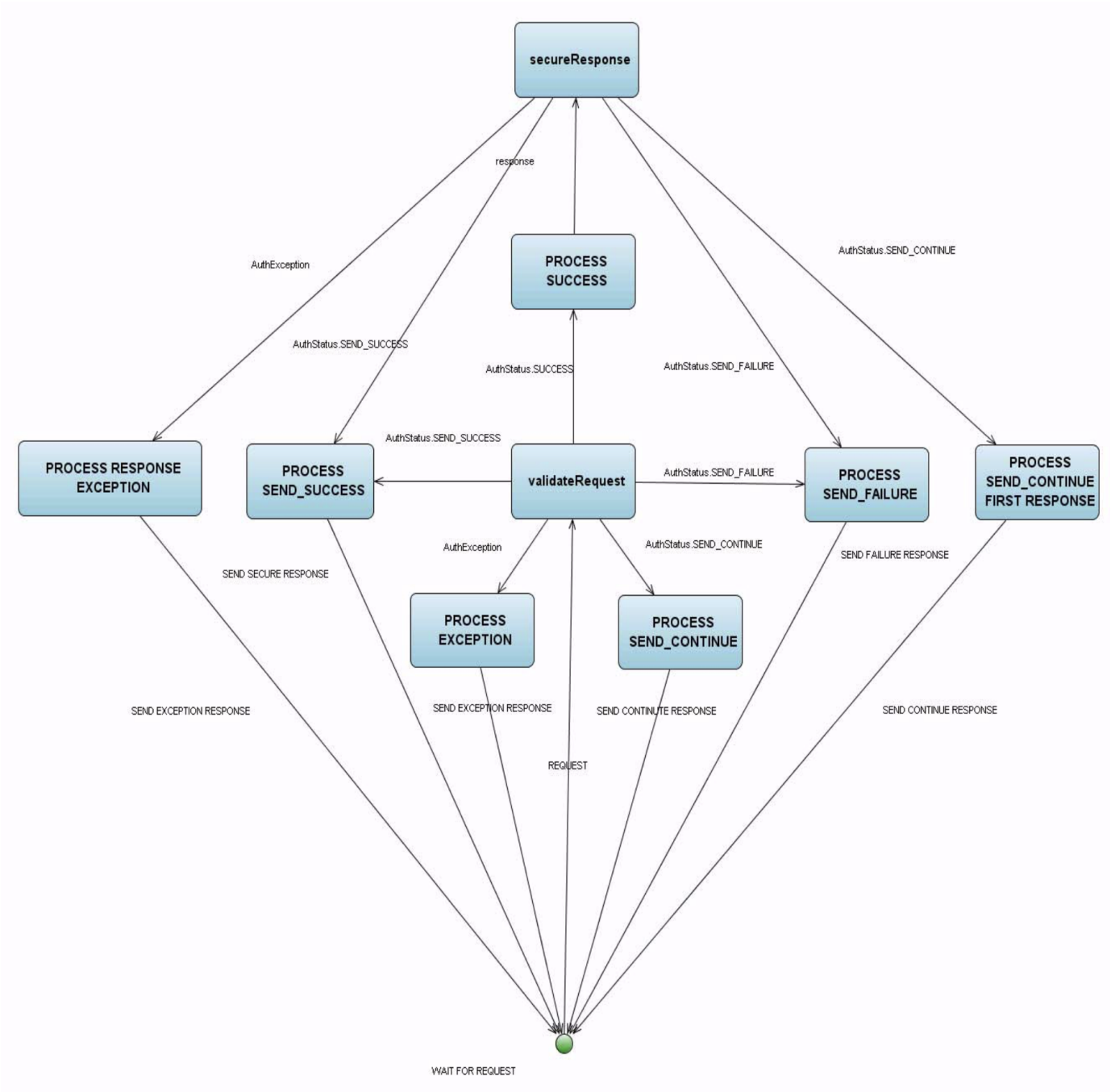

Figure 2-3 State Diagram of Server Message Processing Runtime

2.1.5.3. What the Modules Must Do

The authentication module implementations within the context must satisfy the requirements as defined by the corresponding `ClientAuthModule` or `ServerAuthModule` interface.

[4] The application request processing must not be performed if the request authorization fails. If the runtime intends to return a response message to indicate the failed authorization, the profile of this specification being followed by the runtime must establish whether or not secureResponse must be called prior to sending the authorization failure message.

[5] validateRequest is called to process all received messages, including security mechanism-specific messages sent by clients in response to service response messages.

Chapter 3. Servlet Container Profile

This chapter defines a profile of the use of the interfaces defined in this specification by Servlet containers to enforce the declarative authentication constraints of the Servlet container security model.

This profile focuses on points 2 (and, to a lesser degree), 3 in the message processing model. This profile does not specify the behavior of the corresponding client runtime (that is, points 1 and 4 in the message processing model).

The profile-specific requirements defined in this chapter are to be considered in addition to the generic requirements defined in Chapter 2. A compatible implementation of this profile is a servlet container that satisfies all of the requirements that apply to this profile.

3.1. Message Layer Identifier

The message layer value used to select the `AuthConfigProvider` and `ServerAuthConfig` objects for this profile must be "HttpServlet".

3.2. Application Context Identifier

The application context identifier (that is, the `appContext` parameter value) used to select the `AuthConfigProvider` and `ServerAuthConfig` objects for a specific application shall be the String value constructed by concatenating the host name, a blank separator character, and the decoded context path corresponding to the web module.

```
AppContextID ::= hostname blank context-path

For example: "java-server /petstore"
```

This profile uses the term `host name` to refer to the name of a logical host that processes Servlet requests. Servlet requests may be directed to a logical host using various physical or `virtual host` names or addresses, and a message processing runtime may be composed of multiple logical hosts. Systems or administrators that register `AuthConfigProvider` objects with specific application context identifiers must have an ability to determine the host name for which they wish to perform the registration.

A Jakarta Servlet container that implements a version of the Jakarta Servlet specification that defines the `getVirtualServerName` method on the `ServletContext` interface, must construct its application context identifiers using a value for `hostname` that is equivalent to the value returned by calling `getVirtualServerName` on the ServletContext corresponding to the web application.

3.3. Message Requirements

The `MessageInfo` argument used in any call made by the message processing runtime to `validateRequest` or `secureResponse` must have been initialized such that the non-null objects returned by the `getRequestMessage` and `getResponseMessage` methods of the `MessageInfo` are an `instanceof` `HttpServletRequest` and `HttpServletResponse`, respectively.

3.4. Module Requirements

The `getSupportedMessageTypes` method of all authentication modules integrated for use with this profile must include `jakarta.servlet.http.HttpServletRequest.class` and `jakarta.servlet.http.HttpServeletResponse.class` in its return value.

3.5. CallbackHandler Requirements

The `CallbackHandler` passed to `ServerAuthModule.initialize` is determined by the `handler` argument passed in the `AuthConfigProvider.getServerAuthConfig` call that acquired the corresponding authentication context configuration object. The `handler` argument must not be null, and the argument `handler` and the `CallbackHandler` passed to `ServerAuthModule.initialize` must support the following callbacks:

- CallerPrincipalCallback
- GroupPrincipalCallback
- PasswordValidationCallback

The `CallbackHandler` passed to `ServerAuthModule.initialize` should also support the following callbacks, and it must be possible to configure the runtime such that the `CallbackHandler` passed to `ServerAuthModule.initialize` supports the following callbacks in addition to those listed above.

- CertStoreCallback
- PrivateKeyCallback
- SecretKeyCallback
- TrustStoreCallback

The argument `handler` and the `CallbackHandler` passed through to the authentication modules must be initialized with any application context required to process its supported callbacks on behalf of the corresponding application.

3.6. State

For this profile it is RECOMMENDED that the `CallbackHandler` does not keep any state for a single HTTP request or HTTP session. That is, the `CallbackHandler` SHOULD be considered to have an application

scope lifetime equivalent to an `HttpServlet` instance, and expect to handle calls concurrently from different requests.

To that end it's RECOMMENDED that the `CallbackHandler` uses the `Subject` that's passed in to the following callbacks to store per-request state:

- CallerPrincipalCallback
- GroupPrincipalCallback
- PasswordValidationCallback

3.7. AuthConfigProvider Requirements

The factory implementation returned by calling the `getFactory` method of the abstract `AuthConfigFactory` class must have been configured such that it returns a non-null `AuthConfigProvider` for those application contexts for which pluggable authentication modules have been configured at the "HttpServlet" layer.

For each application context for which it is servicing requests, the runtime must call `getConfigProvider` to acquire the provider object corresponding to the layer and application context. The `layer` and `appContext` arguments to `getConfigProvider` must be as defined in Section 3.1, and Section 3.2 respectively. If a non-null `AuthConfigProvider` is returned, the messaging runtime must call `getServerAuthConfig` on the provider to obtain the authentication context configuration object pertaining to the application context at the layer. The `layer` and `appContext` arguments of the call to `getServerAuthConfig` must be the same as those used to acquire the provider, and the handler argument must be as defined in Section 3.5.

A null return value from `getConfigProvider` indicates that pluggable authentication modules have not been configured at the layer for the application context and that the messaging runtime must proceed to perform servlet security constraint processing (for the application context) without further reliance on this profile.

3.8. Authentication Context Requirements

When a non-null AuthConfigProvider is returned by the factory, the provider must have been configured with the information required to initialize the authentication contexts for the (one or more) authentication context configuration scopes (defined by layer and application context) for which the provider is registered (at the factory). The information (typically) required to initialize authentication contexts is described by example in Section 2.1.4.2.

When a non-null `AuthConfigProvider` is returned by the factory, the messaging runtime must call `getAuthContext` on the authentication context configuration object (obtained from the provider). The `authContextID` argument used in the call to `getAuthContext` must be the value as described in Section 3.8.1.

For all values of the `authContextID` argument that satisfy the requirements of Section 3.8.1, the call to

`getAuthContext` must return a non-null authentication context.

3.8.1. Authentication Context Identifiers

This profile does NOT impose any profile specific requirements on authentication context identifiers. As defined in Section 2.1.3, the authentication context identifier used in the call to `getAuthContext` must be equivalent to the value that would be acquired by calling `getAuthContextID` with the `MessageInfo` that will be used in the call to `validateRequest`.

3.8.2. getAuthContext Subject

A null value may be passed as the `Subject` argument in the `getAuthContext` call.

3.8.3. Module Initialization Properties

If the runtime is a Jakarta Authorization compatible Jakarta Servlet container, the `properties` argument passed in all calls to `getAuthContext` must contain the key-value pair shown in the following table.

Table 3-3 Jakarta Authorization Compatible Module Initialization Properties

key	value
jakarta.security.jacc.PolicyContext	The PolicyContext identifier value that the container must set to satisfy the Jakarta Authorization authorization requirements as described in “Setting the Policy Context” within the Jakarta Authorization specification

When the runtime is not a Jakarta Authorization compatible Jakarta Servlet container, the `properties` argument used in all calls to `getAuthContext` must not include a `jakarta.security.jacc.PolicyContext` key-value pair, and a null value may be passed for the `properties` argument.

3.8.4. MessagePolicy Requirements

Each `ServerAuthContext` obtained through getAuthContext must initialize its encapsulated `ServerAuthModule` objects with a non-null value for `requestPolicy`. The encapsulated authentication modules may be initialized with a null value for `responsePolicy`.

The `requestPolicy` used to initialize the authentication modules of the `ServerAuthContext` must be constructed such that the value obtained by calling `isMandatory` on the `requestPolicy` accurately reflects whether (that is, true return value) or not (that is, false return value) authentication is required to access the web resource corresponding to the `HttpServletRequest` to which the `ServerAuthContext` will be applied. The message processing runtime is responsible for determining if authentication is required and must convey the results of its determination as described in Section 3.9.1.

Calling `getTargetPolicies` on the request `MessagePolicy` must return an array containing at least one `TargetPolicy` whose `ProtectionPolicy` will be interpreted by the modules of the context to mean that the source of the corresponding targets within the message is to be authenticated. To that end, calling

the `getID` method on the `ProtectionPolicy` must return one of the following values:

- `ProtectionPolicy.AUTHENTICATE_SENDER`
- `ProtectionPolicy.AUTHENTICATE_CONTENT`

3.9. Message Processing Requirements

For this profile, point (2) of the messaging processing model occurs after the runtime determines that the connection on which the request was received satisfies the connection requirements[6] that apply to the request and before the runtime enforces the authorization[7] requirements that apply to the request. At point (2) in the message processing model, the runtime must call `validateRequest` on the `ServerAuthContext`. The runtime must not call `validateRequest` if the request does not satisfy the connection requirements that apply to the request. If the request has satisfied the connection requirements, the message processing runtime must call `validateRequest` independent of whether or not access to the resource would be authorized prior to the call to validateRequest[8] The validateRequest method must be called for all requests (to which the Jakarta Servlet security model applies[9] , including submits of a form-based login form.

If the call to `validateRequest` returns any value other than `AuthStatus.SUCCESS`, the runtime should return a response and must discontinue its processing of the request.

If the call to `validateRequest` returns `AuthStatus.SUCCESS`, the runtime must establish return values for `getUserPrincipal`, `getRemoteUser`, and `getAuthType` as defined in Section 3.9.4. After setting the authentication results, the runtime must determine whether the authentication identity established in the `clientSubject` is authorized to access the resource. The identity tested for authorization must be selected based on the nature, with respect to Jakarta Authorization compatibility, of the calling runtime. In a Jakarta Authorization compatible runtime, the identity must be comprised of exactly the `Principal` objects of the `clientSubject`. In a non-Jakarta Authorization compatible Jakarta Servlet runtime, the identity must include the caller `Principal` (established during the `validateRequest` processing using the corresponding `CallerPrincipalCallback`) and may include any of the `Principal` objects of the clientSubject. Independent of the nature of the calling runtime, if the request is NOT authorized, the runtime must set, within the response, an HTTP status code as required by the Jakarta Servlet specification. The request must be dispatched to the resource if the request was determined to be authorized; otherwise it must NOT be dispatched and the runtime must proceed to point (3) in the message processing model.

If the request is dispatched to the resource and the resource invocation throws an exception to the runtime, the runtime must set, within the response, an HTTP status code which satisfies any applicable requirements defined within the Jakarta Servlet specification. In this case, the runtime should complete the processing of the request without calling `secureResponse`.

If invocation of the resource completes without throwing an exception, the runtime must proceed to point (3) in the message processing model. At point (3) in the message processing model, the runtime must call `secureResponse` on the same `ServerAuthContext` used in the corresponding call to `validateRequest` and with the same `MessageInfo` object.

If the request is dispatched to the resource, and the resource was configured to run-as its caller, then for invocations originating from the resource where caller propagation is required, the identity established using the `CallerPrincipalCallback` must be used as the propagated identity.

3.9.1. MessageInfo Requirements

The `messageInfo` argument used in the call to `validateRequest` must have been initialized by the runtime such that its `getRequestMessage` and `getResponseMessage` methods will return the `HttpServletRequest` and `HttpServletResponse` objects corresponding to the messages (respectively) being processed by the runtime. This must be the case even when the target of the request is a static page (that is, not a Servlet).

3.9.1.1. MessageInfo Properties

This profile requires that the message processing runtime conditionally establish the following key-value pair within the `Map` of the `MessageInfo` object passed in the calls to `getAuthContextID`, `validateRequest`, and `secureResponse`.

Table 3-4 MessageInfo Map Properties

key	value
jakarta.security.auth.message.MessagePolicy.isMandatory	Any non-null `String` value, s, for which Boolean.valueOf(s).booleanValue() == true

jakarta.security.auth.message.MessagePolicy.isMandatory

The `MessageInfo` map must contain this key and its associated value, if and only if authentication is required to perform the resource access corresponding to the `HttpServletRequest` to which the `ServerAuthContext` will be applied. Authentication is required if use of the HTTP method of the `HttpServletRequest` at the resource identified by the `HttpServletRequest` is covered by a Jakarta Servlet `auth-constraint` [10], or in a Jakarta Authorization compatible runtime, if the corresponding `WebResourcePermission` is NOT granted[11] to an unauthenticated caller. In a Jakarta Authorization compatible runtime, the corresponding `WebResourcePermission` may be constructed directly from the `HttpServletRequest as follows`:

```
public WebResourcePermission(HttpServletRequest request);
```

The authentication context configuration system must use the value of this property to establish the corresponding value within the `requestPolicy` passed to the authentication modules of the `ServerAuthContext` acquired to process the `MessageInfo`.

3.9.2. Subject Requirements

A new `clientSubject` must be instantiated and passed in the call to `validateRequest`.

3.9.3. ServerAuth Processing

As described in Section 3.9, the profile requires that `validateRequest` be called on every request that satisfies the corresponding connection requirements (and to which the Jakarta Servlet container security model applies). As such, `validateRequest` will be called either before the service invocation (to establish the caller identity) or after the service invocation (when a multi-message dialog is required to secure the response). The module implementation is responsible for recording any state and performing any processing required to differentiate these two different types of calls to `validateRequest`.

3.9.3.1. validateRequest Before Service Invocation

When `validateRequest` is called before the service invocation on a module initialized with a mandatory requestPolicy (as defined by the return value from `requestPolicy.isMandatory()`), the module must only return `AuthStatus.SUCCESS` if it was able to completely satisfy the request authentication policy. In this case, the module (or its context) must also have used the `CallbackHandler` passed to it by the runtime to handle a `CallerPrincipalCallback` using the `clientSubject` as argument to the callback. If more than one module of a context uses the `CallbackHandler` to `handle` this callback, the context is responsible for coordinating the calls such that the appropriate caller principal value is established.

If the module was not able to completely satisfy the request authentication policy, it must:

- return `AuthStatus.SEND_CONTINUE` – If it has established a response (available to the runtime by calling `messageInfo.getResponseMessage`) that must be sent by the runtime for the request validation to be effectively continued by the client. The module must have set the HTTP status code of the response to a value (for example, HTTP 401 unauthorized, HTTP 303 see other, or HTTP 307 temporary redirect) that will indicate to the client that it should retry (or continue) the request. This, however, is solely the responsibility of the module, and the runtime must be liberal in its acceptance of continue responses, including responses with HTTP success status codes; such as might be returned with forms (including login forms and forms that depend on javascript to be relayed through the browser).
- return `AuthStatus.SEND_FAILURE` – If the request validation failed, and when the client should not retry or continue with its processing of the request. The module must have established a response message (available to the runtime by calling messageInfo.getResponseMessage) that may be sent by the runtime to inform the client that the request failed. The module must have set the HTTP status code of the response to a value (for example, HTTP 403 forbidden or HTTP 404 not found) that will indicate to the client that it should NOT continue the request. The runtime may choose not to send a response message, or to send a different response message (given that it also contains an analogous HTTP status code).
- throw an `AuthException` – If the request validation failed, and when the client should not retry the request, and when the module has not defined a response to be sent by the runtime. If the runtime chooses to send a response, it must define the HTTP status code and descriptive content (of the response). The HTTP status code of the response must indicate to the client (for example, HTTP 403 forbidden, HTTP 404 not found, or HTTP 500 internal server error) that the request failed and that it should NOT be retried. The descriptive content set in the response may be obtained from the

AuthException.

When `validateRequest` is called before the service invocation on a module that was initialized with an optional requestPolicy (that is, `requestPolicy.isMandatory()` returns false), the module should attempt to satisfy the request authentication policy, but it must do so without initiating[12] additional message exchanges or interactions involving the client. Independent of whether the authentication policy is satisfied, the module may return `AuthStatus.SUCCESS`. If the module returns `AuthStatus.SUCCESS` (and the authentication policy was satisfied), the module (or its context) must employ a `CallerPrincipalCallback` as described above. If the authentication policy was not satisfied, and yet the module chooses to return `AuthStatus.SUCCESS`, the module (or its context) must use a `CallerPrincipalCallback` to establish the container's representation of the unauthenticated caller within the `clientSubject`. If the module determines that an invalid or incomplete security context was used to secure the request, then the module may return `AuthStatus.SEND_FAILURE`, `AuthStatus.SEND_CONTINUE`, or throw an `AuthException`. If the module throws an `AuthException`, or returns any value other than `AuthStatus.SUCCESS`, the runtime must NOT proceed to the service invocation. The runtime must process an `AuthException` as described above for a request with a mandatory `requestPolicy`. The runtime must process any return value other than `AuthStatus.SUCCESS` as it would be processed if it were returned for a request with a mandatory `requestPolicy`.

3.9.3.2. validateRequest After Service Invocation

When `validateRequest` is called after the service invocation has completed[13], the module must return `AuthStatus.SEND_SUCCESS` when the module has successfully secured the application response message and made it available through `messageInfo.getResponseMessage`. For the request to be successfully completed, the runtime must send the response message returned by the module.

When securing of the application response message has failed, and the response dialog is to be terminated, the module must return `AuthStatus.SEND_FAILURE` or throw an `AuthException`.

If the module returns `AuthStatus.SEND_FAILURE`, it must have established a response message in `messageInfo`, and it must have set the HTTP status code within the response to HTTP 500 (internal server error). The runtime may choose not to send a response message, or to send a different response message (given that it also contains an HTTP 500 status code).

When the module throws an `AuthException`, the runtime may choose not to send a response. If the runtime sends a response, the runtime must set the HTTP status code to HTTP 500 (internal server error), and the runtime must define the descriptive content of the response (perhaps by obtaining it from the `AuthException`).

The module must return `AuthStatus.SEND_CONTINUE` if the response dialog is to continue. This status value is used to inform the calling runtime that, to successfully complete the response processing, it must be capable of continuing the message dialog by processing at least one additional request/response exchange (after having sent the response message returned in `messageInfo`). The module must have established (in `messageInfo`) a response message that will cause the client to continue the response processing (that is, retry the request). For the response processing to be successfully completed, the runtime must send the response message returned by the module.

3.9.3.3. secureResponse Processing

The return value and `AuthException` semantics of secureResponse are as defined in Section 3.9.3.2. This profile places no requirements on authentication modules with respect to interpreting `responsePolicy` values.

3.9.3.4. Forwards and Includes by Server Authentication Modules

The message processing runtime must support the acquisition and use of `RequestDispatcher` objects by authentication modules within their processing of `validateRequest`. Under the constraints defined by `RequestDispatcher`, authentication modules must be able to `forward` and `include` using the request and response objects passed in `MessageInfo`. In particular, an authentication module must be able to acquire a `RequestDispatcher` from the request obtained from `MessageInfo`, and uses it to forward the request (and response) to a login form. Authentication modules should catch and rethrow as an `AuthException` any exception thrown by these methods.

3.9.3.5. Wrapping and UnWrapping of Requests and Responses

A `ServerAuthModule` must only call `MessageInfo.setResponseMessage()` to wrap or unwrap the existing response within `MessageInfo`. That is, if a `ServerAuthModule` calls `MessageInfo.setResponseMessage()`, the response argument must be an `HtppServletResponseWrapper` that wraps the `HttpServletResponse` within `MessageInfo`, or the response argument must be an `HttpServletResponse` that is wrapped by the `HttpServletResponseWrapper` within `MessageInfo`. The analogous requirements apply to `MessageInfo.setRequestMessage()`.

During `secureResponse` processing, a `ServerAuthModule` must unwrap the messages in `MessageInfo` that it wrapped during its `validateRequest` processing. The unwrapped values must be established in `MessageInfo` when `secureResponse` returns. The module should not remove wrappers for which it is not responsible.

During `validateRequest` processing, a `ServerAuthModule` must NOT unwrap a message in `MessageInfo`, and must NOT establish a wrapped message in `MessageInfo` unless the `ServerAuthModule` returns `AuthStatus.SUCCESS`. For example, if during `validateRequest` processing a `ServerAuthModule` calls `MessageInfo.setResponseMessage()`, the response argument must be an `HttpServletResponseWrapper` that wraps the `HttpServletResponse` within `MessageInfo`.

When a `ServerAuthModule` returns a wrapped message in `MessageInfo`, or unwraps a message in `MessageInfo`, the message processing runtime must ensure that the `HttpServletRequest` and `HttpServletResponse` objects established by the `ServerAuthModule` are used in downstream processing.

3.9.4. Setting the Authentication Results on the HttpServletRequest

The requirements defined in this section must be fulfilled by a message processing runtime, when (at point (2) in the messaging model, `validateRequest` returns `AuthStatus.SUCCESS`. The requirements must also be fulfilled by `HttpServletRequest.authenticate` when its call to validateRequest returns `AuthStatus.SUCCESS`. In both cases, the `HttpServletRequest` must be modified as necessary to ensure that

the Principal returned by getUserPrincipal and the String returned by getRemoteUser correspond, respectively, to the Principal established by validateRequest (via the CallerPrincipalCallback) and to the String obtained by calling getName on the established Principal footnote:Except when getUserPrincipal returns null; in which case the value returned by getRemoteUser must be null]. Both cases, must also ensure that the value returned by calling getAuthType on the HttpServletRequest is consistent in terms of being null or non-null with the value returned by getUserPrincipal.

When getAuthType is to return a non-null value, the Map of the MessageInfo object used in the call to validateRequest must be consulted to determine if it contains an entry for the key identified in Table 3-5 . If the Map contains an entry for the key, the corresponding value must be obtained from the Map and established as the getAuthType return value. If the Map does not contain an entry for the key, and an auth-method is defined in the login-config element of the deployment descriptor for the web application, the value from the auth-method must be established as the getAuthType return value. If the Map does not contain an entry for the key, and the deployment descriptor does not define an auth-method, a product defined default non-null value must be established as the getAuthType return value, and the same default value need not be used for both cases.

Table 3-5 Authentication Type (Callback) Property

key	value
jakarta.servlet.http.authType	A non-null String value that identifies the authentication mechanism

If a non-null Principal was established by validateRequest (via the CallerPrincipalCallback), the Map of the MessageInfo object used in the call to validateRequest must be consulted to determine if it contains an entry for the key identified in Table 3-6 . If the Map contains an entry for the key, the authentication session machinery of the container must be used to create (or update) a container authentication session to represent the caller Principal, authType, and the additional container authentication state established by the call to validateRequest. The resulting container authentication session must be bound to the HttpServletResponse such that the container will be able to restore the caller authentication results on subsequent calls to the application.

Table 3-6 Authentication Session Registration (Callback) Property

key	value
jakarta.servlet.http.registerSession	Any non-null String value, s, for which Boolean.valueOf(s).booleanValue() == true

The authentication type and session registration properties are callback properties[14] and are intended to provide a way for an authentication module to request a corresponding service from its encompassing runtime. As such, all authentication modules must ensure that they do not inadvertently relay these properties should they be included in their input MessageInfo arguments.

3.10. Sub-profile for authenticate, login, and logout of HttpServletRequest

The Servlet `HttpServletRequest` interface contains methods related to authentication, namely: the `authenticate`, `login`, and `logout` methods. A compatible implementation of the Servlet Container Profile must satisfy the requirements defined in this sub-profile. This sub-profile differs from the larger profile in which it is contained, in that it describes the handling of calls that would typically be expected to occur within the service invocation; while the focus of the larger profile, is on points (2) and (3) in the messaging model (which occur on either side of the service invocation).

3.10.1. Authentication Configuration Requirements

When an application calls `HttpServletRequest.authenticate`, `HttpServletRequest.login`, or `HttpServletRequest.logout`, the container implementation of the called method must determine (as defined in Section 3.7) if there is an `AuthConfigProvider` configured for the application context and layer. If not, the called method must proceed to perform the required `authenticate`, `login`, or `logout` functionality without further reliance on this sub-profile.

If an `AuthConfigProvider` is determined to be configured, the called method must proceed to obtain the corresponding `ServerAuthConfig` also as defined in Section 3.7.

As described in Section 2.1.1, the called method may reuse the results of a previous `AuthConfigProvider` determination and `ServerAuthConfig` acquisition (such as that performed by the message processing runtime) during its processing of the servlet request within which the `authenticate`, `login`, or `logout` method is being called.

3.10.2. Processing for HttpServletRequest.login

The container implementation of `login` must throw a `ServletException` which may convey that the exception was caused by an incompatibility between the login method and the configured authentication mechanism.

3.10.3. Processing for HttpServletRequest.authenticate

If `authenticate` is called in the context of a call it made to `validateRequest`, it must not recall `validateRequest`, but must perform the container authentication processing that it performs when it determines that an `AuthConfigProvider` is not configured for the application context and layer.

Otherwise, `authenticate` must acquire the corresponding `ServerAuthContext` object as defined in Section 3.8 (and its subsections), while satisfying the additional requirement that the authentication context identifier used to obtain the `ServerAuthContext` must be the identifier that would be acquired by calling `getAuthContextID` with `MessageInfo` as defined in Section 3.9.1 and while satisfying the additional requirement that the `MessageInfo` map must unconditionally contain both the `jakarta.security.auth.message.MessagePolicy.isMandatory` key (with associated `true` value) and the `jakarta.servlet.http.isAuthenticationRequest` key (with associated `true` value).

`Authenticate` must call `validateRequest` on the acquired `ServerAuthContext`. The `MessageInfo` argument to the call to `validateRequest` must be as defined above. The `clientSubject` argument must be a non-null `Subject` and should be the `Subject` resulting from the call to `validateRequest` prior to the service invocation as described in Section 3.9.3.1. If the prior Subject is not used, A new (empty) clientSubject must be instantiated and passed in the call to validateRequest. A null value may be used for the serviceSubject.

If the call to validateRequest returns `AuthStatus.SUCCESS`, the authenticate method must perform the processing defined in Section 3.9.4. This processing includes establishing return values for `getUserPrincipal`, `getRemoteUser`, and `getAuthType` and may include the registration of the authentication results in a container authentication session[15] Following this processing, the `authenticate` method must return the boolean value `true`, and if the calling context is configured to run-as its caller, the results of the authentication must be reflected in the run-as identity.

If the call to `validateRequest` throws an `AuthException`, the `authenticate` method must catch the `AuthException` and throw a `ServletException`.

If the call to `validateRequest` returns any value other than `AuthStatus.SUCCESS`, the `authenticate` method must return `false`.

3.10.4. Processing for HttpServletRequest.logout

If logout is called in the context of a call it made to cleanSubject, it must not recall `cleanSubject`, but it must perform the logout processing that it performs when it determines that an `AuthConfigProvider` is not configured for the application context and layer.

Otherwise, `logout` must acquire the corresponding `ServerAuthContext` object as defined in Section 3.8 (and its subsections), while satisfying the additional requirement that the authentication context identifier used to obtain the `ServerAuthContext` must be the identifier that would be acquired by calling `getAuthContextID` with `MessageInfo` as defined in Section 3.9.1 and while satisfying the additional requirement that the `MessageInfo` map must unconditionally contain the `jakarta.security.auth.message.MessagePolicy.isMandatory` key (with associated `true` value). Logout should attempt to satisfy the requirement of Section 3.9.1, that `MessageInfo` be initialized such that its `getResponseMessage` will return the `HttpServletResponse`, but need not do so if the response is unavailable or committed.

The container implementation of `logout` must call `cleanSubject` on the acquired `ServerAuthContext`. The `MessageInfo` argument to the call to `cleanSubject` must be as defined above. The `clientSubject` argument must be a non-null `Subject` and should be the `Subject` resulting from the most recent call to `validateRequest` which may have occurred either as described in Section 3.9.3.1 or as described in <<a487>. If the prior `Subject` is not used, a new `clientSubject` must be instantiated and passed in the call.

Following the return from `cleanSubject`, `logout` must perform the logout processing that it performs when it determines that an `AuthConfigProvider` is not configured for the application context and layer, and if the calling context is configured to run-as its caller, the results of the logout must be reflected in

the run-as identity.

3.10.5. Calls from within ServerAuthContext

If `HttpServletRequest.authenticate` or `HttpServletRequest.logout` is called from within the methods of the `ServerAuthContext` interface (for example, from within `validateRequest`, `secureResponse`, or `cleanSubject`), it is the responsibility of the implementation of the `ServerAuthContext` to interpret the results of the call and to establish appropriate `ServerAuthContext` return values. This profile is silent on the details of the interpretation and mapping of return values.

3.11. Interaction with other specifications

When this profile is used as part of a Jakarta EE compatible implementation, the requirements as stated in the sub-sections below MUST be satisfied.

When this profile is NOT used in a Jakarta EE compatible implementation, but this implementation uses one or more of the specifications as outlined in the sub-sections below, then the requirements as stated in the relevant sub-sections SHOULD be satisfied.

3.11.1. Availability of Jakarta EE component namespaces

The Jakarta EE JNDI component namespaces (`java:global`, `java:app`, `java:module`, `java:comp`) MUST be made available to code running in the context of a call to `validateRequest`, `secureResponse` and `cleanSubject` on the acquired `ServerAuthContext`.

A practical use case for this is obtaining (application scoped) data sources, which a `ServerAuthModule` could use to validate credentials.

Example:

```
new InitialContext().lookup("java:app/myds")
```

3.11.2. Availability of CDI scopes

The CDI built-in scopes according to "2.4.1. Built-in scope types" of the CDI specification MUST be made available to code running in the context of a call to `validateRequest`, `secureResponse` and `cleanSubject` on the acquired `ServerAuthContext`.

A practical use case for this is obtaining application scoped identity stores, which a `ServerAuthModule` could use to validate credentials.

Example:

```
CDI.current().select(SomeBean.class); // SomeBean is @RequestScoped
```

Note that it is a non-requirement that a `ServerAuthModule` is itself a CDI managed bean, and as such it is not required that services such as injection using the `@Inject` annotation are available to a `ServerAuthModule`. It is only required that programmatic lookup such as shown in the example above works correctly.

[6] In a Jakarta Authorization environment, connection requirements are tested by checking a `WebUserDataPermission` constructed with the `HttpServletRequest`. In a non-Jakarta Authorization environment, connection requirements are tested by comparing the security properties of the connection on which the request was received with the permitted connection types as defined through user-data-constraints in the corresponding web.xml.

[7] In a Jakarta Authorization environment, authorization requirements are enforced by checking if the authenticated caller identity (such as it is) has been granted the `WebResourcePermission` corresponding to the `HttpServletRequest`. In a non-Jakarta Authorization environment, authorization requirements are enforced by checking if the role-mappings of the authenticated caller identity are sufficient to satisfy the auth-constraints (if any) that apply to the request as defined in the corresponding web.xml.

[8] These unconditional calls to `validateRequest` are necessary to allow for delegation of servle tauthentication sessionmanagement to authentication contexts and their contained authentication modules.

[9] Note that the Jakarta Servlet security model does not apply when a servlet uses a `RequestDispatcher` to invoke a static resource or servlet using a forward or an include.

[10] If the auth-constraint is an excluding auth-constraint (that is, an auth-constraint that authorizes no roles), the Servlet Specification requires that no access be permitted independent of authentication. Runtimes should reject requests to ex- cluded resources prior to proceeding to point (2) in the message processing model (that is, prior to the authentication processing).

[11] Jakarta Authorization compatible runtimes should also reject requests to excluded resources prior to proceeding to point (2) in the message processing model (that is, prior to the authentication processing).

[12] The module may continue, or refresh an authentication dialog that has already been initiated (perhaps by the client) in the request, but it must not start an authentication dialog for a request which has not yet been associated with authentication information (as understood by the module).

[13] “After the service invocation” effectively means after the first call to secureResponse; as distinct from the case where authenticate might call validateRequest from within the service invocation and before it completes.

[14] Unlike CallbackHandler processed Callback objects, callback properties are not acted upon until the authentication module returns to the runtime.

[15] Note that the authenticate method must not perform the pre-dispatch container authorization check that the message processing runtime would typically perform on successful return from `validateRequest`.

Chapter 4. SOAP Profile

This chapter defines a profile of the use of the interfaces defined in this specification to secure SOAP message exchanges between web services client runtimes and web service endpoint runtimes. This profile is equally applicable to SOAP versions 1.1 and 1.2.

This profile is composed of two internal profiles that partition the requirements of the profile into those that must be satisfied by client runtimes and those that must be satisfied by server runtimes. The profile-specific requirements defined in this chapter are to be considered in addition to the generic requirements defined in Chapter 2. A compatible implementation of an internal profile of this specification is an implementation that satisfies all of the requirements that apply to that profile.

4.1. Message Layer Identifier

The message layer value used to select the `AuthConfigProvider` and ServerAuthConfig objects for this profile must be “SOAP”.

4.2. Application Context Identifier

The application context identifier (that is, the `appContext` parameter value) used by a client runtime to select the `AuthConfigProvider` and `ClientAuthConfig` objects pertaining to a client-side application context configuration scope must be as defined in See Client-Side Application Context Identifier.

Similarly, the application context identifier used by a server runtime to select the `AuthConfigProvider` and `ClientAuthConfig` objects pertaining to an server-side application context configuration scope must be as defined in Section 4.9.1.

4.3. Message Requirements

The `MessageInfo` argument used in any call made by the message processing runtime to secureRequest, validateResponse, validateRequest, or `secureResponse` must have been initialized such that any non-null objects returned by the getRequestMessage and getResponseMessage methods of the `MessageInfo` are an instanceof jakarta.xml.soap.SOAPMessage.

4.4. Module Requirements

The getSupportedMessageTypes method of all authentication modules integrated for use with this profile must include jakarta.xml.soap.SOAPMessage.class in its return value.

4.5. CallbackHandler Requirements

The CallbackHandler passed to an authentication module’s initialize method is determined by the handler argument passed in the call to `AuthConfigProvider.getClientAuthConfig` or getServerAuthConfig

that acquired the corresponding authentication context configuration object.

The handler argument must not be null, and the argument handler and the CallbackHandler passed to the initialize method of all authentication modules should support the following callbacks, and it must be possible to configure the runtime such that the CallbackHandler passed at module initialization supports the following callbacks (in addition to any others required to be supported by the applicable internal profile):

- CertStoreCallback
- PrivateKeyCallback
- SecretKeyCallback
- TrustStoreCallback

The argument handler and the CallbackHandler passed through to the modules must be initialized with any application context required to process the supported callbacks on behalf of the corresponding application.

4.6. AuthConfigProvider Requirements

The factory implementation returned by calling the getFactory method of the abstract AuthConfigFactory class must be configured such that it returns a non-null AuthConfigProvider for those application contexts for which pluggable authentication modules have been configured at the "SOAP" layer.

For each application context for which it is servicing requests, the runtime must call getConfigProvider to acquire the provider object corresponding to the layer and application context. The layer and appContext arguments to getConfigProvider must be as defined in Section 4.1 and Section 4.2 respectively.

A null return value from `getConfigProvider` indicates that pluggable authentication modules have not been configured at the layer for the application context, and that the messaging runtime must proceed to perform its SOAP message processing (for the application context) without further reliance on this profile.

4.7. Authentication Context Requirements

When a non-null `AuthConfigProvider` is returned by the factory, the provider must have been configured with the information required to initialize the authentication contexts for the one or more authentication context configuration scopes, defined by layer and application context, for which the provider is registered (at the factory). The information typically required to initialize authentication contexts is described by example in Section 2.1.4.2.

When a non-null `AuthConfigProvider` is returned by the factory, the messaging runtime must call `getAuthContext` on the authentication context configuration object (obtained from the provider). The

authContextID argument used in the call to `getAuthContext` must be the value as described in Section 4.7.1.

A null return value from `getAuthContext` indicates that pluggable authentication modules have not been configured for the web service invocation within the authentication context configuration scope, and that the runtime must proceed to perform its SOAP message processing for this request/response without further reliance on this profile.

Effective integration of a session-oriented authentication mechanism for use in an authentication context configuration scope should be expected to require configuration of the corresponding `AuthConfigProvider` such that `getAuthContext` will return non-null authentication context objects for all legitimate authContextID values acquired for the corresponding scope.

4.7.1. Authentication Context Identifiers

This profile does NOT impose any profile specific requirements on authentication context identifiers. As defined in Section 2.1.3, the authentication context identifier used in the call to getAuthContext must be equivalent to the value that would be acquired by calling getAuthContextID with the MessageInfo that will be used in the corresponding call to secureRequest (by a client runtime) or validateRequest (by a server runtime).

4.7.2. MessagePolicy Requirements

Each authentication context object obtained through `getAuthContext` must initialize its encapsulated authentication modules with a non-null requestPolicy and/or a non-null responsePolicy, such that at least one of requestPolicy or responsePolicy is not null.

4.8. Requirements for Client Runtimes

This section defines the requirements of this profile that must be satisfied by a runtime operating in the client role. A runtime may operate in both the client and server roles.

4.8.1. Client-Side Application Context Identifier

The application context identifier used by a client-runtime to acquire the `AuthConfigProvider`and ClientAuthConfig objects pertaining to the client side processing of a web service invocation shall begin with a client scope identifier that identifies the client. If the client-runtime may host multiple client applications, then the client scope identifier must differentiate among the client applications deployed within the runtime. In runtimes where applications are differentiated by unambiguous application identifiers, an application identifier may be used as the client scope identifier. Where application identifiers are not defined or suitable, the location (for example, its file path) of the client archive from which the invocation will originate may be used as the client scope identifier.

In addition to its client scope identifier, the application context identifier must include a client reference to the service. If a service reference is defined for the invocation (for example, by using a

WebServiceRef annotation as defined in the Jakarta XML Web Services specifications), the client reference to the service must be the name value of the service reference. If a service reference was not defined for the invocation, the client reference to the service must be the web service URL.

A client application context identifier must be the String value composed by concatenating the client scope identifier, a blank separator character, and the client reference to the service.

```
AppContextID ::= client-scope-identfier blank client-reference
```

The following are examples of client application context identifiers.

```
"petstoreAppID service/petstore/delivery-service"

"petstoreAppID http://localhost:8080/petstore/delivery-service/fish"

"/home/fishkeeper/petstore-client.jar service/petstore/delivery-service"

"/home/fishkeeper/petstore-client.jar http://localhost:8080/petstore/delivery-
service/fish"
```

Systems or administrators that register `AuthConfigProvider` objects with specific client-side application context identifiers must have an ability to determine the client scope identifier and the client reference for which they wish to perform the registration.

4.8.2. CallbackHandler Requirements

Unless the client runtime is embedded in a server runtime (for example, an invocation of a web service by a servlet running in a Servlet container), the CallbackHandler passed to ClientAuthModule.initialize must support the following callbacks:

- NameCallback
- PasswordCallback

In either event, the CallbackHandler must also support the requirements in Section 4.5

4.8.3. AuthConfigProvider Requirements

If a non-null `AuthConfigProvider`is returned (by the call to getConfigProvider), the messaging runtime must call `getClientAuthConfig` on the provider to obtain the authentication context configuration object pertaining to the application context at the layer. The layer and appContext arguments of the call to getClientAuthConfig must be the same as those used to acquire the provider, and the handler argument must be as defined in Section 4.8.2 for a client runtime.

4.8.4. Authentication Context Requirements

The `getAuthContext` calls made on the `ClientAuthConfig` (obtained by calling `getClientAuthConfig`) must satisfy the requirements defined in the following subsections.

4.8.4.1. getAuthContext Subject

A non-null Subject corresponding to the client must be passed as the clientSubject in the `getAuthContext` call.

4.8.4.2. Module Initialization Properties

A null value may be passed for the properties argument in all calls made to `getAuthContext`.

4.8.4.3. MessagePolicy Requirements

Each ClientAuthContext obtained through getAuthContext must initialize its encapsulated ClientAuthModule objects with requestPolicy and responsePolicy objects (or null values) that are compatible with the requirements and capabilities of the service invocation (at the service). The requirements, preferences, and capabilities of the client may be factored in the context acquisition and may effect the requestPolicy and responsePolicy objects passed to the authentication modules of the context.

4.8.5. Message Processing Requirements

A client runtime, after having prepared (except for security) the SOAP request message to be sent to the service, is operating at point (1) in the message processing model defined by this specification. A client runtime that has received a SOAP response message, and that has not yet performed any transformations on the response message, is operating at point (4) in the message processing model defined by this specification.

If the client runtime obtained a non-null `ClientAuthContext` by using the authentication context identifier corresponding to the request message, then at point (1) in the message processing model, the runtime must call secureRequest on the `ClientAuthContext`, and at point (4) the runtime must call validateResponse on the `ClientAuthContext`.

When processing a one-way application message exchange pattern, the runtime must not proceed to point (4) unless the return value from secureRequest (or a from validateResponse) is `AuthStatus.SEND_CONTINUE`.

4.8.5.1. MessageInfo Requirements

The `messageInfo` argument used in a call to secureRequest must have been initialized by the runtime such that its getRequestMessage will return the SOAP request message being processed by the runtime.

When a corresponding call is made to validateResponse, it must be made with the same `messageInfo` and clientSubject arguments used in the corresponding call to secureRequest, and it must have been

initialized by the runtime such that its `getResponseMessage` method will return the SOAP response message being processed by the runtime.

MessageInfo Properties

This profile requires that the message processing runtime establish the following key-value pairs within the Map of the MessageInfo passed in the calls to secureRequest and validateResponse.

Table 4-7 Client MessageInfo Map Properties

key	value
jakarta.xml.ws.wsdl.service	The value of the qualified service name, represented as a javax.xml.namespace.QName. specification

4.8.5.2. Subject Requirements

The `clientSubject` used in the call to `getAuthContext` must be used in the call to `secureRequest` and for any corresponding calls to `validateResponse`.

4.8.5.3. secureRequest Processing

When secureRequest is called on a module that was initialized with a mandatory request policy (as defined by the return value from requestPolicy.isMandatory()), the module must only return AuthStatus.SEND_SUCCESS if it was able to completely satisfy the request policy. If the module was not able to completely satisfy the request policy, it must:

- Return `AuthStatus.SEND_CONTINUE` – If it has established an initial request (available to the runtime by calling messageInfo.getRequestMessage) that must be sent by the runtime for the request to be effectively continued and when additional message exchanges will be required to achieve successful completion of the secureRequest processing.
- `Return AuthStatus.FAILURE` – If it failed securing the request and only if it established a response message containing a SOAP fault element (available to the runtime by calling `messageInfo.getResponseMessage`) that may be returned to the application to indicate that the request failed.
- Throw an `AuthException` – If it failed securing the request and did not establishing a failure response message. The runtime may choose to return a response message containing a SOAP fault element, in which case, the runtime must define the content of the message and of the fault, and may do so based on the content of the AuthException.

When secureRequest is called on a module that was initialized with an optional requestPolicy (that is, `requestPolicy.isMandatory()` returns false), the module may attempt to satisfy the request policy and may return AuthStatus.SEND_SUCCESS independent of whether the policy was satisfied.

The module should NOT throw an AuthException or return AuthStatus.FAILURE. The module may initiate a security dialog, as described above for AuthStatus.SEND_CONTINUE, but should not do so if the client cannot accommodate the possibility of a failure of an optional security dialog.

When secureRequest is called on a module that was initialized with an undefined request policy (that is, requestPolicy === null), the module must return AuthStatus.SEND_SUCCESS.

4.8.5.4. validateResponse Processing

`validateResponse` may be called either prior to the service invocation to process a response received during the secureRequest processing (when a multi-message dialog is required to secure the request), or after the service invocation and during the process of securing the response generated by the service invocation. The module implementation is responsible for recording any state and performing any processing required to differentiate these contexts.

validateResponse After Service Invocation

When validateResponse is called after the service invocation on a module that was initialized with a mandatory response policy (as defined by the return value from responsePolicy.isMandatory()), the module must only return AuthStatus.SUCCESS if it was able to completely satisfy the response policy. If the module was not able to completely satisfy the response policy, it must:

- Return `AuthStatus.SEND_CONTINUE` – If it has established a request (available to the runtime by calling messageInfo.getRequestMessage) that must be sent by the runtime for the response validation to be effectively continued by the client.
- Return `AuthStatus.FAILURE` – If response validation failed and only if the module has established a response message containing a SOAP fault element (available to the runtime by calling `messageInfo.getResponseMessage`) that may be returned to the application to indicate that the response validation failed.
- Throw an `AuthException` – If response validation failed without establishing a failure response message. The runtime may choose to return a response message containing a SOAP fault element, in which case, the runtime must define the content of the message and of the fault, and may do so based on the content of the AuthException.

When validateResponse is called after the service invocation on a module that was initialized with an optional responsePolicy (that is, responsePolicy.isMandatory() returns false), the module should attempt to satisfy the response policy, but it must do so without initiating[16] additional message exchanges or interactions involving the service. Independent of whether the response policy is satisfied, the module may return AuthStatus.SUCCESS. If the module determines that an invalid or incomplete security context was used to secure the response, then the module may return AuthStatus.FAILURE, AuthStatus.SEND_CONTINUE, or throw an AuthException. The runtime must process an AuthException as described above for a response with a mandatory responsePolicy. The runtime must process any return value other than AuthStatus.SUCCESS as it would be processed if it were returned for a response with a mandatory responsePolicy.

When `validateResponse` is called after the service invocation on a module that was initialized with an undefined response policy (that is, `responsePolicy == null`), the module must return AuthStatus.SUCCESS.

validateResponse Before Service Invocation

When validateResponse is called before the service invocation[17], the module must return AuthStatus.SEND_CONTINUE if the request dialog is to continue. This status value is used to inform the client runtime that, to successfully complete the request processing, it must be capable of continuing the message dialog by processing at least one additional request/response exchange. The module must have established (in `messageInfo`) a request message that will cause the service to continue the request processing. For the request processing to be successfully completed, the runtime must send the request message returned by the module.

If the module returns AuthStatus.FAILURE, it must have established a SOAP message containing a SOAP fault element as the response in `messageInfo` and that may be returned to the application to indicate that the request failed.

If the module throws an AuthException, the runtime may choose to return a response message containing a SOAP fault element, in which case, the runtime must define the content of the message and of the fault, and may do so based on the content of the AuthException.

4.9. Requirements for Server Runtimes

This section defines the requirements of this profile that must be satisfied by a runtime operating in the server role. A runtime may operate in both the client and server roles.

4.9.1. Server-Side Application Context Identifier

The application context identifier used by a server-runtime to acquire the `AuthConfigProvider` and `ServerAuthConfig` objects pertaining to the endpoint side processing of an invocation shall be the String value constructed by concatenating a host name, a blank separator character, and the path[18] component of the service endpoint URI corresponding to the webservice.

```
AppContextID ::= hostname blank service-endpoint-uri

For example: "aquarium /petstore/delivery-service/fish"
```

In the definition of server-side application context identifiers, this profile uses the term `host name` to refer to the logical host that performs the service corresponding to a service invocation. Web service invocations may be directed to a logical host using various physical or `virtual host` names or addresses, and a message processing runtime may be composed of multiple logical hosts. Systems or administrators that register `AuthConfigProvider`objects with specific server-side application context identifiers must have an ability to determine the hostname for which they wish to perform the registration.

4.9.2. CallbackHandler Requirements

The `CallbackHandler` passed to `ServerAuthModule.initialize` must support the following callbacks:

- CallerPrincipalCallback
- GroupPrincipalCallback
- PasswordValidationCallback

The `CallbackHandler` must also support the requirements in Section 4.5

4.9.3. AuthConfigProvider Requirements

If a non-null `AuthConfigProvider` is returned (by the call to `getConfigProvider`), the messaging runtime must call getServerAuthConfig on the provider to obtain the authentication context configuration object pertaining to the application context at the layer. The layer and appContext arguments of the call to getServerAuthConfig must be the same as those used to acquire the provider, and the handler argument must be as defined in Section 4.9.2 for a server runtime.

4.9.4. Authentication Context Requirements

The `getAuthContext` calls made on the ServerAuthConfig object (obtained by calling `getServerAuthConfig`) must satisfy the requirements defined in the following subsections.

4.9.4.1. Module Initialization Properties

If the runtime is a Jakarta Authorization compatible Jakarta Enterprise Beans or Jakarta Servlet endpoint container, the properties argument passed in all calls to `getAuthContext` must contain the key-value pair shown in the following table.

Table 4-8 Jakarta Authorization Compatible Module Initialization Properties

key	value
jakarta.security.jacc.PolicyContext	The PolicyContext identifier value that the container must set to satisfy the Jakarta Authorization authorization requirements as described in “Setting the Policy Context” within the Jakarta Authorization specification

When the runtime is not a Jakarta Authorization compatible endpoint container, the properties argument used in all calls to getAuthContext must not include a `jakarta.security.jacc.PolicyContext` key-value pair, and a null value may be passed for the `properties` argument.

4.9.4.2. MessagePolicy Requirements

When a non-null `requestPolicy` is used to initialize the authentication modules of a `ServerAuthContext`, the requestPolicy must be constructed such that the value obtained by calling `isMandatory` on the requestPolicy accurately reflects whether (that is, true return value) or not (that is, false return value)

message protection within the SOAP messaging layer is required to perform the web service invocation corresponding to the MessageInfo used to acquire the ServerAuthContext. Similarly, the value obtained by calling isMandatory on a non-null responsePolicy must accurately reflect whether or not message protection is required (within the SOAP messaging layer) on the response (if there is one) resulting from the corresponding web service invocation

Calling `getTargetPolicies` on the requestPolicy corresponding to a web service invocation for which a SOAP layer client identity is to be established as the caller identity must return an array containing at least one TargetPolicy for which calling `getProtectionPolicy.getID()` returns one of the following values:

- ProtectionPolicy.AUTHENTICATE_SENDER
- ProtectionPolicy.AUTHENTICATE_CONTENT

When all of the operations of a web service endpoint require client authentication, each `ServerAuthContext` acquired for the endpoint must initialize its contained authentication modules with a requestPolicy that includes a TargetPolicy as described above and that mandates client authentication. When client authentication is required for some, but not all, operations of an endpoint, the requestPolicy used to initialize the authentication modules of a ServerAuthContext acquired for the endpoint must include a TargetPolicy as described above and should only mandate client authentication if client authentication is required for all of the operations mapped to the `ServerAuthContext`. When none of the operations mapped to a `ServerAuthContext` require client authentication, the requestPolicy used to initialize the authentication modules of the `ServerAuthContext` must NOT mandate client authentication.

4.9.5. Message Processing Requirements

A server runtime that has received a SOAP request message, and that has not yet performed any transformations on the SOAP message, is operating at point (2) in the message processing model defined by this specification. A server runtime, after having prepared (except for security) a SOAP response message to be returned to the client, is operating at point (3) in the message processing model defined by this specification.

When processing a one-way application message exchange pattern, the runtime must not proceed to point (3) in the message processing model, and the runtime must only return a response message when validateRequest returns AuthStatus.SEND_CONTINUE (in which case, the response defined by validateRequest is to be returned).

If the server runtime obtained a non-null ServerAuthContext by using the authentication context identifier corresponding to the request message, then at point (2) in the message processing model, the runtime must call validateRequest on the ServerAuthContext, and at point (3) the runtime must call `secureResponse` on the `ServerAuthContext`.

If the call to validateRequest returns AuthStatus.SUCCESS, the runtime must perform any web service authorization processing[19] required as a prerequisite to accessing the target resource. If authentication is required for the request to be authorized, the runtime must determine whether the

authentication identity established in the clientSubject is authorized to access the resource. In a Jakarta Authorization compatible runtime, the identity tested for authorization must be comprised of exactly the Principal objects of the clientSubject. If the request is NOT authorized, and the message-exchange pattern is not one-way, the runtime must set within the response (within `messageInfo`) a SOAP fault element as defined by the runtime. If the request was determined to be authorized, it must be dispatched to the resource. Otherwise the request must NOT be dispatched and the runtime must proceed to point (3) in the message processing model (as appropriate to the message exchange pattern).

If the invocation of the resource results in an exception being thrown by the resource to the runtime and the message exchange pattern is not one-way, the runtime must set within the response (within `messageInfo`) a SOAP fault element as defined by the runtime. Following the resource invocation, and if the message exchange pattern is not one-way, the runtime must proceed to point (3) in the message processing model. At point (3) in the message processing model, the runtime must call `secureResponse` on the same ServerAuthContext used in the corresponding call to `validateRequest` and with the same `MessageInfo` object.

If the request is dispatched to the resource, and the resource was configured to run-as its caller, then for invocations originating from the resource where caller propagation is required, the identity established using the CallerPrincipalCallback must be used as the propagated identity.

4.9.5.1. MessageInfo Requirements

The `messageInfo` argument used in a call to `validateRequest` must have been initialized by the runtime such that its `getRequestMessage` will return the SOAP request message being processed by the runtime.

When a corresponding call is made to `secureResponse`, it must be made with the same `messageInfo` and `serviceSubject` arguments used in the corresponding call to `validateRequest`, and it must have been initialized by the runtime such that its `getResponseMessage` method will return the SOAP response message being processed by the runtime.

MessageInfo Properties

This profile does not define any properties that must be included in the `Map` within the `MessageInfo` passed in calls to `validateRequest` and `secureResponse`.

4.9.5.2. Subject Requirements

A new `clientSubject` must be instantiated and passed in any calls made to `validateRequest`.

4.9.5.3. validateRequest Processing

`validateRequest` may be called either before the service invocation (to validate and authorize the request) or after the service invocation (when a multi-message dialog is required to secure the response). The module implementation is responsible for recording any state and performing any processing required to differentiate these contexts.

validateRequest Before Service Invocation

When `validateRequest` is called before the service invocation on a module initialized with a mandatory request policy (as defined by the return value from `requestPolicy.isMandatory()`), the module must only return AuthStatus.SUCCESS if it was able to completely satisfy the request policy. If the satisfied request policy includes a TargetPolicy element with a ProtectionPolicy of AUTHENTICATE_SOURCE or AUTHENTICATE_CONTENT, then the module (or its context) must employ the CallbackHandler passed to it by the runtime to handle a `CallerPrincipalCallback` using the `clientSubject` as argument to the callback. If more than one module of a context uses the CallbackHandler to handle this callback, the context is responsible for coordinating the calls such that the appropriate caller principal value is established.

If the module was not able to completely satisfy the request policy, it must:

- Return AuthStatus.SEND_CONTINUE – If it has established a response (available to the runtime by calling `messageInfo.getResponseMessage`) that must be sent by the runtime for the request validation to be effectively continued by the client.
- Return AuthStatus.SEND_FAILURE – If the request validation failed, and when the module has established a SOAP message containing a fault element (available to the runtime by calling `messageInfo.getResponseMessage`) that may be sent by the runtime to inform the client that the request failed.
- Throw an AuthException – If the request validation failed, and when the module has NOT defined a response, to be sent by the runtime. If the runtime chooses to send a response, it must define a SOAP message containing a SOAP fault element, and may use the content of the AuthException to do so.

When `validateRequest` is called before the service invocation on a module that was initialized with an optional request policy (that is, `requestPolicy.isMandatory()` returns false), the module should attempt to satisfy the request policy, but it must do so without initiating[20] additional message exchanges or interactions involving the client. Independent of whether the request policy is satisfied, the module may return AuthStatus.SUCCESS. If the module returns AuthStatus.SUCCESS, and the request policy was satisfied (and included a TargetPolicy element as described above), then the module (or its context) must employ the CallerPrincipalCallback as described above. If the request policy was not satisfied (and included a TargetPolicy element as described above), and yet the module chooses to return AuthStatus.SUCCESS, the module (or its context) must use a CallerPrincipalCallback to establish the container's representation of the unauthenticated caller within the clientSubject. If the module determines that an invalid or incomplete security context was used to secure the request, then the module may return AuthStatus.SEND_FAILURE, AuthStatus.SEND_CONTINUE, or throw an AuthException.If the module throws an AuthException, or returns any value other that AuthStatus.SUCCESS, the runtime must NOT proceed to the service invocation. The runtime must process an AuthException as described above for a request with a mandatory requestPolicy. The runtime must process any return value other than AuthStatus.SUCCESS as it would be processed if it were returned for a request with a mandatory requestPolicy.

When `validateRequest` is called before the service invocation on a module that was initialized with an

undefined request policy (that is, `requestPolicy` == null), the module must return AuthStatus.SUCCESS.

validateRequest After Service Invocation

When validateRequest is called after the service invocation[21], the module must return AuthStatus.SEND_SUCCESS when the module has successfully secured the application response message and made it available through `messageInfo.getResponseMessage`. For the request to be successfully completed, the runtime must send the response message returned by the module.

When securing of the application response message has failed, and the response dialog is to be terminated, the module must return AuthStatus.SEND_FAILURE or throw an AuthException.

If the module returns AuthStatus.SEND_FAILURE, it must have established a SOAP message containing a SOAP fault element as the response in `messageInfo`. The runtime may choose not to send a response message, or to send a different response message.

When the module throws an AuthException, the runtime may choose not to send a response. If the runtime sends a response, the runtime must define the content of the response.

The module must return AuthStatus.SEND_CONTINUE if the response dialog is to continue. This status value is used to inform the calling runtime that, to successfully complete the response processing, it will need to be capable of continuing the message dialog by processing at least one additional request/response exchange (after having sent the response message returned in `messageInfo`). The module must have established (in messageInfo) a response message that will cause the client to continue the response processing. For the response processing to be successfully completed, the runtime must send the response message returned by the module.

4.9.5.4. secureResponse Processing

When `secureResponse` is called on a module that was initialized with an undefined responsePolicy (that is, `responsePolicy == null`), the module must return AuthStatus.SEND_SUCCESS. Otherwise, the return value and AuthException semantics of `secureResponse` are as defined in "validateRequest After Service Invocation"

[16] The module may continue, or refresh an authentication dialog that has already been initiated (perhaps by the client) in the request, but it must not start an authentication dialog for a request which has not yet been associated with authentication information (as understood by the module).

[17] Occurs when the module is challenged by the server during secureRequest processing.

[18] For an http or https schema, the path must be the corresponding component of the "generic URI" syntax (that is, <scheme>://<authority><path>?<query>) described in section 3. of RFC 2396 "Uniform Resource Identifiers (URI): Generic Syntax". If the service is implemented as a Servlet, the path must begin with the context-path.

[19] This authorization processing would NOT be expected to include the enforcement of Servlet Auth-Constraints since they are defined at url-pattern granularity.

[20] The module may continue, or refresh an authentication dialog that has already been initiated (perhaps by the client) in the request, but it must not start an authentication dialog for a request which has not yet been associated with

authentication information (as understood by the module).

[21] Occurs when the module is challenged by the client during `secureResponse` processing.

Chapter 5. Future Profiles

This chapter presents initial thoughts on some other profiles that are being considered.

5.1. JMS Profile

This profile would use the interfaces defined in this specification to apply pluggable security mechanisms to JMS message exchanges.

5.1.1. Message Abstraction

This profile would employ `jakarta.jms.Message` as its message abstraction. Properties would be set on the Message to convey security credentials and security results.

5.1.2. Destinations

In this profile, application contexts could be defined for JMS destinations, such that authentication configuration providers could be registered for interactions with destinations, and such that authentication context configuration objects could be defined for interactions with destinations.

5.1.3. Message Processing Model

A client profile could require that `secureRequest` be called when a Message is sent by a MessageProducer to a Destination and that validateResponse be called when a Message is received by a MessageConsumer from a Destination.

A server profile could require that validateRequest be called when a Destination receives a message from a MessageProducer, and that secureResponse be called when a Destination sends a message to a MessageConsumer.

5.2. RMI/IIOP Portable Interceptor Profile

This profile would be implemented within portable interceptors, where it could be used secure RMI/IIOP message exchanges and to serve as security mechanism integration facility within the portable interceptor processing framework.

5.3. Message Abstraction

The profile would employ `org.omg.PortableInterceptor.ClientRequestInfo` for its client-side message abstraction, and `org.omg.PortableInterceptor.ServerRequestInfo` for its server-side message abstraction.

Chapter 6. LoginModule Bridge Profile

This chapter defines an internal contract that specifies how a server-side message layer authentication module (that is, an implementation of the `ServerAuthModule` interface as defined by this specification) may delegate some of its security processing responsibilities to a (JAAS) `LoginModule`. A `LoginModule` is an object that implements the `javax.security.auth.spi.LoginModule` interface in the Java Platform, Standard Edition.

6.1. Processing Model

The `ServerAuthModule` must create an instance of a `javax.security.auth.login.LoginContext`. If the `options` argument passed to the initialize method of the `ServerAuthModule` contains a non-null `String` value for the `String` key "javax.security.auth.login.LoginContext", then the `ServerAuthModule` must pass this value as the name parameter in its calls to the `LoginContext` constructor. If the options argument does not contain a non-null String value for this key, the `ServerAuthModule` must use its own fully qualified class name in its calls to the constructor. In either case, the administrator of the `javax.security.auth.login.Configuration` system of the `LoginContext` is responsible for establishing the `javax.security.auth.login.AppConfigurationEntry` objects (with corresponding login module name, control flag, and initialization options) to be returned for the entry name used by the `ServerAuthModule` and for the default entry name "other".

If the `ServerAuthModule` passes a `Subject` to the LoginContext constructor, it must pass its client `Subject`. The `ServerAuthModule` must pass a `CallbackHandler` to the constructor and the passed `CallbackHandler` must conform to the requirements of Section 6.3

A new LoginContext instance should be created for each new request, and a `LoginContext` instance should not be shared across different requests. Once a `LoginContext` object has been created, the `LoginContext.login` method may be invoked from within the `ServerAuthModule.validateRequest` method to delegate security processing to the `LoginModule` objects configured in the `LoginContext`.

6.2. Division of Responsibility

A `ServerAuthModule` must only interact with a `LoginModule` in a protocol-independent fashion. Specifically, a `ServerAuthModule` is the only entity that may interpret protocol-specific messages (a SOAP request or an HTTP Servlet request, for example). A `LoginModule` must only perform protocol-independent security processing (for example, verifying a username/password that was transmitted in the request).

A `LoginModule` requests information from the `ServerAuthModule` using the `ServerAuthModule` provided `CallbackHandler`. Since the `LoginModule` must only perform protocol-independent operations, it follows that any callback it requests from the handler must also be protocol-independent. It is the responsibility of the provided `CallbackHandler` implementation to return the requested protocol-independent information to the `LoginModule`. The `CallbackHandler` is responsible for any protocol-specific message parsing required to extract the protocol-independent information returned by the

CallbackHandler.

6.3. Standard Callbacks

This profile requires that the CallbackHandler provided by the ServerAuthModule to the LoginContext constructor support the javax.security.auth.callback.NameCallback and the javax.security.auth.callback.PasswordCallback. If the ServerAuthModule passes its client Subject to the LoginContext constructor, the CallbackHandler provided to the LoginContext constructor must also support the GroupPrincipalCallback. Future versions of this profile may require that additional callbacks be supported by the handler.

6.4. Subjects

If authentication succeeds, a LoginModule may update its Subject instance with authenticated Principal and credential objects. If the ServerAuthModule did not pass its client Subject to the LoginContext constructor, then it must transfer the Principals and credentials from the LoginContext Subject to the client Subject.

If the ServerAuthModule is implementing a profile of this specification that requires the module to employ the CallerPrincipalCallback, then the ServerAuthModule must satisfy this requirement using the CallbackHandler provided to the ServerAuthModule, and the CallerPrincipalCallback must be constructed using the name[22] value that would be obtained by the LoginModule if it were to use its CallbackHandler to handle a NameCallback.

6.5. Logout

When ServerAuthModule.cleanSubject is called on the client Subject, the cleanSubject method must invoke the LoginContext.logout method.

6.6. LoginExceptions

If the LoginContext instance throws a LoginException, the ServerAuthModule must throw a corresponding AuthException. The LoginException may be established as the cause of the AuthException.

[22] The CallerPrincipalCallback may be constructed with a String argument containing the name value, or with a Principal argument whose getName method returns the name value.

Appendix A: Related Documents

This specification refers to the following documents. The terms used to refer to the documents in this specification are included in brackets.

S. Bradner, "Key words for use in RFCs to Indicate Requirement Levels," RFC 2119, Harvard University, March 1997, [Keywords]

Jakarta EE 9 Specification [Jakarta EE 9 Specification], available at: https://github.com/eclipse-ee4j/jakartaee-platform

Jakarta Servlet Specification, Version 5.0 [Jakarta Servlet Specification], available at: https://github.com/eclipse-ee4j/servlet-api

Jakarta XML Web Services 3.0 [Jakarta XML Web Services Specification], available at: https://github.com/eclipse-ee4j/jax-ws-api

Jakarta Messaging Specification Version 3.0 [Jakarta Messaging Specification], available at: https://github.com/eclipse-ee4j/jms-api

Jakarta Enterprise Beans, Version 4.0_ [Jakarta Enterprise Beans Specification], available at: *https://github.com/eclipse-ee4j/ejb-api*

Java™, Standard Edition, Version 8.0 API Specification [Java SE 8 Specification], available at: https://docs.oracle.com/javase/8/docs/api

Java™ Authentication and Authorization Service (JAAS) [JAAS Specification], available at: https://docs.oracle.com/javase/8/docs/technotes/guides/security/jaas/JAASRefGuide.html_

SOAP Version 1.2 Part 0: Primer, W3C Recommendation, 24 June 2003 [SOAP Specification], available at: http://www.w3.org/TR/soap12-part0

Common Secure Interoperability, Version 2 (CSIv2),OMG standard [CSIv2 Specification], available at: http://www.omg.org/technology/documents/formal/omg_security.htm

Portable Interceptors, OMG Standard [PI Specification], available at: https://omg.org/spec/CORBA/3.3/Interfaces/PDF

Appendix B: Issues

The following sections document the more noteworthy issues that have historically been discussed by the Expert Group under the JCP. The expectation is that standardization of the interfaces defined by this specification will depend on satisfactory resolution of these issues.

B.1. Implementing getCallerPrincipal and getUserPrincipal

Jakarta EE containers and other messaging runtimes are required to support various forms of these methods. When the authentication identity is provided to the container as a bag of principals in a `Subject`, the container needs some way to recognize which of the principals in the subject should be returned as the caller or user Principal.

Resolution - Defined the `CallerPrincipalCallback` and `GroupPrincipalCallback`. The container provided `CallbackHandler` will handle these callbacks by distinguishing (in some container specific way) the Principals identified in the corresponding Callback within a `Subject` passed in the Callback.

B.2. Alternative Supported Mechanisms at an Endpoint

How does one use this SPI to configure and invoke alternative “sufficient” providers, such that satisfying any alternative within the context results in a successful outcome as seen by the calling container or runtime?

Resolution (Partial) - The `getAuthContext` method of `ClientAuthConfig` and `ServerAuthConfig` was modified to include the credentials of the client or service subject respectively so that they may be applied in the context acquisition. The presence of the credentials during context selection will allow the acquired context to be matched to the credentials, which will eliminate one of the reasons, that is, support for alternative credential types, why a context might need to support alternative (sufficient) modules.`AuthContext` objects could achieve transactional semantics by passing message copies to modules, or they could pass properties requiring transaction behavior of modules. There seems to be consensus within the EG that we should facilitate the use of single module contexts by empowering the config layer to select an appropriate context (containing a single module).

B.3. Access by Module to Other Layer Authentication Results

How does an authentication module gain access to authentication results established at a “lower” authentication layer? For example, acceptance of an identity assertion for subject S conveyed within the message at layer Y may be dependent on being able to authenticate at some lower layer (for example, SSL or perhaps message layer X), the entity (perhaps other than S) providing or making the identity assertion.

Resolution (Partial) - The `ServletRequest` object includes attributes that define the security properties of the transport connection on which a protected request arrived at the Servlet container. For the Servlet profile of this specification, we would expect the existing attribute mechanism to be employed. The general issue remains open, and may be resolved by the definition of one or more new Callback objects (for example, `getTransportProtection` and/or `getLayerSubject`) to be handled by the container or runtime.

B.4. How Are Target Credentials Acquired by Client Authentication Modules?

When a client must obtain a short-lived, service-targeted security token (such as a Kerberos Service Ticket), how are such tokens acquired, and how might the SPI defined by this specification be applied to secure any network interactions required for token acquisition? If the client authentication module is to perform token acquisition directly, it must be provided with sufficient information to acquire a suitable token. If token acquisition is done by the runtime (perhaps) in advance of the authentication module invocation (for example, during name context interpretation), the authentication module must be provided with a means to obtain a suitable token from the runtime.

Resolution - Extended the `AuthConfig` SPI to provide for the communication of properties such as service name at module initialization. Message exchanges required to acquire security tokens may be encapsulated in any of the `AuthConfig`, `AuthContext`, or `AuthModule` elements of the processing model. Also added `Subject` parameter to `getAuthContext` call such that the acquired credential can be passed back to the runtime.

B.5. How Does a Module Issue a Challenge?

How does an authentication module return a message to inform its network peer that it must do some

additional security processing as required by the network authentication mechanism being implemented by the module?

Resolution (Partial) - Defined `AuthStatus.SEND_CONTINUE` and related semantics. Improved the overview and message authentication chapters to describe multi-message exchanges.

B.6. Message Correlation for Multi-Message Dialogs

How are the messages that comprise a multi-message authentication dialog correlated, and where is any state relating to the authentication kept?

Resolution (Partial) - Based on the premise that message-specific knowledge is held within the authentication modules and that authentication modules are responsible for control of the dialog, it is assumed that authentication modules are responsible for tying together or correlating the messages that comprise the multi-message authentication dialog. Modules are expected to record and recover any necessary state, and may do so using the facilities of the containing runtime (for example, persisted sessions). It is also recognized that there are security mechanisms where message correlation is dependent on context external to the exchanged messages, such as the transport connection or session on which the messages were received, and that in such cases authentication modules will be dependent on correlation related facilities provided by the runtime. This draft of the specification does not standardize such facilities. The expert group discussed two alternatives for providing such facilities: 1) provide one or more callbacks to allow a module to set and get state associated with the current transport session; 2) define a module return value to be used to signal the runtime when it must record and reuse the same (stateful) `messageInfo` parameter when it calls the module to process the next message on the same transport session.

B.7. Compatibility With Load-Balancing Mechanisms

In a load-balanced environment, must the messages that comprise a multi-message authentication dialog (for example, the messages of a challenge-response dialog) be processed by the same authentication module instance, and if so how will that be accomplished?

Resolution (Partial) - Modules may choose to persist any state required to complete the dialog in a centralized repository. In other cases, such modules may choose to employ persisted session facilities of the runtime (for example, HttpSession) that have already been reconciled with load balancing. In other cases, it may be feasible to extend train the load-balancer to recognize security-mechanisms

specific correlation identifiers in messages.

B.8. Use of Generics and Typesafe Enums in Interface Definition

Should the SPI be modified to use new Java language features, specifically generics and typesafe enums, introduced in Java SE 5?

Resolution (Partial) - There is a requirement that the SPI be used in J2SE 1.4 environments, and an interest has been expressed in using the SPI in J2ME environments. As such, the specification does not employ these language features. There has been discussion regarding the use of these features in the SPI definition, while allowing for implementations matched to Java environments where these features are not available.

B.9. HttpServletResponse Buffering and Header Commit Semantics

The Servlet Specification defines buffering of the `HttpServletResponse` body such that filling the response body[23] (for the first time) can cause the response status code, HTTP response headers, and first buffer's worth of response body to be sent. Similarly, during processing of an `HttpServletRequest`, methods may be called on the corresponding `HttpServletResponse` (for example, `sendRedirect` or `flushbuffer`) that will cause the analogous content to be sent. In all such cases, the response has effectively been committed with respect to the status code, headers, and first response body buffer that will be returned to the client. After a response has committed, subsequent changes are not permitted to the status code or headers, and change to the response body is only permitted to the extent that more content may be appended. As such, when response buffering triggers a commit, for example during processing within the servlet, a call to `secureResponse`, following return from the servlet, will be unable to effect the response status code, the response headers, or any response body content that has already been sent (any or all of which may be necessary to secure the response).

Resolution - The Jakarta Servlet Specification defines the `HttpServletResponseWrapper` class, which can be used to extend the buffering capacity of the response, and thereby delay commit until the response is complete. When a `ServerAuthModule` requires that responses be buffered until they are explicitly completed, the module's `validateRequest` method should install a response wrapper when it returns `AuthStatus.SUCCESS`. Just prior to its return, the `secureResponse` method of the `ServerAuthModule` should write the completed message to the wrapped response and remove the wrapper.

B.10. Reporting New Issues

The maintenance project for this specification is located on the web at: http://github.com/eclipse-ee4j/authentication where you will find the technology issue tracker at: https://github.com/eclipse-ee4j/authentication/issues

[23] Some `HttpServletResponse` implementations extend the buffering methodology to the response headers, such that the status code and the first buffers worth of response headers are sent when when the header buffer is full. This does not, strictly speaking, cause the response to be committed, but instead creates a situation where attempts to change the status code, or to replace an existing header, would not be expected to succeed.

Appendix C: Revision History

C.1. Early Draft 1 (06/06/2005)

C.2. Significant Changes in Public Draft (08/15/2006)

C.2.1. Changes to API

1. The classes and interfaces of the API were divided into four packages, `message`, `config`, `callback`, and `module`.
2. The `MessageLayer` Interface was removed. Message layers are represented as a `String`.
3. The use of the `URI` type to identify applications (and other things) was replaced by `String`.
4. The `AuthParam` interface was replaced by the `MessageInfo` interface, and concrete message-specific implementations of the `AuthParam` interface were removed from the SPI.
5. The `disposeSubject` methods were renamed `cleanSubject`.
6. The `sharedMap` arguments were removed. `MessageInfo` is now used to convey such context.
7. The parameter names corresponding to subjects were modified to correspond to the service role of the corresponding party (i.e., client or server) as opposed to the message sending role.
8. The `ModuleProperties` interface was removed, and the responsibility for implementing transactional semantics was transferred to the authentication context (if it supports multiple sufficient alternatives).
9. The `PendingException` and `FailureException` classes were removed and a new return value type, `AuthStatus`, was defined to convey the related semantics. A general return value model was provided by the `AuthStatus` class.
10. The `AuthConfigProvider` interface was created to facilitate the integration of alternative module conversation systems, and facilities were added to the `AuthConfigFactory` to support the registration of `AuthConfigProviders`. The `RegistrationListener` interface we defined to support live replacement of configuration systems.
11. The authentication context configuration layer was formalized and methods to acquire authentication contexts (i.e, `getAuthContext`) were moved to the authentication context configuration layer. `Subject` arguments were added to the `getAuthContext` methods to support both the acquisition of credentials by the config system, and to allow the `Subject` and its content to factor in the context acquisition.
12. new callbacks were defined (i.e. `CallerPrincipalCallback` and `GroupPrincipalCallback`).

C.2.2. Changes to Processing Model

1. The `AuthStatus` return model was described and the message processing model of the `Overview` and `Message Authentication` chapters was evolved to describe the processing by runtimes of the

returned `AuthStatus` values, especially in the case of a multi-message authentication dialog.

C.2.3. Changes to Profiles

1. The Jakarta Servlet, SOAP, and Jakarta Messages profiles were added.

C.3. Changes in Proposed Final Draft 1

C.3.1. Changes to Preface

1. Changed Status and Audience to reflect transition to PFD.
2. Added paragraphs to describe relationship to JAAS

C.3.2. Changes to "Overview" Chapter

1. Changed Section 1.2.3 and Section 1.2.4 to reflect change in `AuthConfig` interface from `getOperation` to `getAuthContextID`.
2. Added definition of “message processing runtime” to Section 1.3

C.3.3. Changes to "Message Authentication" Chapter

1. Changed sections Section 2.1, Section 2.1.2.2, Section 2.1.3, Section 2.1.4 to reflect change in `AuthConfig` interface from `getOperation` to `getAuthContextID`.
2. To Section 2.1.1.1, added a requirement that runtimes support the granting to applications and administration utilities of the permissions required to employ the configuration interfaces of the SPI.
3. In subsection “at point (1) in the message processing model:” of Section 2.1.5.2, clarified `clientSubject` requirements, and indicated that a non-null `clientSubject` must not be read-only.
4. In subsection “at point (4) in the message processing model:” of Section 2.1.5.2, clarified `serviceSubject` requirements, and indicated that a non-null serviceSubject must not be read-only.
5. Added “Fig 2.1: State Diagram of Client Message Processing Runtime”
6. In subsection “at point (2) in the message processing model:” of Section 2.1.5.2, clarified `serviceSubject` requirements, and indicated that a non-null serviceSubject must not be read-only.
7. In subsection “at point (3) in the message processing model:” of Section 2.1.5.2, clarified that the call to `secureResponse` should be made independent of the outcome of the application request processing.
8. Added “Fig 2.2: State Diagram of Server Message Processing Runtime”.

C.3.4. Changes to “Servlet Container Profile” Chapter

1. Added last sentence to introductory paragraph to clarify what is required to be a compatible

implementation of the profile.

2. In Section 3.2, extended identifier format to include the logical hostname along with he context path.
3. In Section 3.5, added requirement that the handler argument (passed by the runtime) must not be null.
4. Changed section Section 3.8 to reflect change in `AuthConfig` interface from `getOperation` to `getAuthContextID`.
5. Changed Section 3.8.1, to remove requirements for a specific identifier format.
6. Changed Section 3.8.3,to require that the runtime set the `PolicyContext` in the module initialization properties passed to `getAuthContext` call.
7. In Section 3.8.4, removed requirements relating to `responsePolicy`. Also moved responsibility for determining when (client) authentication is required from the `AuthConfig` subsystem to the message processing runtime.
8. In Section 3.9, clarified the points within the servlet processing model that corresponding to points 2 and 3 of the message module. Added explicit statement to ensure that `validateRequest` is called on all requests including requests to a login form. Moved the comment regarding “delegation of session management” to a footnote. Changed the processing when there is an authorization failure to require that secureResponse be called. Changed the prohibition on calling `secureResponse` when the application throws an exception to a recommendation. Added last sentence to require the use of the principal established using the `CallerPrincipalCallback` where identity propagation is configured.
9. Changed Section 3.9.1, to conditionally require the inclusion of a property within the `MessageInfo` map when client authentication is required. Also placed new requirement on the authentication context configuration system that is use this value to establish the requestPolicy.
10. Added initial sentence to Section 3.9.3, to reiterate that `validateRequest` be called on every request that satisfies the applicable connection requirements.
11. In Section 3.9.3.1, moved responsibility for coordinating disparate uses of the `CallerPrincipalCallback` to the context. Relaxed prohibition on returning `SEND_CONTINUE` from modules initialized with an optional `requestPolicy` by allowing modules to continue a multi-message authentication dialog as long as it was initiated by the client. Added requirement that modules initialized with an optional `requestPolicy`, use the `CallerPrincipalCallback` to established an unauthenticated caller identity (if they return `AuthStatus.SUCCESS` without having satisfied the `TargetPolicy`).
12. In Section 3.9.3.2, removed requirement that the module set the HTTP 200 (OK) status code.
13. In Section 3.9.3.3, removed requirements dependent on responsePolicy.
14. Replaced section “Dealing with Servlet Commit Semantics” with a new Section 3.9.3.5.

C.3.5. Changes to “SOAP Profile” Chapter

1. Added last sentence to introductory paragraph to clarify what is required to be a compatible implementation of the profile.
2. Changed Section 4.2, to refer to subsections within the sub-profiles where the corresponding identifiers are defined.
3. In Section 4.5, added requirement that the handler argument (passed by the runtime) must not be null.
4. In Section 4.7, added clarification of what it means when `getAuthContext` returns a null value, and how the value returned by `getAuthContext` impacts support for a session oriented authentication mechanism.
5. Changed Section 4.7.1, to remove requirements for a specific identifier format.
6. Added new Section 4.8.1, to describe the identifier format as the concatenation of a client scope identifier and a client reference to the service. For client scope identifiers, recommended the use of application identifiers where they are available and suggested the use of the archive URI where application identifiers are not available. Required that the service-ref name be used (if available) for the client reference to the service. Otherwise the service URL is to be used. Included examples, and added a last paragraph indicating that registration would require an ability to predict the client scope identifier and client service reference associated by the runtime with a client invocation.
7. Removed requirements from Section 4.8.4, that were already stated in Section 4.7.
8. In Section 4.8.5, to account for one-way application message exchange patterns, limited the circumstances under which a runtime may proceed to point (4) in the message processing model.
9. In Section 4.8.5.1, changed the description of the value of the `javax.xml.ws.wsdl.service` property such that it must be a `QName` containing the service name. Removed statement of relationship of value to client authentication context identifier.
10. In Section 4.8.5.3, corrected cut an paste errors (i.e., s/response/request/). Relaxed prohibition on returning `SEND_CONTINUE` from `secureRequest` on modules initialized with an optional requestPolicy. Added requirement that a module must return `AuthStatus.SEND_SUCCESS` (from `secureRequest`) if it was initialized with a null requestPolicy.
11. In [a590], on modules initialized with and optional `responsePolicy`, relaxed prohibition on returning `SEND_CONTINUE` from `validateResponse` and clarified the handling of `AuthException` and the various `AuthStatus` return values.
12. Added new Section 4.9.1, to describe the identifier format as the concatenation of the logical hostname of the virtual server, and the service endpoint URI. Also included an example.
13. Removed requirements from Section 4.9.4 that were already stated in Section 4.7.
14. Changed Section 4.9.4.1 to require that `PolicyContext` be set in the module initialization properties (passed to `getAuthContext` call) if the server runtime is a Jakarta Authorization compatible container.

15. In Section 4.9.4.2 removed paragraphs defining when message protection is required by an Jakarta Enterprise Beans web service container. Added requirement for a specific `TargetPolicy` within `requestPolicy` when the `CallerPrincipalCallback` is to be used by the authentication module(s) of the context. Added a requirement that the `requestPolicy` must be mandatory and must include a specific `TargetPolicy` when all the operations of an endpoint require client authentication. Added recommended return values for `isMandatory`, when not all of the operations of an endpoint require client authentication.
16. In Section 4.9.5, to account for one-way application message exchange patterns, limited the circumstances under which a runtime may proceed to point (3) in the message processing model. Moved the comment regarding “delegation of session management” to a footnote. Changed the processing to require that `secureResponse` be called when there is an authorization failure. Changed the prohibition on calling `secureResponse` when the application throws an exception to a requirement that `secureResponse` be called. Added last sentence to require the use of the principal established using the `CallerPrincipalCallback` where identity propagation is configured.
17. In [a642] removed the requirement that the service name property be set in the `MessageInfo` Map.
18. In [a648], moved responsibility for coordinating disparate uses of the `CallerPrincipalCallback` to the context. Relaxed prohibition on returning `SEND_CONTINUE` from modules initialized with an optional `requestPolicy` by allowing modules to continue a multi-message authentication dialog as long as it was initiated by the client. Added requirement that modules initialized with an optional requestPolicy, containing a prescribed `TargetPolicy`, use the `CallerPrincipalCallback` to established an unauthenticated caller identity (if they return `AuthStatus.SUCCESS` without having satisfied the `TargetPolicy`).

in Section 4.9.5.4, corrected the required return value when `responsePolicy == null` to be `AuthStatus.SEND_SUCCESS`.

C.3.6. Changes to JMS Profile Chapter

1. Renamed chapter to "Future Profiles”.
2. Changed chapter to be strictly informative; serving to capture suggestions for additional profiles.
3. Added Section 5.2.

C.3.7. Changes to Appendix B, Issues

1. Added new issue, Section B.9, with resolution which was factored into the Servlet Profile (see Section 3.9.3.5).

C.3.8. Changes to API

1. In javax.security.auth.message.MessagePolicy, changed name of method “isManadatory” to “isMandatory”.
2. In javax.security.auth.message.config.AuthConfig, changed the name of method “getOperation” to “getAuthContextID” and changed the method definition to indicate that it returns the

authentication context identifier corresponding to the request and response objects in the messageInfo argument.

3. In javax.security.auth.message.config.AuthConfigFactory, changed description of the typical sequence of calls to reflect change of “getOperation” to “getAuthContextID”. Also changed description to differentiate registration and self-registration. Added comment to definition of the setFactory method to make it clear that listeners are NOT notified of the change to the registered factory. Added a second form of registerConfigProvider that takes an AuthConfigProvider object (in lieu of an implementation class and properties Map) and that performs an in-memory registration as apposed to a persisted registration. Added support for null registrations. Added the isPersistent method to the AuthConfigFactory.RegistrationContext interface.
4. In javax.security.auth.message.config.AuthConfigProvider, changed description of the typical sequence of calls to reflect change of “getOperation” to “getAuthContextID”. Changed requirement for a “public one argument constructor” to a “public two argument constructor”, where the 2nd argument may be used to pass an AuthConfigFactory to the AuthConfigProvider to allow the provider to self-register with the factory.
5. In javax.security.auth.message.config.ClientAuthConfig, changed method and parameter descriptions to reflect change of “getOperation” to “getAuthContextID”.
6. In javax.security.auth.message.config.ServerAuthConfig, changed method and parameter descriptions to reflect change of “getOperation” to “getAuthContextID”.
7. In javax.security.auth.message.callback.PasswordValidationCallback, added a Subject parameter to the constructor, and a getSubject method to make the Subject available to the CallbackHandler. Also added a sentence describing the expected use of the PasswordValidationCallback.
8. In javax.security.auth.message.callback.PrivateKeyCallback, added PrivateKeyCallback.DigestRequest so that private keys may be requested by certificate digest (or thumbprint). Added a sentence describing the expected use of the PrivateKeyCallback.
9. In javax.security.auth.message.callback.SecretKeyCallback, improved description of the expected use of the SecretKeyCallback.

C.4. Changes in Proposed Final Draft 2

C.4.1. Changes to License

1. Revised date to May 5, 2007

C.4.2. Changes to Servlet Container Profile

1. In Section 3.9, added reference to new section, Section 3.9.4 to describe requirements for setting the authentication results.
2. Added Section 3.9.4 to capture requirements for setting the user principal, remote user, and authentication type on the `HttpServletRequest`.

C.4.3. Changes to SOAP Profile

1. Corrected reference (chapter number) to “Message Authentication” chapter appearing in the chapter introduction.
2. Corrected ambiguity in Section 4.3, to make it clear that the profile does not require that `MessageInfo` contain only non-null request and response objects.

C.4.4. Changes to LoginModule Bridge Profile

1. In Section 6.1, revised the method by which a `ServerAuthModule` chooses the entry name passed to the `LoginContext` constructor. This change allows a single module implementation to be configured to use different entry names, and thus different login modules.
2. In Section 6.3, added requirement that `GroupPrincipalCallback` be supported when `LoginContext` is constructed with `Subject`.
3. In Section 6.4, added requirement that `ServerAuthModule` employ `CallerPrincipalCallback` using same value as that available to `LoginModule` via `NameCallback`.

C.5. Changes in Final Release

C.5.1. Changes to title page

1. Corrected JCP version to 2.6

C.5.2. Changes to Preface

1. Changed Status and Audience to reflect transition to Final Release

C.6. Changes in Maintenance Release A

C.6.1. Changes Effecting Entire Document

Changed document Identifier to Maintenance Release A. Version identifier remains unchanged at 1.0.

C.6.2. Changes to “Message Authentication” Chapter

Clarified definition of baseline compatibility requirements to more explicitly convey that the API is intended to have more general applicability than the specific contexts of its use defined within the specification.

C.6.3. Changes to API

In `javax.security.auth.message.callback.CallerPrincipalCallback`, modified callback definition to allow for principal mapping to occur during the handling of the callback by the `CallbackHandler`.

C.7. Changes in Maintenance Release B

C.7.1. Changes Effecting Entire Document

1. Changed document Identifier to Maintenance Release B, and Version identifier changed to 1.1.
2. Updated JCP version to 2.7
3. Updated the license
4. Replaced Sun logo with Oracle logo
5. Removed paragraph tags from PDF bookmarks

C.7.2. Changes to Preface

1. Changed Status to Maintenance Release B version 1.1
2. Added Will Hopkins, Tim Quinn, Arjan Tijms, and Yi Wang to the list of contributors

C.7.3. Changes to Servlet Container Profile

1. In Section 3.2, described use of `ServletContext.getVirtualServerName` in application context identifier.
2. In Section 3.9 and Section 3.9.3, clarified that `validateRequest` must be called on every request for which the Servlet security model applies. Also included footnote whose text describes that the security model does not apply to forwards and includes.
3. In Section 3.9.3.1, added clarification to description of processing for `SEND_CONTINUE`, especially to allow for forwards to a login page within an authentication module.
4. In Section 3.9.3.1, clarified description of processing for `SEND_FAILURE` to indicate that this return status is returned when the validation failed and the client should not continue or retry the request.
5. Added footnote on header of Section 3.9.3.2 to clarify that “after the service invocation” effectively means after the call to secureResponse, so as to remain distinct from the case where a call to authenticate from within the application results in a call to validateRequest during the service invocation.
6. Added Section 3.9.3.4, to make it clear that authentication modules must be able to use a `RequestDispatcher` to forward to a login page (for example).
7. In Section 3.9.4, amended description to make this section suitable for describing both the case where `validateRequest` is called prior to a request, and the case where `validateRequest` is (presumably) being called during the processing of the request
8. In Section 3.9.4, added Table 3-6 to define the name of the session registration callback property. Also added description of the processing of the property.
9. Added Section 3.10 to define the use of the Jakarta Authentication SPI under

`HttpServletRequest.authenticate`, `login`, and `logout`.

C.7.4. Changes to Appendix B, Issues

1. Added Section B.10 with links to java.net project and JIRA issue tracker.

C.7.5. Changes to API

1. In abstract `AuthConfigFactory` class, made public the static permissions that are used to protect the static `getFactory` and `setFactory` methods, and improved documentation so users of the SPI can know which permissions are used. Also added an additional public `providerRegistrationSecurityPermission` and required that it be used by factory implementations to protect methods like `registerConfigProvider`. Removed incorrect assertion from javadoc of `getFactory`, both forms of `registerConfigProvider`, and `refresh`, that checked `AuthException` could be thrown (by these methods). Changed the javadoc of these four methods to indicate that the conditions for which they were expected to throw an `AuthException` should instead be handled within their existing declarations of throwing an (unchecked) `SecurityException`. Regenerated (mif) javadocs (embedded in spec) from html javadocs, which corrected definition for `layer` and `appContext`parameters of `getConfigProvider(java.lang.String layer, java.lang.String appContext, RegistrationListener listener)`.
2. In `AuthConfig`, and `AuthConfigProvider` interfaces, removed incorrect assertion from javadoc of refresh method that checked `AuthException` could be thrown, and changed javadoc to indicate that the conditions for which `refresh` was expected to throw an `AuthException` should instead be handled within its existing declaration of throwing an (unchecked) `SecurityException`.

C.8. Changes in Jakarta Authentication 3.0

C.8.1. Changes to Servlet Container Profile

1. Added the `jakarta.servlet.http.isAuthenticationRequest` key, so modules can distinguish between being called at the very start of a request and in the middle of it following a call to `HttpServletRequest.authenticate`. This could be needed to determine if certain contexts are effectively active (such as the Faces context in Jakarta Faces).
2. Added requirements regarding the interaction of a `ServerAuthModule` with other specifications. This allows such `ServerAuthModule` to programmatically obtain references to data sources, EJB beans, CDI beans, etc.