Java Authentication SPI
for
Containers

Please send comments to: jsr-196-comments@jcp.org

JSR-196

Java Community Process℠ (JCP℠) 2.6

Final Release

Version 1.0

Specification Lead: Ron Monzillo
Sun Microsystems, Inc.

Specification: JSR 196 JavaTM Authentication Service Provider Interface for Containers ("Specification")
Version: 1.0
Status: Final Release
Release: 9 July 2007

LIMITED LICENSE GRANTS

1. License for Evaluation Purposes. Sun hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under Sun's applicable intellectual property rights to view, download, use and reproduce the Specification only for the purpose of internal evaluation. This includes (i) developing applications intended to run on an implementation of the Specification, provided that such applications do not themselves implement any portion(s) of the Specification, and (ii) discussing the Specification with any third party; and (iii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Specification.

2. License for the Distribution of Compliant Implementations. Sun also grants you a perpetual, non-exclusive, non-transferable, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights or, subject to the provisions of subsection 4 below, patent rights it may have covering the Specification to create and/or distribute an Independent Implementation of the Specification that: (a) fully implements the Specification including all its required interfaces and functionality; (b) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented; and (c) passes the Technology Compatibility Kit (including satisfying the requirements of the applicable TCK Users Guide) for such Specification ("Compliant Implementation"). In addition, the foregoing license is expressly conditioned on your not acting outside its scope. No license is granted hereunder for any other purpose (including, for example, modifying the Specification, other than to the extent of your fair use rights, or distributing the Specification to third parties). Also, no right, title, or interest in or to any trademarks, service marks, or trade names of Sun or Sun's licensors is granted hereunder. Java, and Java-related logos, marks and names are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

3. Pass-through Conditions. You need not include limitations (a)-(c) from the previous paragraph or any other particular "pass through" requirements in any license You grant concerning the use of your Independent Implementation or products derived from it. However, except with respect to Independent Implementations (and products derived from them) that satisfy limitations (a)-(c) from the previous paragraph, You may neither: (a) grant or otherwise pass through to your licensees any licenses under Sun's applicable intellectual property rights; nor (b) authorize your licensees to make any claims concerning their implementation's compliance with the Specification in question.

4. Reciprocity Concerning Patent Licenses.

a. With respect to any patent claims covered by the license granted under subparagraph 2 above that would be infringed by all technically feasible implementations of the Specification, such license is conditioned upon your offering on fair, reasonable and non-discriminatory terms, to any party seeking it from You, a perpetual, non-exclusive, non-transferable, worldwide license under Your patent rights which are or would be infringed by all technically feasible implementations of the Specification to develop, distribute and use a Compliant Implementation.

b With respect to any patent claims owned by Sun and covered by the license granted under subparagraph 2, whether or not their infringement can be avoided in a technically feasible manner when implementing the Specification, such license shall terminate with respect to such claims if You initiate a claim against Sun that it has, in the course of performing its responsibilities as the Specification Lead, induced any other entity to infringe Your patent rights.

c Also with respect to any patent claims owned by Sun and covered by the license granted under subparagraph 2 above, where the infringement of such claims can be avoided in a technically feasible manner when implementing the Specification such license, with respect to such claims, shall terminate if You initiate a claim against Sun that its making, having made, using, offering to sell, selling or importing a Compliant Implementation infringes Your patent rights.

5. Definitions. For the purposes of this Agreement: "Independent Implementation" shall mean an implementation of the Specification that neither derives from any of Sun's source code or binary code materials nor, except with an appropriate and separate license from Sun, includes any of Sun's source code or binary code materials; "Licensor Name Space" shall mean the public class or interface declarations whose names begin with "java", "javax", "com.sun" or their equivalents in any subsequent naming convention adopted by Sun through the Java Community Process, or any recognized successors or replacements thereof; and "Technology Compatibility Kit" or "TCK" shall mean the test suite and accompanying TCK User's Guide provided by Sun which corresponds to the Specification and that was available either (i)

from Sun's 120 days before the first release of Your Independent Implementation that allows its use for commercial purposes, or (ii) more recently than 120 days from such release but against which You elect to test Your implementation of the Specification.

This Agreement will terminate immediately without notice from Sun if you breach the Agreement or act outside the scope of the licenses granted above.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS". SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT (INCLUDING AS A CONSEQUENCE OF ANY PRACTICE OR IMPLEMENTATION OF THE SPECIFICATION), OR THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE. This document does not represent any commitment to release or implement any portion of the Specification in any product. In addition, the Specification could include technical inaccuracies or typographical errors.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED IN ANY WAY TO YOUR HAVING, IMPELEMENTING OR OTHERWISE USING USING THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
You will indemnify, hold harmless, and defend Sun and its licensors from any claims arising or resulting from: (i) your use of the Specification; (ii) the use or distribution of your Java application, applet and/or implementation; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

U.S. Government: If this Specification is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

If you provide Sun with any comments or suggestions concerning the Specification ("Feedback"), you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose.

GENERAL TERMS

Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

Rev. April, 2006

Preface

Status of Document

This document is the Final Release of the Java™ Authentication Service Provider Interface for Containers Version 1.0 specification

Audience

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

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, JMS, and HttpServlet) and message processing runtimes (including J2EE and Java 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.

Keywords

The keywords “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC2119 [KEYWORDS].

Acknowledgements

This draft of the specification incorporates the contributions of the JSR 196 Expert Group which included the following members:

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
Misun Yoon	Tmax Soft, Inc.

The following people also contributed to this specification:

Venu Gopal	Sun Microsystems
V. B. Kumar Jayanti	Sun Microsystems
Manveen Kaur	Sun Microsystems
Raja Perumal	Sun Microsystems
Gursharan Singh	Sun Microsystems
Anil Tappetla	Sun Microsystems
Kai Xu	Sun Microsystems

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.

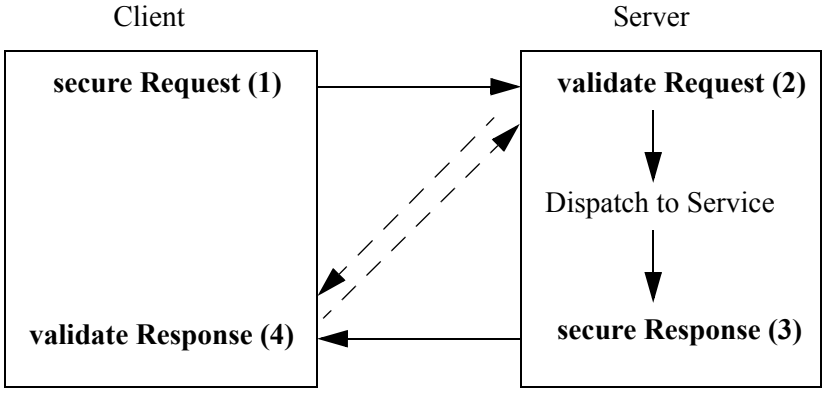

Figure 1.1 Message Processing Model[1]

[1] 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.

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, javax.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 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
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 Section 2.1.5.2, “What the Runtime Must Do,” on page 19. 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);
```

[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.

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);
```

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

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 Java Enterprise Edition (Java EE version 1.4 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 1.4.
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 J2EE and Java 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.

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 requirements defined in this chapter serve as the baseline on which profiles establish the requirements pertaining to the use of the specification in a particular message processing context.

2.1 Authentication

As defined in Section 1.2, "Typical Runtime Use Model," 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 A*uthConfigProvider*. 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 on page 18. 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 c*lientSubject* 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.

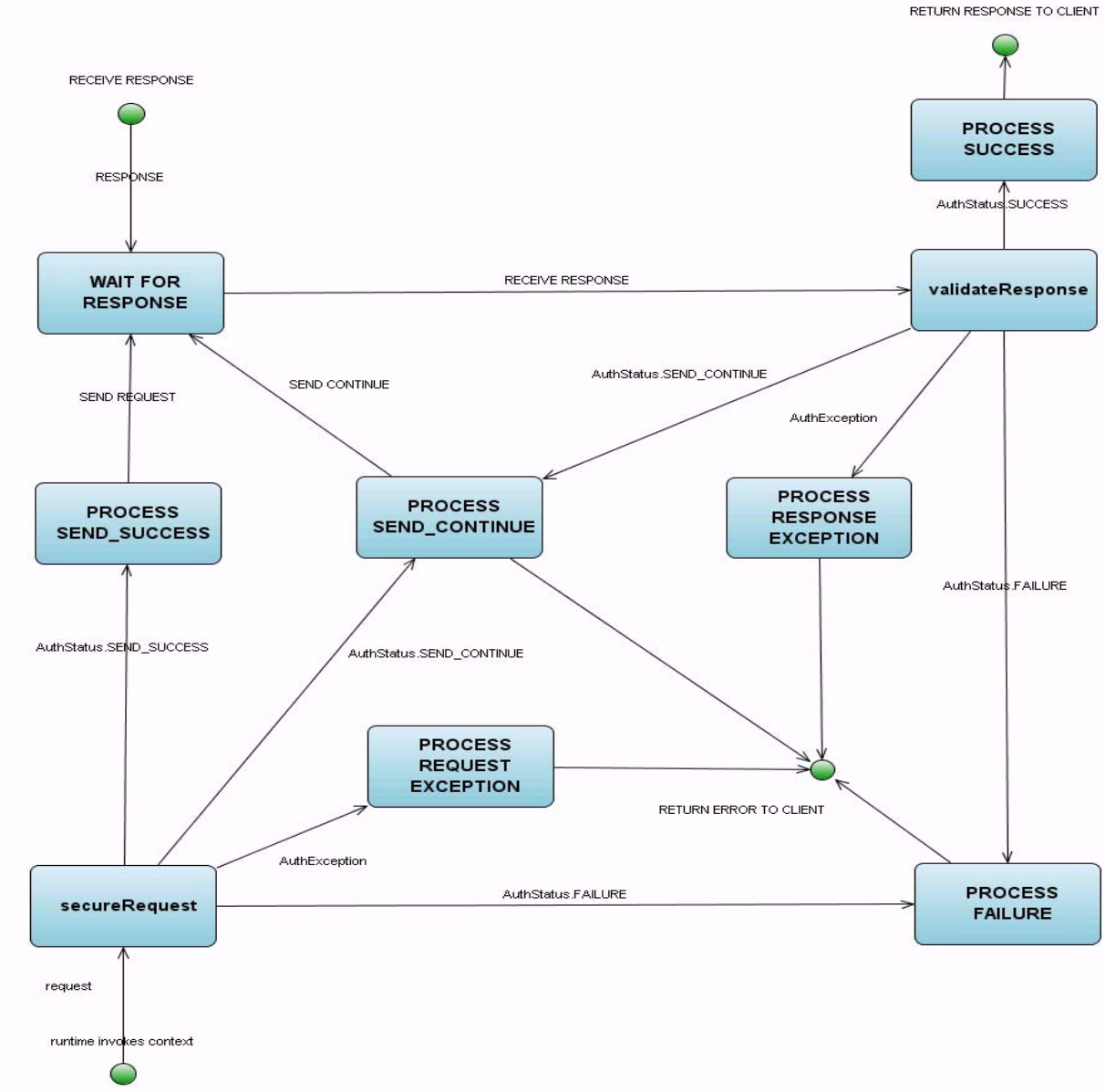

Figure 2.1 State Diagram of Client Message Processing Runtime

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 m*essageInfo* 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 service*Subject* 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[1].

    - 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[2] 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

---

1. 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.
2. validateRequest is called to process all received messages, including security mechanism-specific messages sent by clients in response to service response messages.

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.

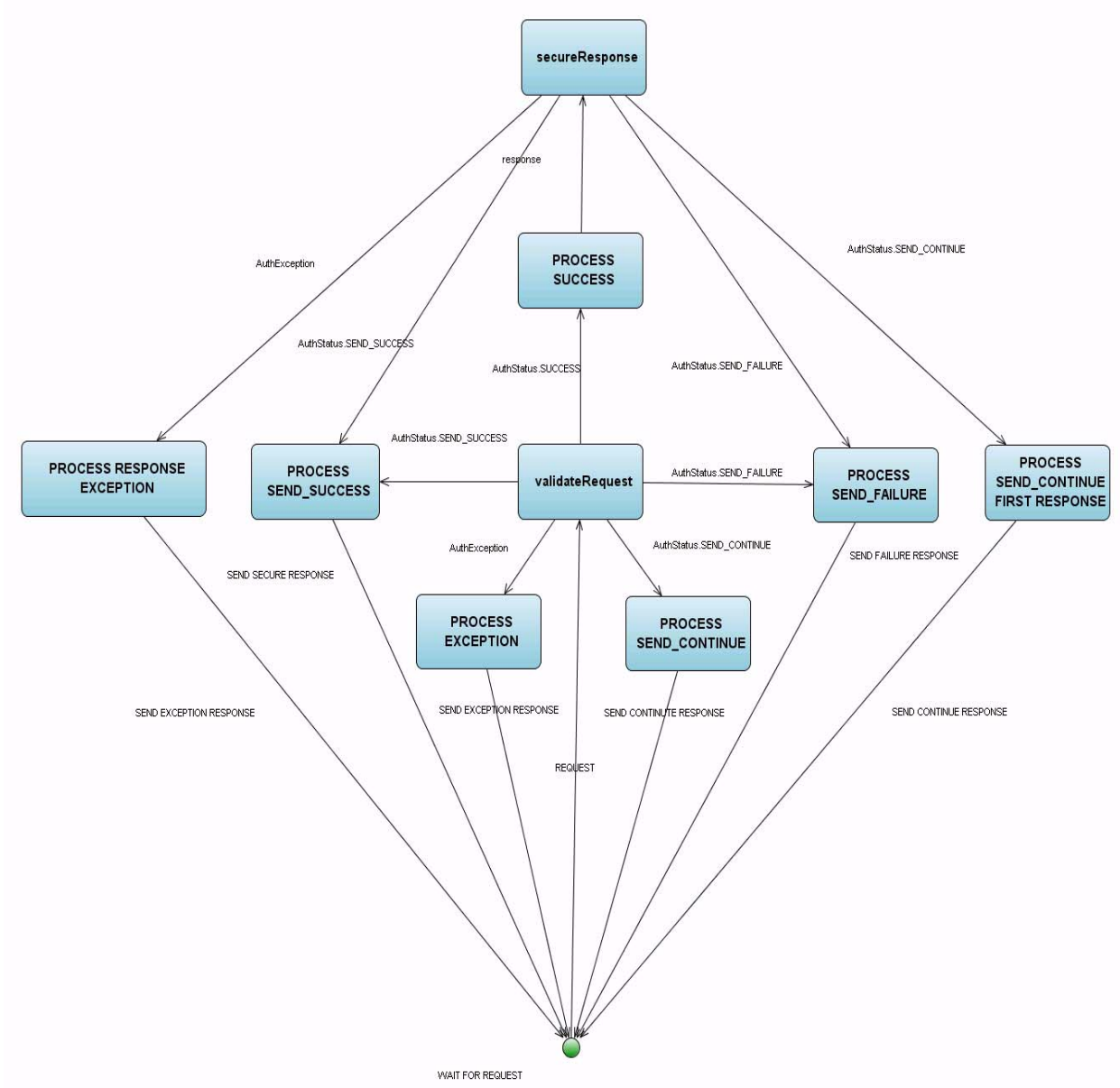

Figure 2.2 State Diagram of Server Message Processing Runtime

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.

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.

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 *javax.servlet.http.HttpServletRequest.class* and *javax.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 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, “Message Layer Identifier,” and Section 3.2, “Application Context Identifier,” 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, "CallbackHandler Requirements."

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.

3.7 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 on page 18.

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 3.7.1, "Authentication Context Identifiers."

For all values of the *authContextID* argument that satisfy the requirements of Section 3.7.1, the call to *getAuthContext* must return a non-null authentication context.

3.7.1 Authentication Context Identifiers

This profile does NOT impose any profile specific requirements on authentication context identifiers. As defined in Section 2.1.3, "Acquire AuthContext Identifier," on page 17, 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*.

3.7.2 *getAuthContext* Subject

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

3.7.3 Module Initialization Properties

If the runtime is a JSR 115 compatible Servlet container, the *properties* argument passed in all calls to *getAuthContext* must contain the key-value pair shown in the following table.

TABLE 3-1 JSR 115 Compatible Module Initialization Properties

key	*value*
javax.security.jacc.PolicyContext	The PolicyContext identifier value that the container must set to satisfy the JSR 115 authorization requirements as described in "Setting the Policy Context" within the JSR 115 specification

When the runtime is not a JSR 115 compatible Servlet container, the *properties* argument used in all calls to getAuthContext must not include a *javax.security.jacc.PolicyContext* key-value pair, and a null value may be passed for the *properties* argument.

3.7.4 MessagePolicy Requirements

Each *ServerAuthCon*text 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 i*sMandatory* 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 3.8.1, "MessageInfo Requirements," on page 29.

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

3.8 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[1] that apply to the

[1] In a JSR 115 environment, connection requirements are tested by checking a WebUserDataPermission constructed with the HttpServletRequest. In a non-JSR 115 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.

request and before the runtime enforces the authorization[2] 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*[3]. validateRequest must be called for all requests, including requests to 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 3.8.4, "Setting the Authentication Results on the HttpServletRequest". 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 JSR 115 compatibility, of the calling runtime. In a JSR 115 compatible runtime, the identity must be comprised of exactly the Principal objects of the *clientSubject*. In a non-JSR 115 compatible 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 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 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.

3.8.1 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).

---

2. In a JSR 115 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-JSR 115 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.
3. These unconditional calls to *validateRequest* are necessary to allow for delegation of servlet authentication session management to authentication contexts and their contained authentication modules.

3.8.1.1 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 3-2 MessageInfo Map Properties

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

javax.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 Servlet auth-constraint[4], or in a JSR 115 compatible runtime, if the corresponding *WebResourcePermission* is NOT granted[5] to an unauthenticated caller. In a JSR 115 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*.

3.8.2 Subject Requirements

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

3.8.3 *ServerAuth* Processing

As described in *Section 3.8, "Message Processing Requirements," on page 28*, the profile requires that *validateRequest* be called on every request that satisfies the corresponding connection requirements. 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*.

---

4. 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 excluded resources prior to proceeding to point (2) in the message processing model (that is, prior to the authentication processing).
5. JSR 115 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).

3.8.3.1 *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 the request.
- return AuthStatus.SEND_FAILURE – If the request validation failed, and when the client should not retry 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 retry 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[6] 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 that AuthStatus.SUCCESS, the runtime must NOT proceed to the service invocation. The runtime must process an AuthException as described above for a

6. 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).

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 request with a mandatory *requestPolicy.*

3.8.3.2 *validateRequest* After Service Invocation

When *validateRequest* is called after the service invocation, 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.

3.8.3.3 *secureResponse* Processing

The return value and AuthException semantics of *secureResponse* are as defined in Section 3.8.3.2, “validateRequest After Service Invocation.” This profile places no requirements on authentication modules with respect to interpreting *responsePolicy* values.

3.8.3.4 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.

3.8.4 Setting the Authentication Results on the HttpServletRequest

The message processing runtime must fulfill the requirements defined in this section when, at point (2) in the messaging model, *validateRequest* returns *AuthStatus.SUCCESS*. In this case, the runtime must modify the *HttpServletRequest* 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*[7]. The runtime 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 runtime must consult the *Map* of the *MessageInfo* object used in the call to *validateRequest* to determine if it contains an entry for the key identified in the following table. If the *Map* contains an entry for the key, the runtime must obtain (from the *Map*) the value corresponding to the key and establish it as the *getAuthTyp*e 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 runtime must establish the value from the *auth-method* as the value returned by *getAuthType*. If the *Map* does not contain an entry for the key, and the deployment descriptor does not define an *auth-method*, the runtime must establish a non-null value of its choice as the value returned by *getAuthType*.

TABLE 3-3 Authentication Type Property Returned in MessageInfo Map

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

[7] Except when getUserPrincipal returns null; in which case the value returned by getRemoteUser must be null

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 Section 4.8.1, "Client-Side Application Context Identifier," on page 37.

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, "Server-Side Application Context Identifier," on page 42.

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 javax.xml.soap.SOAPMessage.*

4.4 Module Requirements

The *getSupportedMessageTypes* method of all authentication modules integrated for use with this profile must include *javax.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, "Message Layer Identifier" and Section 4.2, "Application Context Identifier" 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 on page 18.

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, "Authentication Context Identifiers."

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, "Acquire AuthContext Identifier," on page 17, 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 JAXWS 2.0 or JAXWS 2.1 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, “CallbackHandler Requirements.”

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, “CallbackHandler Requirements,” 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 *ClientAuthCon*text 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 4-1 Client MessageInfo Map Properties

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

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[1] 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[2]

When *validateResponse* is called before the service invocation, 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.

---

1. 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).
2. Occurs when the module is challenged by the server during *secureRequest* processing.

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[3] 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, “CallbackHandler Requirements.”

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, “CallbackHandler Requirements,” for a server runtime.

---

[3] 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.

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 JSR 115 compatible EJB or Servlet endpoint container, the *properties* argument passed in all calls to *getAuthContext* must contain the key-value pair shown in the following table.

TABLE 4-2 JSR 115 Compatible Module Initialization Properties

key	*value*
javax.security.jacc.PolicyContext	The PolicyContext identifier value that the container must set to satisfy the JSR 115 authorization requirements as described in "Setting the Policy Context" within the JSR 115 specification

When the runtime is not a JSR 115 compatible endpoint container, the *properties* argument used in all calls to getAuthContext must not include a *javax.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 i*sMandatory* 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 i*sMandatory* 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[4] 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 JSR 115 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.

---

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

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[5] 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 [6]

When *validateRequest* is called after the service invocation, 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

5. 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).
6. Occurs when the module is challenged by the client during *secureResponse* processing.

value and AuthException semantics of *secureResponse* are as defined in "validateRequest After Service Invocation."

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 *javax.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 Message-Consumer.

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.

5.2.1 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, “Standard Callbacks."

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 *CallbackHandle*r. 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*[1] 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*.

[1.] The *CallerPrincipalCallback* may be constucted 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].

Java™ 2 Platform, Enterprise Edition Specification Version 1.4 [J2EE Specification], available at: http://java.sun.com/j2ee/docs.html.

Java™ Platform, Enterprise Edition 5 Specification, [Java EE 5 Specification], available at: http://java.sun.com/javaee

Java™ 2 Platform, Standard Edition, Version 5.0 API Specification, [Java SE 5 Specification], available at: http://java.sun.com/javase

Enterprise JavaBeans™ Specification, Version 3.0 [EJB Specification], available at: http://java.sun.com/products/ejb

Java™ Servlet Specification, Version 2.5 [Servlet Specification], available at: http://java.sun.com/products/servlet

Java™ Authentication and Authorization Service (JAAS) 1.0 [JAAS Specification], available at: http://java.sun.com/products/jaas

Java™ API for XML-Based Web Services (JAX-WS) 2.0 [JAXWS-2.0 Specification], available at: https://jax-ws.dev.java.net/ or http://jcp.org/aboutJava/communityprocess/final/jsr224

Java™ API for XML-Based Web Services (JAX-WS) 2.1 [JAXWS-2.1 Specification], available at: https://jax-ws.dev.java.net/ or http://jcp.org/aboutJava/communityprocess/final/jsr224

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

Java™ Message Service Specification Version 1.1 [JMS Specification], available at: http://java.sun.com/products/jms/docs.html

*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: http://www.omg.org/docs/formal/04-03-19.pdf

APPENDIX B

Issues

The following sections document the more noteworthy issues that have been discussed by the Expert Group (EG). 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

J2EE 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[1] (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 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.

1. 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 (dated 06/06/2005)

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

C.2.1 Changes to API

- The classes and interfaces of the API were divided into four packages, *message, config, callback,* and *module.*
- The *MessageLayer* Interface was removed. Message layers are represented as a *String*.
- The use of the *URI* type to identify applications (and other things) was replaced by *String*,
- The *AuthParam* Interface was replaced by the *MessageInfo* Interface, and concrete message-specific implementations of the *AuthParam* interface were removed from the SPI.
- The *disposeSubject* methods were renamed *cleanSubject*.
- The *sharedMap* arguments were removed. *MessageInfo* is now used to convey such context.
- The parameter names corresponding to subjects were modified to correspond to the service role of the corresponding party (i.e., client or server) as 0pposed to the message sending role.
- The *ModuleProperties* Interface was removed, and the responsibility for implementing transactional semantics was transferred to the authentication context (if it supports multiple sufficient alternatives).
- 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.
- The *AuthConfigProvider* interface was created to facilitate the integration of alternative module conversation systems, and facilities were added to the A*uthConfigFactory* to support the registration of *AuthConfigProviders*. The *RegistrationListener* interface we defined to support live replacement of configuration systems.
- 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.
- new callbacks were defined (i.e. *CallerPrincipalCallback and GroupPrincipalCallback*).

C.2.2 Changes to Processing Model

- 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

- The Servlet, SOAP, and JMS profiles were added.

C.3 Changes in Proposed Final Draft 1

C.3.1 Changes to Preface

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

C.3.2 Changes to "Overview" Chapter

- Changed sections 1.2.3 and 1.2.4 to reflect change in *AuthConfig* interface from *getOperation* to *getAuthContextID*.
- Added definition of "message processing runtime" to "Terminology" section.

C.3.3 Changes to "Message Authentication" Chapter

- Changed sections 2.1, 2.1.2.2, 2.1.3, 2.1.4o reflect change in *AuthConfig* interface from *getOperation* to *getAuthContextID*.
- to Section 2.1.1.1, "What the Runtime Must Do", 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.
- In subsection "***at point (1) in the message processing model:"*** of Section 2.1.5.2, "What the Runtime Must Do", clarified *clientSubject* requirements, and indicated that a non-null *clientSubject* must not be read-only.
- In subsection "***at point (4) in the message processing model:"*** of Section 2.1.5.2, "What the Runtime Must Do", clarified *serviceSubject* requirements, and indicated that a non-null *serviceSubject* must not be read-only.
- added "Fig 2.1: State Diagram of Client Message Processing Runtime"
- In subsection "***at point (2) in the message processing model:"*** of Section 2.1.5.2, "What the Runtime Must Do", clarified serviceSubject requirements, and indicated that a non-null *serviceSubject* must not be read-only.
- In subsection "***at point (3) in the message processing model:"*** of Section 2.1.5.2, "What the Runtime Must Do", clarified that the call to *secureResponse* should be made independent of the outcome of the application request processing.
- added "Fig 2.2: State Diagram of Server Message Processing Runtime".

C.3.4 Changes to "Servlet Container Profile" Chapter

- Added last sentence to introductory paragraph to clarify what is required to be a compatible implementation of the profile.
- In Section 3.2, "Application Context Identifier", extended identifier format to include the logical hostname along with he context path.
- In Section 3.5, "CallbackHandler Requirements", added requirement that the handler argument (passed by the runtime) must not be null.
- Changed section 3.7 to reflect change in *AuthConfig* interface from *getOperation* to *getAuthContextID*.
- Changed Section 3.7.1, "Authentication Context Identifiers", to remove requirements for a specific identifier format.
- Changed Section 3.7.3, "Module Initialization Properties",to require that the runtime set the *PolicyContext* in the module initialization properties passed to *getAuthContext* call.
- In Section 3.7.4, "MessagePolicy Requirements", removed requirements relating to *responsePolicy.* Also moved responsibility for determining when (client) authentication is required from the AuthConfig subsystem to the message processing runtime.
- In Section 3.8, "Message Processing Requirements", 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.
- Changed Section 3.8.1, "MessageInfo Requirements", 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.*
- Added initial sentence to Section 3.8.3, "ServerAuth Processing", to reiterate that *validateRequest* be called on every request that satisfies the applicable connection requirements.
- In Section 3.8.3.1, "validateRequest Before Service Invocation",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).
- In Section 3.8.3.2, "validateRequest After Service Invocation", removed requirement that the module set the HTTP 200 (OK) status code.
- In Section 3.8.3.3, "secureResponse Processing", removed requirements dependent on *responsePolicy.*
- Replaced section "Dealing with Servlet Commit Semantics" with a new Section 3.8.3.4, "Wrapping and UnWrapping of Requests and Responses.

C.3.5 Changes to "SOAP Profile" Chapter

- Added last sentence to introductory paragraph to clarify what is required to be a compatible implementation of the profile.
- Changed Section 4.2, "Application Context Identifier", to refer to subsections within the sub-profiles where the corresponding identifiers are defined.
- In Section 4.5, "CallbackHandler Requirements", added requirement that the handler argument (passed by the runtime) must not be null.
- In Section 4.7, "Authentication Context Requirements", 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.
- Changed Section 4.7.1, "Authentication Context Identifiers", to remove requirements for a specific identifier format.
- Added new Section 4.8.1, "Client-Side Application Context Identifier", 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.
- Removed requirements from Section 4.8.4, "Authentication Context Requirements", that were already stated in Section 4.7, "Authentication Context Requirements".
- In Section 4.8.5, "Message Processing Requirements", 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.
- In Section 4.8.5.1, "MessageInfo Requirements", 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.
- In Section 4.8.5.3, "secureRequest Processing", 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.
- In "validateResponse After Service Invocation", 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.
- Added new Section 4.9.1, "Server-Side Application Context Identifier", 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.
- Removed requirements from Section 4.9.4, "Authentication Context Requirements", that were already stated in Section 4.7, "Authentication Context Requirements".
- Changed Section 4.9.4.1, "Module Initialization Properties", to require that *PolicyContext* be set in the module initialization properties (passed to *getAuthContext* call) if the server runtime is a JSR 115 compatible container.

- In Section 4.9.4.2, “MessagePolicy Requirements”, removed paragraphs defining when message protection is required by an EJB 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.
- In Section 4.9.5, “Message Processing Requirements”, 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.
- In Section , “MessageInfo Properties” removed the requirement that the service name property be set in the MessageInfo Map.
- In Section , “validateRequest Before Service Invocation”, 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, “secureResponse Processing”, corrected the required return value when responsePolicy == null to be AuthStatus.SEND_SUCCESS.

C.3.6 Changes to JMS Profile Chapter

- Renamed chapter to “Future Profiles”.
- Changed chapter to be strictly informative; serving to capture suggestions for additional profiles.
- Added Section 5.2, “RMI/IIOP Portable Interceptor Profile”.

C.3.7 Changes to Appendix B, Issues

- Added new issue, B.9,“HttpServletResponse Buffering and Header Commit Semantics", with resolution which was factored into the Servlet Profile (see 3.8.3.4,“Wrapping and UnWrapping of Requests and Responses").

C.3.8 Changes to Appendix D, API

- In *javax.security.auth.message.MessagePolicy,* changed name of method “*isManadatory*” to “*isMandatory*”.
- 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.

- In j*avax.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 *isPersisten*t method to the *AuthConfigFactory.RegistrationContext* interface.
- 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.
- In *javax.security.auth.message.config.ClientAuthConfig*, changed method and parameter descriptions to reflect change of "getOperation" to "getAuthContextID".
- In *javax.security.auth.message.config.ServerAuthConfig*, changed method and parameter descriptions to reflect change of "getOperation" to "getAuthContextID".
- 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*.
- 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*.
- 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

- Revised date to May 5, 2007

C.4.2 Changes to Servlet Container Profile

- In Section 3.8, "Message Processing Requirements", added reference to new section, Section 3.8.4, "Setting the Authentication Results on the HttpServletRequest" to describe requirements for setting the authentication results.
- Added Section 3.8.4, "Setting the Authentication Results on the HttpServletRequest" to capture requirements for setting the user principal, remote user, and authentication type on the HttpServletRequest.

C.4.3 Changes to SOAP Profile

- Corrected reference (chapter number) to “Message Authentication” chapter appearing in the chapter introduction.
- Corrected ambiguity in Section 4.3, “Message Requirements”, 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

- In Section 6.1, “Processing Model”, 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.
- In Section 6.3, “Standard Callbacks”, added requirement that GroupPrincipalCallback be supported when LoginContext is constructed with Subject.
- In Section 6.4, “Subjects”, 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

- Corrected JCP version to 2.6

C.5.2 Changes to Preface

- Changed Status and Audience to reflect transition to Final Release
- Changed “including J2EE containers” to “including J2EE and Java EE containers”

C.5.3 Changes to Overview

- Changed “exchanged by J2EE containers” to “exchanged by J2EE and Java EE containers”

C.5.4 Changes to References

- Changed “[J2SE Specification]” to “[Java SE 5 Specification]”

C.5.5 Changes to Issues

- Changed “introduced in J2SE 5.0” to “introduced in Java SE 5”

APPENDIX D

API

Overview

Package Summary	
Packages	
javax.security.auth.message[68]	This package defines the core interfaces of the JSR 196 message authentication SPI.
javax.security.auth.message.callback[89]	This package defines callback interfaces that may be used by a pluggable message authentication module to interact with the message processing runtime that invoked the module.
javax.security.auth.message.config[119]	This package defines the interfaces implemented by JSR 196 compatible configuration systems.
javax.security.auth.message.module[143]	This package defines the interfaces implemented by JSR 196 compatible authentication modules.

Package javax.security.auth.message

Description

This package defines the core interfaces of the JSR 196 message authentication SPI. The SPI defines the interfaces by which pluggable message authentication modules may be configured and invoked by message processing runtimes.

Class Summary

Interfaces

ClientAuth 73	An implementation of this interface is used to secure service request messages, and validate received service response messages.
MessageInfo 76	A message processing runtime uses this interface to pass messages and message processing state to authentication contexts for processing by authentication modules.
MessagePolicy.ProtectionPolicy 80	This interface is used to represent message authentication policy.
MessagePolicy.Target 82	This interface is used to represent and perform message targeting.
ServerAuth 86	An implementation of this interface is used to validate received service request messages, and to secure service response messages.

Classes

AuthStatus 71	The AuthStatus class is used to represent return values from Authentication modules and Authentication Contexts.
MessagePolicy 78	This class defines a message authentication policy.
MessagePolicy.TargetPolicy 84	This class defines the message protection policies for specific Targets.

Exceptions

AuthException 69	A generic authentication exception.

javax.security.auth.message
AuthException

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--java.security.GeneralSecurityException
                    |
                    +--javax.security.auth.login.LoginException
                          |
                          +--javax.security.auth.message.AuthException
```

All Implemented Interfaces: `java.io.Serializable`

Declaration

```
public class AuthException extends javax.security.auth.login.LoginException
```

Description

A generic authentication exception.

Member Summary	
Constructors	
	AuthException[69]()
	AuthException[70](java.lang.String msg)

Inherited Member Summary
Methods inherited from class `Object`
`clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), wait(), wait(long), wait(long, int)`
Methods inherited from class `Throwable`
`fillInStackTrace(), getCause(), getLocalizedMessage(), getMessage(), getStackTrace(), initCause(Throwable), printStackTrace(), printStackTrace(PrintStream), printStackTrace(PrintWriter), setStackTrace(StackTraceElement[]), toString()`

Constructors

AuthException()

```
public AuthException()
```

Constructs an AuthException with no detail message. A detail message is a String that describes this particular exception.

AuthException(`java.lang.String` msg)

```
public AuthException(java.lang.String msg)
```

Constructs an AuthException with the specified detail message. A detail message is a String that describes this particular exception.

Parameters:

`msg` - The detail message.

javax.security.auth.message
AuthStatus

```
java.lang.Object
  |
  +--javax.security.auth.message.AuthStatus
```

Declaration
```
public class AuthStatus
```

Description
The AuthStatus class is used to represent return values from Authentication modules and Authentication Contexts. An AuthStatus value is returned when the module processing has established a corresponding request or response message within the message parameters exchanged with the runtime.

See Also: `java.util.Map<K,V>`

Member Summary
Fields
static AuthStatus[71] FAILURE[71]
static AuthStatus[71] SEND_CONTINUE[72]
static AuthStatus[71] SEND_FAILURE[72]
static AuthStatus[71] SEND_SUCCESS[72]
static AuthStatus[71] SUCCESS[72]

Inherited Member Summary
Methods inherited from class `Object`
`clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)`

Fields

FAILURE
```
public static final AuthStatus[71] FAILURE
```
Indicates that the message processing by the authentication module was NOT successful, and that the module replaced the application message with an error message.

SEND_CONTINUE

```
public static final AuthStatus71 SEND_CONTINUE
```

Indicates the message processing by the authentication module is NOT complete, that the module replaced the application message with a security message, and that the runtime is to proceed by sending the security message.

SEND_FAILURE

```
public static final AuthStatus71 SEND_FAILURE
```

Indicates that the message processing by the authentication module was NOT successful, that the module replaced the application message with an error message, and that the runtime is to proceed by sending the error message.

SEND_SUCCESS

```
public static final AuthStatus71 SEND_SUCCESS
```

Indicates that the message processing by the authentication module was successful and that the runtime is to proceed by sending a message returned by the authentication module.

SUCCESS

```
public static final AuthStatus71 SUCCESS
```

Indicates that the message processing by the authentication module was successful and that the runtime is to proceed with its normal processing of the resulting message.

javax.security.auth.message
ClientAuth

All Known Subinterfaces: ClientAuthContext[137], ClientAuthModule[144]

Declaration
```
public interface ClientAuth
```

Description
An implementation of this interface is used to secure service request messages, and validate received service response messages.

See Also: MessageInfo[76], javax.security.auth.Subject

Member Summary	
Methods	
void	cleanSubject[73](MessageInfo[76] messageInfo, javax.security.auth.Subject subject)
AuthStatus[71]	secureRequest[73](MessageInfo[76] messageInfo, javax.security.auth.Subject clientSubject)
AuthStatus[71]	validateResponse[74](MessageInfo[76] messageInfo, javax.security.auth.Subject clientSubject, javax.security.auth.Subject serviceSubject)

Methods

cleanSubject(MessageInfo[76] messageInfo, javax.security.auth.Subject subject)

```
public void cleanSubject(MessageInfo[76] messageInfo, javax.security.auth.Subject subject)
            throws AuthException
```

Remove implementation specific principals and credentials from the subject.

Parameters:

messageInfo - A contextual object that encapsulates the client request and server response objects, and that may be used to save state across a sequence of calls made to the methods of this interface for the purpose of completing a secure message exchange.

subject - The Subject instance from which the Principals and credentials are to be removed.

Throws:

AuthException[69] - If an error occurs during the Subject processing.

secureRequest(MessageInfo[76] messageInfo, javax.security.auth.Subject clientSubject)

```
public AuthStatus[71] secureRequest(MessageInfo[76] messageInfo, javax.security.auth.Subject
            clientSubject)
            throws AuthException
```

Secure a service request message before sending it to the service.

This method is called to transform the request message acquired by calling getRequestMessage (on messageInfo) into the mechanism-specific form to be sent by the runtime.

This method conveys the outcome of its message processing either by returning an AuthStatus value or by throwing an AuthException.

Parameters:

`messageInfo` - A contextual object that encapsulates the client request and server response objects, and that may be used to save state across a sequence of calls made to the methods of this interface for the purpose of completing a secure message exchange.

`clientSubject` - A Subject that represents the source of the service request, or null. It may be used by the method implementation as the source of Principals or credentials to be used to secure the request. If the Subject is not null, the method implementation may add additional Principals or credentials (pertaining to the source of the service request) to the Subject.

Returns: An AuthStatus object representing the completion status of the processing performed by the method. The AuthStatus values that may be returned by this method are defined as follows:

- AuthStatus.SUCCESS when the application request message was successfully secured. The secured request message may be obtained by calling getRequestMessage on messageInfo.
- AuthStatus.SEND_CONTINUE to indicate that the application request message (within messageInfo) was replaced with a security message that should elicit a security-specific response from the peer security system. This status value also indicates that the application message has not yet been secured. This status value serves to inform the calling runtime that (to successfully complete the message exchange) it will need to be capable of continuing the message dialog by conducting at least one additional request/response exchange after having received the security-specific response elicited by sending the security message. When this status value is returned, the corresponding invocation of `validateResponse` must be able to obtain the original application request message.
- AuthStatus.FAILURE to indicate that a failure occured while securing the request message, and that an appropriate failure response message is available by calling getResponseMessage on messageInfo.

Throws:

`AuthException`[69] - When the message processing failed without establishing a failure response message (in messageInfo).

validateResponse(`MessageInfo`[76] messageInfo, `javax.security.auth.Subject` clientSubject, `javax.security.auth.Subject` serviceSubject)

```
public AuthStatus[71] validateResponse(MessageInfo[76] messageInfo,
            javax.security.auth.Subject clientSubject, javax.security.auth.Subject
            serviceSubject)
            throws AuthException
```

Validate a received service response.

This method is called to transform the mechanism-specific response message acquired by calling getResponseMessage (on messageInfo) into the validated application message to be returned to the message processing runtime. If the response message is a (mechanism-specific) meta-message, the method implementation must attempt to transform the meta-message into the next mechanism-specific request message to be sent by the runtime.

This method conveys the outcome of its message processing either by returning an AuthStatus value or by throwing an AuthException.

Parameters:

`messageInfo` - A contextual object that encapsulates the client request and server response objects, and that may be used to save state across a sequence of calls made to the methods of this interface for the purpose of completing a secure message exchange.

`clientSubject` - A Subject that represents the recipient of the service response, or null. It may be used by the method implementation as the source of Principals or credentials to be used to validate the response. If the Subject is not null, the method implementation may add additional Principals or credentials (pertaining to the recipient of the service request) to the Subject.

`serviceSubject` - A Subject that represents the source of the service response, or null. If the Subject is not null, the method implementation may add additional Principals or credentials (pertaining to the source of the service response) to the Subject.

Returns: An AuthStatus object representing the completion status of the processing performed by the method. The AuthStatus values that may be returned by this method are defined as follows:

- AuthStatus.SUCCESS when the application response message was successfully validated. The validated message is available by calling getResponseMessage on messageInfo.
- AuthStatus.SEND_CONTINUE to indicate that response validation is incomplete, and that a continuation request was returned as the request message within messageInfo. This status value serves to inform the calling runtime that (to successfully complete the message exchange) it will need to be capable of continuing the message dialog by conducting at least one additional request/response exchange.
- AuthStatus.FAILURE to indicate that validation of the response failed, and that a failure response message has been established in messageInfo.

Throws:

`AuthException` 69 - When the message processing failed without establishing a failure response message (in messageInfo).

javax.security.auth.message
MessageInfo

Declaration
```
public interface MessageInfo
```

Description
A message processing runtime uses this interface to pass messages and message processing state to authentication contexts for processing by authentication modules.

This interface encapsulates a request message object and a response message object for a message exchange. This interface may also be used to associate additional context in the form of key/value pairs, with the encapsulated messages.

Every implementation of this interface should provide a zero argument constructor, and a constructor which takes a single Map argument. Additional constructors may also be provided.

See Also: `java.util.Map<K,V>`

Member Summary	
Methods	
`java.util.Map`	`getMap`$_{76}$`()`
`java.lang.Object`	`getRequestMessage`$_{76}$`()`
`java.lang.Object`	`getResponseMessage`$_{77}$`()`
`void`	`setRequestMessage`$_{77}$`(java.lang.Object request)`
`void`	`setResponseMessage`$_{77}$`(java.lang.Object response)`

Methods

getMap()
```
public java.util.Map getMap()
```
Get (a reference to) the Map object of this MessageInfo. Operations performed on the acquired Map must effect the Map within the MessageInfo.

Returns: the Map object of this MessageInfo. This method never returns null. If a Map has not been associated with the MessageInfo, this method instantiates a Map, associates it with this MessageInfo, and then returns it.

getRequestMessage()
```
public java.lang.Object getRequestMessage()
```
Get the request message object from this MessageInfo.

Returns: An object representing the request message, or null if no request message is set within the MessageInfo.

getResponseMessage()

```
public java.lang.Object getResponseMessage()
```

Get the response message object from this MessageInfo.

Returns: an object representing the response message, or null if no response message is set within the MessageInfo.

setRequestMessage(java.lang.Object request)

```
public void setRequestMessage(java.lang.Object request)
```

Set the request message object in this MessageInfo.

Parameters:
`request` - An object representing the request message.

setResponseMessage(java.lang.Object response)

```
public void setResponseMessage(java.lang.Object response)
```

Set the response message object in this MessageInfo.

Parameters:
`response` - An object representing the response message.

javax.security.auth.message
MessagePolicy

```
java.lang.Object
  |
  +--javax.security.auth.message.MessagePolicy
```

Declaration

```
public class MessagePolicy
```

Description

This class defines a message authentication policy.

A ClientAuthContext uses this class to communicate (at module initialization time) request and response message protection policies to its ClientAuthModule objects. A ServerAuthContext uses this class to communicate request and response message protection policies to its ServerAuthModule objects.

See Also: ClientAuthContext[137], ServerAuthContext[141], ClientAuthModule[144], ServerAuthModule[146]

Member Summary

Nested Classes

static	MessagePolicy.ProtectionPolicy[80]
static	MessagePolicy.Target[82]
static class	MessagePolicy.TargetPolicy[84]

Constructors

	MessagePolicy[79](MessagePolicy.TargetPolicy[84][] targetPolicies, boolean mandatory)

Methods

	getTargetPolicies[79]()
MessagePolicy.TargetPolicy[84][]	
boolean	isMandatory[79]()

Inherited Member Summary

Methods inherited from class `Object`

`clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)`

Constructors

MessagePolicy(MessagePolicy.TargetPolicy[] targetPolicies, boolean mandatory)

```
public MessagePolicy(MessagePolicy.TargetPolicy[] targetPolicies, boolean mandatory)
```

Create a MessagePolicy instance with an array of target policies.

Parameters:

`targetPolicies` - an array of target policies.

`mandatory` - A boolean value indicating whether the MessagePolicy is mandatory or optional.

Throws:

`java.lang.IllegalArgumentException` - if the specified targetPolicies is null.

Methods

getTargetPolicies()

```
public MessagePolicy.TargetPolicy[] getTargetPolicies()
```

Get the target policies that comprise the authentication policy.

Returns: An array of target authentication policies, where each element describes an authentication policy and the parts of the message to which the authentication policy applies. This method returns null to indicate that no security operations should be performed on the messages to which the policy applies. This method never returns a zero-length array.

When this method returns an array of target policies, the order of elements in the array represents the order that the corresponding message transformations or validations described by the target policies are to be performed by the authentication module.

isMandatory()

```
public boolean isMandatory()
```

Get the MessagePolicy modifier.

Returns: A boolean indicating whether the MessagePolicy is optional(false) or required(true).

javax.security.auth.message
MessagePolicy.ProtectionPolicy

Enclosing Class: MessagePolicy[78]

Declaration
```
public static interface MessagePolicy.ProtectionPolicy
```

Description
This interface is used to represent message authentication policy.

The internal state of a ProtectionPolicy object defines the message authentication requirements to be applied to the associated Target.

Member Summary	
Fields	
static	AUTHENTICATE_CONTENT[80]
java.lang.String	
static	AUTHENTICATE_RECIPIENT[80]
java.lang.String	
static	AUTHENTICATE_SENDER[80]
java.lang.String	
Methods	
java.lang.String	getID[81]()

Fields

AUTHENTICATE_CONTENT
```
public static final java.lang.String AUTHENTICATE_CONTENT
```
The identifer for a ProtectionPolicy that indicates that the origin of data within the message is to be authenticated (that is, the message is to be protected such that its recipients can establish who defined the message content).

AUTHENTICATE_RECIPIENT
```
public static final java.lang.String AUTHENTICATE_RECIPIENT
```
The identifer for a ProtectionPolicy that indicates that the message recipient is to be authenticated.

AUTHENTICATE_SENDER
```
public static final java.lang.String AUTHENTICATE_SENDER
```
The identifer for a ProtectionPolicy that indicates that the sending entity is to be authenticated.

Methods

getID()

```
public java.lang.String getID()
```

Get the ProtectionPolicy identifier. An identifier may represent a conceptual protection policy (as is the case with the static identifiers defined within this interface) or it may identify a procedural policy expression or plan that may be more difficult to categorize in terms of a conceptual identifier.

Returns: A String containing a policy identifier. This interface defines some policy identifier constants. Configuration systems may define and employ other policy identifiers values.

javax.security.auth.message
MessagePolicy.Target

Enclosing Class: MessagePolicy[78]

Declaration
```
public static interface MessagePolicy.Target
```

Description
This interface is used to represent and perform message targeting. Targets are used by message authentication modules to operate on the corresponding content within messages.

The internal state of a Target indicates whether it applies to the request or response message of a MessageInfo and to which components it applies within the identified message.

Member Summary	
Methods	
java.lang.Object	get[82](MessageInfo[76] messageInfo)
void	put[82](MessageInfo[76] messageInfo, java.lang.Object data)
void	remove[82](MessageInfo[76] messageInfo)

Methods

get(MessageInfo[76] messageInfo)

```
public java.lang.Object get(MessageInfo[76] messageInfo)
```

Get the Object identified by the Target from the MessageInfo.

Parameters:
`messageInfo` - The MessageInfo containing the request or response message from which the target is to be obtained.

Returns: An Object representing the target, or null when the target could not be found in the MessageInfo.

put(MessageInfo[76] messageInfo, java.lang.Object data)

```
public void put(MessageInfo[76] messageInfo, java.lang.Object data)
```

Put the Object into the MessageInfo at the location identified by the target.

Parameters:
`messageInfo` - The MessageInfo containing the request or response message into which the object is to be put.

remove(MessageInfo[76] messageInfo)

```
public void remove(MessageInfo[76] messageInfo)
```

Remove the Object identified by the Target from the MessageInfo.

Parameters:

`messageInfo` - The MessageInfo containing the request or response message from which the target is to be removed.

javax.security.auth.message
MessagePolicy.TargetPolicy

```
java.lang.Object
  |
  +--javax.security.auth.message.MessagePolicy.TargetPolicy
```

Enclosing Class: MessagePolicy[78]

Declaration

```
public static class MessagePolicy.TargetPolicy
```

Description

This class defines the message protection policies for specific Targets.

This class is used to associate a message protection policy with targets within a message. Message targets are represented using an implementation of the *Target* interface matched to the message types in MessageInfo. The message protection policy is identified by an implementation of the *ProtectionPolicy* interface.

See Also: ClientAuthModule[144], ServerAuthModule[146]

Member Summary

Constructors

	MessagePolicy.TargetPolicy[85](MessagePolicy.Target[82][] targets, MessagePolicy.ProtectionPolicy[80] protectionPolicy)

Methods

MessagePolicy.ProtectionPolicy[80]	getProtectionPolicy[85]()
MessagePolicy.Target[82][]	getTargets[85]()

Inherited Member Summary

Methods inherited from class `Object`

`clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)`

Constructors

MessagePolicy.TargetPolicy(MessagePolicy.Target[] targets, MessagePolicy.ProtectionPolicy protectionPolicy)

```
public MessagePolicy.TargetPolicy(MessagePolicy.Target[] targets,
            MessagePolicy.ProtectionPolicy protectionPolicy)
```

Create a TargetPolicy instance with an array of Targets and with a ProtectionPolicy.

Parameters:

`targets` - An array of Targets. This argument may be null.

`protectionPolicy` - The object that describes the message authentication policy that applies to the targets.

Throws:

`java.lang.IllegalArgumentException` - if the specified targets or protectionPolicy is null.

Methods

getProtectionPolicy()

```
public MessagePolicy.ProtectionPolicy getProtectionPolicy()
```

Get the ProtectionPolicy that applies to the targets.

Returns: A ProtectionPolicy object that identifies the message authentication requirements that apply to the targets.

getTargets()

```
public MessagePolicy.Target[] getTargets()
```

Get the array of layer-specific target descriptors that identify the one or more message parts to which the specified message protection policy applies.

Returns: An array of *Target* that identify targets within a message. This method returns null when the specified policy applies to the whole message (excluding any metadata added to the message to satisfy the policy). This method never returns a zero-length array.

javax.security.auth.message
ServerAuth

All Known Subinterfaces: ServerAuthContext[141], ServerAuthModule[146]

Declaration
```
public interface ServerAuth
```

Description
An implementation of this interface is used to validate received service request messages, and to secure service response messages.

See Also: MessageInfo[76], javax.security.auth.Subject

Member Summary	
Methods	
void	cleanSubject[86](MessageInfo[76] messageInfo, javax.security.auth.Subject subject)
AuthStatus[71]	secureResponse[86](MessageInfo[76] messageInfo, javax.security.auth.Subject serviceSubject)
AuthStatus[71]	validateRequest[87](MessageInfo[76] messageInfo, javax.security.auth.Subject clientSubject, javax.security.auth.Subject serviceSubject)

Methods

cleanSubject(MessageInfo[76] messageInfo, javax.security.auth.Subject subject)

```
public void cleanSubject(MessageInfo[76] messageInfo, javax.security.auth.Subject subject)
            throws AuthException
```

Remove method specific principals and credentials from the subject.

Parameters:

messageInfo - a contextual object that encapsulates the client request and server response objects, and that may be used to save state across a sequence of calls made to the methods of this interface for the purpose of completing a secure message exchange.

subject - the Subject instance from which the Principals and credentials are to be removed.

Throws:

AuthException[69] - If an error occurs during the Subject processing.

secureResponse(MessageInfo[76] messageInfo, javax.security.auth.Subject

serviceSubject)

```
public AuthStatus71 secureResponse(MessageInfo76 messageInfo, javax.security.auth.Subject
            serviceSubject)
            throws AuthException
```

Secure a service response before sending it to the client. This method is called to transform the response message acquired by calling getResponseMessage (on messageInfo) into the mechanism-specific form to be sent by the runtime.

This method conveys the outcome of its message processing either by returning an AuthStatus value or by throwing an AuthException.

Parameters:

`messageInfo` - A contextual object that encapsulates the client request and server response objects, and that may be used to save state across a sequence of calls made to the methods of this interface for the purpose of completing a secure message exchange.

`serviceSubject` - A Subject that represents the source of the service response, or null. It may be used by the method implementation to retrieve Principals and credentials necessary to secure the response. If the Subject is not null, the method implementation may add additional Principals or credentials (pertaining to the source of the service response) to the Subject.

Returns: An AuthStatus object representing the completion status of the processing performed by the method. The AuthStatus values that may be returned by this method are defined as follows:

- AuthStatus.SEND_SUCCESS when the application response message was successfully secured. The secured response message may be obtained by calling getResponseMessage on messageInfo.
- AuthStatus.SEND_CONTINUE to indicate that the application response message (within messageInfo) was replaced with a security message that should elicit a security-specific response (in the form of a request) from the peer. This status value serves to inform the calling runtime that (to successfully complete the message exchange) 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). When this status value is returned, the application response must be saved by the authentication module such that it can be recovered when the module's validateRequest message is called to process the elicited response.
- AuthStatus.SEND_FAILURE to indicate that a failure occurred while securing the response message and that an appropriate failure response message is available by calling getResponseMeessage on messageInfo.

Throws:

`AuthException69` - When the message processing failed without establishing a failure response message (in messageInfo).

validateRequest(MessageInfo76 messageInfo, javax.security.auth.Subject clientSubject, javax.security.auth.Subject serviceSubject)

```
public AuthStatus71 validateRequest(MessageInfo76 messageInfo,
            javax.security.auth.Subject clientSubject, javax.security.auth.Subject
            serviceSubject)
            throws AuthException
```

Authenticate a received service request. This method is called to transform the mechanism-specific request message acquired by calling getRequestMessage (on messageInfo) into the validated application message to be returned to the message processing runtime. If the received message is a (mechanism-specific) meta-message, the method implementation must attempt to transform the meta-message into a corresponding

mechanism-specific response message, or to the validated application request message. The runtime will bind a validated application message into the the corresponding service invocation.

This method conveys the outcome of its message processing either by returning an AuthStatus value or by throwing an AuthException.

Parameters:

`messageInfo` - A contextual object that encapsulates the client request and server response objects, and that may be used to save state across a sequence of calls made to the methods of this interface for the purpose of completing a secure message exchange.

`clientSubject` - A Subject that represents the source of the service request. It is used by the method implementation to store Principals and credentials validated in the request.

`serviceSubject` - A Subject that represents the recipient of the service request, or null. It may be used by the method implementation as the source of Principals or credentials to be used to validate the request. If the Subject is not null, the method implementation may add additional Principals or credentials (pertaining to the recipient of the service request) to the Subject.

Returns: An AuthStatus object representing the completion status of the processing performed by the method. The AuthStatus values that may be returned by this method are defined as follows:

- AuthStatus.SUCCESS when the application request message was successfully validated. The validated request message is available by calling getRequestMessage on messageInfo.
- AuthStatus.SEND_SUCCESS to indicate that validation/processing of the request message successfully produced the secured application response message (in messageInfo). The secured response message is available by calling getResponseMessage on messageInfo.
- AuthStatus.SEND_CONTINUE to indicate that message validation is incomplete, and that a preliminary response was returned as the response message in messageInfo. When this status value is returned to challenge an application request message, the challenged request must be saved by the authentication module such that it can be recovered when the module's validateRequest message is called to process the request returned for the challenge.
- AuthStatus.SEND_FAILURE to indicate that message validation failed and that an appropriate failure response message is available by calling getResponseMessage on messageInfo.

Throws:

`AuthException`[69] - When the message processing failed without establishing a failure response message (in messageInfo).

Package javax.security.auth.message.callback

Description

This package defines callback interfaces that may be used by a pluggable message authentication module to interact with the message processing runtime that invoked the module.

Class Summary	
Interfaces	
PrivateKeyCallback.Request 109	Marker interface for private key request types.
SecretKeyCallback.Request 116	Marker interface for secret key request types.
Classes	
CallerPrincipalCallback 90	Callback for setting the container's caller (or Remote user) principal.
CertStoreCallback 93	Callback for CertStore.
GroupPrincipalCallback 95	Callback establishing group principals within the argument subject.
PasswordValidationCallback 97	Callback for PasswordValidation.
PrivateKeyCallback 100	Callback for acquiring a Public Key Infrastructure (PKI) private key and its corresponding certificate chain.
PrivateKeyCallback.AliasRequest 103	Request type for private keys that are identified using an alias.
PrivateKeyCallback.DigestRequest 105	Request type for private keys that are identified using a certificate digest or thumbprint.
PrivateKeyCallback.IssuerSerialNumRequest 107	Request type for private keys that are identified using an issuer/serial number.
PrivateKeyCallback.SubjectKeyIDRequest 110	Request type for private keys that are identified using a SubjectKeyID
SecretKeyCallback 112	Callback for acquiring a shared secret from a key repository.
SecretKeyCallback.AliasRequest 114	Request type for secret keys that are identified using an alias.
TrustStoreCallback 117	Callback for trusted certificate KeyStore.

javax.security.auth.message.callback
CallerPrincipalCallback

```
java.lang.Object
  |
  +--javax.security.auth.message.callback.CallerPrincipalCallback
```

All Implemented Interfaces: `javax.security.auth.callback.Callback`

Declaration

```
public class CallerPrincipalCallback implements javax.security.auth.callback.Callback
```

Description

Callback for setting the container's caller (or Remote user) principal. This callback is intended to be called by a `serverAuthModule` during its `validateRequest` processing.

Member Summary	
Constructors	
	CallerPrincipalCallback[90](javax.security.auth.Subject s, java.security.Principal p)
	CallerPrincipalCallback[91](javax.security.auth.Subject s, java.lang.String n)
Methods	
java.lang.String	getName[91]()
java.security.Principal	getPrincipal[91]()
javax.security.auth.Subject	getSubject[91]()

Inherited Member Summary
Methods inherited from class `Object`
`clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)`

Constructors

CallerPrincipalCallback(`javax.security.auth.Subject` s, `java.security.Principal`

p)

```
public CallerPrincipalCallback(javax.security.auth.Subject s, java.security.Principal p)
```

Create a CallerPrincipalCallback to set the container's representation of the caller principal

Parameters:

`s` - The Subject in which the container will distinguish the caller identity.

`p` - The Principal that will be distinguished as the caller principal. This value may be null.

The CallbackHandler must establish the argument Principal as the caller principal associated with the invocation being processed by the container. When the argument Principal is null, the handler will establish the container's representation of the unauthenticated caller principal.

CallerPrincipalCallback(`javax.security.auth.Subject` s, `java.lang.String` n)

```
public CallerPrincipalCallback(javax.security.auth.Subject s, java.lang.String n)
```

Create a CallerPrincipalCallback to set the container's representation of the caller principal.

Parameters:

`s` - The Subject in which the container will distinguish the caller identity.

`n` - The String value that will be returned when getName() is called on the principal established as the caller principal or null.

When the n argument is null, the handler will establish the container's representation of the unauthenticated caller principal (which may or may not be equal to null, depending on the requirements of the container type). When the container type requires that a non-null principal be established as the caller principal, the value obatined by calling getName on the principal may not match the arument value.

Methods

getName()

```
public java.lang.String getName()
```

Get the caller principal name.

Returns: The principal name or null.

When the values returned by this method and the getPrincipal methods are null, the handler must establish the container's representation of the unauthenticated caller principal within the Subject.

getPrincipal()

```
public java.security.Principal getPrincipal()
```

Get the caller principal.

Returns: The principal or null.

When the values returned by this method and the getName methods are null, the handler must establish the container's representation of the unauthenticated caller principal within the Subject.

getSubject()

```
public javax.security.auth.Subject getSubject()
```

Get the Subject in which the handler will distinguish the caller principal

Returns: The subject.

javax.security.auth.message.callback
CertStoreCallback

```
java.lang.Object
  |
  +--javax.security.auth.message.callback.CertStoreCallback
```

All Implemented Interfaces: `javax.security.auth.callback.Callback`

Declaration
```
public class CertStoreCallback implements javax.security.auth.callback.Callback
```

Description
Callback for CertStore.

A CertStore is a generic repository for certificates. CertStores may be searched to locate public key certificates, as well as to put together certificate chains. Such a search may be necessary when the caller needs to verify a signature.

Member Summary	
Constructors	
	CertStoreCallback93()
Methods	
	getCertStore94()
java.security.cert.CertStore	
void	setCertStore94(java.security.cert.CertStore certStore)

Inherited Member Summary
Methods inherited from class `Object`
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)

Constructors

CertStoreCallback()
```
public CertStoreCallback()
```
Create a CertStoreCallback.

Methods

getCertStore()

```
public java.security.cert.CertStore getCertStore()
```

Used by the CertStore user to obtain the CertStore set within the Callback.

Returns: The CertStore, or null.

setCertStore(java.security.cert.CertStore certStore)

```
public void setCertStore(java.security.cert.CertStore certStore)
```

Used by the CallbackHandler to set the CertStore within the Callback.

Parameters:

`certStore` - The certificate store, which may be null

javax.security.auth.message.callback
GroupPrincipalCallback

```
java.lang.Object
  |
  +--javax.security.auth.message.callback.GroupPrincipalCallback
```

All Implemented Interfaces: `javax.security.auth.callback.Callback`

Declaration
```
public class GroupPrincipalCallback implements javax.security.auth.callback.Callback
```

Description
Callback establishing group principals within the argument subject. This callback is intended to be called by a `serverAuthModule` during its `validateRequest` processing.

Member Summary	
Constructors	
	GroupPrincipalCallback95(javax.security.auth.Subject s, java.lang.String[] g)
Methods	
java.lang.String[]	getGroups96()
javax.security.auth.Subject	getSubject96()

Inherited Member Summary
Methods inherited from class `Object`
`clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)`

Constructors

GroupPrincipalCallback(`javax.security.auth.Subject` s, `java.lang.String[]` g)

```
public GroupPrincipalCallback(javax.security.auth.Subject s, java.lang.String[] g)
```

Create a GroupPrincipalCallback to establish the container's representation of the corresponding group principals within the Subject.

Parameters:

`s` - The Subject in which the container will create group principals.

`g` - An array of Strings, where each element contains the name of a group that will be used to create a corresponding group principal within the Subject.

When a null value is passed to the g argument, the handler will establish the container's representation of no group principals within the Subject. Otherwise, the handler's processing of this callback is additive, yielding the union (without duplicates) of the principals existing within the Subject, and those created with the names occuring within the argument array. The CallbackHandler will define the type of the created principals.

Methods

getGroups()

```
public java.lang.String[] getGroups()
```

Get the array of group names.

Returns: Null, or an array containing 0 or more String group names.

When the return value is null, the handler will establish the container's representation of no group principals within the Subject. Otherwise, the handler's processing of this callback is additive, yielding the union (without duplicates) of the principals created with the names in the returned array and those existing within the Subject.

getSubject()

```
public javax.security.auth.Subject getSubject()
```

Get the Subject in which the handler will establish the group principals.

Returns: The subject.

javax.security.auth.message.callback
PasswordValidationCallback

```
java.lang.Object
  |
  +--javax.security.auth.message.callback.PasswordValidationCallback
```

All Implemented Interfaces: `javax.security.auth.callback.Callback`

Declaration

```
public class PasswordValidationCallback implements javax.security.auth.callback.Callback
```

Description

Callback for PasswordValidation. This callback may be used by an authentication module to employ the password validation facilities of its containing runtime. This Callback would typically be called by a `ServerAuthModule` during `validateRequest` processing.

Member Summary	
Constructors	
	PasswordValidationCallback$_{97}$(javax.security.auth.Subject subject, java.lang.String username, char[] password)
Methods	
void	clearPassword$_{98}$()
char[]	getPassword$_{98}$()
boolean	getResult$_{98}$()
javax.security.auth.Subject	getSubject$_{98}$()
java.lang.String	getUsername$_{98}$()
void	setResult$_{98}$(boolean result)

Inherited Member Summary
Methods inherited from class `Object`
`clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)`

Constructors

PasswordValidationCallback(`javax.security.auth.Subject` subject, `java.lang.String`

username, char[] password)

```
public PasswordValidationCallback(javax.security.auth.Subject subject, java.lang.String
            username, char[] password)
```

Create a PasswordValidationCallback.

Parameters:

`subject` - The subject for authentication

`username` - The username to authenticate

`password` - tTe user's password, which may be null.

Methods

clearPassword()

```
public void clearPassword()
```

Clear the password.

getPassword()

```
public char[] getPassword()
```

Get the password.

Note that this method returns a reference to the password. If a clone of the array is created it is the caller's responsibility to zero out the password information after it is no longer needed.

Returns: The password, which may be null.

getResult()

```
public boolean getResult()
```

Get the authentication result.

Returns: True if authentication succeeded, false otherwise

getSubject()

```
public javax.security.auth.Subject getSubject()
```

Get the subject.

Returns: The subject.

getUsername()

```
public java.lang.String getUsername()
```

Get the username.

Returns: The username.

setResult(boolean result)

```
public void setResult(boolean result)
```

Set the authentication result.

Parameters:

`result` - True if authentication succeeded, false otherwise

javax.security.auth.message.callback
PrivateKeyCallback

```
java.lang.Object
  |
  +--javax.security.auth.message.callback.PrivateKeyCallback
```

All Implemented Interfaces: `javax.security.auth.callback.Callback`

Declaration

```
public class PrivateKeyCallback implements javax.security.auth.callback.Callback
```

Description

Callback for acquiring a Public Key Infrastructure (PKI) private key and its corresponding certificate chain. This Callback may be used by client or server authentication modules to obtain private keys or private key references, from key repositories available to the CallbackHandler that processes the Callback.

Member Summary

Nested Classes

static class	PrivateKeyCallback.AliasRequest[103]
static class	PrivateKeyCallback.DigestRequest[105]
static class	PrivateKeyCallback.IssuerSerialNumRequest[107]
static	PrivateKeyCallback.Request[109]
static class	PrivateKeyCallback.SubjectKeyIDRequest[110]

Constructors

	PrivateKeyCallback[101](PrivateKeyCallback.Request[109] request)

Methods

	getChain[101]()
java.security.cert.Certificate[]	
	getKey[101]()
java.security.PrivateKey	
	getRequest[101]()
PrivateKeyCallback.Request[109]	
void	setKey[101](java.security.PrivateKey key, java.security.cert.Certificate[] chain)

Inherited Member Summary

Methods inherited from class `Object`

Inherited Member Summary
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)

Constructors

PrivateKeyCallback(PrivateKeyCallback.Request[109] request)

```
public PrivateKeyCallback(PrivateKeyCallback.Request[109] request)
```

Constructs this PrivateKeyCallback with a private key Request object.

The *request* object identifies the private key to be returned. The corresponding certificate chain for the private key is also returned.

If the *request* object is null, the handler of the callback relies on its own default.

Parameters:
`request` - Identifier for the private key, or null.

Methods

getChain()

```
public java.security.cert.Certificate[] getChain()
```

Used to obtain the certicicate chain set within the Callback.

Returns: The certificate chain, or null if the chain could not be found.

getKey()

```
public java.security.PrivateKey getKey()
```

Used to obtain the private key set within the Callback.

Returns: The private key, or null if the key could not be found.

getRequest()

```
public PrivateKeyCallback.Request[109] getRequest()
```

Used by the CallbackHandler to get the Request object that identifies the private key to be returned.

Returns: The Request object which identifies the private key to be returned, or null. If null, the handler of the callback relies on its own default.

setKey(java.security.PrivateKey key, java.security.cert.Certificate[] chain)

```
public void setKey(java.security.PrivateKey key, java.security.cert.Certificate[] chain)
```

Used by the CallbackHandler to set the requested private key and the corresponding certificate chain within the Callback.

If the requested private key or chain could not be found, then both values must be set to null.

Parameters:
`key` - The private key, or null.

`chain` - The corresponding certificate chain, or null.

javax.security.auth.message.callback
PrivateKeyCallback.AliasRequest

```
java.lang.Object
  |
  +--javax.security.auth.message.callback.PrivateKeyCallback.AliasRequest
```

All Implemented Interfaces: PrivateKeyCallback.Request[109]

Enclosing Class: PrivateKeyCallback[100]

Declaration

```
public static class PrivateKeyCallback.AliasRequest implements PrivateKeyCallback.Request[109]
```

Description

Request type for private keys that are identified using an alias.

Member Summary	
Constructors	
	PrivateKeyCallback.AliasRequest[103](java.lang.String alias)
Methods	
java.lang.String	getAlias[104]()

Inherited Member Summary
Methods inherited from class Object
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)

Constructors

PrivateKeyCallback.AliasRequest(java.lang.String alias)

```
public PrivateKeyCallback.AliasRequest(java.lang.String alias)
```

Construct an AliasRequest with an alias.

The alias is used to directly identify the private key to be returned. The corresponding certificate chain for the private key is also returned.

If the alias is null, the handler of the callback relies on its own default.

Parameters:
`alias` - Name identifier for the private key, or null.

Methods

getAlias()

```
public java.lang.String getAlias()
```

Get the alias.

Returns: The alias, or null.

javax.security.auth.message.callback
PrivateKeyCallback.DigestRequest

```
java.lang.Object
  |
  +--javax.security.auth.message.callback.PrivateKeyCallback.DigestRequest
```

All Implemented Interfaces: PrivateKeyCallback.Request[109]

Enclosing Class: PrivateKeyCallback[100]

Declaration

```
public static class PrivateKeyCallback.DigestRequest implements PrivateKeyCallback.Request[109]
```

Description

Request type for private keys that are identified using a certificate digest or thumbprint.

Member Summary	
Constructors	
	PrivateKeyCallback.DigestRequest[105](byte[] digest, java.lang.String algorithm)
Methods	
java.lang.String	getAlgorithm[106]()
byte[]	getDigest[106]()

Inherited Member Summary
Methods inherited from class `Object`
`clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)`

Constructors

PrivateKeyCallback.DigestRequest(byte[] digest, `java.lang.String` algorithm)

```
public PrivateKeyCallback.DigestRequest(byte[] digest, java.lang.String algorithm)
```

Constructs a DigestRequest with a digest value and algorithm identifier.

The digest of the certificate whose private key is returned must match the provided digest. The certificate digest is computed by applying the specified algorithm to the bytes of the certificate. For example: `MessageDigest.getInstance(algorithm).digest(cert.getEncoded())` .

The corresponding certificate chain for the private key is also returned. If the digest or algorithm parameters are null, the handler of the callback relies on its own defaults.

Parameters:

`digest` - The digest value to use to select the corresponding certificate and private key (or null).

`algorithm` - A string value identifying the digest algorithm. The value passed to this parameter may be null. If it is not null, it must conform to the requirements for the algorithm parameter of `java.security.MessageDigest.getInstance()`.

Methods

getAlgorithm()

```
public java.lang.String getAlgorithm()
```

Get the algorithm identifier.

Returns: The identifer of the algorithm used to compute the digest.

getDigest()

```
public byte[] getDigest()
```

Get the digest value.

Returns: The digest value which must match the digest of the certificate corresponding to the returned private key.

javax.security.auth.message.callback

PrivateKeyCallback.IssuerSerialNumRequest

```
java.lang.Object
  |
  +--javax.security.auth.message.callback.PrivateKeyCallback.IssuerSerialNumRequest
```

All Implemented Interfaces: PrivateKeyCallback.Request[109]

Enclosing Class: PrivateKeyCallback[100]

Declaration

```
public static class PrivateKeyCallback.IssuerSerialNumRequest implements
    PrivateKeyCallback.Request[109]
```

Description

Request type for private keys that are identified using an issuer/serial number.

Member Summary	
Constructors	
	PrivateKeyCallback.IssuerSerialNumRequest[107](javax.security.auth.x500.X500Principal issuer, java.math.BigInteger serialNumber)
Methods	
javax.security.auth.x500.X500Principal	getIssuer[108]()
java.math.BigInteger	getSerialNum[108]()

Inherited Member Summary
Methods inherited from class `Object`
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)

Constructors

PrivateKeyCallback.IssuerSerialNumRequest(javax.security.auth.x500.X500Principa

l issuer, java.math.BigInteger serialNumber)

```
public PrivateKeyCallback.IssuerSerialNumRequest(javax.security.auth.x500.X500Principal
            issuer, java.math.BigInteger serialNumber)
```

Constructs a IssuerSerialNumRequest with an issuer/serial number.

The issuer/serial number is used to identify a public key certificate. The corresponding private key is returned in the callback. The corresponding certificate chain for the private key is also returned. If the issuer/serialNumber parameters are null, the handler of the callback relies on its own defaults.

Parameters:

`issuer` - The X500Principal name of the certificate issuer, or null.

`serialNumber` - The serial number of the certificate, or null.

Methods

getIssuer()

```
public javax.security.auth.x500.X500Principal getIssuer()
```

Get the issuer.

Returns: The issuer, or null.

getSerialNum()

```
public java.math.BigInteger getSerialNum()
```

Get the serial number.

Returns: The serial number, or null.

javax.security.auth.message.callback

PrivateKeyCallback.Request

All Known Implementing Classes: PrivateKeyCallback.AliasRequest[103], PrivateKeyCallback.DigestRequest[105], PrivateKeyCallback.IssuerSerialNumRequest[107], PrivateKeyCallback.SubjectKeyIDRequest[110]

Enclosing Class: **PrivateKeyCallback**[100]

Declaration

```
public static interface PrivateKeyCallback.Request
```

Description

Marker interface for private key request types.

javax.security.auth.message.callback
PrivateKeyCallback.SubjectKeyIDRequest

```
java.lang.Object
  |
  +--javax.security.auth.message.callback.PrivateKeyCallback.SubjectKeyIDRequest
```

All Implemented Interfaces: PrivateKeyCallback.Request[109]

Enclosing Class: **PrivateKeyCallback[100]**

Declaration

```
public static class PrivateKeyCallback.SubjectKeyIDRequest implements
    PrivateKeyCallback.Request[109]
```

Description

Request type for private keys that are identified using a SubjectKeyID

Member Summary	
Constructors	
	PrivateKeyCallback.SubjectKeyIDRequest[110](byte[] subjectKeyID)
Methods	
byte[]	getSubjectKeyID[111]()

Inherited Member Summary
Methods inherited from class `Object`
`clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)`

Constructors

PrivateKeyCallback.SubjectKeyIDRequest(byte[] subjectKeyID)

```
public PrivateKeyCallback.SubjectKeyIDRequest(byte[] subjectKeyID)
```

Construct a SubjectKeyIDRequest with an subjectKeyID.

The subjectKeyID is used to directly identify the private key to be returned. The corresponding certificate chain for the private key is also returned.

If the subjectKeyID is null, the handler of the callback relies on its own default.

Parameters:

`subjectKeyID` - Identifier for the private key, or null.

Methods

getSubjectKeyID()

```
public byte[] getSubjectKeyID()
```

Get the subjectKeyID.

Returns: The subjectKeyID, or null.

javax.security.auth.message.callback
SecretKeyCallback

```
java.lang.Object
  |
  +--javax.security.auth.message.callback.SecretKeyCallback
```

All Implemented Interfaces: `javax.security.auth.callback.Callback`

Declaration

```
public class SecretKeyCallback implements javax.security.auth.callback.Callback
```

Description

Callback for acquiring a shared secret from a key repository. This Callback may be used by client or server authentication modules to obtain shared secrets (for example, passwords) without relying on a user during the Callback processing. This Callback is typically empoyed by `ClientAuthModules` invoked from intermediate components that need to acquire a password to authenticate to their target service.

Member Summary

Nested Classes

static class	SecretKeyCallback.AliasRequest[114]
static	SecretKeyCallback.Request[116]

Constructors

	SecretKeyCallback[113](SecretKeyCallback.Request[116] request)

Methods

	getKey[113]()
javax.crypto.SecretKey	
	getRequest[113]()
SecretKeyCallback.Request[116]	
void	setKey[113](javax.crypto.SecretKey key)

Inherited Member Summary

Methods inherited from class `Object`

`clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)`

Constructors

SecretKeyCallback(SecretKeyCallback.Request[116] request)

```
public SecretKeyCallback(SecretKeyCallback.Request[116] request)
```

Constructs this SecretKeyCallback with a secret key Request object.

The *request* object identifies the secret key to be returned. If the alias is null, the handler of the callback relies on its own default.

Parameters:

`request` - Request object identifying the secret key, or null.

Methods

getKey()

```
public javax.crypto.SecretKey getKey()
```

Used to obtain the secret key set within the Callback.

Returns: The secret key, or null if no key was found.

getRequest()

```
public SecretKeyCallback.Request[116] getRequest()
```

Used by the CallbackHandler to get the Request object which identifies the secret key to be returned.

Returns: The Request object which identifies the private key to be returned, or null. If null, the handler of the callback relies on its own default.

setKey(javax.crypto.SecretKey key)

```
public void setKey(javax.crypto.SecretKey key)
```

Used by the CallbackHandler to set the requested secret key within the Callback.

Parameters:

`key` - The secret key, or null if no key was found.

javax.security.auth.message.callback
SecretKeyCallback.AliasRequest

```
java.lang.Object
  |
  +--javax.security.auth.message.callback.SecretKeyCallback.AliasRequest
```

All Implemented Interfaces: SecretKeyCallback.Request[116]

Enclosing Class: SecretKeyCallback[112]

Declaration

```
public static class SecretKeyCallback.AliasRequest implements SecretKeyCallback.Request[116]
```

Description

Request type for secret keys that are identified using an alias.

Member Summary	
Constructors	
	SecretKeyCallback.AliasRequest[114](java.lang.String alias)
Methods	
java.lang.String	getAlias[115]()

Inherited Member Summary
Methods inherited from class `Object`
`clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)`

Constructors

SecretKeyCallback.AliasRequest(java.lang.String alias)

```
public SecretKeyCallback.AliasRequest(java.lang.String alias)
```

Construct an AliasRequest with an alias.

The alias is used to directly identify the secret key to be returned.

If the alias is null, the handler of the callback relies on its own default.

Parameters:

`alias` - Name identifier for the secret key, or null.

Methods

getAlias()

```
public java.lang.String getAlias()
```

Get the alias.

Returns: The alias, or null.

javax.security.auth.message.callback
SecretKeyCallback.Request

All Known Implementing Classes: `SecretKeyCallback.AliasRequest`[114]

Enclosing Class: **`SecretKeyCallback`**[112]

Declaration

```
public static interface SecretKeyCallback.Request
```

Description

Marker interface for secret key request types.

javax.security.auth.message.callback
TrustStoreCallback

```
java.lang.Object
  |
  +--javax.security.auth.message.callback.TrustStoreCallback
```

All Implemented Interfaces: `javax.security.auth.callback.Callback`

Declaration

```
public class TrustStoreCallback implements javax.security.auth.callback.Callback
```

Description

Callback for trusted certificate KeyStore.

A trusted certificate KeyStore may be used to determine whether a given certificate chain can be trusted.

Member Summary	
Constructors	
	TrustStoreCallback[117]()
Methods	
	getTrustStore[118]()
java.security.KeyStore	
void	setTrustStore[118](java.security.KeyStore trustStore)

Inherited Member Summary
Methods inherited from class `Object`
`clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)`

Constructors

TrustStoreCallback()

```
public TrustStoreCallback()
```

Create a TrustStoreCallback.

Methods

getTrustStore()

```
public java.security.KeyStore getTrustStore()
```

Used by the TrustStore user to obtain the TrustStore set within the Callback.

Returns: The trusted certificate KeyStore. The KeyStore is guaranteed to already be loaded.

setTrustStore(`java.security.KeyStore` trustStore)

```
public void setTrustStore(java.security.KeyStore trustStore)
```

Used by the CallbackHandler to set the trusted certificate keystore within the Callback.

Parameters:

`trustStore` - The trusted certificate KeyStore, which must already be loaded.

Package javax.security.auth.message.config

Description

This package defines the interfaces implemented by JSR 196 compatible configuration systems.

Class Summary

Interfaces

AuthConfig 120	This interface defines the common functionality implemented by Authentication context configuration objects.
AuthConfigFactory.RegistrationContext 130	Represents the layer identifier, application context identifier, and description components of an AuthConfigProvider registration at the factory.
AuthConfigProvider 132	This interface is implemented by objects that can be used to obtain authentication context configuration objects, that is, `ClientAuthConfig` or `ServerAuthConfig` objects.
ClientAuthConfig 135	This interface encapsulates the configuration of ClientAuthContext objects for a message layer and application context (for example, the messaging context of a specific application, or set of applications).
ClientAuthContext 137	This ClientAuthContext class encapsulates ClientAuthModules that are used to secure service requests made by a client, and to validate any responses received to those requests.
RegistrationListener 138	An implementation of this interface may be associated with an AuthConfigProvider registration at an AuthConfigFactory at the time the AuthConfigProvider is obtained for use from the factory.
ServerAuthConfig 139	This interface describes a configuration of ServerAuthConfiguration objects for a message layer and application context (for example, the messaging context of a specific application, or set of applications).
ServerAuthContext 141	This ServerAuthContext class encapsulates ServerAuthModules that are used to validate service requests received from clients, and to secure any response returned for those requests.

Classes

AuthConfigFactory 122	This class is used to obtain `AuthConfigProvider` objects that can be used to obtain authentication context configuration objects, that is, `ClientAuthConfig` and `ServerAuthConfig` objects.

javax.security.auth.message.config
AuthConfig

All Known Subinterfaces: ClientAuthConfig[135], ServerAuthConfig[139]

Declaration
```
public interface AuthConfig
```

Description
This interface defines the common functionality implemented by Authentication context configuration objects.

See Also: ClientAuthContext[137], ServerAuthContext[141]

Member Summary	
Methods	
java.lang.String	getAppContext[120]()
java.lang.String	getAuthContextID[120](MessageInfo[76] messageInfo)
java.lang.String	getMessageLayer[121]()
boolean	isProtected[121]()
void	refresh[121]()

Methods

getAppContext()

```
public java.lang.String getAppContext()
```

Get the application context identifier of this authentication context configuration object.

Returns: The String identifying the application context of this configuration object, or null if the configuration object pertains to an unspecified application context.

getAuthContextID(MessageInfo[76] messageInfo)

```
public java.lang.String getAuthContextID(MessageInfo[76] messageInfo)
```

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

Parameters:
`messageInfo` - A contextual Object that encapsulates the client request and server response objects.

Returns: The authentication context identifier corresponding to the encapsulated request and response objects, or null.

Throws:

`java.lang.IllegalArgumentException` - If the type of the message objects incorporated in messageInfo are not compatible with the message types supported by this authentication context configuration object.

getMessageLayer()

```
public java.lang.String getMessageLayer()
```

Get the message layer name of this authentication context configuration object.

Returns: The message layer name of this configuration object, or null if the configuration object pertains to an unspecified message layer.

isProtected()

```
public boolean isProtected()
```

Used to determine whether the authentication context configuration object encapsulates any protected authentication contexts.

Returns: True if the configuration object encapsulates at least one protected authentication context. Otherwise, this method returns false.

refresh()

```
public void refresh()
```

Causes a dynamic anthentication context configuration object to update the internal state that it uses to process calls to its `getAuthContext` method.

Throws:

`AuthException`[69] - If an error occured during the update.

`java.lang.SecurityException` - If the caller does not have permission to refresh the configuration object.

javax.security.auth.message.config
AuthConfigFactory

```
java.lang.Object
  |
  +--javax.security.auth.message.config.AuthConfigFactory
```

Declaration

```
public abstract class AuthConfigFactory
```

Description

This class is used to obtain `AuthConfigProvider` objects that can be used to obtain authentication context configuration objects, that is, `ClientAuthConfig` and `ServerAuthConfig` objects.

Authentication context configuration objects are used to obtain authentication context objects. Authentication context objects, that is, `ClientAuthContext` and `ServerAuthContex` objects, encapsulate authentication modules. Authentication modules are pluggable components that perform security-related processing of request and response messages.

Callers do not operate on modules directly. Instead they rely on an authentication context to manage the invocation of modules. A caller obtains an authentication context by calling the `getAuthContext` method on a `ClientAuthConfig` or `ServerAuthConfig` obtained from an AuthConfigProvider.

The following represents a typical sequence of calls for obtaining a client authentication context, and then using it to secure a request.

1. AuthConfigFactory factory = AuthConfigFactory.getFactory();
2. AuthConfigProvider provider = factory.getConfigProvider(layer,appID,listener);
3. ClientAuthConfig config = provider.getClientAuthConfig(layer,appID,cbh)
4. String authContextID = config.getAuthContextID(messageInfo);
5. ClientAuthContext context = config.getAuthContext(authContextID,subject,properties);
6. context.secureRequest(messageInfo,subject);

A system-wide AuthConfigFactory implementation can be set by invoking `setFactory`, and retrieved using `getFactory`.

Every implementation of this abstract class must offer a public, zero argument constructor. This constructor must support the construction and registration (including self-registration) of AuthConfigProviders from a persistent declarative representation. For example, a factory implementation class could interpret the contents of a file containing a sequence of configuration entries, with one entry per AuthConfigProvider, and with each entry representing:

- The fully qualified name of the provider implementation class (or null)
- The list of provider initialization properties (which could be empty)

Any provider initialization properties must be specified in a form that can be passed to the provider constructor within a Map of key, value pairs, and where all keys and values within the Map are of type String.

The entry syntax must also provide for the optional inclusion of information sufficient to define a RegistrationContext. This information would only be present when the factory will register the provider. For

example, each entry could provide for the inclusion of one or more RegistrationContext objects of the following form:

- The message layer name (or null)
- The application context identifier (or null)
- The registration description (or null)

When a RegistrationContext is not included, the factory must make it convenient for the provider to self-register with the factory during the provider construction (see `registerConfigProvider(AuthConfigProvider provider, ...)`).

An AuthConfigFactory implementation is free to choose is own persistent declarative syntax as long as it conforms to the requirements defined by this class.

See Also: `ClientAuthContext`[137], `ServerAuthContext`[141], `ClientAuthConfig`[135], `ServerAuthConfig`[139], `java.util.Properties`

Member Summary

Nested Classes

static	`AuthConfigFactory.RegistrationContext`[130]

Fields

static java.lang.String	`DEFAULT_FACTORY_SECURITY_PROPERTY`[124]

Constructors

	`AuthConfigFactory`[124]()

Methods

abstract java.lang.String[]	`detachListener`[124](`RegistrationListener`[138] listener, java.lang.String layer, java.lang.String appContext)
abstract AuthConfigProvider[132]	`getConfigProvider`[124](java.lang.String layer, java.lang.String appContext, `RegistrationListener`[138] listener)
static AuthConfigFactory[122]	`getFactory`[125]()
abstract AuthConfigFactory.RegistrationContext[130]	`getRegistrationContext`[126](java.lang.String registrationID)
abstract java.lang.String[]	`getRegistrationIDs`[126](`AuthConfigProvider`[132] provider)
abstract void	`refresh`[126]()
abstract java.lang.String	`registerConfigProvider`[127](`AuthConfigProvider`[132] provider, java.lang.String layer, java.lang.String appContext, java.lang.String description)
abstract java.lang.String	`registerConfigProvider`[126](java.lang.String className, java.util.Map properties, java.lang.String layer, java.lang.String appContext, java.lang.String description)
abstract boolean	`removeRegistration`[128](java.lang.String registrationID)
static void	`setFactory`[129](`AuthConfigFactory`[122] factory)

Inherited Member Summary
Methods inherited from class `Object`
`clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(long), wait(long, int)`

Fields

DEFAULT_FACTORY_SECURITY_PROPERTY

```
public static final java.lang.String DEFAULT_FACTORY_SECURITY_PROPERTY
```

The name of the Security property used to define the default AuthConfigFactory implementation class.

Constructors

AuthConfigFactory()

```
public AuthConfigFactory()
```

Methods

detachListener(RegistrationListener[138] listener, java.lang.String layer, java.lang.String appContext)

```
public abstract java.lang.String[] detachListener(RegistrationListener[138] listener,
            java.lang.String layer, java.lang.String appContext)
```

Disassociate the listener from all the provider registrations whose layer and appContext values are matched by the corresponding arguments to this method.

Factories should periodically notify Listeners to effectively detach listeners that are no longer in use.

Parameters:

`listener` - The RegistrationListener to be detached.

`layer` - A String identifying the message layer or null.

`appContext` - A String value identifying the application contex or null.

Returns: An array of String values where each value identifies a provider registration from which the listener was removed. This method never returns null; it returns an empty array if the listener was not removed from any registrations.

Throws:

`java.lang.SecurityException` - If the caller does not have permission to detach the listener from the factory.

getConfigProvider(java.lang.String layer, java.lang.String appContext, RegistrationListener[138] listener)

```
public abstract AuthConfigProvider[132] getConfigProvider(java.lang.String layer,
            java.lang.String appContext, RegistrationListener[138] listener)
```

Get a registered AuthConfigProvider from the factory. Get the provider of ServerAuthConfig and ClientAuthConfig objects registered for the identified message layer and application context.

All factories shall employ the following precedence rules to select the registered AuthConfigProvider that matches the layer and appContext arguments:

- The provider that is specifically registered for both the corresponding message layer and appContext shall be selected.
- If no provider is selected according to the preceding rule, the provider specifically registered for the corresponding appContext and for all message layers shall be selected.
- If no provider is selected according to the preceding rules, the provider specifically registered for the corresponding message layer and for all appContexts shall be selected.
- If no provider is selected according to the preceding rules, the provider registered for all message layers and for all appContexts shall be selected.
- If no provider is selected according to the preceding rules, the factory shall terminate its search for a registered provider.

The above precedence rules apply equivalently to registrations created with a null or non-null `className` argument.

Parameters:

`layer` - A String identifying the message layer for which the registered AuthConfigProvider is to be returned. This argument may NOT be null.

`appContext` - A String that identifies the application messaging context for which the registered AuthConfigProvider is to be returned. This argument may NOT be null.

`listener` - The RegistrationListener whose `notify` method is to be invoked if the corresponding registration is unregistered or replaced. The value of this argument may be null.

Returns: The implementation of the AuthConfigProvider interface registered at the factory for the layer and appContext, or null if no AuthConfigProvider is selected. An argument listener is attached even if the return value is null.

getFactory()

```
public static AuthConfigFactory122 getFactory()
```

Get the system-wide AuthConfigFactory implementation.

If a non-null system-wide factory instance is defined at the time of the call, for example, with `setFactory`, it will be returned. Otherwise, an attempt will be made to construct an instance of the default AuthConfigFactory implementation class. The fully qualified class name of the default factory implementation class is obtained from the value of the authconfigprovider.factory security property. When an instance of the default factory implementation class is successfully constructed by this method, this method will set it as the system-wide factory instance.

The absolute pathname of the Java security properties file is JAVA_HOME/lib/security/java.security, where JAVA_HOME refers to the directory where the JDK was installed.

Returns: The non-null system-wide AuthConfigFactory instance set at the time of the call, or if that value was null, the value of the system-wide factory instance established by this method. This method returns null when the system-wide factory was not defined when this method was called and no default factory name was defined via the security property.

Throws:

AuthException[69] - If an error occurred during the class loading, or construction of the default AuthConfigFactory implementation class.

java.lang.SecurityException - If the caller does not have permission to retrieve the factory, or set it as the system-wide instance.

getRegistrationContext(java.lang.String registrationID)

```
public abstract AuthConfigFactory.RegistrationContext[130]
            getRegistrationContext(java.lang.String registrationID)
```

Get the the registration context for the identified registration.

Parameters:

registrationID - A String that identifies a provider registration at the factory

Returns: A RegistrationContext or null. When a Non-null value is returned, it is a copy of the registration context corresponding to the registration. Null is returned when the registration identifier does not correpond to an active registration

getRegistrationIDs(AuthConfigProvider[132] provider)

```
public abstract java.lang.String[] getRegistrationIDs(AuthConfigProvider[132] provider)
```

Get the registration identifiers for all registrations of the provider instance at the factory.

Parameters:

provider - The AuthConfigurationProvider whose registration identifiers are to be returned. This argument may be null, in which case it indicates that the the IDs of all active registrations within the factory are to be returned.

Returns: An array of String values where each value identifies a provider registration at the factory. This method never returns null; it returns an empty array when there are no registrations at the factory for the identified provider.

refresh()

```
public abstract void refresh()
```

Cause the factory to reprocess its persisent declarative representation of provider registrations.

A factory should only replace an existing registration when a change of provider implementation class or initialization properties has occurred.

Throws:

AuthException[69] - If an error occured during the reinitialization.

java.lang.SecurityException - If the caller does not have permission to refresh the factory.

registerConfigProvider(java.lang.String className, java.util.Map properties, java.lang.String layer, java.lang.String appContext, java.lang.String description)

```
public abstract java.lang.String registerConfigProvider(java.lang.String className,
            java.util.Map properties, java.lang.String layer, java.lang.String
            appContext, java.lang.String description)
```

Registers within the factory and records within the factory's persistent declarative representation of provider registrations a provider of ServerAuthConfig and/or ClientAuthConfig objects for a message layer and application context identifier. This method typically constructs an instance of the provider before registering it with the factory. Factories may extend or modify the persisted registrations of existing provider instances, if those instances were registered with ClassName and properties arguments equivalent to those passed in the current call.

This method employs the two argument constructor required to be supported by every implementation of the AuthConfigProvider interface, and this method must pass a null value for the factory argument of the constructor. `AuthConfigProviderImpl AuthConfigProviderImpl(Map properties, AuthConfigFactory factory)`.

At most one registration may exist within the factory for a given combination of message layer and appContext. Any pre-existing registration with identical values for layer and appContext is replaced by a subsequent registration. When replacement occurs, the registration identifier, layer, and appContext identifier remain unchanged, and the AuthConfigProvider (with initialization properties) and description are replaced.

Within the lifetime of its Java process, a factory must assign unique registration identifiers to registrations, and must never assign a previously used registration identifier to a registration whose message layer and or appContext identifier differ from the previous use.

Programmatic registrations performed by using this method must update (according to the replacement rules described above) the persistent declarative representation of provider registrations employed by the factory constructor.

Parameters:

`className` - The fully qualified name of an AuthConfigProvider implementation class (or null). Calling this method with a null value for this parameter shall cause `getConfigProvider` to return null when it is called with layer and appContext values for which the resulting registration is the best match.

`properties` - A Map object containing the initialization properties to be passed to the properties argument of the provider constructor. This argument may be null. When this argument is not null, all the values and keys occuring in the Map must be of type String.

`layer` - A String identifying the message layer for which the provider will be registered at the factory. A null value may be passed as an argument for this parameter, in which case the provider is registered at all layers.

`appContext` - A String value that may be used by a runtime to request a configuration object from this provider. A null value may be passed as an argument for this parameter, in which case the provider is registered for all configuration ids (at the indicated layers).

`description` - A text String describing the provider. This value may be null.

Returns: A String identifier assigned by the factory to the provider registration, and that may be used to remove the registration from the factory.

Throws:

`java.lang.SecurityException` - If the caller does not have permission to register a provider at the factory.

`AuthException`[69] - If the provider construction (given a non-null `className`) or registration fails.

registerConfigProvider(`AuthConfigProvider`[132] provider, `java.lang.String` layer,

java.lang.String appContext, java.lang.String description)

```
public abstract java.lang.String registerConfigProvider(AuthConfigProvider132 provider,
            java.lang.String layer, java.lang.String appContext, java.lang.String
            description)
```

Registers within the (in-memory) factory, a provider of ServerAuthConfig and/or ClientAuthConfig objects for a message layer and application context identifier. This method does NOT effect the factory's persistent declarative representation of provider registrations, and is intended to be used by providers to perform self-Registration.

At most one registration may exist within the factory for a given combination of message layer and appContext. Any pre-existing registration with identical values for layer and appContext is replaced by a subsequent registration. When replacement occurs, the registration identifier, layer, and appContext identifier remain unchanged, and the AuthConfigProvider (with initialization properties) and description are replaced.

Within the lifetime of its Java process, a factory must assign unique registration identifiers to registrations, and must never assign a previously used registration identifier to a registration whose message layer and or appContext identifier differ from the previous use.

Parameters:

`provider` - The AuthConfigProvider to be registered at the factory (or null). Calling this method with a null value for this parameter shall cause `getConfigProvider` to return null when it is called with layer and appContext values for which the resulting registration is the best match.

`layer` - A String identifying the message layer for which the provider will be registered at the factory. A null value may be passed as an argument for this parameter, in which case the provider is registered at all layers.

`appContext` - A String value that may be used by a runtime to request a configuration object from this provider. A null value may be passed as an argument for this parameter, in which case the provider is registered for all configuration ids (at the indicated layers).

`description` - A text String describing the provider. This value may be null.

Returns: A String identifier assigned by the factory to the provider registration, and that may be used to remove the registration from the factory.

Throws:

`java.lang.SecurityException` - If the caller does not have permission to register a provider at the factory.

`AuthException`69 - If the provider registration fails.

removeRegistration(java.lang.String registrationID)

```
public abstract boolean removeRegistration(java.lang.String registrationID)
```

Remove the identified provider registration from the factory (and from the persistent declarative representation of provider registrations, if appropriate) and invoke any listeners associated with the removed registration.

Parameters:

`registrationID` - A String that identifies a provider registration at the factory

Returns: True if there was a registration with the specified identifier and it was removed. Return false if the registrationID was invalid.

Throws:

`java.lang.SecurityException` - If the caller does not have permission to unregister the provider at the factory.

setFactory(AuthConfigFactory[122] factory)

```
public static void setFactory(AuthConfigFactory[122] factory)
```

Set the system-wide AuthConfigFactory implementation.

If an implementation was set previously, it will be replaced.

Listeners are not notified of a change to the registered factory.

Parameters:

`factory` - The AuthConfigFactory instance, which may be null.

Throws:

`java.lang.SecurityException` - If the caller does not have permission to set the factory.

javax.security.auth.message.config

AuthConfigFactory.RegistrationContext

Enclosing Class: AuthConfigFactory[122]

Declaration

```
public static interface AuthConfigFactory.RegistrationContext
```

Description

Represents the layer identifier, application context identifier, and description components of an AuthConfigProvider registration at the factory.

Member Summary	
Methods	
java.lang.String	getAppContext[130]()
java.lang.String	getDescription[130]()
java.lang.String	getMessageLayer[130]()
boolean	isPersistent[131]()

Methods

getAppContext()

```
public java.lang.String getAppContext()
```

Get the application context identifier from the registration context

Returns: A String identifying the application context for which the AuthConfigProvider was registered. The returned value may be null.

getDescription()

```
public java.lang.String getDescription()
```

Get the description from the registration context

Returns: The description String from the registration, or null if no description string was included in the registration.

getMessageLayer()

```
public java.lang.String getMessageLayer()
```

Get the layer name from the registration context

Returns: A String identifying the message layer for which the AuthConfigProvider was registered. The returned value may be null.

isPersistent()

```
public boolean isPersistent()
```

Get the persisted status from the registration context.

Returns: A boolean indicating whether the registration is the result of a className based registration, or an instance-based (for example, self-) registration. Only registrations performed using the five argument `registerConfigProvider` method are persistent.

javax.security.auth.message.config
AuthConfigProvider

Declaration

```
public interface AuthConfigProvider
```

Description

This interface is implemented by objects that can be used to obtain authentication context configuration objects, that is, `ClientAuthConfig` or `ServerAuthConfig` objects.

Authentication context configuration objects serve as sources of the authentication context objects, that is, `ClientAuthContext` or `ServerAuthContext` objects, for a specific message layer and messaging context.

Authentication context objects encapsulate the initialization, configuration, and invocation of authentication modules, that is, `ClientAuthModule` or `ServerAuthModule` objects, for a specific message exchange within a specific message layer and messaging context.

Callers do not directly operate on authentication modules. Instead, they rely on a ClientAuthContext or ServerAuthContext to manage the invocation of modules. A caller obtains an instance of ClientAuthContext or ServerAuthContext by calling the respective `getAuthContext` method on a `ClientAuthConfig` or `ServerAuthConfig` object obtained from an AuthConfigProvider.

The following represents a typical sequence of calls for obtaining a client authentication context object, and then using it to secure a request.

1. AuthConfigProvider provider;
2. ClientAuthConfig config = provider.getClientAuthConfig(layer,appID,cbh);
3. String authContextID = config.getAuthContextID(messageInfo);
4. ClientAuthContext context = config.getAuthContext(authContextID,subject,properties);
5. context.secureRequest(messageInfo,subject);

Every implementation of this interface must offer a public, two argument constructor with the following signature:

```
public AuthConfigProviderImpl(Map properties, AuthConfigFactory factory);
```

where the properties argument may be null, and where all values and keys occurring in a non-null properties argument must be of type String. When the factory argument is not null, it indicates that the provider is to self-register at the factory by calling the following method on the factory:

```
public String
registerConfigProvider(AuthConfigProvider provider, String layer,
                       String appContext, String description);
```

See Also: `ClientAuthContext`[137], `ServerAuthContext`[141], `AuthConfigFactory`[122]

Member Summary	
Methods	
ClientAuthConfig[135]	getClientAuthConfig[133](java.lang.String layer, java.lang.String appContext, javax.security.auth.callback.CallbackHandler handler)
ServerAuthConfig[139]	getServerAuthConfig[133](java.lang.String layer, java.lang.String appContext, javax.security.auth.callback.CallbackHandler handler)
void	refresh[134]()

Methods

getClientAuthConfig(java.lang.String layer, java.lang.String appContext, javax.security.auth.callback.CallbackHandler handler)

```
public ClientAuthConfig[135] getClientAuthConfig(java.lang.String layer, java.lang.String
            appContext, javax.security.auth.callback.CallbackHandler handler)
            throws AuthException
```

Get an instance of ClientAuthConfig from this provider.

The implementation of this method returns a ClientAuthConfig instance that describes the configuration of ClientAuthModules at a given message layer, and for use in an identified application context.

Parameters:

`layer` - A String identifying the message layer for the returned ClientAuthConfig object. This argument must not be null.

`appContext` - A String that identifies the messaging context for the returned ClientAuthConfig object. This argument must not be null.

`handler` - A CallbackHandler to be passed to the ClientAuthModules encapsulated by ClientAuthContext objects derived from the returned ClientAuthConfig. This argument may be null, in which case the implementation may assign a default handler to the configuration. The CallbackHandler assigned to the configuration must support the Callback objects required to be supported by the profile of this specification being followed by the messaging runtime. The CallbackHandler instance must be initialized with any application context needed to process the required callbacks on behalf of the corresponding application.

Returns: A ClientAuthConfig Object that describes the configuration of ClientAuthModules at the message layer and messaging context identified by the layer and appContext arguments. This method does not return null.

Throws:

AuthException[69] - If this provider does not support the assignment of a default CallbackHandler to the returned ClientAuthConfig.

`java.lang.SecurityException` - If the caller does not have permission to retrieve the configuration.

getServerAuthConfig(java.lang.String layer, java.lang.String appContext, javax.security.auth.callback.CallbackHandler handler)

```
public ServerAuthConfig[139] getServerAuthConfig(java.lang.String layer, java.lang.String
            appContext, javax.security.auth.callback.CallbackHandler handler)
            throws AuthException
```

Get an instance of ServerAuthConfig from this provider.

The implementation of this method returns a ServerAuthConfig instance that describes the configuration of ServerAuthModules at a given message layer, and for a particular application context.

Parameters:

`layer` - A String identifying the message layer for the returned ServerAuthConfig object. This argument must not be null.

`appContext` - A String that identifies the messaging context for the returned ServerAuthConfig object. This argument must not be null.

`handler` - A CallbackHandler to be passed to the ServerAuthModules encapsulated by ServerAuthContext objects derived from the returned ServerAuthConfig. This argument may be null, in which case the implementation may assign a default handler to the configuration. The CallbackHandler assigned to the configuration must support the Callback objects required to be supported by the profile of this specification being followed by the messaging runtime. The CallbackHandler instance must be initialized with any application context needed to process the required callbacks on behalf of the corresponding application.

Returns: A ServerAuthConfig Object that describes the configuration of ServerAuthModules at a given message layer, and for a particular application context. This method does not return null.

Throws:

`AuthException`[69] - If this provider does not support the assignment of a default CallbackHandler to the returned ServerAuthConfig.

`java.lang.SecurityException` - If the caller does not have permission to retrieve the configuration.

refresh()

```
public void refresh()
```

Causes a dynamic configuration provider to update its internal state such that any resulting change to its state is reflected in the corresponding authentication context configuration objects previously created by the provider within the current process context.

Throws:

`AuthException`[69] - If an error occured during the refresh.

`java.lang.SecurityException` - If the caller does not have permission to refresh the provider.

javax.security.auth.message.config
ClientAuthConfig

All Superinterfaces: AuthConfig[120]

Declaration
```
public interface ClientAuthConfig extends AuthConfig[120]
```

Description
This interface encapsulates the configuration of ClientAuthContext objects for a message layer and application context (for example, the messaging context of a specific application, or set of applications).

Implementations of this interface are returned by an AuthConfigProvider.

Callers interact with a ClientAuthConfig to obtain ClientAuthContext objects suitable for processing a given message exchange at the layer and within the application context of the ClientAuthConfig. Each ClientAuthContext object is responsible for instantiating, initializing, and invoking the one or more ClientAuthModules encapsulated in the ClientAuthContext.

After having acquired a ClientAuthContext, a caller operates on the context to cause it to invoke the encapsulated ClientAuthModules to secure client requests and to validate server responses.

See Also: AuthConfigProvider[132]

Member Summary	
Methods	
ClientAuthContext[137]	getAuthContext[135](java.lang.String authContextID, javax.security.auth.Subject clientSubject, java.util.Map properties)

Inherited Member Summary
Methods inherited from interface AuthConfig[120]
getAppContext()[120], getAuthContextID(MessageInfo)[120], getMessageLayer()[121], isProtected()[121], refresh()[121]

Methods

getAuthContext(java.lang.String authContextID, javax.security.auth.Subject

clientSubject, `java.util.Map` properties)

```
public ClientAuthContext137 getAuthContext(java.lang.String authContextID,
            javax.security.auth.Subject clientSubject, java.util.Map properties)
            throws AuthException
```

Get a ClientAuthContext instance from this ClientAuthConfig.

The implementation of this method returns a ClientAuthContext instance that encapsulates the ClientAuthModules used to secure and validate requests/responses associated with the given *authContextID*.

Specifically, this method accesses this ClientAuthConfig object with the argument *authContextID* to determine the ClientAuthModules that are to be encapsulated in the returned ClientAuthContext instance.

The ClientAuthConfig object establishes the request and response MessagePolicy objects that are passed to the encapsulated modules when they are initialized by the returned ClientAuthContext instance. It is the modules' responsibility to enforce these policies when invoked.

Parameters:

`authContextID` - An String identifier used to index the provided *config*, or null. This value must be identical to the value returned by the `getAuthContextID` method for all `MessageInfo` objects passed to the `secureRequest` method of the returned ClientAuthContext.

`clientSubject` - A Subject that represents the source of the service request to be secured by the acquired authentication context. The principals and credentials of the Subject may be used to select or acquire the authentication context. If the Subject is not null, additional Principals or credentials (pertaining to the source of the request) may be added to the Subject. A null value may be passed for this parameter.

`properties` - A Map object that may be used by the caller to augment the properties that will be passed to the encapsulated modules at module initialization. The null value may be passed for this parameter.

Returns: A ClientAuthContext instance that encapsulates the ClientAuthModules used to secure and validate requests/responses associated with the given *authContextID*, or null (indicating that no modules are configured).

Throws:

`AuthException`69 - If this method fails.

javax.security.auth.message.config
ClientAuthContext

All Superinterfaces: ClientAuth 73

Declaration

```
public interface ClientAuthContext extends ClientAuth 73
```

Description

This ClientAuthContext class encapsulates ClientAuthModules that are used to secure service requests made by a client, and to validate any responses received to those requests. A caller typically uses this class in the following manner:

1. Retrieve an instance of this class by using ClientAuthConfig.getAuthContext.
2. Invoke *secureRequest*.
   ClientAuthContext implementation invokes secureRequest of one or more encapsulated ClientAuthModules. Modules might attach credentials to request (for example, a user name and password), and/or secure the request (for example, sign and encrypt the request).
3. Send request and receive response.
4. Invoke *validateResponse*.
   ClientAuthContext implementation invokes validateResponse of one or more encapsulated ClientAuthModules. Modules verify or decrypt response as necessary.
5. Invoke *cleanSubject* method (as necessary) to clean up any authentication state in Subject.

A ClientAuthContext instance may be used concurrently by multiple callers.

Implementations of this interface are responsible for constructing and initializing the encapsulated modules. The initialization step includes passing the relevant request and response MessagePolicy objects to the encapsulated modules. The MessagePolicy objects are obtained by the ClientAuthConfig instance used to obtain the ClientAuthContext object. See `ClientAuthConfig.getAuthContext` for more information.

Implementations of this interface are instantiated by their associated configuration object such that they know which modules to invoke, in what order, and how results returned by preceding modules are to influence subsequent module invocations.

Calls to the inherited methods of this interface delegate to the corresponding methods of the encapsulated authentication modules.

See Also: ClientAuthConfig 135, ClientAuthModule 144

Inherited Member Summary

Methods inherited from interface ClientAuth 73

cleanSubject(MessageInfo, Subject) 73, secureRequest(MessageInfo, Subject) 73, validateResponse(MessageInfo, Subject, Subject) 74

javax.security.auth.message.config
RegistrationListener

Declaration

```
public interface RegistrationListener
```

Description

An implementation of this interface may be associated with an AuthConfigProvider registration at an AuthConfigFactory at the time the AuthConfigProvider is obtained for use from the factory. The AuthConfigFactory will invoke the notify method of the RegistrationListener if the corresponding provider registration is unregistered or replaced at the factory.

Member Summary	
Methods	
	void notify138(java.lang.String layer, java.lang.String appContext)

Methods

notify(java.lang.String layer, java.lang.String appContext)

```
public void notify(java.lang.String layer, java.lang.String appContext)
```

Notify the listener that a registration with which it was associated was replaced or unregistered.

When a RegistrationListener is associated with a provider registration within the factory, the factory must call its `notify` method when the corresponding registration is unregistered or replaced.

Parameters:

`layer` - A String identifying the one or more message layers corresponding to the registration for which the listerner is being notified.

`appContext` - A String value identifying the application contexts corresponding to the registration for which the listener is being notified. The factory detaches the listener from the corresponding registration once the listener has been notified for the registration. The `detachListerner` method must be called to detach listeners that are no longer in use.

javax.security.auth.message.config
ServerAuthConfig

All Superinterfaces: AuthConfig[120]

Declaration
```
public interface ServerAuthConfig extends AuthConfig[120]
```

Description
This interface describes a configuration of ServerAuthConfiguration objects for a message layer and application context (for example, the messaging context of a specific application, or set of applications).

Implementations of this interface are returned by an AnthConfigProvider.

Callers interact with a ServerAuthConfig to obtain ServerAuthContext objects suitable for processing a given message exchange at the layer and within the application context of the ServerAuthConfig. Each ServerAuthContext object is responsible for instantiating, initializing, and invoking the one or more ServerAuthModules encapsulated in the ServerAuthContext.

After having acquired a ServerAuthContext, a caller operates on the context to cause it to invoke the encapsulated ServerAuthModules to validate service requests and to secure service responses.

See Also: AuthConfigProvider[132]

Member Summary	
Methods	
ServerAuthContext[141]	getAuthContext[139](java.lang.String authContextID, javax.security.auth.Subject serviceSubject, java.util.Map properties)

Inherited Member Summary
Methods inherited from interface AuthConfig[120]
getAppContext()[120], getAuthContextID(MessageInfo)[120], getMessageLayer()[121], isProtected()[121], refresh()[121]

Methods

getAuthContext(java.lang.String authContextID, javax.security.auth.Subject

serviceSubject, `java.util.Map` properties)

```
public ServerAuthContext141 getAuthContext(java.lang.String authContextID,
            javax.security.auth.Subject serviceSubject, java.util.Map properties)
            throws AuthException
```

Get a ServerAuthContext instance from this ServerAuthConfig.

The implementation of this method returns a ServerAuthContext instance that encapsulates the ServerAuthModules used to validate requests and secure responses associated with the given *authContextID*.

Specifically, this method accesses this ServerAuthConfig object with the argument *authContextID* to determine the ServerAuthModules that are to be encapsulated in the returned ServerAuthContext instance.

The ServerAuthConfig object establishes the request and response MessagePolicy objects that are passed to the encapsulated modules when they are initialized by the returned ServerAuthContext instance. It is the modules' responsibility to enforce these policies when invoked.

Parameters:

`authContextID` - An identifier used to index the provided *config*, or null. This value must be identical to the value returned by the `getAuthContextID` method for all `MessageInfo` objects passed to the `validateRequest` method of the returned ServerAuthContext.

`serviceSubject` - A Subject that represents the source of the service response to be secured by the acquired authentication context. The principal and credentials of the Subject may be used to select or acquire the authentication context. If the Subject is not null, additional Principals or credentials (pertaining to the source of the response) may be added to the Subject. A null value may be passed for this parameter.

`properties` - A Map object that may be used by the caller to augment the properties that will be passed to the encapsulated modules at module initialization. The null value may be passed for this parameter.

Returns: A ServerAuthContext instance that encapsulates the ServerAuthModules used to secure and validate requests/responses associated with the given *authContextID*, or null (indicating that no modules are configured).

Throws:

`AuthException`69 - If this method fails.

javax.security.auth.message.config
ServerAuthContext

All Superinterfaces: ServerAuth[86]

Declaration

```
public interface ServerAuthContext extends ServerAuth[86]
```

Description

This ServerAuthContext class encapsulates ServerAuthModules that are used to validate service requests received from clients, and to secure any response returned for those requests. A caller typically uses this class in the following manner:

1. Retrieve an instance of this class via ServerAuthConfig.getAuthContext.
2. Invoke *validateRequest*.
   ServerAuthContext implementation invokes validateRequest of one or more encapsulated ServerAuthModules. Modules validate credentials present in request (for example, decrypt and verify a signature).
3. If credentials valid and sufficient, authentication complete.
   Perform authorization check on authenticated identity and, if successful, dispatch to requested service application.
4. Service application finished.
5. Invoke *secureResponse*.
   ServerAuthContext implementation invokes secureResponse of one or more encapsulated ServerAuthModules. Modules secure response (sign and encrypt response, for example), and prepare response message.
6. Send secured response to client.
7. Invoke *cleanSubject* (as necessary) to clean up any authentication state in Subject(s).

A ServerAuthContext instance may be used concurrently by multiple callers.

Implementations of this interface are responsible for constructing and initializing the encapsulated modules. The initialization step includes passing the relevant request and response MessagePolicy objects to the encapsulated modules. The MessagePolicy objects are obtained by the ServerAuthConfig instance used to ontain the ServerAuthContext object. See `ServerAuthConfig.getAuthContext` for more information.

Implementations of this interface are instantiated by their associated configuration object such that they know which modules to invoke, in what order, and how results returned by preceding modules are to influence subsequent module invocations.

Calls to the inherited methods of this interface delegate to the corresponding methods of the encapsulated authentication modules.

See Also: ServerAuthConfig[139], ServerAuthModule[146]

Inherited Member Summary

Methods inherited from interface ServerAuth[86]

cleanSubject(MessageInfo, Subject)[86], secureResponse(MessageInfo, Subject)[86], validateRequest(MessageInfo, Subject, Subject)[87]

Package javax.security.auth.message.module

Description

This package defines the interfaces implemented by JSR 196 compatible authentication modules.

Class Summary	
Interfaces	
ClientAuthModule 144	A ClientAuthModule secures request messages, and validates received response messages.
ServerAuthModule 146	A ServerAuthModule validates client requests and secures responses to the client.

javax.security.auth.message.module
ClientAuthModule

All Superinterfaces: ClientAuth[73]

Declaration

```
public interface ClientAuthModule extends ClientAuth[73]
```

Description

A ClientAuthModule secures request messages, and validates received response messages.

A module implementation should assume it may be used to secure different requests as different clients. A module should also assume it may be used concurrently by multiple callers. It is the module implementation's responsibility to properly save and restore any state as necessary. A module that does not need to do so may remain completely stateless.

Every implementation of the interface must provide a public zero argument contructor.

See Also: ClientAuthContext[137]

Member Summary	
Methods	
java.lang.Class[]	getSupportedMessageTypes[144]()
void	initialize[145](MessagePolicy[78] requestPolicy, MessagePolicy[78] responsePolicy, javax.security.auth.callback.CallbackHandler handler, java.util.Map options)

Inherited Member Summary
Methods inherited from interface ClientAuth[73]
cleanSubject(MessageInfo, Subject)[73], secureRequest(MessageInfo, Subject)[73], validateResponse(MessageInfo, Subject, Subject)[74]

Methods

getSupportedMessageTypes()

```
public java.lang.Class[] getSupportedMessageTypes()
```

Get the one or more Class objects representing the message types supported by the module.

Returns: An array of Class objects where each element defines a message type supported by the module. A module should return an array containing at least one element. An empty array indicates that the module will attempt to support any message type. This method never returns null.

initialize(MessagePolicy[78] requestPolicy, MessagePolicy[78] responsePolicy, javax.security.auth.callback.CallbackHandler handler, java.util.Map options)

```
public void initialize(MessagePolicy[78] requestPolicy, MessagePolicy[78] responsePolicy,
            javax.security.auth.callback.CallbackHandler handler, java.util.Map options)
            throws AuthException
```

Initialize this module with request and response message policies to enforce, a CallbackHandler, and any module-specific configuration properties.

The request policy and the response policy must not both be null.

Parameters:

`requestPolicy` - The request policy this module must enforce, or null.

`responsePolicy` - The response policy this module must enforce, or null.

`handler` - CallbackHandler used to request information.

`options` - A Map of module-specific configuration properties.

Throws:

`AuthException`[69] - If module initialization fails, including for the case where the options argument contains elements that are not supported by the module.

javax.security.auth.message.module
ServerAuthModule

All Superinterfaces: ServerAuth[86]

Declaration

```
public interface ServerAuthModule extends ServerAuth[86]
```

Description

A ServerAuthModule validates client requests and secures responses to the client.

A module implementation should assume it may be used to secure different requests as different clients. A module should also assume it may be used concurrently by multiple callers. It is the module implementation's responsibility to properly save and restore any state as necessary. A module that does not need to do so may remain completely stateless.

Every implementation of the interface must provide a public zero argument contructor.

See Also: ServerAuthContext[141]

Member Summary	
Methods	
java.lang.Class[]	getSupportedMessageTypes[146]()
void	initialize[147](MessagePolicy[78] requestPolicy, MessagePolicy[78] responsePolicy, javax.security.auth.callback.CallbackHandler handler, java.util.Map options)

Inherited Member Summary
Methods inherited from interface ServerAuth[86]
cleanSubject(MessageInfo, Subject)[86], secureResponse(MessageInfo, Subject)[86], validateRequest(MessageInfo, Subject, Subject)[87]

Methods

getSupportedMessageTypes()

```
public java.lang.Class[] getSupportedMessageTypes()
```

Get the one or more Class objects representing the message types supported by the module.

Returns: An array of Class objects, with at least one element defining a message type supported by the module.

initialize(MessagePolicy[78] requestPolicy, MessagePolicy[78] responsePolicy, javax.security.auth.callback.CallbackHandler handler, java.util.Map options)

```
public void initialize(MessagePolicy[78] requestPolicy, MessagePolicy[78] responsePolicy,
            javax.security.auth.callback.CallbackHandler handler, java.util.Map options)
            throws AuthException
```

Initialize this module with request and response message policies to enforce, a CallbackHandler, and any module-specific configuration properties.

The request policy and the response policy must not both be null.

Parameters:

`requestPolicy` - The request policy this module must enforce, or null.

`responsePolicy` - The response policy this module must enforce, or null.

`handler` - CallbackHandler used to request information.

`options` - A Map of module-specific configuration properties.

Throws:

`AuthException`[69] - If module initialization fails, including for the case where the options argument contains elements that are not supported by the module.

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
650 960-1300

For U.S. Sales Office locations, call:
800 821-4643
In California:
800 821-4642

Australia: (02) 844 5000
Belgium: 32 2 716 7911
Canada: 416 477-6745
Finland: +358-0-525561
France: (1) 30 67 50 00
Germany: (0) 89-46 00 8-0
Hong Kong: 852 802 4188
Italy: 039 60551
Japan: (03) 5717-5000
Korea: 822-563-8700
Latin America: 650 688-9464
The Netherlands: 033 501234
New Zealand: (04) 499 2344
Nordic Countries: +46 (0) 8 623 90 00
PRC: 861-849 2828
Singapore: 224 3388
Spain: (91) 5551648
Switzerland: (1) 825 71 11
Taiwan: 2-514-0567
UK: 0276 20444

Elsewhere in the world,
call Corporate Headquarters:
650 960-1300
Intercontinental Sales: 650 688-9000