



A Layered Security Approach to Enable...

Consumers



Enterprises



Governments



Entrust IdentityGuard 12.0

Programming Guide for the Java Platform

Document Issue: 4.0
August 2018

Entrust is a registered trademark of Entrust, Inc. in the United States and certain other countries. Entrust is a registered trademark of Entrust Limited in Canada. All other company and product names are trademarks or registered trademarks of their respective owners. The material provided in this document is for information purposes only. It is not intended to be advice. You should not act or abstain from acting based upon such information without first consulting a professional. ENTRUST DOES NOT WARRANT THE QUALITY, ACCURACY OR COMPLETENESS OF THE INFORMATION CONTAINED IN THIS GUIDE. SUCH INFORMATION IS PROVIDED "AS IS" WITHOUT ANY REPRESENTATIONS AND/OR WARRANTIES OF ANY KIND, WHETHER EXPRESS, IMPLIED, STATUTORY, BY USAGE OF TRADE, OR OTHERWISE, AND ENTRUST SPECIFICALLY DISCLAIMS ANY AND ALL REPRESENTATIONS, AND/OR WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT, OR FITNESS FOR A SPECIFIC PURPOSE.

© Copyright 2018 Entrust Datacard Ltd. All rights reserved

Contents

About this guide	7
Revision information.....	7
Documentation conventions	9
Note and Attention text	9
Related documentation.....	10
Obtaining documentation	11
Documentation feedback	11
Obtaining technical assistance	11
Technical support	11
Professional Services	12
Training.....	12
Chapter 1: API overview and samples.....	13
Entrust IdentityGuard APIs.....	13
Feature history	13
V11 services.....	14
Authentication V11 API.....	15
Administration V11 API.....	15
V11 Web service definition files	15
V11 Java files and libraries.....	16
V11 Java API toolkit help	17
Sample applications	17
Sample Web-based application.....	17
Sample command-line applications.....	18
Running the sample authentication client.....	18
Running the sample administration client	20
Using the administration commands	25
Chapter 2: Client application setup	26
Setting up your application	26
Using the Web services directly	26
Using the Entrust IdentityGuard APIs.....	27
Supported Java SDK and JRE versions	28
Using SSL communication	29
Disabling hostname verification.....	29
Configuring trust	30
Configuring SSL with Entrust IdentityGuard replicas	32
Create a binding object.....	33
Create an authentication binding object	33
Create an administration binding object	34
Create a failover authentication binding object	35
Create a failover administration binding object	37
Performance optimizations.....	39
Reuse a single binding object	39
Migrating V9 or V10 services to V11	39
Update service URLs.....	39
Update proxy class library.....	39
Update the import statements	40
Changes to the Authentication API	40
Changes to the Administration API.....	41

Chapter 3: Authentication approaches.....	45
Anonymous grid authentication.....	45
Anonymous grid authentication method	45
Security considerations.....	46
Anonymous grid authentication sample	46
Challenge retention	47
String conversion sample.....	47
Generic authentication	48
Generic API methods.....	48
Grids	50
Tokens	50
One time password (OTP)	50
Knowledge-based questions and answers	53
External authentication	54
Password authentication.....	54
Certificate challenge response authentication	54
Generic API code sample	55
Mutual authentication	61
Grid and token serial number and location replay.....	61
Knowledge-based authentication	62
Image and caption replay.....	62
Image and caption replay samples.....	62
Image management.....	67
Step-up authentication	67
Machine authentication	68
Machine Authentication API methods.....	68
Machine Authentication API code example	69
Sources of machine information.....	70
Storing and retrieving machine information	74
Machine authentication Web sample.....	75
Risk-based authentication (RBA)	78
Transaction authentication.....	80
Chapter 4: Administration tasks	84
Administration setup and login.....	84
Basic administration tasks	87
Create and register a user	87
Rename a user	87
Get grid contents for a user	88
Create and activate a user's grid card.....	88
Create and assign preproduced grid cards.....	91
Create and send an OTP	91
Retrieve delivery configuration of an OTP	94
Create and modify user contact information	94
Assign and modify a token.....	95
Create and modify a temporary PIN.....	98
Create and modify a personal verification number (PVN)	99
Set up a user's questions and answers	101
Unlock users	101
Administer machine secrets.....	102
Administrative monitoring tasks.....	103
Check for expiring grid cards	103
Check grid card inventory	104
Check token inventory	106

Check for unused assigned grid cards or tokens	107
Administration of smart credentials.....	109
Create smart credentials for a user	109
Approve smart credentials	110
Issue smart credentials.....	110
Check the status of smart credentials issuance request	111
Modify smart credentials.....	113
Modify enrollment values in smart credentials	114
Change the state of a smart credential.....	115
Chapter 5: Programming additional Entrust IdentityGuard functionality	118
Customizing out-of-band delivery of OTPs.....	118
Adding properties to identityguard.properties to support new OOB delivery methods.....	118
Creating the Java class to implement the Entrust IdentityGuard Java interface.....	119
Setting up the Java class for use	123
Integrating external Q&A providers	123
Creating the Java class to implement the Entrust IdentityGuard Java interface for external Q&A...	123
Setting up the Java class for use for external Q&A.....	127
Chapter 6: Performing Identity Assured operations with smart credentials	128
Security challenges	128
Security challenge integration	130
Configuration.....	130
Security challenges for authentication and transaction verification.....	131
Security challenges for digital signing	135
Anonymous smart credential security challenges	139
Chapter 7: Integrating Entrust mobile soft tokens.....	145
Soft token activation.....	145
Token activation methods	145
userTokenActivateComplete	145
Token activation integration.....	146
Soft token transaction verification.....	147
Online transaction verification	147
Authentication.....	148
Transaction verification.....	148
Online transaction verification integration	149
Configuration.....	149
Online authentication and transaction verification.....	149
Offline transaction verification	153
Offline transaction verification integration	155
Chapter 8: API exceptions.....	157
System and service faults	157
Error output sample	157
ErrorCode class.....	158
Authentication faults.....	159
AuthenticationFault base class	159
AuthenticationServiceFault class	159
AuthenticationSystemFault class	160
Authentication warning faults	161
Authentication operation exceptions.....	161
Administration faults	162

AdminServiceFault base class	162
Appendix A: UtilityMethods class.....	164
Index	171

About this guide

The *Entrust IdentityGuard Programming Guide* provides detailed information about how to use the Java version of the Entrust IdentityGuard Authentication API and the Administration API to integrate an existing secure application with Entrust IdentityGuard.

This guide discusses the V11 version of these APIs.

Note: If you are using a programming environment other than .NET or Java, you can still connect applications to Entrust IdentityGuard. Entrust IdentityGuard exposes a standard Web-services interface for authentication and administration. The server install ships the WSDL for these services in the Entrust IdentityGuard installation directory structure:

- on UNIX: `$IG_HOME/client/doc/`

- on Windows: `<IG_HOME>\client\doc`

You can translate example code in the .NET and Java guides into other languages.

Attention: The V8 and earlier Authentication and Administration APIs are no longer supported.

Revision information

Revisions in this document

Document issue and date	Section	Description
4.0 August 2018	Online activation and transaction verification	Updated to support Entrust IdentityGuard Release 12.0 Patch 78604. Note about use of priority transaction detail for queued transactions.
3.0 March 2018	Create a failover administration binding object	Fixed typo (authentication > administration)
	Security challenges Online transaction verification	Improved readability of sequence diagrams
2.0 June 2017	Generic authentication > tokens	Small update about support for Android mobile phones.
	Supported Java SDK and JRE versions	Changed Sun JRE to Oracle JRE.
1.0 March 2017	All sections	First issue of this guide for Entrust IdentityGuard release 12.0.

Documentation conventions

The following typographic conventions appear in this guide:

Typographic conventions

Convention	Purpose	Example
Bold text (other than headings)	Indicates graphical user interface elements and wizards	Click Next .
<i>Italicized text</i>	Used for book or document titles	<i>Entrust IdentityGuard Administration Guide</i>
Blue text	Used for hyperlinks to other sections in the document	For more information about initialization see, Initializing IdentityGuard .
<u>Underlined blue text</u>	Used for Web links	For more information, visit our Web site at www.entrustdatacard.com .
Courier type	Indicates installation paths, file names, Windows registry keys, commands, code, and text you must enter	<code>init [-sernum <num>] [-overwrite]</code>
Angle brackets < >	Indicates variables (text you must replace with your organization's correct values)	<code>userDelete <userid> [-import <file>]</code>
Square brackets [courier type]	Indicates optional parameters	<code>init [-sernum <num>] [-overwrite]</code>

Note and Attention text

Throughout this guide, there are paragraphs set off by ruled lines above and below the text. These paragraphs provide key information with two levels of importance, as shown below.

Note: Information to help you maximize the benefits of your Entrust product.

Attention: Issues that, if ignored, may seriously affect performance, security, or the operation of your Entrust product.

Related documentation

Entrust IdentityGuard is supported by a complete documentation suite:

- For instructions about installing and configuring the Entrust IdentityGuard Server, see the *Entrust IdentityGuard Installation Guide*.
- For instructions about administering Entrust IdentityGuard users and groups, see the *Entrust IdentityGuard Server Administration Guide*.
- For information about smart credentials authentication, see the *Entrust IdentityGuard Smart Credentials Guide*.
- For a full list and descriptions of the Entrust IdentityGuard master user shell commands, see the *Entrust IdentityGuard Master User Shell Reference*.
- For information about configuring Entrust IdentityGuard to work with a supported LDAP repository, see the *Entrust IdentityGuard Directory Configuration Guide*.
- For information about configuring Entrust IdentityGuard to work with a supported JDBC database, see the *Entrust IdentityGuard Database Configuration Guide*.
- For information about Entrust IdentityGuard error messages, see the *Entrust IdentityGuard Error Messages*.
- For information about new features, limitations and known issues in the latest release, see the *Entrust IdentityGuard Release Notes*.
- For information about the Self-Service Module, see:
 - *Entrust IdentityGuard Self-Service Module Installation and Configuration Guide*
 - *Entrust IdentityGuard Self-Service Module Customization Guide*
 - *Entrust IdentityGuard Self-Service Module User Guide*
- For information about integrating the authentication and administration processes of your applications with Entrust IdentityGuard, see the *Entrust IdentityGuard Programming Guide* that applies to your development platform (either Java Platform or .NET).

Note: If you are using a programming environment other than .NET or Java, you can still connect applications to Entrust IdentityGuard. Entrust IdentityGuard exposes a standard web-services interface for authentication and administration. The server install ships the WSDL for these services in the <IG_HOME>/client/doc directory. You can translate example code in the .NET and Java guides into other languages.

- For information about the Entrust IdentityGuard Device Fingerprint Software Development Kit, see:
 - *Entrust IdentityGuard Device Fingerprint SDK Programmer's Guide*
 - *Entrust IdentityGuard Device Fingerprint SDK Readme files for Android, iOS, and Java*
- For Entrust IdentityGuard product information and a data sheet, go to <https://www.entrustdatacard.com/products>

Obtaining documentation

Entrust product documentation, white papers, technical notes, and a comprehensive Knowledge Base are available through Entrust TrustedCare Online. If you are registered for our support programs, you can use our Web-based Entrust TrustedCare Online support services at:

<https://trustedcare.entrustdatacard.com>

Documentation feedback

You can rate and provide feedback about Entrust product documentation by completing the online feedback form. You can access this form by following [this link](#)

Feedback concerning documentation can also be directed to the Customer Support email address.

support@entrustdatacard.com

Obtaining technical assistance

Entrust recognizes the importance of providing quick and easy access to our support resources. The following subsections provide details about the technical support and professional services available to you.

Technical support

Entrust offers a variety of technical support programs to help you keep Entrust products up and running. To learn more about the full range of Entrust technical support services, visit our Web site at:

<http://www.entrustdatacard.com>

If you are registered for our support programs, you can use our Web-based support services.

Entrust TrustedCare Online offers technical resources including Entrust product documentation, white papers and technical notes, and a comprehensive Knowledge Base at:

<https://trustedcare.entrustdatacard.com>

If you contact Entrust Customer Support, please provide as much of the following information as possible:

- your contact information
- product name, version, and operating system information
- your deployment scenario
- description of the problem
- copy of log files containing error messages
- description of conditions under which the error occurred
- description of troubleshooting activities you have already performed

Email address

The email address for Customer Support is: support@entrustdatacard.com

Professional Services

The Entrust team assists organizations around the world to deploy and maintain secure transactions and communications with their partners, customers, suppliers, and employees. Entrust offers a full range of professional services to deploy our solutions successfully for wired and wireless networks, including planning and design, installation, system integration, deployment support, and custom software development.

Whether you choose to operate your Entrust solution in-house or subscribe to hosted services, Entrust Professional Services will design and implement the right solution for your organization's needs. For more information about Entrust Professional Services please visit our Web site at:

<http://www.entrust.com/services>

Training

Through a variety of hands-on courses, Entrust delivers effective training for deploying, operating, administering, extending, customizing and supporting any variety of Entrust digital identity and information security solutions. Delivered by training professionals, Entrust's professional training services help to equip you with the knowledge you need to speed the deployment of your security platforms and solutions. Please visit our training website at:

<https://www.entrust.com/training>

Chapter 1:

API overview and samples

This chapter describes the various APIs available for use with a client application. It also includes details on the sample applications included with Entrust IdentityGuard.

Entrust IdentityGuard APIs

Entrust IdentityGuard includes two Web services:

- Authentication service and Authentication API
- Administration service and Administration API

Use the Authentication service to integrate Entrust IdentityGuard authentication methods, such as grid or token authentication, into your Web applications.

Use the Administration service to add end-user services to your application. The services include features such as user self-registration, user requests for cards or tokens, a self-reporting mechanism for lost cards or tokens, and similar functionality. Alternatively, you can deploy the Entrust IdentityGuard Self-Service Module to perform these tasks.

Feature history

The following table summarizes changes to the features supported by the APIs over several versions.

API version	Entrust IdentityGuard version	API features
API V6	9.3	Administration API: <ul style="list-style-type: none">• soft token support• support for multiple tokens per user (token sets)• tokens support storing custom data as token parameters• support for federated user identities• high availability enhancements to support automatic server failover Authentication API: <ul style="list-style-type: none">• support for multiple tokens per user (token sets)• support for soft token transaction delivery and authentication• high-availability enhancements to support automatic server failover
API V7	10.0	Added smart credentials in the Administration API
API V8	10.1	Added features to administer the smart credential APIs. There was no change to the Authentication API in IdentityGuard 10.1.

API version	Entrust IdentityGuard version	API features
		A patch to the V8 APIs introduced support for multiple passwords
API V9	10.2	Introduced support for smart credential identity assured operations (security challenges for authentication, transaction verification, and digital signing). This version of the APIs is compatible with Entrust IdentityGuard 10.2 Feature Pack 1, however, it does not support the new features introduced in that release
API V9	10.2 FP1	The V9 version of the APIs introduced with Entrust IdentityGuard 10.2 Feature Pack 1 introduced the following features: <ul style="list-style-type: none"> • biometric authentication • biometric administration • support for online token transaction verification and online activation • new license types and new license administration features Use of these features requires Entrust IdentityGuard 10.2 Feature Pack 1.
API V10	11.0	Introduced enhancements for risk based assessment with finger prints and smart credential handling
API V11	12.0	Introduced support for anonymous smart credential security challenges, which allow users to perform smart credential authentication without first entering a user name and password.

V11 services

When you upgrade to Entrust IdentityGuard 12.0, you have access to the new V11 versions of the Web services and APIs. The V11 versions are the latest for Entrust IdentityGuard and include the new features available in Entrust IdentityGuard release 12.0.

- The V9 and V10 services are still available in Entrust IdentityGuard 12.0 and use the same naming.
- You can continue to use client applications you developed to work with Entrust IdentityGuard 10.1, 10.2, 10.2 FP1 and 11.0 applications, but you must run them using the V9, V10, or V11 services.
- If you decide to upgrade to the V11 services, your applications must be changed. See [Migrating V9 or V10 services to V11](#).
- You can run V9, V10 and V11 client code in the same application.
- The new Entrust IdentityGuard functionality introduced in release 12.0 is available only through the V11 services.

To use the V11 services and any of the release 12.0 features, you must update your application, and run it using the V11 services. Your Entrust IdentityGuard release 12.0 applications can access the V11 services from:

`http://<host>:8080/IdentityGuardAuthService/services/AuthenticationServiceV11`

`https://<host>:8443/IdentityGuardAuthService/services/AuthenticationServiceV11`

`https://<host>:8444/IdentityGuardAdminService/services/AdminServiceV11`

where <host> refers to the server where you installed Entrust IdentityGuard.

The ports shown are the defaults for installations with embedded Tomcat server and will differ with installations on an existing IBM WebSphere and Oracle WebLogic application server.

You must update the Entrust IdentityGuard Authentication and Administration interface names in the client code if you want to use the V11 services. For upgraded coding samples, see [Migrating V9 or V10 services to V11](#).

Note: This guide discusses only the V11 APIs.

Authentication V11 API

The Entrust IdentityGuard Authentication service is a set of Web service methods used for retrieving challenge requests and authenticating user responses. It is designed to integrate with your existing authentication applications to provide multifactor authentication.

You can create an application that calls the Authentication API using its Web service, Java Platform, or .NET interfaces to authenticate users.

For more information about programming on the .NET Platform, see the *Entrust IdentityGuard Programming Guide for the .NET Platform*.

Administration V11 API

The Entrust IdentityGuard Administration service (also referred to as the Administration API) is a Web service running on the Entrust IdentityGuard server that manages groups, policies, administrators, users, grid cards, tokens, PINs, PVNs, smart credentials, biometrics and other Entrust IdentityGuard data.

You can create a client application that uses the Administration service to automate Entrust IdentityGuard user administration tasks and incorporate these tasks into existing user management systems.

For information about programming on the .NET Platform, see the *Entrust IdentityGuard Programming Guide for the .NET Platform*.

V11 Web service definition files

The Authentication API is defined in `AuthenticationServiceV11.wsdl`. The Administration API is defined in `AdminServiceV11.wsdl`. Common data types are found in `ServiceV11Common.xsd`.

You can locate these files:

- On UNIX: `$IG_HOME/client/doc`
where:
`$IG_HOME` is usually `/opt/entrust/identityguard120`
- On Windows: `<IG_HOME>\client\doc`
where:
`<IG_HOME>` is usually `C:\Program Files\Entrust\IdentityGuard\identityguard120`.

You can also view a Web service definition file from a browser after you complete the following steps.

To view a Web service definition file from a browser

- 1 Log In to the Entrust IdentityGuard Properties Editor.
- 2 In the Properties Editor Table of Contents, click **Service Settings**.
- 3 Under **WSDL Query Returns WSDL**, select **True**.
- 4 Click **Validate & Save**.
- 5 Restart the server.
- 6 Enter one of the following URLs in a browser:

`http://<host>:8080/IdentityGuardAuthService/services/AuthenticationServiceV11?wsdl`

`https://<host>:8443/IdentityGuardAuthService/services/AuthenticationServiceV11?wsdl`

`https://<host>:8444/IdentityGuardAdminService/services/AdminServiceV11?wsdl`

where <host> refers to the server where you installed Entrust IdentityGuard.

V11 Java files and libraries

The Entrust IdentityGuard Authentication API Java library is found in `IdentityGuardAuthAPIV11.jar`. The Entrust IdentityGuard Administration API Java library is found in `IdentityGuardAdminAPIV11.jar`.

You can locate these:

- on UNIX, in `$IG_HOME/client/lib`
- on Windows, in `<IG_HOME>\client\lib`

Both Entrust IdentityGuard APIs use the following third-party libraries, also located in the `lib` directory:

- `axis.jar`
- `commons-discovery.jar`
- `commons-logging-1.0.4.jar`
- `commons-httpclient-3.1.jar`
- `commons-codec-1.4.jar`
- `jaxrpc.jar`
- `saaj.jar`
- `wsdl4j-1.5.1.jar`
- `mailapi.jar`
- `activation.jar`

Java system properties

You can set the following Java system properties when you are running your Entrust IdentityGuard client application:

- `identityguard.disable.strict.ssl`: When set to `true`, turns off SSL hostname validation for both `commons-httpclient` and the previous approach to HTTP communication. The default is `false`.
- `identityguard.disable.commons.http`: When set to `true`, the client code uses the previous approach to HTTP communication for sending SOAP messages. The default is `false`, meaning that the `HTTPClient` library from the Apache Commons project is used.

For example:

```
java -classpath <insert_classpath_here> -
Didentityguard.disable.strict.ssl=true -
Didentityguard.disable.commons.http=true <insert_program_name_here>
```

V11 Java API toolkit help

Your Entrust IdentityGuard installation also includes a set of zipped HTML files (referred to as Javadocs), `IdentityGuardAuthAPIV11.zip` and `IdentityGuardAdminAPIV11.zip`, that explain the Java API toolkits. You can locate these:

- on UNIX, in `$IG_HOME/client/doc/`
- on Windows, in `<IG_HOME>\client\doc`

Sample applications

Entrust provides Web-based and command-line sample applications to help guide your Entrust IdentityGuard implementation.

Sample Web-based application

Entrust IdentityGuard includes a sample Web-based application that you can enable after installation and initialization. The sample uses the Authentication service to authenticate users, and the Administration service to register users and to activate cards and tokens.

This application demonstrates:

- how to register users and machines for different authentication types (grid, knowledge-based, one-time passwords, and token)
- how to authenticate users using different authentication commands (grid, knowledge-based, one-time passwords, and token)
- how to activate an Entrust IdentityGuard grid or token
- how to authenticate using machine authentication
- how to set up mutual authentication

The Web sample provides guidance for how your application should behave once you integrate it with Entrust IdentityGuard.

See the *Entrust IdentityGuard Installation Guide* for information about configuring and running the sample Web application.

After you install and enable the sample Web application, you can access it on any supported platform using one of the following URLs:

`https://<host>:8443/IdentityGuardSampleApp/`

`http://<host>:8080/IdentityGuardSampleApp/`

where `<host>` is the name of the computer on which Entrust IdentityGuard is installed.

The port shown in the URL is the default port for installations with embedded Tomcat server. Replace the port number in the examples with the one you specified during installation of Entrust IdentityGuard.

You can access the client authenticated SSL location for the sample at:

`https://<host>:8447/IdentityGuardSampleApp/`

This sample prompts the end-user to select a certificate when accessing the site.

For installations on Windows, click **Start > All Programs > Entrust > IdentityGuard > Sample Application**.

Note: If you cannot reach the Entrust IdentityGuard Authentication service or Administration service, verify that firewall rules are not blocking their HTTPS ports.

To view the source code of the sample application, open the `IdentityGuardSampleApp-SRC.zip` file. You can locate this:

- on UNIX, in `$IG_HOME/client/sample`
- on Windows, in `<IG_HOME>\client\sample`

Sample command-line applications

Entrust IdentityGuard includes sample applications that you can use to test the APIs that perform challenge, authentication, and administration requests.

- For the Authentication API, the file `IdentityGuardAuthServiceClient.java` contains the source code for a sample command-line Java application.

The compiled version is `IdentityGuardAuthServiceClient.class`. For instructions for running this sample, see “Running the sample authentication client” on page [18](#).

- For the Administration API, the file `IGAdminTest.java` contains the source code for a sample command-line Java application.

The compiled version is `IGAdminTest.class`. For instructions on running this sample, see “Running the sample administration client” on page [20](#).

Running the sample authentication client

Complete the following procedure to run the sample Entrust IdentityGuard command-line authentication sample. It provides examples that show how different authentication approaches work.

Before running the samples, you may want to create sample users with grids, tokens, or other authentication methods. See the *Entrust IdentityGuard Server Administration Guide* for explanations of the Normal and Enhanced security level authentication types, and the Machine authentication policies. The *Entrust IdentityGuard Server Administration Guide* also includes information about setting the allowed and default authentication methods for users.

To run the authentication sample on UNIX

- 1** Log in to the hardware server hosting Entrust IdentityGuard as the user that owns the Entrust IdentityGuard installation.
- 2** Change to the Entrust IdentityGuard installation directory. For example:

```
cd /opt/entrust/identityguard120
```
- 3** Set the Entrust IdentityGuard environment variables:

```
./env_settings.sh
```
- 4** Change to the Entrust IdentityGuard sample directory:

```
cd $IG_HOME/client/sample
```
- 5** Execute the script to start the authentication client:

```
./runSampleClient.sh
```

To run the authentication sample on Microsoft Windows

- 1** Ensure that the Entrust IdentityGuard server is running.
- 2** Change to the <IG_HOME>\client\sample directory.
- 3** Start the authentication client using one of the following:
 - If you are running the Authentication service on port 8080 (the default port), double-click the following file to start the authentication client:

```
runSampleClient.bat
```
 - If you are not running the Authentication service on port 8080, open a DOS shell, and enter:

```
runSampleClient.bat -port <auth-service-port>
```

 where <auth-service-port> is the number of the port you are using.

Using the authentication commands

When you start the command-line authentication sample in either UNIX or Windows, the following welcome message and list of available commands appears:

```
=====
== Entrust IdentityGuard Authentication Client Sample Java App ==
=====

Welcome to the Entrust IdentityGuard Authentication Service sample
application.

This application provides example usages of the Entrust IdentityGuard
Authentication API, an interface that can be used to retrieve challenge
requests and authenticate user responses.

The following authentication mechanisms can be implemented using the
APIs:

- one-step authentication:
  1. getAnonymousChallenge or getAnonymousChallengeForGroup
  2. authenticateAnonymousChallenge
- two-step generic authentication
  1. getGenericChallenge
  2. authenticateGenericChallenge

To display all available commands, type 'help'.
```

```
Connected to Entrust IdentityGuard authentication service URL:
http://localhost:8080/IdentityGuardAuthService/services/AuthenticationServiceV11
```

For more information about the authentication mechanisms listed, see “Authentication approaches” on page [45](#).

Enter **Help** on the command line to see the syntax for all commands.

In the syntax:

- The asterisk (*) means you can include zero or more occurrences of an attribute.
- The plus sign (+) means you must include one but may include more than one occurrence of an attribute.
- Many commands require a user ID.

A user ID consists of both the user's name and group, expressed in the following format: `<groupname>/<username>`. If you do not include the group name, Entrust IdentityGuard finds the correct group if the user name is unique; otherwise it returns an error.

See “Documentation conventions” on page [9](#) for an explanation of standard syntax conventions.

Running the sample administration client

Complete the following procedure to run the sample Entrust IdentityGuard command-line administration client. This client lets you test how different administration commands work.

To run the sample administration client on UNIX

- 1 Log in to the hardware server hosting Entrust IdentityGuard as the user who owns the Entrust IdentityGuard installation.
- 2 Change to the Entrust IdentityGuard installation directory. For example:

```
cd /opt/entrust/identityguard120
```
- 3 Set the Entrust IdentityGuard environment variables:

```
./env_settings.sh
```
- 4 Change to the Entrust IdentityGuard sample directory:

```
cd $IG_HOME/client/sample
```
- 5 Give the administrator access to the Administration API. There are two ways to do this.

- You can edit the administration credentials to use an administrator with proper permissions. To use this method, complete the following steps:

- a Open the `igadmintest.properties` file in a text editor.

The `igadmintest.properties` file contains the administrator credentials that the sample administration client uses when it authenticates to the administration Web service.

- b Edit the file as shown. In the following example, `samplegroup/superadmin` has the super user role, so it can invoke any administrative function.

```
# the URL of the IdentityGuard Admin Service
igadmintest.url=https://localhost:8444/IdentityGuardAdminService/services/AdminServiceV11
# Specify additional urls for failover by appending
```

```
# incrementing numbers after igadmintest.url starting from 1.
# For example:
# igadmintest.url=
# igadmintest.url1=
# igadmintest.url2=
# the admin id of the admin used to log in to the admin service
igadmintest.adminid=samplegroup/superadmin
# the password of the admin
igadmintest.adminpassword=superadminpassword
```

- You can tell the sample to use WS-Security authentication rather than using the standard login call. To use this method, add the following setting:

```
# Use a WS-Security UsernameToken header to authenticate
# instead of a call to the login API?
# Generally, using WS-Security headers is less efficient than
# calling the login API and maintaining a session, but is
# useful in programs that cannot guarantee a session will
# be successfully maintained.
igadmintest.usewssecurity=false
```

6 Save and close the file.

7 Create a list of command line arguments based on the API call you wish to invoke.

The list of available API calls are the methods associated with the `AdminServicesBindingStub` class. These methods are documented in the Javadocs (see “V11 Java API toolkit help ” on page [17](#)).

For example, you can make one of the following calls:

- `groupList`
- `userCreate <userID> -group <name>`
- `userOTPGet <userID>`

Ensure that the administrator you selected in }Step [5](#) on page 20 has the permissions to complete the request.

8 Execute the script to start the administration client:

```
./runAdmin.sh <list of arguments>
```

where <list of arguments> is the list of Administration API commands to run. The list of available API calls are the methods associated with the `AdminServiceBindingStub` class. These methods are documented in the Javadocs (see “V11 Java API toolkit help ” on page [17](#)).

For example, to list all Entrust IdentityGuard groups, enter:

```
./runAdmin.sh groupList
```

Output similar to the following appears:

```
igadmintest.url =
https://localhost:8444/IdentityGuardAdminService/services/AdminServiceV11
igadmintest.adminid = igadmin
igadmintest.adminpassword = 6Frosted_flakes
igadmintest.usewssecurity = false
```

```

login
Administrator igadmin in group default logged in.
  Password expiry date: 2009-08-19 11:46:14
  Password allow change after date: 2009-06-24 11:46:14
Group:
  Name: groupAB
  Default: false
  Policy: default
  Partition: No Partition Assigned
  User Repositories: sb2, sb3
  Card Repository: No Card Repository Assigned
  Token Repository: No Token Repository Assigned
  Comments:
Group:
  Name: default
  Default: true
  Policy: default
  Partition: No Partition Assigned
  User Repositories: default
  Card Repository: No Card Repository Assigned
  Token Repository: No Token Repository Assigned
  Comments:
  2 groups returned.

```

If you do not provide arguments, or if you provide incorrect arguments, the sample administration client displays one of the following:

- an error message, such as:
 "Unknown command <your command>"
 "Missing arguments"
- the actual Service Fault returned by the Web service, for example:
 AdminServiceFault: OBJECT_NOT_EXIST 5204056 Policy abcd does not exist.

To run the sample administration client on Windows

- 1 Ensure that the Entrust IdentityGuard server is running.
- 2 If you are running the sample client from another computer, copy the following files from <IG_HOME>:
 - client\sample\igadmintest.properties
 - client\sample\runAdmin.bat
 - etc\keystore
 - client\lib*.jar

Note: Do not copy the keystore file from a production Entrust IdentityGuard system as this file contains the private key of the server and could be used to decrypt traffic to the server. If you are running sample code against a production server (not recommended), you should

export the Entrust IdentityGuard server public certificate from the keystore using the Java keytool command.

3 Open the `igadmintest.properties` file in a text editor.

The `igadmintest.properties` file contains the administrator credentials that the sample administration client uses when it authenticates to the Administration service.

4 Edit the administration credentials to use an administrator with proper permissions.

In the following example, `samplegroup/superadmin` has the super user role, and so can invoke any administrative function.

```
# the URL of the Entrust IdentityGuard Admin Service
igadmintest.url=https://localhost:8444/IdentityGuardAdminService/services/AdminService
V11
# Specify additional urls for failover by appending
# incrementing numbers after igadmintest.url starting from 1.
# For example:
# igadmintest.url=
# igadmintest.url1=
# igadmintest.url2=
# the admin id of the admin used to log in to the admin service
igadmintest.adminid=samplegroup/superadmin
# the password of the admin
igadmintest.adminpassword=superadminpassword
# Use a WS-Security UsernameToken header to authenticate instead
# of a call to the login API?
# Generally, using WS-Security headers is less efficient than calling the login
# API and maintaining a session, but is useful in programs that cannot guarantee
# a session will be successfully maintained.
igadmintest.usewssecurity=false
```

5 Save and close the file.

6 If you are running the sample application on a server other than the one where you installed Entrust IdentityGuard, follow these steps:

a Open the `runAdmin.bat` file in a text editor.

b Check the `-classpath` to ensure that it points to the location on the computer where you copied the JAR files.

By default, the script expects the JAR files in the same folder.

c <http://xml.apache.org/xerces2-j/>

The command line arguments for the sample application must include:

`-Djavax.net.ssl.trustStore=<path_to_keystore>`

where `<path_to_keystore>` is:

- on UNIX, `$IG_HOME/etc/keystore`
 - on Microsoft Windows, `<IG_HOME>etc\keystore`
-

For example:

```
-Djavax.net.ssl.trustStore=<IG_HOME>\etc\keystore
```

where

<IG_HOME> is the path to your Entrust IdentityGuard installation (for example, C:\Program Files\Entrust\IdentityGuard\identityguard102).

- 7** Run each API call you wish to invoke, substituting in the required command line arguments. For example:

```
runAdmin.bat groupList
```

The list of available API calls are the methods associated with the AdminServiceBindingStub class. These methods are documented in the Javadocs (see "V10 Java API toolkit help " on page 17).???

For example, you can make one of the following calls:

- groupList
- userCreate <userID> -group <name>
- userOTPGet <userID>

Ensure that the administrator you selected in Step 4 on page 23 has the permissions to complete the request.

- 8** Open a command-line window and change to the <IG_HOME>\client\sample directory.

- 9** Execute the script to start the authentication client:

```
runAdmin.bat
```

If you ran userOTPGet user1 -all to obtain one-time password information for a particular user, you would see output similar to the following:

```
=====
== Entrust IdentityGuard Administration Client Java Sample App ==
=====

AdminServiceV11
igadmintest.adminid = sampleadmin
igadmintest.adminpassword = samplepassword
login
Administrator sampleadmin in group default logged in.
OTP:
  User: default/iguser1
  Name: iguser1
  Group: default
  Create Date: 2009-06-23 15:28:14
  Expire Date: never
  OTP: 6184CQ65
```

If you do not provide any arguments, or if you provide incorrect arguments, then the sample administration client displays one of the following:

- an error message, such as one of the following:


```
"Unknown command <your command>"
```

```
"Missing arguments"
```

- the actual Service Fault returned by the Web service, for example:

```
AdminServiceFault: OBJECT_NOT_EXIST 5204056 Policy abcd does not exist.
```

If you attempt to run this sample on a computer with more than one drive, where Entrust IdentityGuard is installed on a drive other than the drive where the command window is installed, edit `runAdmin.bat` to include the full path for each directory setting. For example, change:

```
set IG_INSTALL_DIR=%cd%
```

to:

```
set IG_INSTALL_DIR=D:\Program Files\Entrust\IdentityGuard\
```

Using the administration commands

In most cases, the command options are identical to the attributes of similarly-named master user shell commands. For example, the administration sample command `userCreate` does the same thing as the master user shell command `user create`, although the latter has a few more attributes.

See the *Entrust IdentityGuard Master User Shell Command Reference* for applicable master user shell information.

Note: Not all attributes available on master user shell commands are available as options on administration sample commands.

In the syntax:

- The asterisk (*) means you can include zero or more occurrences of an attribute.
- The plus sign (+) means you must include one, but may include more than one occurrence of an attribute.
- Many commands require a user ID.

A user ID consists of both the user's name and the group the user belongs to, in the following format: `<groupname>/<username>`. If you do not include the group name, Entrust IdentityGuard finds the correct group if the user name is unique; otherwise it returns an error.

Chapter 2:

Client application setup

The Entrust IdentityGuard Java APIs are a set of services and operations used for retrieving challenge requests, authenticating user responses, and administering users and authentication mechanisms. They are designed to integrate with an existing client application.

This chapter describes how to set up a client application to use the APIs.

Setting up your application

You can set up your application in one of two ways:

- Set it up to communicate directly with the Entrust IdentityGuard Web services using Simple Object Access Protocol (SOAP) over HTTP/HTTPS.

In this scenario, your application must comply with the WSDL definition and is directly responsible for all SOAP communications to the Web service. Apart from the WSDL file, your client application is not required to use any other Entrust IdentityGuard supplied files or JARs.

- Set it up to use the Entrust IdentityGuard Java JAR files, which provide a convenient API for your client application.

In this scenario, your client application makes API calls to the Entrust IdentityGuard Java classes. The details of conforming to the WSDL definition and communicating with the Entrust IdentityGuard Web services are handled automatically.

Using the Web services directly

Entrust IdentityGuard provides two Web services:

- Authentication Web service
- Administration Web service

The Authentication Web service allows client applications to perform second-factor authentication requests to Entrust IdentityGuard. This Web service provides:

- SSL communication (on default port 8443 for installations with embedded Tomcat)
- non-SSL communication (on default port 8080 for installations with embedded Tomcat).

Note: Installations on an existing application server use the ports already defined for the IBM WebSphere or Oracle WebLogic application server.

The Administration Web service allows client applications to perform administration-related requests to Entrust IdentityGuard (for example, creating, modifying, or deleting users).

This Web service communicates using SSL only (on default port 8444 for installations with embedded Tomcat or the SSL port defined with your installation on an existing application server).

To ensure that your client application can communicate with the Entrust IdentityGuard Web services, design and configure your application so that:

- It complies with the Web service definition associated with the Entrust IdentityGuard Web services.

There are separate Web service definitions for the Authentication Web service and the Administration Web service.

- It trusts the Entrust IdentityGuard certificate, if you are using SSL to communicate with the Web service. See “Using SSL communication” on page [29](#).

The definitions for the Entrust IdentityGuard Authentication and Administration Web services are stored in the files `AuthenticationService.wsdl` and `AdminService.wsdl`, respectively. They are located:

- on UNIX, in `$IG_HOME/client/doc`
- on Windows, in `<IG_HOME>\client\doc`

Using the Entrust IdentityGuard APIs

This section provides instructions for setting up the Entrust IdentityGuard APIs so that your client application can use them.

Note: Ensure that your client application uses a Java SDK or JRE supported by Entrust IdentityGuard. For information, see “Supported Java SDK and JRE versions” on page [28](#).

To use the Entrust IdentityGuard APIs

- 1 Copy the following JAR files from the `/client/lib` directory, and make them available to your client application:

- `IdentityGuardAuthAPIV11.jar` (if you are using the Authentication API)
- `IdentityGuardAdminAPIV11.jar` (if you are using the Administration API)
- `axis.jar`
- `commons-discovery.jar`
- `commons-logging-1.0.4.jar`
- `commons-httpclient-3.1.jar`
- `commons-codec-1.4.jar`
- `jaxrpc.jar`
- `saaaj.jar`
- `wsdl4j-1.5.1.jar`
- `mailapi.jar`
- `activation.jar`

These JAR files provide the necessary SOAP-related functionality required to communicate with the Entrust IdentityGuard Web services.

- 2 Update your Java classpath to include these JAR files.

The details of updating your Java classpath depends on your specific environment.

- If your client application is a standalone application running under its own JRE, then update the Java classpath argument directly. The UNIX script file `$IG_HOME/client/sample/runSampleClient.sh` shows how this can be done. Specify the following on one line:

```
java -classpath
.:<location_of_copied_jars>/IdentityGuardAuthAPIV11.jar:<location_of_
copied_jars>/IdentityGuardAdminAPIV11.jar:<location_of_copied_jars>/a
```

```
xis.jar:<location_of_copied_jars>/activation.jar:<location_of_copied_jars>/jaxrpc.jar:<location_of_copied_jars>/commons-logging-1.0.4.jar:<location_of_copied_jars>/commons-httpclient-3.1.jar:<location_of_copied_jars>/commons-codec-1.4.jar:<location_of_copied_jars>/commons-discovery.jar:<location_of_copied_jars>/mailapi.jar:<location_of_copied_jars>/saaj.jar:<location_of_copied_jars>/wsdl4j-1.5.1.jar <name of your client application>
```

- The Windows batch file <IG_HOME>\client\sample\ runSampleClient.bat uses the same list of JAR files in its classpath.
- If your client application is running on an application server, such as IBM WebSphere or Oracle WebLogic, then update the Java classpath on the application server. Refer to your vendor documentation for instructions.

Note: Some application servers provide their own versions of XML parsers, SOAP clients/servers, and loggers. You may encounter a version conflict between the vendor-supplied JAR files and the Entrust IdentityGuard supplied JAR files. The files IdentityGuardAuthAPIV11.jar and IdentityGuardAdminAPIV11.jar support the following versions:

- axis.jar 1.4
- jaxrpc.jar 1.1
- commons-logging.jar 1.0.4
- commons-discovery.jar 0.2
- saaj.jar 1.2
- wsdl4j.jar 1.5.1
- mailapi.jar 1.4.1
- activation.jar 1.1.1- commons-httpclient-3.1.jar
- commons-codec-1.4.jar

If you encounter a JAR conflict (that is, if you receive Java errors of the form `java.lang.NoSuchFieldError` or `java.lang.NoSuchMethodError` during execution), modify your application level classpath to ensure that it is using the correct versions of the JAR files (while the server level classpath continues to use its native JAR files). Refer to your vendor's documentation on configuring a class loading order for your application server.

- 3 Enable SSL communication, if required, see "Using SSL communication" on page [29](#).

Supported Java SDK and JRE versions

This section only applies if your client application is using the Entrust IdentityGuard APIs and is not contacting the Web service directly using WSDL.

Apache Axis Compatibility

The Entrust IdentityGuard V11 APIs use version 1.4 of Apache Axis. If the Java SDK version or the application server that you are using includes Apache Axis, then ensure that it is using the 1.4 version, otherwise you must edit your Java classpath to include the Apache Axis 1.4 version.

Oracle JRE

Apart from including the Apache Axis JAR files in your Java classpath and the Entrust IdentityGuard API JAR files, no other classpath changes are required. Use the JRE version that is available with your Entrust IdentityGuard server installation.

IBM JRE

Apart from including the Apache Axis JAR files in your Java classpath and the Entrust IdentityGuard API JAR files, no other classpath changes are required.

If you are using the IBM JRE on Sun, you need to set the following property to enable SSL:

```
java -Djava.protocol.handler.pkgs=com.ibm.net.ssl.www.protocol
```

Using SSL communication

The Entrust IdentityGuard Authentication and Administration Web services can communicate with client applications using SSL.

Note: This section applies only to installations of Entrust IdentityGuard with embedded Tomcat server. Certificates, keys, and trust for installations using an existing WebSphere or WebLogic application server are stored outside of Entrust IdentityGuard. See the *Entrust IdentityGuard Installation Guide* and the SSL documentation for your application server for information about trust stores.

Note: The Administration Web service requires a secure connection. The Authentication Web service does not.

Disabling hostname verification

By default, Entrust IdentityGuard performs hostname verification when establishing a secure SSL channel between your client application and its Web services. Hostname verification works as follows:

- Entrust IdentityGuard looks at the `CN=` entry in the `subject` attribute in the server's certificate. If the `CN=` entry contains a hostname that matches the hostname of the server HTTPS URL, then hostname verification succeeds. Wildcards are supported. For example, if the certificate contains a `subject` of:

```
CN=*.example.com
```

and the hostname in your URL is

```
igserverprimary.example.com or  
igserversecondary.example.com or  
igservern.example.com
```

then this is considered a match, and hostname verification succeeds.

- If a match cannot be found in the `subject`, Entrust IdentityGuard next looks in the `subjectAltName` extension for a `dNSName` entry that contains a hostname that matches the one in the URL. Again, wildcards are supported.
- If a match is found, hostname verification succeeds. If a match is not found, hostname verification fails.

If you want to disable hostname verification, specify the following Java startup option for your client application:

```
-Didentityguard.disable.strict.ssl=true
```

With this startup option set to `true`, the Entrust IdentityGuard APIs do not verify that the host name specified in the HTTPS URL to the Web service matches the `subject` attribute or the `subjectAltName` extension of the server certificate when it creates the SSL connection to the Entrust IdentityGuard server. In this case, your client application is responsible for verifying the server host name.

Configuring trust

Entrust IdentityGuard with embedded Tomcat server stores certificates in the `keystore` file under `$IG_HOME/etc/`. During installation, Entrust IdentityGuard creates its own self-signed certificate, and stores it under the `tomcat` alias. It uses this certificate to perform SSL communications.

See the *Entrust IdentityGuard Installation Guide* if you want to replace the self-signed certificate with your own certificate (or if you want to generate a new self-signed certificate).

Your client application must trust the certificate associated with the `tomcat` alias for it to communicate with the Entrust IdentityGuard Web services using SSL. You can configure your client application to trust the Entrust IdentityGuard certificate directly, or to trust the associated CA certificate.

Note: The Java SE installed with your Entrust IdentityGuard system includes the `keytool` application. Use it to manage the Java keystore containing private keys and SSL certificates (X.509 chains and public keys). For complete documentation about `keytool`, see the `keytool` information at <http://java.sun.com/javase/>

To configure your client application to trust the Entrust IdentityGuard certificate

- 1** On UNIX, set the Entrust IdentityGuard environment variables:
 - a** On the Entrust IdentityGuard server, change to the Entrust IdentityGuard installation directory. For example:

```
cd /opt/entrust/identityguard120
```
 - b** Set the Entrust IdentityGuard environment variable:

```
.. ./env_settings.sh
```
- 2** On Windows:
 - a** Open a command window.
 - b** If necessary, change to the directory where the `keytool` is installed (under your Java installation).
- 3** Enter the following command (on one line) to export the keystore:

```
keytool -export -alias tomcat -file <path_to_file.cer> -keystore <path_to_keystore> -keypass entrust
```

where `<path_to_keystore>` is:
 - on UNIX, `$IG_HOME/etc/keystore`
 - on Microsoft Windows, `<IG_HOME>\etc\keystore`
- 4** Copy the generated file to the client application server, and import the Entrust IdentityGuard certificate to the client application's keystore.

Alternatively, you may import the associated CA certificate to your client application's keystore instead. The details of importing the certificate into your client application's keystore depend on your specific environment.

For example:

- If this is a standalone Java client application, you can import the certificate into the Java keystore as:

```
keytool -import -alias <your alias> -file <path_to_file.crt>
-keystore <enter full path to your client application keystore>
```

Enter the password when prompted.

- If this is a Java client application running in an application server, then refer to your product documentation for instructions for updating the application server's keystore.

Modifying certificate trust settings

There are some additional Java settings that you may wish to use to have your client application trust the Entrust IdentityGuard certificate. To enable these settings, modify the following Java startup options associated with the client application's JRE.

- `-Djavax.net.ssl.trustStore:`

Use this option to specify which certificate keystore your client application should use. Ensure that this keystore contains the Entrust IdentityGuard certificate. If the Entrust IdentityGuard certificate was not imported properly into the appropriate keystore, you may see an error message like this during execution of your client application:

```
"javax.net.ssl.SSLHandshakeException:
sun.security.validator.ValidatorException: No trusted certificate
found".
```

The format to specify this option is: -

```
Djavax.net.ssl.trustStore=<path_to_keystore>
```

where `<path_to_keystore>` is:

- on UNIX, `$IG_HOME/etc/keystore`
- on Microsoft Windows, `<IG_HOME>\etc\keystore`

- `-Didentityguard.disable.strict.ssl:`

Use this option to disable the Entrust IdentityGuard APIs from checking the Entrust IdentityGuard server name. By default, the fully qualified domain name of the Entrust IdentityGuard server must match the subject DN attribute of the certificate; otherwise, the Entrust IdentityGuard APIs will not trust the certificate presented by the Entrust IdentityGuard server. During execution, you may receive an error message similar to the following:

```
"javax.net.ssl.SSLPeerUnverifiedException: Hostname wrong"
```

To disable this check, specify this option as:

```
-Didentityguard.disable.strict.ssl=true
```

- `-Djavax.net.ssl:`

Use this option for troubleshooting purposes only. This option provides an SSL trace from the JVM (the trace is directed to standard out).

To enable tracing, specify this option as: `-Djavax.net.ssl=all`

In summary, your Java startup options may look similar to the following example:

```
java -classpath
.:IdentityGuardAdminAPIV11.jar:IdentityGuardAuthAPIV11.jar:axis.jar:activation.jar:jaxrpc.jar:commons-logging-1.0.4.jar:commons-
```

```
discovery.jar:commons-codec-1.4.jar:commons-httpclient-  
3.1.jar:mailapi.jar:saaj.jar:wsdl4j-1.5.1.jar -  
Didentityguard.disable.strict.ssl=true -  
Djavax.net.ssl.trustStore=/path/to/your/idgkeystore <name of your client  
application>
```

If your client application runs on an application server, then refer to the vendor's documentation for instructions on modifying the Java startup options.

Configuring SSL with Entrust IdentityGuard replicas

An Entrust IdentityGuard deployment consists of one primary server and zero or more replica servers. The Authentication and Administration Web services are available on all Entrust IdentityGuard servers (the primary and all replicas).

The following guidelines apply when using Entrust IdentityGuard replicas:

- Your client application can contact the Web services on any of these Entrust IdentityGuard servers; it is up to your client application to decide which Entrust IdentityGuard servers to contact.

Note: Depending on your Entrust IdentityGuard deployment scenario, the Web services running on the Entrust IdentityGuard replica servers may not provide the same functionality as the primary Entrust IdentityGuard service.

For example, if you are using an LDAP Directory or Active Directory repository and the file-based preproduced card functionality is enabled, any Administration API functions associated with these preproduced cards work only on the primary server. The same applies for unassigned tokens stored in a file-base repository. See the *Entrust IdentityGuard Server Administration Guide* for details.

- Configure your application to trust the certificates for each Entrust IdentityGuard instance. You can either import each Entrust IdentityGuard certificate individually into the keystore of your client application, or you can import the CA certificate associated with the Entrust IdentityGuard certificates.
 - If all Entrust IdentityGuard instances use their own self-signed certificates, import all the required Entrust IdentityGuard certificates into your client application keystore.
 - If you installed your own certificates after the Entrust IdentityGuard installation, you can import them.

If these certificates are all generated by the same CA, then you need to import only the CA certificate into your client application keystore.

Create a binding object

By using specific Java classes, you can retrieve a binding object that provides the logic necessary for connecting to the Entrust IdentityGuard services. These services perform all the transformations necessary to send a SOAP XML request to the server.

Create an authentication binding object

To create a new binding object that invokes authentication operations, you need to know the location of the authentication service. The following code sample illustrates the use of the `AuthenticationService_Service` interface class:

```
// Create the URL where the authentication service is located:
String urlString =
"http://localhost:8080/IdentityGuardAuthService/services/AuthenticationServiceV11";
java.net.URL authServiceUrl = new URL(urlString);
// Create a new binding using the URL just created:
try {
    AuthenticationService_ServiceLocator locator = new
AuthenticationService_ServiceLocator();

    AuthenticationServiceBindingStub binding = (AuthenticationServiceBindingStub)
locator.getAuthenticationService(authServiceUrl);

    return binding;
} catch (javax.xml.rpc.ServiceException se) {
    // Handle exception
    if (se.getLinkedCause() != null) {
        se.getLinkedCause().printStackTrace();
    }
    throw se;
}
```

AuthenticationService_ServiceLocator

The `AuthenticationService_ServiceLocator` class implements the interface defined by the `AuthenticationService_Service` class. Its role is to retrieve the appropriate binding class which can communicate with the service. In this case, it returns an instance of the `AuthenticationServiceBindingStub` class.

AuthenticationServiceBindingStub

The `AuthenticationServiceBindingStub` provides the necessary logic for creating the SOAP XML Web service request, and translates the XML response from the server back into Java objects.

Consult the Javadocs for a list of methods available under the `AuthenticationServiceBindingStub` class.

Create an administration binding object

To create a new binding object that invokes administration operations, you need to know the location of the Administration service. The following code sample illustrates the use of the `AdminService_Service` interface class:

```
// Create the URL where the administration service is located:
String urlString =
"https://localhost:8444/IdentityGuardAdminService/services/AdminServiceV11";
java.net.URL adminServiceUrl = new URL(urlString);
// Create a new binding using the URL just created:
try {
    AdminService_ServiceLocator locator = new AdminService_ServiceLocator();
    AdminServiceBindingStub binding = (AdminServiceBindingStub)
locator.getAdminService(adminServiceUrl);
    binding.setMaintainSession(true);
    return binding;
} catch (javax.xml.rpc.ServiceException se) {
    // Handle exception
    if (se.getLinkedCause() != null) {
        se.getLinkedCause().printStackTrace();
    }
    throw se;
}
```

Note: An administrator must be logged in before performing any administration operation. The `changePassword` and `ping` operations are the only ones that do not require an administrator to log in first.

AdminService_ServiceLocator

The `AdminService_ServiceLocator` class implements the interface defined by the `AdminService_Service` class. Its role is to retrieve the appropriate binding class, which can communicate with the service. In this case, it returns an instance of the `AdminServiceBindingStub` class.

AdminServiceBindingStub

The `AdminServiceBindingStub` provides the necessary logic for creating the SOAP XML Web service request, and translates the XML response from the server back into Java objects.

There are two ways to use the Administration service:

- Instruct the binding to maintain a session and call the login API using the Apache Axis APIs.

Note: This method may not be possible if you are using other SOAP/Web Service software.

- Use WS-Security headers to provide the administrator name and password with every request. This is shown in the Sample application using Entrust APIs.

Consult the third-party documentation if you are using different SOAP software.

Consult the Javadocs for a list of methods available under the `AdminServiceBindingStub` class.

Create a failover authentication binding object

To create a failover authentication binding object, you need to know the locations of the Authentication services. The following code sample illustrates the use of the `IGAAuthServiceFailoverFactory` interface class:

```
// Number of times to retry a service before marking as failed.
// (optional)
int numRetries = 2;
// How long to wait between retries in milliseconds (optional)
int delayBetweenRetries = 500;
// How long before allowing a failed service to be retried in
// seconds (optional)
long holdOffTime = 600;
// How long to wait before attempting to revert back to the
// primary service (optional)
long restorePrimaryTime = 3600;
// Commons logging implementation (optional)
Log logger = null;
// The implementation for pinging a service URL
TestConnection testCon = new TestConnectionImpl();
// The service URLs
String[] authServiceUrls = {
"http://localhost:8080/IdentityGuardAuthService/services/AuthenticationServiceV11",

"http://identityguard2.example.com:8080/IdentityGuardAuthService/services/Authenticati
onServiceV11", };

try {
    // Create the URI Failover Factory
    URIFailoverFactory failoverFactory = new URIFailoverFactory(authServiceUrls, new
TimeInterval(restorePrimaryTime), new TimeInterval(holdOffTime), logger, testCon);

    // Create Failover Configuration
    FailoverCallConfigurator failoverConfig = new FailoverCallConfigurator(numRetries,
delayBetweenRetries);

    AuthenticationService_ServiceLocator locator = new
AuthenticationService_ServiceLocator(failoverFactory, logger, failoverConfig);
    AuthenticationServiceBindingStub authBinding = (AuthenticationServiceBindingStub)
locator.getAuthenticationService();
```

```
        return authBinding;
    } catch (ServiceException se) {
        // Handle exception
        if (se.getLinkedCause() != null) {
            se.getLinkedCause().printStackTrace();
        }
        throw se;
    }
}
```

Create a failover administration binding object

To create a failover authentication binding object, you need to know the locations of the Authentication services. The following code sample illustrates the use of the `IGAdminServiceFailoverFactory` interface class:

```
// Number of times to retry a service before marking as failed.
// (optional)
int numRetries = 2;
// How long to wait between retries in milliseconds (optional)
int delayBetweenRetries = 500;
// How long before allowing a failed service to be retried in seconds
// (optional)
long holdOffTime = 600;
// How long to wait before attempting to revert back to the primary
// service (optional)
long restorePrimaryTime = 3600;
// Commons logging implementation (optional)
Log logger = null;
// The implementation for pinging a service URL
TestConnection testCon = new TestConnectionImpl();
// The service URLs
String[] adminServiceUrls = {
    "https://identityguard1.example.com:8444/IdentityGuardAdminService/services/AdminServiceV1",
    "https://identityguard2.example.com:8444/IdentityGuardAdminService/services/AdminServiceV1", };
// Admin credentials (store encrypted)
String adminid = "admin";
String adminpw = "Apple1995!";

try {
    // Create the URI Failover Factory
    URIFailoverFactory failoverFactory = new URIFailoverFactory(adminServiceUrls, new
    TimeInterval(restorePrimaryTime), new TimeInterval(holdOffTime), logger, testCon);

    AdminCredentialsImpl credentials = new AdminCredentialsImpl(adminid, adminpw);
    FailoverCallConfigurator failoverConfig = new
    FailoverCallConfigurator(credentials, numRetries, delayBetweenRetries);
    AdminService_ServiceLocator locator = new
    AdminService_ServiceLocator(failoverFactory, logger, failoverConfig);

    adminBinding = (AdminServiceBindingStub) locator.getAdminService();
}
```

```
adminBinding.setMaintainSession(true);
credentials.setBinding(adminBinding);

return adminBinding;
} catch (ServiceException se) {
    // Handle exception
    if (se.getLinkedCause() != null) {
        se.getLinkedCause().printStackTrace();
    }
    throw se;
}
```

Performance optimizations

Reuse a single binding object

When making API calls to Entrust IdentityGuard, it is recommended that you initialize a single authentication or administration binding object and reuse that binding for all invocations of the Entrust IdentityGuard APIs. By reusing the binding you avoid creating additional networking overhead and load on the server.

Migrating V9 or V10 services to V11

After upgrading to Entrust IdentityGuard release 12.0, your Java authentication client application can still access the V9 or V10 authentication services.

You can also update your client application to access the V11 authentication service.

This section provides you with the information you need to upgrade from V9 or V10 services to V11.

You can run V9, V10 and V11 client code in the same application. Entrust IdentityGuard 12.0 does not support these earlier versions of the Authentication and Administration APIs (V1 through V8).

Update service URLs

The URLs for accessing the V11 services are the same Entrust IdentityGuard URLs with `v11` rather than `v10` appended on the end. You must update the URLs to access V11 services.

Alternatively, you can use the failover APIs and specify the URLs of multiple IG servers. See “Create a binding object” on page [33](#) for examples.

For example, instead of accessing the Authentication service using

```
http://<host>:8080/IdentityGuardAuthService/services/AuthenticationServiceV10  
use
```

```
http://<host>:8080/IdentityGuardAuthService/services/AuthenticationServiceV11
```

To access the new Administration service, update the URL from

```
https://<host>:8444/IdentityGuardAdminService/services/AdminServiceV10  
to
```

```
https://<host>:8444/IdentityGuardAdminService/services/AdminServiceV11
```

Note: Accessing the old URL with the new V11 API will result in errors.

Update proxy class library

Update the V10 authentication proxy class library to V11. Change the authentication proxy class library from `IdentityGuardAuthAPIV10.jar` to `IdentityGuardAuthAPIV11.jar`.

Change the administration proxy class library from `IdentityGuardAdminAPIV10.jar` to `IdentityGuardAdminAPIV11.jar`

Update the import statements

Change the import statements in your Java programs.

Change

```
import com.entrust.identityGuard.authenticationManagement.wsv10.*
```

to

```
import com.entrust.identityGuard.authenticationManagement.wsv11.*
```

Change

```
import com.entrust.identityGuard.userManagement.wsv10.*
```

to

```
import com.entrust.identityGuard.userManagement.wsv11.*
```

Changes to the Authentication API

For full descriptions of these new and changed classes, see the HTML documentation for Java included with your installation.

All V10 Extended APIs were amalgamated into their associated V11 classes.

No Authentication API methods were removed.

The following table describes classes have been modified or added. For “New Fields”, getters and setter are not explicitly listed but are implied.

Class	Changes
AuthenticationServiceBindingImpl	Removed usage of extended methods from V10 Added authenticateAnonymousCertChallenge getAnonymousCertChallenge
AuthenticationServiceBindingSkeleton	Removed usage of extended methods from V10 Added authenticateAnonymousCertChallenge getAnonymousCertChallenge
AuthenticationServiceBindingStub	Removed usage of extended methods from V10 Added authenticateAnonymousCertChallenge getAnonymousCertChallenge
AuthenticationService_PortType	Removed usage of extended methods from V10 Added authenticateAnonymousCertChallenge

Class	Changes
	getAnonymousCertChallenge
AuthenticateAnonymousCertChallengeCallParms	New class Contains the parameters passed in a call to authenticateAnonymousCertChallenge.
GetAnonymousCertChallengeCallParms	New class Contains the parameters passed in a call to getAnonymousCertChallenge.
GenericAuthenticateResponse	Added Fields accessGroups userName
GenericChallenge	Added Fields anonymousChallengeURL QRCode
GenericChallengeParms	Added Fields anonymousCertChallengeCallback anonymousCertChallengeQRCodeSize
MachineSecret	Added Fields machineExternalId
RiskScoringResult	Added Fields externalRiskParameters externalRiskScore externalRiskScoreStatus

Changes to the Administration API

For full descriptions of these new and changed classes, see the HTML documentation for Java included with your installation.

All V10 Extended APIs were amalgamated into their associated V11 classes.

The following classes have been removed.

- UserSmartCredentialCreateCallParmsEx
- UserSmartCredentialParmsEx

The following table describes classes that have been added. For added fields, getters and setter are not explicitly listed but are implied.

Class name	Description
ExternalRiskResetAccountRelationshipsCallParms	New class This structure contains the parameters passed in to an external risk engine to reset any relationships a user's account may have to external entities, such as mobile devices.
ExternalRiskScoreGetCallParms	New class This structure includes parameters passed to an external risk score application such as Entrust TransactionGuard to generate an external risk score used as part of risk-based authentication.
ExternalRiskScoringReturn	New class This structure includes details on the results of external risk engine scoring.
ManagedCaChangeDNMode	New class Defines how the Directory is treated during a digital identity create/recover DN change operation.

The following classes have been modified: For “New Fields”, getters and setter are not explicitly listed but are implied.

Class name	Description
AdminService_PortType	Added Methods externalRiskResetAccountRelationships externalRiskScoreGet
AdminServiceBindingImpl	Added Methods externalRiskResetAccountRelationships externalRiskScoreGet
AdminServiceBindingSkeleton	Added Methods externalRiskResetAccountRelationships externalRiskScoreGet
AdminServiceBindingStub	Added Methods externalRiskResetAccountRelationships externalRiskScoreGet\
DigitalIdConfigEntrustSpecificInfo	changeDNMode supportsDNChange
DigitalIdConfigEntrustSpecificParms	Added Fields changeDNMode supportsDNChange

Class name	Description
MachineSecretInfo	Added Field machineExternalId
MachineSecretParms	Added Field machineExternalId
RepositoryInfo	Added Field userEnrollmentSearchAttributes
TrustedExecutionEnvType	Added Field SECURE_TRANSFER
UserDigitalIdInfo	Added Field previousDN
UserEnrollFilter	Added Field searchValue
UserFilter	Added Fields pvnExpiryEndDate pvnExpiryStartDate
UserInfo	Added Fields accessGroups resetAccountRelationshipsAllowed usePolicyForAccessGroups resetAccountRelationshipsAllowed
UserParms	Added Fields accessGroups addAccessGroups removeAccessGroups usePolicyForAccessGroups
UserPVNCreateParms	Added Field PVNLifetime
UserPVNInfo	Added Field expireDate
UserPVNSetParms	Added Field PVNLifetime
UserSmartCredentialInfo	Added Fields derivedCredentialAuthenticatingCertificate derivedCredentialAuthenticatingCertificateValid derivedCredentialAuthenticatingCertificateValidityCheckDate

Class name	Description
UserSmartCredentialParms	Added Field derivedCredentialAuthenticatingCertificate
UserSpecInfo	Added Fields accessGroups maxUserAccessGroups PVNLifetime
UserSpecParms	Added Fields accessGroups addAccessGroups maxUserAccessGroups PVNLifetime PVNLifetime removeAccessGroups
UserTokenActivateCallParms	Added Field secureTokenExchangeReq
UserTokenActivateResult	Added Field secureTokenExchangeRsp
UserTokenFilter	Added Field hasTokenDrift
UserTokenParms	Added Field resetDrift

Chapter 3:

Authentication approaches

This chapter explains the various Entrust IdentityGuard authentication approaches you can take using the Entrust IdentityGuard Authentication API.

Anonymous grid authentication

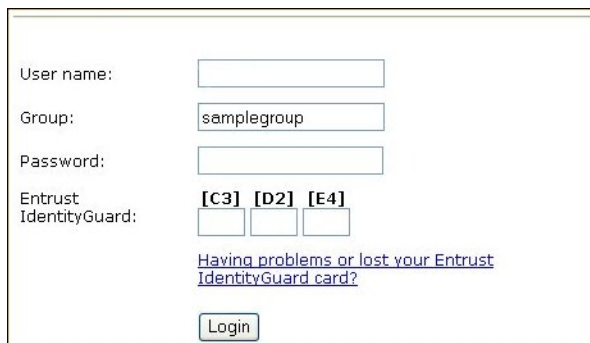
Note: This approach applies to a grid or temporary PIN authentication only.

Use this approach if you want to combine first and second-factor authentication on a single page—that is, you do not want to present your users with two authentication pages. In this approach, the existing system does not know the identity of the user until after login and authentication; the user is anonymous until both first and second-factor authentication are complete. This is sometimes called one-step authentication because both first and second-factor authentication are presented on one page.

Note: You can disable Anonymous authentication. See the *Entrust IdentityGuard Server Administration Guide* for details on setting policy to disable anonymous grid authentication.

In anonymous grid authentication, you add an Entrust IdentityGuard challenge to your existing authentication page, as the following figure.

Anonymous grid authentication example



The screenshot shows a web form for login. It contains the following fields and elements:

- User name:** A text input field.
- Group:** A text input field containing the value "samplegroup".
- Password:** A text input field.
- Entrust IdentityGuard:** A section containing three small square icons labeled [C3], [D2], and [E4].
- Link:** A blue hyperlink text that reads "Having problems or lost your Entrust IdentityGuard card?".
- Login button:** A button labeled "Login" at the bottom of the form.

Attention: When you use anonymous grid authentication, Entrust IdentityGuard does not track challenges per user. Your own authentication application must ensure that the challenge returned to Entrust IdentityGuard by `authenticateAnonymousChallenge` is the same as the challenge returned for the user by `getAnonymousChallenge`. Otherwise, a previously used challenge response can be successfully used again. This increases the risk of an attacker capturing and reusing a challenge.

Anonymous grid authentication method

If the user and group are unknown, implement anonymous grid authentication using `getAnonymousChallenge` to issue the challenge, and `authenticateAnonymousChallenge`

to authenticate the response. The policy associated with the default group is used to generate the challenge.

Use `getAnonymousChallengeForGroup` if your organization uses grids of different size or number of characters per grid cell for users in different Entrust IdentityGuard groups. Use `authenticateAnonymousChallenge` to authenticate the response. The policy associated with the specified group is used to generate the challenge.

Note: It is possible for the client application to construct its own challenge set and bypass `getAnonymousChallenge`. This procedure requires in-depth knowledge of the applicable user policy, and is not recommended.

Security considerations

One-step authentication is not as secure as two-step authentication in the situation where an attacker has obtained a portion of a user's grid. When writing applications using anonymous grid authentication, consider the following:

- Ensure that the grid coordinates specified as part of the challenge response are the same grid coordinates that were delivered in the original anonymous challenge request. Without this checking, an attacker can choose challenge coordinates for which they have the answers.
- Because the user has not been identified, there is no way to associate a challenge response with a specific user. One approach to stopping an attacker from cycling to another set of grid coordinates (in the hope of finding something they can answer) is to associate the challenge with the user session until their current session expires.

Keep in mind, however, that knowledgeable attackers realize that they can terminate an existing session by restarting their Web browser, or by deleting the cookie that contains their current session ID.

Anonymous grid authentication sample

The following code fragments show how to issue an anonymous challenge for a user or a user in a specific group, and how to authenticate the response.

If the user is part of the default group, or all groups use grids of the same size and with the same number of characters per grid, get the challenge set this way:

```
GenericChallenge challenge = authBinding.getAnonymousChallenge();
ChallengeSet challengeSet = challenge.getGridChallenge();
```

For a user in a group other than the default, or if different groups use grids of a different size or number of characters per cell, get the challenge set this way:

```
GetAnonymousChallengeForGroupCallParms callParms;
callParms.setGroup(group);

GenericChallenge challenge =
    authBinding.getAnonymousChallengeForGroup(callParms);
```

```
ChallengeSet challengeSet = challenge.getGridChallenge();
```

Convert the challenge if required by the client application interface (see [String conversion sample](#) on page 47).

After the client application receives the user ID and challenge response, authenticate the response as follows:

```
GenericAuthenticateParms parms = new GenericAuthenticateParms();
Response response = new Response();
response.setResponse(challengeResponse);
AuthenticateAnonymousChallengeCallParms
    authenticateAnonymousChallengeCallParms
        = new AuthenticateAnonymousChallengeCallParms();
authenticateAnonymousChallengeCallParms.setUserId(userId);
authenticateAnonymousChallengeCallParms.setChallengeSet(
    ms_cachedAnonymousChallengeSet);
authenticateAnonymousChallengeCallParms.setResponse(response);
GenericAuthenticateResponse resp =
    authBinding.authenticateAnonymousChallenge(
        authenticateAnonymousChallengeCallParms);
```

Challenge retention

When you use `getGenericChallenge` for a user, the challenge information is retained. That is, Entrust IdentityGuard issues the same challenge each time the user requests one until the user answers the original challenge correctly.

When you use `getAnonymousChallenge` or `getAnonymousChallengeForGroup`, the challenge information is not retained. Entrust IdentityGuard does not track challenges for the user and issues a new challenge each time the user requests one. Therefore, anonymous challenges create a potential security risk. Attackers who have already captured some challenge responses can cycle through challenges until they get a challenge they can answer.

As mentioned earlier, this can be a security risk; see [Security considerations](#) on page 46.

String conversion sample

Entrust IdentityGuard returns a grid challenge as a set of integers. The client application can convert the challenge to anything it needs before displaying it. This example converts the challenge to a string, such as "[A,1] [B,2] [C,3]".

Note: The cell coordinates used in the challenge depend on the row and column header used on the grid.

```
Challenge[] challArr = challengeSet.getChallenge();
StringBuilder challengeString = new StringBuilder();
```

```

for (int i = 0; i < challArr.length; i++) {
    if (i != 0) {
        challengeString.append(" ");
    }
    Challenge chall = challArr[i];
    challengeString.append('[');
    // Cards with more than 26 columns need a more sophisticated algorithm
    challengeString.append((char) (chall.getColumn() + (int) 'A'));
    challengeString.append(',');
    challengeString.append(chall.getRow() + 1);
    challengeString.append(']');
}

```

Through conversion, you can apply additional security to make challenges difficult to steal. For instance, you can obfuscate entries and avoid machine-readable characters by converting the challenge to images rather than text.

Generic authentication

This approach can include one or more authentication methods: grid, token, one-time password (OTP), knowledge-based authentication, password, certificate, and external authentication.

You can use a personal verification number (PVN) when authenticating with grids, token, OTPs, and temporary PINs.

Generic API methods

To implement generic authentication, use `getGenericChallenge` to issue the challenge and `authenticateGenericChallenge` to authenticate the response.

When calling `getGenericChallenge`, you can select which authentication type to use in one of three ways.

If you do not specify an authentication type in the `GenericChallengeParms authenticationType` parameter, and you do not use the `authenticationTypeList` parameter, Entrust IdentityGuard returns a challenge for the first authentication type in the allowed authentication type policy.

If you specify a single authentication type in the `GenericChallengeParms authenticationType` parameter, IdentityGuard returns a challenge of that authentication type.

If you specify a list of authentication types in the `GenericChallengeParms authenticationTypeList` parameter, IdentityGuard selects an authentication type to use based on the following algorithm:

- The intersection of the policy authentication type list and the specified list is taken. If the specified list is empty, all authentication types in the policy list are taken. Each authentication type in the resulting list is considered in the order it appears in the policy.
 - for GRID, return a challenge only if the user has an active grid card or a temporary PIN.

- for TOKENRO, return a challenge only if the user has an active read-only token or a temporary PIN.
- for TOKENCR, return a challenge only if the user has an active challenge-response token or a temporary PIN.
- for QA, return a challenge only if the user has question and answer values defined.
- for EXTERNAL, return a challenge only if the user has external authentication configured.
- for OTP, return a challenge only if the user has OTP enabled. If the application requested delivery, return a one-time password only if the user has the necessary delivery mechanisms configured.
- for CERTIFICATE, return a challenge only if the user has a valid certificate.

The authentication method or methods available to `getGenericChallenge` are set based on the setting of the `userspec` policy attribute Normal Security Authentication Types or Enhanced Security Authentication Types, depending on whether the challenge is to be issued at the Normal or Enhanced security level. You can also overwrite the default setting and specify the authentication type as shown in the sample code in “Generic API code sample” on page 55.

See the *Entrust IdentityGuard Server Administration Guide* for more information on the `userspec` policy.

Grids

You can provide your users with grid cards for authentication and, optionally, a PVN. Grid authentication uses cards, each printed with a grid, as the authentication lookup tool. When asked to authenticate with a grid, the challenge presents the user with coordinates: **B3**, **H1**, for instance. The user looks up those coordinates on their grid cards, and responds by typing the corresponding value. Each grid card is unique and carries a serial number, so every user can be uniquely identified and authenticated.

See [Generic API code sample](#) on page 55 for a detailed example.

Grid challenges also accept a temporary PIN as a response.

Two-step authentication with a grid

	A	B	C	D	E	F	G	H	I	J
1	1	7	3	9	5	4	8	5	4	9
2	9	2	5	3	6	2	8	4	1	3
3	4	6	9	1	7	6	8	0	7	
4	1	5	7	8	5	1	7	2		
5	6	8	6	8	1	7	4			

Serial #1234567

Tokens

You can authenticate users with a dynamic password generated by a token device and, optionally, a personal verification number (PVN). Tokens provide an alternative to grid card authentication. Token challenges accept temporary PINs as a response, just as grid cards do.

See [Generic API code sample](#) on page 55 for a detailed example.

Entrust soft tokens can also be used to sign transactions details. To perform transaction delivery, simply include the transaction details in the `GenericChallengeParams` when requesting a `TOKENRO` authentication from Entrust IdentityGuard.

See [Transaction authentication](#) on page 80 for a code sample that delivers and authenticates a transaction.

One time password (OTP)

You can authenticate users with a one-time password (OTP).

Your organization can issue a one-time password by email, a text message, or phone call. The user then enters the password online to enter your site or to initiate a secure transaction.

Entrust IdentityGuard provides three out-of-band delivery mechanisms for one-time passwords:

- JavaMail

Delivers OTPs by email or SMS.

- Authenticate

Delivers OTPs using a voice call to a telephone.

- Clickatell

Delivers OTPS using SMS messages to a mobile phone.

Example code showing how to use out-of-band authentication using OTPs is shown below. These are the steps the code follows:

- 1 The application calls `getGenericChallenge`, requesting an OTP challenge and specifying that the challenge only be generated if the OTP can be delivered out-of-band. This returns a list of available values for `deliveryconfig`; these are the delivery mechanisms.

Note: If OTP delivery is requested, but is not allowed for the user, and there are no other authentication types available, then an error message is returned. Possible reasons for OTP delivery failure include:

- The user does not have contact information that is mapped to a valid delivery instance.
- OTP Delivery Enabled is false at the individual user setting level or by associated policy.

- 2 The user selects one or more delivery mechanisms.
 - 3 The application calls `getGenericChallenge` again, this time specifying the chosen delivery mechanisms. If a PVN is also required (PVN is required by Authenticate telephone OTP delivery), Entrust IdentityGuard also prompts for a PVN.
 - 4 Entrust IdentityGuard generates the OTP and delivers it using one or more out-of-band delivery mechanisms.
-

```
// Call getGenericChallenge without specifying OOB delivery
// configuration info. A list of available delivery mechanisms
// is returned.

GenericChallengeParms genericChallengeParms = new GenericChallengeParms();
genericChallengeParms.setAuthenticationType(AuthenticationType.OTP);
GetGenericChallengeCallParms callParms = new GetGenericChallengeCallParms();
callParms.setUserId(userid);
callParms.setParms(genericChallengeParms);
GenericChallenge genericChallenge = null;
// Call getGenericChallenge; the OTP will be generated,
// but not delivered.
try

{
    genericChallenge = UtilityMethods.getAuthBinding().getGenericChallenge(callParms);
} catch (AuthenticationFault fault) {
    // Handle IdentityGuard authentication fault
```

```

        fault.printStackTrace();
        return;
    } catch (Exception ex) {
        // Handle general exception
        ex.printStackTrace();
        return;
    }
    // Get the available contact info from the response, if any
    OTPChallenge otpChallenge = genericChallenge.getOTPChallenge();
    // It is possible that manual delivery is required
    if (otpChallenge.getManualDeliveryRequired() != null &&
        otpChallenge.getManualDeliveryRequired()) {
        // Implement strategy for getting OTP delivered to the user
        System.err.println("The OTP must be delivered manually to the user!");
        return;
    }
    DeliveryMechanism[] deliveryConfigList = otpChallenge.getDeliveryMechanism();
    // Just get the first available; a real app would prompt
    // the user for a choice if there is more than one option.
    assert deliveryConfigList != null;
    String[] contactinfoLabelsToUse = new String[] {
        deliveryConfigList[0].getContactInfoLabel() };

    // Ask again, this time requesting delivery of the OTP
    genericChallengeParms.setContactInfoLabel(contactinfoLabelsToUse);
    try {
        genericChallenge = UtilityMethods.getAuthBinding().getGenericChallenge(callParms);
        // Delivery can be successful or it can fail.
        otpChallenge = genericChallenge.getOTPChallenge();
        DeliveryMechanism[] successList = otpChallenge.getDeliveryMechanismUsed();
        if (successList != null) {
            for (int i = 0; i < successList.length; i++) {
                System.out.println("Successfully delivered OTP to " +
                    successList[i].getContactInfoLabel());
            }
        }
        DeliveryMechanism[] failureList = otpChallenge.getDeliveryMechanismFailed();
        if (failureList != null) {
            AuthenticationFault[] failureReasons =
                otpChallenge.getDeliveryMechanismFailureReason();
            for (int i = 0; i < failureList.length; i++) {
                System.err.println("Failed to deliver OTP to " +
                    failureList[i].getContactInfoLabel() + " because: " +
                    failureReasons[i].getErrorMessage());
            }
        }
    }

```

```
    }  
  } catch (AuthenticationFault fault) {  
    // handle IdentityGuard authentication fault  
    fault.printStackTrace();  
  } catch (Exception ex) {  
    // handle general exception  
    ex.printStackTrace();  
  }  
}
```

You can use the delivery implementations provided by Entrust, as described above, or you can develop your own delivery system using the Administration API.

Also see [“Create and send an OTP”](#) on page 91 for more information about using the Administration API for managing OTPs.

Knowledge-based questions and answers

Your organization can construct a question list for users logging in. Users are asked questions based on information that they entered in the past.

What year did you buy your first car?

Which historical figure do you most admire?

Who is your most memorable cartoon character?

For example, during enrollment the consumer may select and provide answers to easily-remembered questions such as those shown in the preceding figure.

See [Generic API code sample](#) on page 55 for a detailed example.

External authentication

The external authentication feature provided with Entrust IdentityGuard lets you manage first-factor authentication using a Windows domain controller or an LDAP directory. Typically, you would use external authentication as the first layer of a multifactor Entrust IdentityGuard authentication regime. First-factor authentication identifies users and lets you direct them to the appropriate risk-based or second-factor authentication method.

You can configure the Entrust IdentityGuard Radius proxy to use external authentication for first-factor authentication.

See [Generic API code sample](#) on page 55 for a detailed example.

Password authentication

Instead of relying on an external resource such as a Radius server or other external authentication manager for first-factor authentication, your application can use the password authentication feature provided with Entrust IdentityGuard. Password authentication is always limited to first-factor authentication when authenticating through the Radius proxy.

See [Generic API code sample](#) on page 55 for a detailed example.

Certificate challenge response authentication

There are two forms of certificate authentication: one is integrated with Risk-Based Authentication (see "[Risk-based authentication \(RBA\)](#)" on page 78), and the second is a challenge response authentication.

Your application requests a certificate challenge by specifying an authentication mechanism of type CERTIFICATE. Entrust IdentityGuard returns random data to be signed by the application. Upon receipt of the encoded signature, Entrust IdentityGuard checks it to ensure that the signature is valid and that the certificate used for the signature is valid. Certificate validation conforms to RFC 2459, the X.509 standard.

Entrust IdentityGuard does not supply an off-the-shelf client application that can sign the challenge. However, a command line sample application called `IGCertAuth.java` is provided in:

- On Windows: `<IG_HOME>/client/sample`
- On Unix: `$IG_HOME/client/sample`

This sample demonstrates the use of the Entrust Java toolkit to read a certificate from an Entrust Security Manager profile. To use this sample as the basis for your own code, you must license the Java toolkit from Entrust separately.

Note: There is no corresponding sample in available in C#.

Generic API code sample

The sample shows how to request a generic challenge for a given user, and how to authenticate using the response.

```
// As an optional step, retrieve the allowed authentication types for the
// generic authentication operation. You could also retrieve the allowed types
// for a specific group. Remember that even if an authentication type is allowed
// for a user, it will not be usable unless the user has been assigned the
// appropriate authentication mechanism.
GetAllowedAuthenticationTypesCallParms getTypeCallParms = new
GetAllowedAuthenticationTypesCallParms();
getTypeCallParms.setUserId(userid);
AllowedAuthenticationTypes types = null;
try {
    types =
UtilityMethods.getAuthBinding().getAllowedAuthenticationTypes(getTypeCallParms);
} catch (AuthenticationFault fault) {
    // handle IdentityGuard authentication error
    fault.printStackTrace();
    return;
} catch (Exception ex) {
    // handle non-IdentityGuard error
    ex.printStackTrace();
    return;
}

// Prefer QA if it is acceptable,
// otherwise use the default type (first one supported by policy).
AuthenticationType authType = AuthenticationType.QA;
AuthenticationType[] genericAuthTypes = types.getGenericAuth();
boolean acceptableAuthType = false;
for (int i = 0; i < genericAuthTypes.length; i++) {
    if (genericAuthTypes[i].equals(authType)) {
        acceptableAuthType = true;
        break;
    }
}
if (!acceptableAuthType) {
    // Cause first type in the generic authentication list to be used
    authType = null;
}

// Create the generic challenge parameters as follows:
```

```

GenericChallengeParms genericChallengeParms = new GenericChallengeParms();
genericChallengeParms.setAuthenticationType(authType);
// If OTP is chosen, assume user has a default delivery mechanism
genericChallengeParms.setUseDefaultDelivery(Boolean.TRUE);
// Get the generic challenge
GetGenericChallengeCallParms callParms = new GetGenericChallengeCallParms();
callParms.setUserId(userid);
callParms.setParms(genericChallengeParms);
GenericChallenge genericChallenge = null;
try {
    genericChallenge = UtilityMethods.getAuthBinding().getGenericChallenge(callParms);
} catch (AuthenticationFault fault) {
    // handle IdentityGuard authentication error
    System.err.println("Error requesting authentication challenge for type " +
authType + ": " + fault.getErrorMessage());
    return;
} catch (Exception ex) {
    // handle non-IdentityGuard error
    ex.printStackTrace();
    return;
}

// Fill up the authentication parameters and response with user input
GenericAuthenticateParms authParms = new GenericAuthenticateParms();
Response response = new Response();
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

// Process the challenge:
authType = genericChallenge.getType();
// If the authentication method is a grid:
if (authType.equals(AuthenticationType.GRID)) {
    ChallengeSet challengeSet = genericChallenge.getGridChallenge();
    // Code to construct a challenge prompt
    String prompt =
UtilityMethods.getGridChallengePrompt(challengeSet.getChallenge());
    System.out.print("Please answer the grid challenge " + prompt + ": ");
    response.setResponse(in.readLine().split(" "));
}
// If the authentication method is a read-only token:
else if (authType.equals(AuthenticationType.TOKENRO)) {
    TokenChallenge tokenChallenge = genericChallenge.getTokenChallenge();
    // For simplicity, we assume user only has one response only token
    System.out.print("Enter your response for the token with serial number " +
tokenChallenge.getTokens()[0].getSerialNumber() + ": ");

```



```

        response.setResponse(new String[] { in.readLine() });
    }
    // If the authentication method is a challenge-response token
    else if (authType.equals(AuthenticationType.TOKENCR)) {
        TokenChallenge tokenChallenge = genericChallenge.getTokenChallenge();
        // Get the token challenge
        // For simplicity, we assume user only has one CR token
        System.out.println("For the challenge " + tokenChallenge.getChallenge() + ", enter
your response for the token with serial number " +
tokenChallenge.getTokens()[0].getSerialNumber()
        + ": ");
        response.setResponse(new String[] { in.readLine() });
    }
    // If the authentication method is knowledge-based
    // (questions and answers):
    else if (authType.equals(AuthenticationType.QA)) {
        String[] questions = genericChallenge.getQChallenge();
        String[] answers = new String[questions.length];
        System.out.println("Please provide answers for the following questions:");
        for (int i = 0; i < questions.length; i++) {
            System.out.print(questions[i] + " ");
            answers[i] = in.readLine();
        }
        response.setResponse(answers);
    }
    // If the authentication method is one-time password:
    else if (authType.equals(AuthenticationType.OTP)) {
        // Depending on the delivery configuration setup, IdentityGuard may deliver
        // the OTP to user via out-of-band mechanism
        System.out.print("Please enter your OTP: ");
        response.setResponse(new String[] { in.readLine() });
    }
    // If the authentication method is Entrust IdentityGuard password:
    else if (authType.equals(AuthenticationType.PASSWORD)) {
        // Display the password challenge
        char[] currentpwd = UtilityMethods.getPassword("Please enter your current
password: ");
        response.setResponse(new String[] { new String(currentpwd) });
        Arrays.fill(currentpwd, ' ');
        // Add code to handle the password change requirement
        PasswordChallenge pswdChall = genericChallenge.getPasswordChallenge();
        if (pswdChall.getChangeRequired() != null && pswdChall.getChangeRequired()) {
            // Optionally, add code to display the password rules

```

```

        // PasswordRules pswdRules = pswdChall.getPasswordRules();
        char[] newpwd, confirmpwd;
        do {
            newpwd = UtilityMethods.getPassword("Please enter your new password: ");
            confirmpwd = UtilityMethods.getPassword("Please confirm your new password:
");
            } while (!Arrays.equals(newpwd, confirmpwd));
        authParms.setNewPassword(new String(newpwd));
        Arrays.fill(newpwd, ' ');
        Arrays.fill(confirmpwd, ' ');
    }
} else if (authType.equals(AuthenticationType.CERTIFICATE)) {
    // getGenericChallenge() has returned a certificate challenge
    // which is a random string with size specified by policy.
    // This challenge value is retained by IdentityGuard until it is
    // successfully authenticated.
    // The application should generate the response as follows:
    // o prepend the string "Entrust IdentityGuard:"
    // (not including the quotes) to the challenge.
    // o hash the resulting string using the hashing algorithm
    // specified in the certificate challenge.
    // o sign the resulting hash in PKCS#7 SignedData format.
    // o return the Base-64 encoded signature as the authentication response.
    // For more details, see the example program IGCertAuth.java in the
    // client/sample folder.
    System.err.println("This program does not support certificate authentication!");
    return;
} else if (authType.equals(AuthenticationType.EXTERNAL)) {
    // Add code to handle the external authentication
    // Assume external authentication is password based.
    response.setResponse(new String[] { new String(UtilityMethods.getPassword("Please
enter your external password: ")) });
}

// If the authentication method is NONE, do the following:
else if (authType.equals(AuthenticationType.NONE)) {
    // For NONE, do not call authenticatGenericChallenge --
    // the user does not require authentication
    System.out.println("User does not require authentication.");
    return;
}

PVNInfo pvnInfo = genericChallenge.getPVNInfo();
if (pvnInfo != null) {
    // Determine if the PVN is required

```

```

    if (pvnInfo.isRequired()) {
        // If the PVN is available, prompt the user to enter it
        boolean changeRequired;
        if (pvnInfo.getAvailable() != null && pvnInfo.getAvailable()) {
            // Prompt the user to enter their PVN
            char pvn[] = UtilityMethods.getPassword("Please enter your Personal
Verification Number: ");
            response.setPVN(new String(pvn));
            Arrays.fill(pvn, ' ');
            // If PVN requires updating, prompt the user to update it
            changeRequired = pvnInfo.getChangeRequired() != null &&
pvnInfo.getChangeRequired();
        } else {
            // Need to provision a PVN since one doesn't exist.
            changeRequired = true;
        }
        if (changeRequired) {
            // Please enter a new x-digit PVN and confirm it
            char[] newpvn, confirmpvn;
            do {
                newpvn = UtilityMethods.getPassword("Please enter a new " +
pvnInfo.getLength() + " digit Personal Verification Number: ");
                confirmpvn = UtilityMethods.getPassword("Please confirm your new PVN:
");
            } while (!Arrays.equals(newpvn, confirmpvn));
            authParms.setNewPVN(new String(newpvn));
            Arrays.fill(newpvn, ' ');
            Arrays.fill(confirmpvn, ' ');
        }
    }
}

// Once the client application gets a challenge response from the
// user, authenticate the response
try {
    authParms.setAuthenticationType(authType);
    AuthenticateGenericChallengeCallParms authCallParms = new
AuthenticateGenericChallengeCallParms();
    authCallParms.setUserId(userid);
    authCallParms.setParms(authParms);
    authCallParms.setResponse(response);
    GenericAuthenticateResponse authResponse =
UtilityMethods.getAuthBinding().authenticateGenericChallenge(authCallParms);
    String name = authResponse.getFullName();
}

```

```
        if (name == null)
            name = userid;
        System.out.println("The user " + name + " has successfully authenticated!");
    } catch (AuthenticationFault fault) {
        // handle IdentityGuard authentication error
        System.err.println("Error performing authentication with type " +
            authType.getValue() + ": " + fault.getErrorMessage());
    } catch (Exception ex) {
        // handle non-IdentityGuard error
        ex.printStackTrace();
    }
```

Mutual authentication

Your organization needs to have confidence in the identity of the user trying to log in. Likewise, users need to be confident that they are initiating a transaction with the intended organization. Entrust IdentityGuard provides ways for both parties to authenticate each other.

Mutual authentication refers to replay features or specific authentication methods that allow the user to validate the organization at the same time the organization authenticates the user.

Grid and token serial number and location replay

Grid and token authentication include built-in mechanisms for mutual authentication; serial number replay and location replay.

Serial number replay is based on the serial number of the grid or token. Each grid card and each token has a unique serial number that is known only to the issuing organization and the user. During login, you can display this number to the user before prompting for user authentication.

Before entering a password or challenge response, users confirm that the serial number displayed on the Web site matches the one on their grid card or token. If it does, users can be confident they are on the legitimate Web site.

For a grid challenge, you can display the grid card serial number to the user in the challenge, using code like the following:

```
//Get the card serial number
String[] sernum = challengeSet.getCardSerialNumbers();
```

Users may have more than one valid grid card (active and pending), so this returns a string array.

To display the token serial number, use code similar to the following:

```
// Get the token serial numbers
TokenData[] tokens = tokenChallenge.getTokens();
String[] tokenSernum = new String[tokens.length];
for (int i = 0; i < tokens.length; i++)
{
    tokenSernum[i] = tokens[i].getSerialNumber();
}
```

Location replay displays specific grid coordinates to the user. This confirms to the user that the site has specific knowledge of the contents of the user's grid and, therefore, must be legitimate.

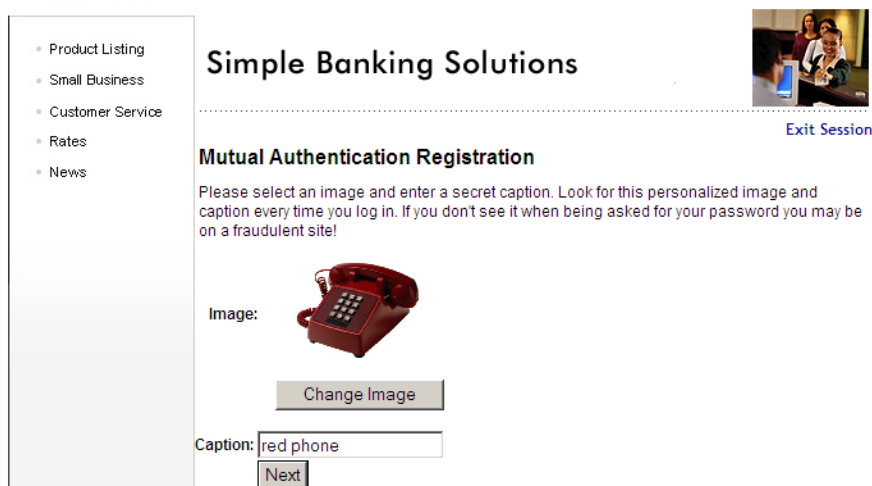
Knowledge-based authentication

You can provide questions that challenge users to provide information that only they know. This helps an organization verify the user but, since the user recognizes the source or origin of the questions, the user also recognizes the site is legitimate. (See “Knowledge-based questions and answers” on page 53 for more information.)

Image and caption replay

Another feature available with generic authentication is image and caption replay. In this case, as part of the registration process, a user selects an image from a gallery and enters a custom image caption that is later shown during login. By personalizing the login with the image and message, as shown in the following figure, the user recognizes the site is legitimate during login because a fraudulent one would not have this information to replay.

Choosing a custom image and caption




Product Listing
Small Business
Customer Service
Rates
News

Simple Banking Solutions

[Exit Session](#)

Mutual Authentication Registration

Please select an image and enter a secret caption. Look for this personalized image and caption every time you log in. If you don't see it when being asked for your password you may be on a fraudulent site!

Image: 

[Change Image](#)

Caption:

[Next](#)

Image and caption replay samples

In generic authentication, both the generic challenge response and the generic authentication response can contain authentication secrets such as images and captions used for mutual authentication.

There are two operations you can perform with mutual authentication secrets:

- Fetch them so that the user knows they have contacted a legitimate Web site.
- Allow users to set or change their mutual authentication secrets.

Mutual authentication secrets are usually set and changed through the Administration API using an application like Entrust IdentityGuard Self-Service Module, but it is also possible to set and save mutual authentication secrets with the Authentication API.

Fetching mutual authentication secrets

You can fetch mutual authentication secrets using the `getGenericChallenge` call. (The user policy for the user group must have the `returnauthsecretwithchallenge` attribute set to `true`.)

When using `getGenericChallenge`, you are using mutual authentication secrets to prevent a user from responding to a second-factor challenge, unless the Web site correctly displays the user's secrets.

In this case, the user should already be authenticated using first-factor authentication, so you can be confident that the user is who they say they are. This prevents the application from showing a user's mutual authentication secrets to an attacker who has learned the user's user ID.

You can also fetch mutual authentication secrets using the `authenticateGenericChallenge` call. If you are fetching mutual authentication secrets as part of the `authenticateGenericChallenge` call, then it is assumed that you are performing second-factor authentication before first-factor authentication, and the mutual authentication secrets are protecting the user from disclosing their first-factor password to a possible attacker.

Setting and changing mutual authentication secrets

The standard approach to setting mutual authentication secrets is to separate this operation from the authentication process, instead using an administration application such as Entrust IdentityGuard Self-Service Module.

If you wish to set and change mutual authentication secrets using the Authentication API, you can use the `getGenericChallenge` call and the `authenticateGenericChallenge` call. In both cases, the set operation succeeds only if authentication succeeds.

In the case of `getGenericChallenge`, this means that RBA must be in force and the request for a challenge must result in an automatic AUTHENTICATED—the user does not have to respond to a challenge, since the RBA settings, which include machine authentication, require that the user is already authenticated.

When the user is not automatically authenticated, it is not recommended that you offer users the ability to set or change mutual authentication secrets before requesting a challenge, unless the user's choices were saved and automatically applied on the `authenticateGenericChallenge` call.

Mutual authentication secrets code samples

The following code samples show how to retrieve mutual authentication secrets and how to set and change them using the Authentication API.

To retrieve authentication secrets

Do one of the following:

- If you have set the **Return Mutual Authentication Secrets With Challenge** policy to **True**, and you want to display the mutual authentication secrets after getting the challenge:
 - Set `AuthenticationSecretParms` so that the authentication secrets are retrieved with the challenge. If you want to retrieve all secrets, use a line like this:

```
authSecParms.setGetAllSecrets( Boolean.TRUE );
```
 - If you want to retrieve the secrets by specifying their names, use a line like this:

```
authSecParms.setGetSecrets( auth_sec_name_array );
```
 - Add the `AuthenticationSecretParms` object to `GenericChallengeParms`, which is used in the `getGenericChallenge` operation:

```
genericChallengeParms.setAuthSecretParms( authSecParms );
```

- Call `getGenericChallenge` and retrieve the authentication secrets from the resulting `GenericChallenge` object:

```
genericChallenge.getAuthenticationSecrets();
```
- If you want to display the mutual authentication secrets after a successful authentication:
 - Set `AuthenticationSecretParms` so that the authentication secrets are retrieved with a successful authentication response. If you want to retrieve all secrets, use a line like this:

```
authSecParms.setGetAllSecrets(Boolean.TRUE);
```
 - If you want to retrieve the secrets by specifying their names, use a line like this:

```
authSecParms.setGetSecrets(auth_sec_name_array);
```
 - Add the `AuthenticationSecretParms` object to `GenericAuthenticateParms`, which is used in the `authenticateGenericChallenge` operation:

```
genericAuthenticateParms.setAuthSecretParms(authSecParms);
```
 - Call `authenticateGenericChallenge` and retrieve the authentication secrets from the resulting `GenericAuthenticateResponse`:

```
authResponse.getAuthenticationSecrets();
```

To retrieve authentication secrets using `getGenericChallenge`

The following sample code shows how to retrieve authentication secrets using the `getGenericChallenge` operation:

```
// The following code retrieves authentication secrets
// from a getGenericChallenge operation

// Set AuthenticationSecretsParms to get secrets
AuthenticationSecretParms authSecParms = new AuthenticationSecretParms();
authSecParms.setGetAllSecrets(Boolean.TRUE);
// Or specifically retrieve authentication secrets by name
// using something along the lines of:
// authSecParms.setGetSecrets(new String[] {"IMAGE_SECRET", "CAPTION_SECRET"});

// Add authSecParm to genericChallengeParms
GenericChallengeParms genChallparms = new GenericChallengeParms();
genChallparms.setAuthSecretParms(authSecParms);
// Get a challenge
GetGenericChallengeCallParms callParms = new GetGenericChallengeCallParms();
callParms.setUserId(userid);
callParms.setParms(genChallparms);
GenericChallenge genericChallenge = authBinding.getGenericChallenge(callParms);
// Separate the authentication secrets from the challenge
NameValue[] secrets = genericChallenge.getAuthenticationSecrets();
// Add code to display the authentication secrets to the user
```

To retrieve authentication secrets using authenticateGenericChallenge

The following sample code shows how to retrieve authentication secrets using the authenticateGenericChallenge operation:

```
// The following code retrieves the authentication secrets
// from the authenticate generic challenge operation

// Set AuthenticationSecretsParams to get secrets
AuthenticationSecretParams authSecretParams = new AuthenticationSecretParams();
authSecretParams.setGetAllSecrets(Boolean.TRUE);
// Or specifically retrieve authentication secrets by name
// using something along the lines of:
// authSecParams.setGetSecrets(new String[] { "IMAGE_SECRET", "CAPTION_SECRET" });

// Authenticate challenge response
AuthenticateGenericChallengeCallParams authCallParams = new
AuthenticateGenericChallengeCallParams();
authCallParams.setUserId(userid);
authCallParams.setResponse(response);
GenericAuthenticateParams genAuthParams = new GenericAuthenticateParams();
genAuthParams.setAuthSecretParams(authSecretParams);
authCallParams.setParams(genAuthParams);
GenericAuthenticateResponse authResponse =
authBinding.authenticateGenericChallenge(authCallParams);
// Separate the authentication secrets from the challenge
NameValue[] secrets = authResponse.getAuthenticationSecrets();
// Add code to display the authentication secrets to the user
```

To set mutual authentication secrets

To set authentication secrets using the Authentication API, you must do one of the following:

- If you want to set mutual authentication secrets as part of requesting a challenge (this works only if the request for a challenge results in the user being automatically authenticated):
 - Present the user with an interface that allows them to choose one or more authentication secrets. Since current mutual authentication secrets cannot be displayed without using the Administration API, you must ascertain whether existing mutual authentication secrets are to be replaced, or new authentication secrets are to be added.
 - Set AuthenticationSecretParams so that the mutual authentication secrets chosen by the user are saved if authentication is successful. This line requests that an authentication secret be added:

```
authSecParams.setSet(new NameValue[] { authSecret });
```
 - If you want to merge the new secret with existing secrets, set the **Merge** flag to **True**. A secret with the same name still replaces any existing secret with the same name:

```
authSecParams.setMerge(Boolean.TRUE);
```

- If you do not set the **Merge** flag discussed above, then the entire collection of mutual authentication secrets currently associated with the user is replaced with the new collection of secrets provided in this call.
- Add the `AuthenticationSecretParams` object to `GenericChallengeParams`, which is used in the `getGenericChallenge` operation.
- Call `getGenericChallenge`.
- If you want to set mutual authentication secrets as part of successfully responding to a challenge:
 - If you have the **Return Mutual Authentication Secrets With Challenge** policy value set to **True**, then when requesting a challenge, you must get the user's current collection of mutual authentication secrets. See "To retrieve authentication secrets" on page [63](#) for more information.

Alternatively, you can use the Administration API to fetch a user's mutual authentication secrets.

- You may or may not be able to show the existing set of mutual authentication secrets belonging to a user, but you should indicate to the user whether the newly chosen secrets will replace the existing ones or be merged with those that already exist:

```
authSecParams.setMergeSecrets(Boolean.TRUE);
```

- Set `AuthenticationSecretParams` so that the mutual authentication secrets chosen by the user are saved if authentication is successful. This line requests that an authentication secret be added:

```
authSecParams.setSetSecrets(new NameValue[] {authSecret});
```

- Add the `AuthenticationSecretParams` object to `AuthenticateGenericChallengeCallParams`, which is used in the `authenticateGenericChallenge` operation.
- Call `authenticateGenericChallenge`.

To set mutual authentication secrets using `authenticateGenericChallenge`

The following code sample sets mutual authentication secrets as part of an `authenticateGenericChallenge` request:

```
// The following code adds an authentication secret
// for a user when performing generic authentication.

NameValue[] authSecretsToAdd = new NameValue[] { authSecret };
AuthenticationSecretParams authSecretParams = new AuthenticationSecretParams();
authSecretParams.setMergeSecrets(Boolean.TRUE);
authSecretParams.setSetSecrets(authSecretsToAdd);
GenericAuthenticateParams genAuthParams = new GenericAuthenticateParams();
genAuthParams.setAuthSecretParams(authSecretParams);
AuthenticateGenericChallengeCallParams authenticateGenericChallengeCallParams = new
AuthenticateGenericChallengeCallParams(userid, response, genAuthParams);
// Code to authenticate the generic challenge
```

Image management

You can set up your application to allow users to upload their own images. As part of your application programming, you need to create the application code to upload and store the images.

Alternatively, you can allow users to choose from a preselected set of images to be used for mutual authentication. Images are stored as binary data, so the format does not matter, as long as the browser can display the image.

You can store user images in two ways:

- Use Administration API operations before the first user authentication process starts. You can purchase Entrust IdentityGuard Self-Service Module to perform this function.
- Use the Authentication API during the generic challenge request or generic challenge response process.

To store or update any authentication secrets, see “Mutual authentication secrets code samples” on page [63](#). All authentication secrets are stored as strings, so images must be converted.

To store and retrieve images

- 1 Read the image as bytes.
- 2 Convert bytes to string using an encoding method such as Base 64.
- 3 Send the string, along with the name to associate with it, to Entrust IdentityGuard as an authentication secret.

To retrieve and display an image

- 1 Get the authentication secret string from Entrust IdentityGuard.
- 2 Convert string to bytes by performing a Base 64 decode.

Note: To allow users to see the authentication secrets before they are authenticated, see “To retrieve authentication secrets” on page [63](#).

For more information about the mutual authentication policies, see the *Entrust IdentityGuard Server Administration Guide*.

Step-up authentication

If your organization has an e-commerce application for which the risk associated with a transaction increases with its monetary value or potential for fraud, you may want to implement step-up authentication (also called layered authentication).

The Web interface to a bank serves as a good example of where step-up authentication can heighten security. First-factor authentication alone may be an acceptable way to authenticate a user if the user is just checking account balances or transferring small amounts of money. For large fund transfers or for signing up for new services (like a brokerage account), the bank can add a additional authentication step (such as an out-of-band password or grid card) for extra security. Additionally, the bank can add serial number replay or image replay to ensure mutual authentication.

There are no specific APIs for step-up authentication. To implement step-up authentication, extend the logic of your application to call one or more of the authentication methods documented in this chapter. Ensure that you track the authentication methods already used in the transaction so that the step-up authentication method chosen is different from the ones already successfully answered by the user.

Machine authentication

Machine authentication provides seamless authentication without any noticeable impact to the user experience. It is an especially attractive method if users usually access their accounts from the same computer.

This approach is typically combined with one or more of these authentication approaches:

- grid
- token
- one-time password
- knowledge-based authentication

To establish the machine identity, you first generate a fingerprint of the user's computer. This fingerprint is based on a set of machine parameters chosen by your code that is transparently read from the user's computer. After it obtains this fingerprint, Entrust IdentityGuard generates a machine identity reference and stores it on the Entrust IdentityGuard server for future authentication. This machine registration process is similarly performed for all computers a user wishes to register.

In the following figure, the simple action of selecting the option **Remember me** activates machine authentication.

Login page with machine authentication

Machine Authentication API methods

To register a machine for machine authentication, call `authenticateGenericChallenge`, supplying the `machineSecret` property of the `GenericAuthenticateParms`.

To check fingerprints, call `getGenericChallenge`, supplying the `machineSecret` property of `GenericChallengeParms`.

The result of calling `getGenericChallenge` includes a `MachineSecretPolicy`. The `MachineSecretPolicy` structure includes details about various machine secret policies. Your application can use this information to determine what to put in a machine secret when registering a new one.

Rather than storing the full value for each piece of machine-specific application data gathered, you can choose to save space in your application by compressing the application data. You can configure the application to store the hash of an application value rather than the full value, for example.

Machine Authentication API code example

The code sample provided demonstrates how to check whether or not a machine is registered. If it is registered, the fingerprint is updated. If it is not registered, a challenge is issued. The workflow in machine authentication follows these steps:

Check for machine registration.

- If the machine is registered:
 - a Update the fingerprint.
 - b Display the authentication secrets (the image and caption replay).
 - c Prompt for the user name and password.
- If the machine is not registered:
 - a Register the machine.
 - b Prompt for the user name and password.

To integrate machine authentication with a client application

The following sample shows how to integrate machine authentication with a client application.

```
// To authenticate using a machine secret
MachineSecret machineSecret = buildMachineSecret();
GenericChallengeParms parms = new GenericChallengeParms();
parms.setMachineSecret(machineSecret);
GetGenericChallengeCallParms getGenericChallengeCallParms = new
GetGenericChallengeCallParms();
getGenericChallengeCallParms.setUserId(userid);
getGenericChallengeCallParms.setParms(parms);
GenericChallenge genericChallenge =
authBinding.getGenericChallenge(getGenericChallengeCallParms);
if
(genericChallenge.getChallengeRequestResult().equals(ChallengeRequestResult.AUTHENTICA
TED)) {
    // No second-factor authentication is required as
    // the machine authentication successfully passed
    // Get the updated machine secret (for sequence nonce)
    // and update the local fingerprint
}
```

```

        machineSecret = genericChallenge.getMachineSecret();
    } else if
    (genericChallenge.getChallengeRequestResult().equals(ChallengeRequestResult.CHALLENGE)
    ) {
        // require second-factor challenge
        // add code to display challenge and retrieve user response
    } else if
    (genericChallenge.getChallengeRequestResult().equals(ChallengeRequestResult.REJECT)) {
        // authentication has been rejected
    }
}

```

To register a machine secret

This sample shows how to register a machine secret:

```

// To register a machine secret
GenericAuthenticateParams genAuthParams = new GenericAuthenticateParams();
// The following two lines are mutually exclusive:
// Use this line if you want Entrust IdentityGuard to create a machine secret
// that includes no application data (the policy must allow machine secrets
// without application data)
genAuthParams.setRegisterMachineSecret(registerMachine);
// Use this line if you are passing in a machine secret that was
// created in your application (it may include application data)
// genAuthParams.setMachineSecret(machineSecret);
Response response = new Response();
response.setResponse(challengeResponse);
AuthenticateGenericChallengeCallParams authenticateGenericChallengeCallParams = new
AuthenticateGenericChallengeCallParams();
authenticateGenericChallengeCallParams.setUserId(userId);
authenticateGenericChallengeCallParams.setResponse(response);
authenticateGenericChallengeCallParams.setParams(genAuthParams);
GenericAuthenticateResponse resp =
authBinding.authenticateGenericChallenge(authenticateGenericChallengeCallParams);

// handle the response
// Get the updated machine secret and update the local fingerprint
machineSecret = resp.getMachineSecret();

```

Sources of machine information

There are several ways to create a fingerprint of a particular computer. The choice depends on the method chosen to gather fingerprint data.

Basic Web browser without client-side software

This requires only a Web browser. From the user's perspective, it is the least invasive method of gathering the information for a machine fingerprint.

Your program needs to set a cookie within the browser for subsequent authentication comparisons of the user's machine fingerprint. This may not always be possible; some users set their browsers so that cookies cannot be saved.

There are two ways of gathering data from a Web browser without requiring client-side software. You can use the browser `Get` request or JavaScript.

Through a Web browser `Get` request, the application can identify a browser using the HTTP headers present in the browser's request to the server. Unfortunately, all data returned is quite predictable, even to an attacker who has never seen a particular browser's request. The following figure shows a sample `Get` request.

Sample browser `Get` request

```
GET /cgi-bin/inputdump.exe HTTP/1.1
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322; .NET CLR 2.0.50727)
Host: anyserver.anybank.com
Connection: Keep-Alive
Cookie: intranetredirectURL=; GA_SHOW_TABS=; LASTSITE=intranet
```

Due to the predictability of standard `Get` requests from a browser, it is recommended that you do not use these fields on their own. Some fields (such as user-agent) may be useful as part of a broader machine fingerprint. Use other methods described in this section to create a unique machine fingerprint.

Instead of `Get` requests, your Web application can use standard JavaScript calls to gather information. This involves a minor modification to the application's login page to collect the wider range of data needed for the machine fingerprint. All the following pieces of information are available through standard JavaScript calls without requiring any client-side software.

Note: The properties in the following table were collected using JavaScript on an Internet Explorer browser running on Windows. Similar properties are available on other browsers, but the names and values may be different.

General properties

Property	Value
navigator.appCodeName	Mozilla
navigator.appName	Microsoft Internet Explorer
navigator.appMinorVersion	;SP2;

Property	Value
navigator.cpuClass	x86
navigator.platform	Win32
navigator.systemLanguage	en-us
navigator.userLanguage	en-us
navigator.appVersion	4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322; .NET CLR 2.0.50727)
navigator.userAgent	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322; .NET CLR 2.0.50727)
navigator.onLine	true
navigator.cookieEnabled	true
screen.availHeight	1170
screen.avaiWidth	1600
screen.bufferDepth	0
screen.colorDepth	32
screen.deviceXDPI	96
screen.deviceYDPI	96
screen.fontSmoothingEnabled	true
screen.height	1200
screen.logicalXDPI	96
screen.logicalYDPI	96
screen.updateInterval	0
screen.width	1600

Note: The properties in the following tables show just a portion of the MIME and plug-in information available. They were collected using JavaScript on a Firefox browser running on Microsoft Windows. Similar properties are available on other browsers, but the names and values may be different.

MIME properties (partial list)

Property	Value
----------	-------

navigator.mimeTypes[0].description	Mozilla Default Plug-in
navigator.mimeTypes[0].suffixes	*
navigator.mimeTypes[0].type	*
navigator.mimeTypes[1].description	Java
navigator.mimeTypes[1].enabledPlugin.filename	NPOJI610.dll

Plug-in information (partial list)

Property	Value
navigator.plugins[0].description	Default Plug-in
navigator.plugins[0].filename	npnui32.dll
navigator.plugins[0].length	1
navigator.plugins[0].name	Mozilla Default Plug-in
navigator.plugins[1].description	Java Plug-in 1.5.0 for Netscape Navigator (DLL Helper)

Given the wide range of information available, some of which may be too common to be useful, it is recommended that organizations consider the use of a combination of elements gathered through JavaScript such as:

- browser version
- browser plug-ins present
- browser language being used
- browser platform (user's operating system)
- screen size of user's computer (height and width)
- screen color depth

Basic Web browser with client-side software

Organizations can deploy signed Java applets or ActiveX controls that can go beyond basic browser security. This involves the user seeing and accepting security notifications on a regular basis. While more secure, it is less than ideal for large-scale deployments. However, there may be instances where this is the best practice since it allows organizations to gather more detailed physical machine data for use in a machine fingerprint.

Elements that could be gathered in this scenario include:

- media access control (MAC) address of the user's Ethernet card
- exact operating system (OS) information including the service pack and patch level

- system information including native byte order and number of available processors
- hardware information (manufacturer, model, version, and so on) of various hardware devices (network card, video card, hard drive, CD reader/writer, processor type)
- CPU processor ID (if enabled)
- user information (account name and home directory)

These elements can be combined with other available elements to create the machine fingerprint.

Web application (server-side)

The information available through JavaScript and client-side software can be augmented by data available from the Web application. The following figure shows information gathered by a simple server-side CGI.

Sample Web application data

```
HTTP_USER_AGENT=Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US;
rv:1.7.10) Gecko/20050716 Firefox/1.0.6
HTTP_ACCEPT=text/xml,application/xml,application/xhtml+xml,text/html;q=0.
9,text/plain;q=0.8,image/png,*/*;q=0.5
HTTP_ACCEPT_LANGUAGE=en-us,en;q=0.5
HTTP_ACCEPT_ENCODING=gzip,deflate
HTTP_ACCEPT_CHARSET=ISO-8859-1,utf-8;q=0.7,*;q=0.7
HTTP_KEEP_ALIVE=300
HTTP_CONNECTION=keep-alive
HTTP_COOKIE=LASTSITE=anybank;
intranetredirectURL=https%3A//anyserver.anybank.com/download/cnbc.htm;
GA_SHOW_TABS=0%2C1%2C2%2C4
REMOTE_ADDR=10.4.132.9
REMOTE_PORT=1294
```

This information is similar to that found in a `Get` request. This list includes a port and IP address. Port information may change each time and is not a useful property for a machine fingerprint. A user's IP address can be used to look up geolocation information, but should not be used directly.

Entrust IdentityGuard can store additional application data specified by your organization, including data that may be gathered with standard APIs through external data sources. (For example, geolocation services can estimate the geographic location of the user based on their PC's IP address.)

Storing and retrieving machine information

For machine authentication, organizations need to modify their application to first gather the information available as described in "Sources of machine information" on page [70](#). After the information is gathered, you can pass it to Entrust IdentityGuard through standard Web service APIs for storage.

The contents of a machine fingerprint in Entrust IdentityGuard include at least the machine nonce, and optionally a sequence nonce and application data.

Machine nonce

This is an arbitrary number generated by Entrust IdentityGuard for authentication purposes when Entrust IdentityGuard registers the machine. This nonce must be stored on the client machine by the application, typically in a cookie or an Adobe Flash shared object. This nonce value does not change.

The machine nonce is not required if application data is required.

Note: Flash shared objects are a feature in Adobe Flash that allow applications to store information similar to cookies on a machine and retrieve it at a later time. It stores information without the need to enable cookies. For Entrust IdentityGuard, Flash shared objects can store both the machine nonce and the sequence nonce.

Entrust IdentityGuard normally creates the machine nonce, but it is possible for an application to create the machine nonce from something that does not need to be stored in a persistent object like a cookie or Flash object (a MAC address that uniquely identifies a machine, for example). The application provides the machine nonce to Entrust IdentityGuard when registering the machine secret. This allows the application to use a machine nonce without requiring that users enable cookies or Flash in their browsers.

Optional sequence nonce

Entrust IdentityGuard generates and changes the sequence nonce each time authentication occurs. A sequence nonce assures that the machine secret is only valid until the next login attempt. This increases security by reducing the validity period of the machine information, and making it more difficult for an attacker to use the cookies without being detected.

The sequence nonce must be stored on the client machine by the application, typically in a cookie or a Flash shared object. The inclusion of a sequence nonce is recommended for strengthening machine authentication.

Optional application data

When Entrust IdentityGuard first creates the machine secret, the client application specifies and sets a list of name and value pairs. A client application can provide Entrust IdentityGuard with application data specific to the user's computer. This can include operating system and browser versions gathered through simple methods that do not require client-side software, as described in "Sources of machine information" on page [70](#).

During authentication, the application must retrieve and pass the contents of the fingerprint to Entrust IdentityGuard for comparison and validation.

Organizations can decide how many properties must be successfully matched in the fingerprint for successful authentication (for example, five of six must be correct). Failure may depend on the property in question. If one of the properties captured is the browser version and in subsequent authentications that version changes (perhaps the user upgraded the browser), it may still make sense to allow that user access. To maximize security and overall usability, it is recommended that organizations examine their user base carefully before configuring this option.

Machine authentication Web sample

The following code samples contain the components required to implement a fully functional Entrust IdentityGuard Authentication application using machine authentication. This sample consists of two levels of processing: client-side, through HTML and JavaScript; and server-side, through JSP and Java servlet technology.

The samples demonstrate the approach taken for machine authentication with respect to cookie handling and persistent storage of machine and sequence nonces.

- 1 Get the machine fingerprint (based on client browser details).
- 2 Determine if the browser supports cookies. If so, the browser is responsible for transmitting cookie information. If not, determine if the browser supports Adobe® Flash® Player 9 (or greater). If so, the client API is used to gather nonce values.
- 3 Post this data to the Web server to attempt the machine authentication.
- 4 Return the result back to the client, and if Flash cookies are used, write the machine and optional sequence nonce information back to the Flash cookie.

Note: All JavaScript and Flash files can be found in the file `IdentityGuardMachineAuthenticationWebSample.zip` under `/scripts` stored within your Entrust IdentityGuard installation under `/IdentityGuard/identityguard120/client/sample`.

The relevant files are:

- `/scripts/FlashInterface.js` (client-side Flash cookie interface)
- `/scripts/MachineInfo.js` (client-side Machine Fingerprinting interface)
- `/scripts/LocalStore.swf` (Flash Object to read/write Flash cookies. Requires Adobe Flash Player 9 or greater)
- `/java/*` (All the components required to establish a Machine Authentication J2EE Web Application)

Client-side processing for machine authentication

The client-side processing is responsible for gathering client-side information such as a machine fingerprint based on the client browser properties, and whether Flash cookies are to be used as a method for persistent storage of the machine and sequence nonces (see “Machine nonce” and “Optional sequence nonce”) as opposed to browser cookies. In the case of Flash cookies, client-side processing is also responsible for writing machine and sequence nonce values to the persistent store.

The following code samples use the API exposed through `MachineInfo.js` and `FlashInterface.js`. See the `MachineInfo.js` and `FlashInterface.js` files for implementation details.

- 1 Initialize the Machine Authentication client-side. This requires JavaScript to be enabled in the user's Web browser.

```
// Call to machineInfo.js
// Create a MachineSecret object providing a boolean
// parameter set to true which indicates this MachineSecret
// is intended for reading only (i.e. Machine Fingerprint
// and Flash Cookies are read)
var secret = new MachineSecret(true);
```

- 2 Get a machine fingerprint using browser details. This requires JavaScript to be enabled.

```
// Call to machineInfo.js
// Get the machine fingerprint from the MachineSecret object
var machineFingerprint = secret.getMachineFingerprint();
```

- 3 Determine whether Flash is to be used as an alternative for browser persistent storage as opposed to cookies. This flag is only ever true if the browser has cookie support disabled and Adobe Flash Player 9 or greater is supported. This requires JavaScript to be enabled.

```
// Call to machineInfo.js
var supportsFlash = secret.getUseFlash();
```

- 4 Get the Entrust IdentityGuard Machine Nonce values if, and only if, Flash cookies are used. This is because when cookies are enabled, browsers, by default, include cookie information within HTTP requests, where Flash requires an independent method of data transmission. This requires JavaScript to be enabled.

```
// Call to machineInfo.js
if (supportsFlash) {
    // Get the machine label, machine nonce and
    // sequence nonce stored within the Flash cookie
    machineLabel = secret.getMachineLabel();
    machineNonce = secret.getMachineNonce();
    sequenceNonce = secret.getSequenceNonce();
}
```

- 5 Include all collected machine information within the HTTP request during the `getChallenge` or authentication call. This can be done by using hidden form fields within HTML.

See `/java/MachineAuthStart.html` for implementation details.

Server-side processing for machine authentication

Server-side processing is responsible for handling the authentication request submitted by the client. This includes processing the form data, which initiates gathering the machine fingerprint and the machine and sequence nonces (if Flash is used). After the form data is processed, a call is made through the Entrust IdentityGuard Authentication Web Service to determine the authentication result.

- 1 Process the form data submitted by the client and re-encapsulate the data to be re-submitted to the business logic responsible for the machine authentication. This is done through a JSP bean object used to encapsulate the form data, and a servlet to perform the authentication.

Refer to `/java/jsp/Regiser_Authentication.jsp` and `/java/src/com/entrust/identityguard/example/MachineDataBean.java` for implementation details with respect to bean processing and posting to the Authentication Servlet.
- 2 The business logic can now be called to perform the appropriate authentication call. See `/java/src/com/entrust/identityguard/example/GetChallenge.java` for implementation details for performing a challenge request (machine authentication) using a J2EE Servlet Context.
- 3 When attempting to return a challenge, if machine authentication is successful and persistent storage is supported, the machine and sequence nonces need to be written to the client. No client processing is required if cookies are being used; however, in the case of Flash, client processing is required to write the provided nonce values to the appropriate Flash object.

See `/java/jsp/ApplyResults.jsp` for implementation details for creating an HTML form object based on the machine and sequence nonce set within the `MachineDataBean` from Step 2 above, and writing it to a Flash cookie.

Note: Machine authentication requires the use of nonce values if the policies, `machineSecretReqMachineNonce` and `machineSecretReqSequenceNonce` are set to `True`.

Risk-based authentication (RBA)

There may be situations where you want to present users with an additional authentication challenge, or reject users outright for security reasons. On the other hand, you may also want to automatically authenticate users who pose no risk.

With Entrust IdentityGuard, you can configure your application to provide normal or enhanced risk assessment levels based on the sensitivity of the information the user wants to access, or the potential for fraud a transaction poses. The risk posed by any specific user is determined, in large part, using IP/Geolocation data. See also “Step-up authentication” on page 67.

To use all the features of RBA, you can use IP/Geolocation data, machine authentication, and certificate authentication. You can, however, configure RBA to use only machine authentication or only IP/Geolocation authentication, or only certificate authentication.

Certificate RBA is based on a user certificate. The `getGenericChallenge` method can include a certificate parameter set with the `setCertificate` method of the `GenericChallengeParms` class. The certificate parameter must be a Base-64 encoded X.509 certificate that is associated with the user.

When doing RBA analysis, if a certificate is included in the list of values provided by the client, it is processed as follows:

- 1 The certificate is validated. Validation is a multi-step process, which includes the following checks:
 - The provided value must be a certificate.
 - The certificate must be within its validity period.
 - According to policy, the certificate may be required to be issued by a registered CA certificate. If this policy setting is **True**, and the certificate was not issued by a registered CA certificate, then certificate validation fails.
 - If a registered CA certificate is found, it must be in the active state and it must be allowed for the user as defined by policy.
 - According to policy, self-signed certificates may not be allowed. If this is true and the certificate was self-signed, then validation fails.
 - If the certificate was issued by a registered CA certificate or is self-signed, then the certificate signatures must be valid up to the root.
 - If the certificate was issued by a registered CA, then all of the CA certificates up to the root CA must be within their validity period.
 - User certificates must have the `digitalSignature` or `nonRepudiation` key usage set.
 - If the registered CA certificate defines LDAP or OCSP parameters, then the user certificate and CAs must not be revoked according to that revocation source.
- 2 Entrust IdentityGuard checks to see if the certificate is registered against the user.

Based on the results of these tests, the RBA policy determines if the request is rejected, challenged, or authenticated. The certificate validation and certificate registered checks are

independent of each other. It is possible for a registered certificate to be invalid, indicating that the certificate had been revoked or expired, or that its CA certificate had been inactivated or deleted. It is also possible for a valid certificate not to be registered.

The RBA results returned from a `getGenericChallenge` call (the `RiskScoringResult` object), indicate whether certificate authentication passed or not, and if not, the reason for the failure (the certificate was invalid, or it was not registered, or both).

Entrust IdentityGuard includes a default configuration for risk-based authentication at both the normal and enhanced security levels.

To perform RBA for a user with known IP address

This sample code shows how to perform risk-based authentication for a specific user, given the user's IP address.

Note: This sample code displays `ipAddress` as "x.x.x.x", but your code must provide the IP address associated with the connecting user. You can do this programmatically:

For a Web application in general, you can use the value associated with the HTTP header named `REMOTE_ADDR`.

The Java servlet API makes this value available through the method `getRemoteAddr()` associated with an `HttpServletRequest`.

```
// Create the generic challenge parameters:
GenericChallengeParms genericChallengeParms = new GenericChallengeParms();
// Replace the following with code that determines the IP address
// associated with the connecting user.
String ipAddress = "x.x.x.x";
// Set the ip address of the requesting client.
// This value is used to perform risk-based analysis based
// on the geolocation associated with the IP address.
// Machine secrets can also be included for risk based analysis.
genericChallengeParms.setIPAddress(ipAddress);
// set the security level used for risk based authentication
// SecurityLevel.NORMAL is the default.
genericChallengeParms.setSecurityLevel(SecurityLevel.ENHANCED);
GetGenericChallengeCallParms callParms = new GetGenericChallengeCallParms();
callParms.setUserId(userid);
callParms.setParms(genericChallengeParms);
GenericChallenge genericChallenge = authBinding.getGenericChallenge(callParms);
ChallengeRequestResult challengeResult = genericChallenge.getChallengeRequestResult();
if (challengeResult == ChallengeRequestResult.AUTHENTICATED) {
    // no second-factor authentication is required
} else if (challengeResult == ChallengeRequestResult.REJECT) {
    // deny access to the user
} else if (challengeResult == ChallengeRequestResult.CHALLENGE) {
    // Authenticate the user using the current challenge.
```

```
// Refer to the Generic API code sample for challenge authentication
}
```

Challenge history

The challenge history is used for skipping risk-based authentication during `getGenericChallenge` calls. The generic challenge history is a list of authentication types for which the user has already answered challenges.

The challenge history is used as follows:

- Call `getGenericChallenge` with the IP address, the machine secret, or both, and a `challengehistory` value. (The challenge history value is optional.)
 - If the authentication type that would be returned is in the `challengehistory` value, Entrust IdentityGuard returns a `ChallengeRequestResult` object for which the value is set to `AUTHENTICATED`.
 - Otherwise, perform risk-based authentication as normal, which results in a `ChallengeRequestResult` object for which the value is set to `AUTHENTICATED`, `CHALLENGE`, or `REJECT`.

To set the challenge history

To set the challenge history, use code like the following:

```
AuthenticationType[] challHistory =
    new AuthenticationType[] { AuthenticationType.GRID };
genericChallengeParams.setChallengeHistory(challHistory);
```

Similarly, during the `authenticateGenericChallenge` call, the application can also pass the generic challenge history information to the Entrust IdentityGuard server. This information is not used for authentication, but if the authentication succeeds, the authentication type being used is added to the input challenge history and is returned to the application.

To set the generic challenge history during authentication, use code like the following:

```
genericAuthenticateParams.setChallengeHistory(challHistory);
```

To retrieve the challenge history information from the `authenticate` response use code like the following:

```
AuthenticationType[] newChallHistory =
    genericAuthenticateResponse.getChallengeHistory();
```

Remember, it is the responsibility of the application to store the challenge history. Entrust IdentityGuard server does not store the challenge history information.

Transaction authentication

Transaction authentication is available for use with Entrust IdentityGuard Mobile soft tokens. When enabled, users receive notifications on their mobile devices asking them to confirm pending transactions they have started at your Web site—a money transfer, for example. By having a confirmation notice sent to a secondary device, man-in-the-browser attacks are mitigated. If users choose to confirm the transaction, a confirmation code is presented to them on their mobile

device. Users then enter the code onto your Web site to authorize the transaction and allow it to proceed.

Note: Transaction authentication is supported on iPhone, iPod touch and Android devices.

A record of each transaction is kept on the mobile device, so users can review them as needed.

The following code sample illustrates delivering and authenticating a transaction.

```
// This code assumes that the user's token has been enrolled for transaction
// delivery through the Entrust IdentityGuard Self-Service Module Transaction
// Component.

BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

// Define transaction details
NameValue[] transactionDetails = new NameValue[2];
transactionDetails[0] = new NameValue();
transactionDetails[0].setName("Detail 1");
transactionDetails[0].setValue("Value 1");
transactionDetails[1] = new NameValue();
transactionDetails[1].setName("Detail 2");
transactionDetails[1].setValue("Value 2");

// Prompt for the userid
System.out.print("Enter UserId: ");
String userid = in.readLine();

// Request a generic challenge, passing in the transaction
// details to be signed by the token.

GenericChallengeParms genericChallengeParms = new GenericChallengeParms();
GetGenericChallengeCallParms callParms = new GetGenericChallengeCallParms();

// Transaction delivery uses the authentication type TOKENRO
genericChallengeParms.setAuthenticationType(AuthenticationType.TOKENRO);
// Include the transaction details to be signed.
genericChallengeParms.setTransactionDetails(transactionDetails);
callParms.setParms(genericChallengeParms);
callParms.setUserId(userid);

// call getGenericChallenge, the transaction details will attempt to be
// delivered
GenericChallenge genericChallenge = null;
```

```

try {
    genericChallenge = authBinding.getGenericChallenge(callParms);
} catch (AuthenticationFault fault) {
    // handle IdentityGuard authentication error
    System.err.println("Error requesting authentication challenge: " +
        fault.getErrorMessage());
    return;
} catch (Exception ex) {
    // handle non-IdentityGuard error
    ex.printStackTrace();
    return;
}

// Ensure that the transaction was delivered to at least one token
boolean delivered = false;
if (genericChallenge.getTokenChallenge() != null &&
    genericChallenge.getTokenChallenge().getTokens() != null) {

    TokenSetParser parser = new
TokenSetParser(genericChallenge.getTokenChallenge().getTokens());
    Iterator tokenSetIt = parser.getTokenSets().iterator();
    while (tokenSetIt.hasNext()) {
        String set = (String) tokenSetIt.next();
        Iterator tokenIt = parser.getTokensForSet(set).iterator();
        while (tokenIt.hasNext()) {
            TokenData tok = (TokenData) tokenIt.next();
            if (genericChallenge.getType().equals(AuthenticationType.TOKENRO) &&
                tok.getDeliveryStatus().equals(DeliveryStatus.OK)) {
                // delivery successful.
                delivered = true;
            }
        }
    }
}

if (!delivered) {
    // Handle the scenario where a transaction could not be delivered.
    // A normal token authentication could be done instead.
    System.out.println("The transaction could not be delivered to any tokens.");
    return;
}

System.out.print("Please enter the confirmation code generated by your token:");
String[] userResponse = { in.readLine() };

```

```

// Authenticate the response
try {
    Response response = new Response();
    response.setResponse(userResponse);

    GenericAuthenticateParms authParms = new GenericAuthenticateParms();
    // The same transaction details must be included in
    // authenticateGenericChallenge
    authParms.setTransactionDetails(transactionDetails);
    authParms.setAuthenticationType(AuthenticationType.TOKENRO);

    AuthenticateGenericChallengeCallParms authCallParms = new
AuthenticateGenericChallengeCallParms();
    authCallParms.setUserId(userid);
    authCallParms.setParms(authParms);
    authCallParms.setResponse(response);

    GenericAuthenticateResponse authResponse =
authBinding.authenticateGenericChallenge(authCallParms);

    String name = authResponse.getFullName();
    if (name == null)
        name = userid;

    System.out.println("The user " + name + " has successfully confirmed the
transaction details!");
} catch (AuthenticationFault fault) {
    // handle IdentityGuard authentication error

    System.err.println("Error authenticating the reponse: " +
fault.getErrorMessage());
    return;
} catch (Exception ex) {
    // handle non-IdentityGuard error
    ex.printStackTrace();
    return;
}

```

Chapter 4:

Administration tasks

This chapter explains the various Entrust IdentityGuard administration tasks you can manage using the Entrust IdentityGuard Administration APIs.

Administration setup and login

For a client application to use the Administration API, the client must have administrator credentials. Its administrator role must include all of the privileges required to complete the tasks assigned.

For instructions on setting up an administrator, see the *Entrust IdentityGuard Server Administration Guide*.

Note: The login call is not required when using the Admin API Failover Binding. The failover API automatically handles calling the login command when required.

The following sample code generates an Administration service binding object that connects to the Administration services URL:

```
// The following sample code generates an administration service
// binding object that connects to the Administration services
// URL, and logs in the provided admin ID

URL adminServiceUrl = new URL(ADMIN_SERVICE_URL);

// Create a new binding using the URL just created:
try {
    AdminService_ServiceLocator locator = new AdminService_ServiceLocator();
    adminBinding = (AdminServiceBindingStub) locator.getAdminService(adminServiceUrl);
    adminBinding.setMaintainSession(true);
} catch (ServiceException se) {
    // Handle exception
    if (se.getLinkedCause() != null) {
        se.getLinkedCause().printStackTrace();
    }
    throw se;
}

// login an administrative id
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
LoginParms loginParms = new LoginParms();
LoginCallParms callParms = new LoginCallParms();
```

```

callParms.setParms(loginParms);
LoginResult loginResult = null;
try {
    char[] password;

    System.out.print("Admin Id: ");
    loginParms.setAdminId(in.readLine());
    password = getPassword("Password: ");
    loginParms.setPassword(new String(password));
    Arrays.fill(password, ' ');
    loginResult = adminBinding.login(callParms);
} catch (AdminPasswordChangeRequiredFault apcrf) {
    // Optionally, add code to display the password rules
    // PasswordRulesInfo pwdinfo = apcrf.getPasswordRules();
    char[] newpwd, confirmpwd;
    do {
        newpwd = UtilityMethods.getPassword("Please enter your new password: ");
        confirmpwd = UtilityMethods.getPassword("Please confirm your new password: ");
    } while (!Arrays.equals(newpwd, confirmpwd));
    ChangePasswordCallParms changePasswordCallParms = new ChangePasswordCallParms();
    changePasswordCallParms.setNewpassword(new String(newpwd));
    changePasswordCallParms.setParms(loginParms);
    Arrays.fill(newpwd, ' ');
    Arrays.fill(confirmpwd, ' ');
    try {
        loginResult = adminBinding.changePassword(changePasswordCallParms);
        if (loginResult.getState().equals(LoginState.NEED_SECOND_FACTOR)) {
            // Need to answer 2nd-factor authentication
            loginParms.setPassword(null);
            loginParms.setResponse(getSecondFactorChallengeResponse(in,
loginResult.getLoginChallenge()));
            loginResult = adminBinding.changePassword(changePasswordCallParms);
            if (!loginResult.getState().equals(LoginState.COMPLETE)) {
                // Cannot complete 2nd-factor authentication
                throw new Exception("Cannot complete 2nd-factor authentication.");
            }
        }
    } catch (AdminServiceFault fault) {
        // Handle password change failed
        System.err.println("Password Change Failed: " + fault.getErrorMessage());
        return null;
    } catch (Exception e) {

```

```

        System.err.println("Password Change Failed: " + e.getMessage());
        return null;
    }
} catch (AdminServiceFault fault) {
    // Handle login failed
    System.err.println("Login Failed: " + fault.getErrorMessage());
    return null;
} catch (RemoteException re) {
    // Handle other types of exceptions
    System.err.println("Login Failed: " + re.getMessage());
    return null;
}

// We could get here and still require 2nd-factor authentication
if (loginResult.getState().equals(LoginState.NEED_SECOND_FACTOR)) {
    loginParams.setPassword(null);
    loginParams.setResponse(getSecondFactorChallengeResponse(in,
loginResult.getLoginChallenge()));
    try {
        loginResult = adminBinding.login(callParams);
        if (!loginResult.getState().equals(LoginState.COMPLETE)) {
            // Cannot complete 2nd-factor authentication
            throw new Exception("Cannot complete 2nd-factor authentication.");
        }
    } catch (AdminServiceFault fault) {
        System.err.println("Second-factor Authentication Failed: " +
fault.getErrorMessage());
        return null;
    } catch (Exception re) {
        System.err.println("Second-factor Authentication Failed: " + re.getMessage());
        return null;
    }
}

System.out.println("Successfully authenticated to admin service with user " +
loginResult.getUserName() + " in group " + loginResult.getGroupName());
return adminBinding;

```

Basic administration tasks

This section explains how to use the Administration API to handle day-to-day administration tasks.

Create and register a user

You must add new users to the Entrust IdentityGuard repository when they register through your application.

This example shows one approach:

```
UserParms userParms = new UserParms();
// set attributes
userParms.setGroup(groupid);
userParms.setAliases(aliases);
userParms.setContactInfoList(contactInfoList);
userParms.setAuthenticationSecrets(authenticationSecrets);
userParms.setQaSecrets(qaSecrets);
userParms.setSharedSecrets(sharedSecrets);

UserCreateCallParms callParms = new UserCreateCallParms();
callParms.setUserid(userid);
callParms.setParms(userParms);
adminBinding.userCreate(callParms);
```

Note: If your application stores users in a directory repository, make sure a user entry exists in the repository before you create the user in Entrust IdentityGuard.

Rename a user

You can rename users in the Entrust IdentityGuard repository through your application. This example shows one approach.

```
UserParms userParms = new UserParms();
// set attributes
userParms.setUserid(olduserid);
userParms.setGroup(groupid);
userParms.setAliases(aliases);
userParms.setContactInfoList(contactInfoList);
userParms.setAuthenticationSecrets(authenticationSecrets);
userParms.setQaSecrets(qaSecrets);
userParms.setSharedSecrets(sharedSecrets);

UserSetCallParms callParms = new UserSetCallParms();
```

```
callParms.setUserid(userid);
callParms.setParms(userParms);
adminBinding.setUserSet(callParms);
```

Get grid contents for a user

Use this code in your administration application to display a user's grid.

To fetch the contents of a user grid, the administrator must have the `userCardView` permission.

```
UserCardGetCallParms callParms = new UserCardGetCallParms();
callParms.setUserid(userid);

// Set the flag to get grid content
CardGetParms cardGetParms = new CardGetParms();
cardGetParms.setGetGrid(Boolean.TRUE);
callParms.setParms(cardGetParms);

UserCardFilter userCardFilter = new UserCardFilter();
// Look for a user car with a specific serial number
userCardFilter.setSerialNumber(serialNumber);
callParms.setFilter(userCardFilter);

UserCardInfo[] cardInfoArray = adminBinding.userCardGet(callParms);

// if a user card is returned
UserCardInfo cardInfo = cardInfoArray[0];
Grid grid = cardInfo.getGrid();
```

Create and activate a user's grid card

There are two ways to create grid cards for your end users. You can either create a set of unassigned grid cards and assign them to users, as described in "Create and assign preproduced grid cards" on page [91](#), or you can create and assign a single grid card in one step using the `userCardCreate` method.

You can also set an expiry date on a user's current method of authentication to force the user to activate the grid card by a certain date.

The following example creates a single grid card for a user in `HOLD_PENDING` state and activates it to the Current state.

Create a user grid card

```
// create user card
UserCardParms userCardParms = new UserCardParms();
```

```

// instead of creating a new card, set serial number
// if you want to assign a preproduced card
// userCardParms.setSerialNumber(serialNumber);
userCardParms.setComment(comment);
// set card state to HOLD_PENDING if you want to force the user to
// go through an activation process to activate their new card.
userCardParms.setState(State.HOLD_PENDING);
UserCardCreateCallParms callParms = new UserCardCreateCallParms();
callParms.setUserid(userid);
callParms.setParms(userCardParms);
adminBinding.userCardCreate(callParms);

```

Activate a user grid card

```

// the following code activates a user card
// if the card state is HOLD_PENDING, change it to PENDING first
UserCardFilter userCardFilter = new UserCardFilter();
userCardFilter.setSerialNumber(cardSerialNumber);

UserCardParms userCardParms = new UserCardParms();
userCardParms.setState(State.PENDING);
UserCardSetCallParms userCardSetCallParms = new UserCardSetCallParms();
userCardSetCallParms.setUserid(userid);
userCardSetCallParms.setFilter(userCardFilter);
userCardSetCallParms.setParms(userCardParms);
int count = adminBinding.userCardSet(userCardSetCallParms);
assert count == 1;
try {
    // authenticate the user using authentication service
    // see grid authentication; auth service API example
    // Authentication is required to verify that the user who has activated the
    // grid has possession of the grid.
}
//
catch (Exception e) {
    // if authentication failed, change the card state back
    // to HOLD_PENDING
    userCardParms.setState(State.HOLD_PENDING);
    adminBinding.userCardSet(userCardSetCallParms);
}

```

Advanced grid card properties

If you need advanced functionality, there are additional grid card properties you can set while creating grid cards.

The lifetime of an activated grid card is based on the policy, and the policy defaults to unlimited. To set the lifetime of the grid card to another value, add the following line:

```
userCardParms.setLifetime(lifetime);
```

where `lifetime` is the active life of the card, in days.

If a new grid card is assigned to a user, it supersedes the existing card. When the new grid card is used, the superseded card is cancelled. The superseded card remains active until the new grid card is used, since it may take some time for the new card to arrive in the hands of the user. The default lifetime policy value for a superseded grid card is unlimited. To set the superseded card lifetime to a shorter time, add the following line:

```
userCardParms.setSupersede(supersede);
```

where `supersede` is the number of days the superseded card remains active while waiting for the new grid card to be used.

Note: You can never unassign a grid card created specifically for a user. You can only unassign reproduced grid cards.

Create and assign preproduced grid cards

You can create a specific number of unassigned grid cards in one operation using the `cardCreate` method. Grid cards are assigned to users later using the `userCardCreate` method, as shown:

Create preproduced grid cards

```
// The following code creates preproduced cards for a group.
PreproducedCardParms preproducedCardParms = new PreproducedCardParms();
preproducedCardParms.setNumCards(numCards);
preproducedCardParms.setGroup(group);
PreproducedCardCreateCallParms preproducedCardCreateCallParms = new
PreproducedCardCreateCallParms();
preproducedCardCreateCallParms.setParms(preproducedCardParms);
PreproducedCardCreateResult result =
adminBinding.preproducedCardCreate(preproducedCardCreateCallParms);
```

Assign preproduced grid card

```
//The following code assigns a preproduced card to a user.
UserCardParms userCardParms = new UserCardParms();
userCardParms.setSerialNumber(serialNumber);
userCardParms.setLifetime(lifetime);
// set the card state to HOLD_PENDING
userCardParms.setState(State.HOLD_PENDING);
UserCardCreateCallParms userCardCreateCallParms = new UserCardCreateCallParms();
userCardCreateCallParms.setUserid(userid);
userCardCreateCallParms.setParms(userCardParms);
adminBinding.userCardCreate(userCardCreateCallParms);
```

Create and send an OTP

In some multifactor authentication scenarios, your application may require an OTP to authenticate a user for a single transaction, or to tie a login to the application. Optionally, you create and update one or more OTPs using the Administration API. You should distribute the OTPs to your end user through an out-of-band method, such as by text message, voice mail, or email.

Note: This step is optional as `getGenericChallenge` in the Authentication API automatically creates an OTP if required. Using the method shown in this example guarantees the return of a valid OTP in the case where `getGenericChallenge` is not used.

The following example creates and retrieves one or more OTPs. Use this example only if you are not using `getGenericChallenge`.

Create an OTP

Depending on the policy settings, this code may result in more than one OTP being created.

```
// The following code generates the OTP for a given user ID
// The OTP is not delivered to end user
UserOTPParms userOTPParms = new UserOTPParms();
// The lifetime of the OTP in milliseconds
// If set to 0, the OTP remains valid until it is used.
userOTPParms.setLifetime(lifetime);
UserOTPCreateCallParms userOTPCreateCallParms = new UserOTPCreateCallParms();
userOTPCreateCallParms.setUserid(userid);
userOTPCreateCallParms.setParms(userOTPParms);
UserOTPInfo[] otp = adminBinding.userOTPCreate(userOTPCreateCallParms);
```

Retrieve an OTP

```
// The following code retrieves an existing OTP for a given user ID.
// Note that if the user has more than one OTP, the oldest one is
// retrieved. It is also possible that the user doesn't have any
// OTPs.
UserOTPGetCallParms userOTPGetCallParms = new UserOTPGetCallParms();
userOTPGetCallParms.setUserid("userid");
userOTPGetCallParms.setFilter(new UserOTPFilter());
UserOTPInfo[] otp = adminBinding.userOTPGet(userOTPGetCallParms);
// Make sure the user has an OTP
if (otp != null && otp.length == 1) {
    // Only able to get the OTP value if
    // admin user has the userOtpView permission
    String otpString = otp[0].getOTP();
    if (otpString != null) {
        // Deliver the OTP to the user using an
        // application-specific mechanism
    }
}
```

Send an OTP using Entrust IdentityGuard delivery methods

```
// The following code retrieves the users contact information for
// OOB Delivery of OTP
UserContactInfoGetCallParms contactInfoParms = new UserContactInfoGetCallParms();
contactInfoParms.setUserid("userid");
```

```

ContactInfo[] infos = adminBinding.userContactInfoGet(contactInfoParms);
String contactInfoLabel = null;
for (int i = 0; i < infos.length; i++) {
    // Choose the first contact info with a valid delivery
    // mechanism assigned. Better approach would be to
    // give the user a choice as to where they want
    // their OTP delivered, possibly multiple places.
    String label = infos[i].getDeliveryConfigLabel();
    if (label != null && !label.equalsIgnoreCase("none")) {
        contactInfoLabel = label;
        break;
    }
}
if (contactInfoLabel == null) {
    // Return some kind of error detailing the fact that no
    // contact information has been configured for the user, or
    // none has been associated with a delivery mechanism, so
    // therefore the OTP cannot be delivered.
} else {
    // Deliver the OTP
    UserOTPParms userOTPParms = new UserOTPParms();
    userOTPParms.setContactInfoLabel(new String[] { contactInfoLabel });

    UserOTPDeliverCallParms otpDeliverParms = new UserOTPDeliverCallParms();
    otpDeliverParms.setUserid("user");
    otpDeliverParms.setParms(userOTPParms);
    adminBinding.userOTPDeliver(otpDeliverParms);
}

```

Retrieve delivery configuration of an OTP

You can retrieve the available delivery configurations for specified users. The delivery configurations are initiated when you start the Entrust IdentityGuard server. Your application can automate the delivery of an OTP through email, a text message to a cell phone, or voice message to a standard phone or cell phone.

The following sample code shows how to retrieve the delivery configuration list:

```
// This sample retrieves the entire delivery config list
// and chooses the label for the first entry
DeliveryConfigInfoList configList = adminBinding.deliveryConfigList(new NameValue[0]);
// Retrieve array of delivery configuration info
DeliveryConfigInfo[] configs = configList.getDeliveryConfigInfo();
String deliveryConfigLabel = null;
if (configs != null && configs.length > 0) {
    // Take the first one
    deliveryConfigLabel = configs[0].getConfigLabel();
}
```

Create and modify user contact information

You can create and modify a user's contact information, (phone number and email address, for example). If a valid delivery configuration is assigned to a contact information entry, it can be used for automatic OTP delivery.

Create contact information

```
// The following code creates a contact info entry with
// a specified label, value, and config label.
UserContactInfoParms userContactInfoParms = new UserContactInfoParms();
userContactInfoParms.setContactInfoLabel(contactinfoLabel);
userContactInfoParms.setValue(contactinfoValue);
userContactInfoParms.setDeliveryConfigLabel(deliveryConfigLabel);

// Optionally, you can set this contactinfo as
// the default one for out-of-band delivery.
userContactInfoParms.setDefaultContactInfo(Boolean.TRUE);

UserContactInfoCreateCallParms userContactInfoCreateCallParms = new
UserContactInfoCreateCallParms();
userContactInfoCreateCallParms.setUserid(userid);
userContactInfoCreateCallParms.setParms(userContactInfoParms);
adminBinding.userContactInfoCreate(userContactInfoCreateCallParms);
```

Modify contact information

```
// The following code modifies a value for the contactinfo,
// and sets the default contactinfo for the user
UserContactInfoParms userContactInfoParms = new UserContactInfoParms();
userContactInfoParms.setValue(contactinfoValue);
// set the contactinfo as default
userContactInfoParms.setDefaultContactInfo(Boolean.TRUE);

UserContactInfoSetCallParms userContactInfoSetCallParms = new
UserContactInfoSetCallParms();
userContactInfoSetCallParms.setUserid(userid);

userContactInfoSetCallParms.setLabel(contactinfoLabel);
userContactInfoSetCallParms.setParms(userContactInfoParms);
adminBinding.userContactInfoSet(userContactInfoSetCallParms);
```

Delete contact information

```
// The following code deletes user contactinfo
UserContactInfoDeleteCallParms userContactInfoDeleteCallParms = new
UserContactInfoDeleteCallParms();
userContactInfoDeleteCallParms.setUserid(userid);
userContactInfoDeleteCallParms.setLabels(new String[] { contactinfoLabel });
adminBinding.userContactInfoDelete(userContactInfoDeleteCallParms);
```

Retrieve contact information

```
// This sample retrieves a user contact info entry named
// Work Email and displays its value
UserContactInfoGetCallParms userContactInfoGetCallParms = new
UserContactInfoGetCallParms();
userContactInfoGetCallParms.setUserid(userid);
userContactInfoGetCallParms.setLabels(new String[] { "Work Email" });
ContactInfo[] contactInfo =
adminBinding.userContactInfoGet(userContactInfoGetCallParms);
if (contactInfo != null && contactInfo.length == 1) {
    System.out.println("Address for Work Email is: " + contactInfo[0].getValue());
}
```

Assign and modify a token

After tokens are loaded into Entrust IdentityGuard, you can assign them to your end users. Token devices generate a dynamic password for use in authenticating users. Token authentication may also require a PVN.

Note: A token is specified by both a token vendor and a serial number. In administration operation, if the vendor is not specified, the default token vendor is used. If you specify a serial number without a token vendor, Entrust IdentityGuard does not search among all token vendors to match a token with that serial number.

The following examples show how to assign and modify a token.

Assign a token

```
// The following code assigns a token to the given user and sets the
// token state to HOLD_PENDING
UserTokenParms userTokenParms = new UserTokenParms();
userTokenParms.setState(State.HOLD_PENDING);
UserTokenAssignCallParms userTokenAssignCallParms = new UserTokenAssignCallParms();
userTokenAssignCallParms.setUserid(userid);
userTokenAssignCallParms.setSerialNumber(serialNumber);
// if not specified, the token vendor defaults to the default token vendor
userTokenAssignCallParms.setVendorId(tokenVendor);
userTokenAssignCallParms.setParms(userTokenParms);
adminBinding.userTokenAssign(userTokenAssignCallParms);
```

Modify a token

```
// The following code changes the state of a user token to PENDING
UserTokenFilter userTokenFilter = new UserTokenFilter();
userTokenFilter.setSerialNumber(serialNumber);
// if not specified, the token vendor defaults to the default token vendor
userTokenFilter.setVendorId(tokenVendor);
UserTokenParms userTokenParms = new UserTokenParms();
userTokenParms.setState(State.PENDING);
UserTokenSetCallParms userTokenSetCallParms = new UserTokenSetCallParms();
userTokenSetCallParms.setUserid(userid);
userTokenSetCallParms.setFilter(userTokenFilter);
userTokenSetCallParms.setParms(userTokenParms);
adminBinding.userTokenSet(userTokenSetCallParms);
```

Reset a user token

If a user's token drifts, either through clock drift or through token event drift, it can no longer be used to authenticate with Entrust IdentityGuard.

To fix token clock drift or token event drift, you must reset the token by synchronizing the token with the Entrust IdentityGuard server.

```
// The following code resets the token for the user
UserTokenParms userTokenParms = new UserTokenParms();
userTokenParms.setResetToken(Boolean.TRUE);
// Some token implementations additionally require one or two token responses to
// be provided. If your deployment supports tokens with different reset
// requirements, you must look up what the current token
// requires using the TokenVendorInfo object
userTokenParms.setResetTokenResponse1(resetTokenResponse1);
userTokenParms.setResetTokenResponse2(resetTokenResponse2);
UserTokenSetCallParms userTokenSetCallParms = new UserTokenSetCallParms();
userTokenSetCallParms.setUserid(userid);
UserTokenFilter filter = new UserTokenFilter();
// This filter indicates which token to reset.
// This filter must specify a single token.
filter.setSerialNumber(serialNumber);
// if not specified, the token vendor defaults to the default token vendor
filter.setVendorId(tokenVendor);
userTokenSetCallParms.setFilter(filter);
userTokenSetCallParms.setParms(userTokenParms);
int numSet = adminBinding.userTokenSet(userTokenSetCallParms);
```

Unlock a user token

Challenge-response tokens, such as the Entrust Pocket Token, require that the user enter a token PIN before the token will generate a token password. If the user enters an incorrect token PIN repeatedly (four times for the Entrust Pocket Token), the token locks.

The next time the user clicks the token button to generate a response, a “locked” message appears, along with an unlock challenge. The user can no longer authenticate to the Entrust IdentityGuard server, and must contact an administrator and ask for the token to be unlocked.

```
// The following code unlocks the specified token
UserTokenUnlockCallParms userTokenUnlockCallParms = new UserTokenUnlockCallParms();
userTokenUnlockCallParms.setUserid(userid);
UserTokenFilter filter = new UserTokenFilter();
// The filter indicates which token is to be unlocked.
// This filter must specify a single token.
filter.setSerialNumber(serialNumber);
// if not specified, the token vendor defaults to the default token vendor
filter.setVendorId(tokenVendor);
userTokenUnlockCallParms.setFilter(filter);
// The unlock challenge is generated by the token, and
// must be obtained from the user.
```

```
userTokenUnlockCallParams.setChallenge(challenge);
UserTokenUnlockResult result = adminBinding.userTokenUnlock(userTokenUnlockCallParams);
// Then communicate the unlock code to the user.
// The user must enter this number to unlock the token.
String unlockCode = result.getUnlockCode();
```

Create and modify a temporary PIN

If users lose a grid card or token, or if users are waiting for their first grid card or token, you can issue them a temporary PIN. The following code samples show how to create and retrieve a temporary PIN.

Create a user temporary PIN

The following example creates a temporary PIN for a user and sets its lifetime to one day.

```
// The following code creates a temporary PIN for
// a given user and replaces the existing temporary PIN
UserPINParams userPINParams = new UserPINParams();
// The lifetime of the PIN in milliseconds
userPINParams.setLifetime(1000L * 60 * 60 * 24);
// Replace any existing temporary PIN.
userPINParams.setForce(true);
UserPINCreateCallParams callParams = new UserPINCreateCallParams();
callParams.setUserid(userid);
callParams.setParams(userPINParams);
adminBinding.userPINCreate(callParams);
```

Retrieve a user temporary PIN

The administrator must have `userPinView` permission to retrieve the temporary PIN.

```
// The following code retrieves a temporary PIN for a given user.
UserPINGetCallParams callParams = new UserPINGetCallParams();
callParams.setUserid(userid);
UserPINInfo pin = adminBinding.userPINGet(callParams);
// user PIN could have be composed of multiple cells
String[] pinValues = pin.getPIN();
```

Modify a user temporary PIN

```
// The following code modifies the lifetime and maximum use times
// of the user's temporary PIN
```

```
UserPINParms userPINParms = new UserPINParms();
// The lifetime of the PIN in milliseconds
userPINParms.setLifetime(lifetime);
userPINParms.setMaxUses(maxUses);
UserPINSetCallParms callParms = new UserPINSetCallParms();
callParms.setUserid(userid);
callParms.setParms(userPINParms);
adminBinding.userPINSet(callParms);
```

Create and modify a personal verification number (PVN)

The PVN feature provided with Entrust IdentityGuard lets you add an extra level of security when using grids, tokens, and one-time passwords (OTP). That is, any grid, token, or OTP challenge issued to a user can also include a PVN challenge.

The following examples allow you to manage the user PVN:

Create a user PVN

```
// The following code creates a PVN for a given user
// and forces the user to change the PVN value.
UserPVNCreateParms userPVNCreateParms = new UserPVNCreateParms();
userPVNCreateParms.setAutoGenerate(Boolean.TRUE);
// Optionally, you can also specify the PVN value
// userPVNCreateParms.setPVN("1234");

// Require the user to change the PVN value
userPVNCreateParms.setChangeRequired(Boolean.TRUE);
UserPVNCreateCallParms userPVNCreateCallParms = new UserPVNCreateCallParms();
userPVNCreateCallParms.setUserid(userid);
userPVNCreateCallParms.setParms(userPVNCreateParms);
adminBinding.userPVNCreate(userPVNCreateCallParms);
```

Retrieve a user PVN

To retrieve a user PVN, the administrator must have the `userPvnView` permission, and the user must not have changed the PVN.

```
// The following code retrieves a PVN for a given user
UserPVNGetCallParms userPVNGetCallParms = new UserPVNGetCallParms();
userPVNGetCallParms.setUserid(userid);
UserPVNInfo pvnInfo = adminBinding.userPVNGet(userPVNGetCallParms);
String pvnString = pvnInfo.getPVN();
```

Update a user PVN

```
// The following code updates a PVN for a given user
UserPVNSetParms userPVNSetParms = new UserPVNSetParms();
userPVNSetParms.setPVN(newPVN);
UserPVNSetCallParms userPVNSetCallParms = new UserPVNSetCallParms();
userPVNSetCallParms.setUserid(userid);
userPVNSetCallParms.setParms(userPVNSetParms);
adminBinding.userPVNSet(userPVNSetCallParms);
```

Delete a user PVN

```
// The following code deletes a PVN for a given user
UserPVNDeleteCallParms userPVNDeleteCallParms = new UserPVNDeleteCallParms();
userPVNDeleteCallParms.setUserid(userid);
adminBinding.userPVNDelete(userPVNDeleteCallParms);
```

Set up a user's questions and answers

For knowledge-based authentication to work, you must first ask users a series of questions to which the user must respond with answers. Your organization creates these questions, which are usually personalized to ensure that only the user can respond correctly. Users provide their answers at registration. Entrust IdentityGuard stores the questions and answers for later user authentication.

The following shows how to present questions, collect the answers, and store the results.

```
// Set the questions for the user's QA authentication
NameValue[] qaPairs = new NameValue[questionArray.length];
for (int i = 0; i < qaPairs.length; i++) {
    qaPairs[i] = new NameValue();
    qaPairs[i].setName(questionArray[i]);
    qaPairs[i].setValue(answerArray[i]);
}
UserParms userParms = new UserParms();
userParms.setQaSecrets(qaPairs);
UserSetCallParms userSetCallParms = new UserSetCallParms();
userSetCallParms.setUserid(userid);
userSetCallParms.setParms(userParms);
adminBinding.userSet(userSetCallParms);
```

Unlock users

Entrust IdentityGuard locks out users if they fail a specified number of authentication attempts. When lockout occurs, your application can provide a way for the locked-out users to request an unlock if they can prove their identity some other way.

When a user is locked out using one authentication method, they may not be locked out of all other authentication methods. Unlocking a user resets the lock counter for all authentication methods.

The following shows one approach to unlocking users:

```
// The following code unlocks the user
UserParms userParms = new UserParms();
userParms.setLockoutParms(new UserLockoutParms(Boolean.TRUE, Boolean.FALSE, null));
```

```
UserSetCallParms userSetCallParms = new UserSetCallParms();
userSetCallParms.setUserid(userid);
userSetCallParms.setParms(userParms);
adminBinding.userSet(userSetCallParms);
```

Administer machine secrets

User's machine secrets are stored at the Entrust IdentityGuard repository. Administrators can remove all the machine secrets for a user, or delete specific machine secrets for a user.

Clear machine secrets

An administrator can remove all machine secrets for the user. If the machine secrets are removed, users are required to re-register the machine during their next authentication attempt.

```
// The following code clears all machine secrets for the user
UserParms userParms = new UserParms();
userParms.setClearMachineSecrets(Boolean.TRUE);

UserSetCallParms callParms = new UserSetCallParms();
callParms.setUserid(userid);
callParms.setParms(userParms);
adminBinding.userSet(callParms);
```

Delete machine secrets

Administrators can delete some of the machine secrets belonging to a user. You would delete a machine secret to force a user to reauthenticate, or if a machine secret has been established in error; a machine secret was associated with a public computer or one that does not belong to the user, for example.

The following code shows one approach for deleting a machine secret, by specifying the nonce. To determine what nonce is associated with what machine secret, it is generally necessary to retrieve all the machine secrets associated with a user using `userMachineSecretList`, and then display their attributes: create date, last used date, and machine label. Based on that information, determine the machine secret or secrets to be deleted, and obtain the associated machine nonce from the corresponding `MachineSecretInfo` object.

```
// The following code deletes the specified machine secret for the user
UserMachineSecretDeleteCallParms userMachineSecretDeleteCallParms = new
UserMachineSecretDeleteCallParms();
userMachineSecretDeleteCallParms.setUserid(userid);
// you can only delete the machine secret by specifying the machine nonce
userMachineSecretDeleteCallParms.setMachineNonce(machineNonce);
adminBinding.userMachineSecretDelete(userMachineSecretDeleteCallParms);
```

Administrative monitoring tasks

This section explains how to monitor your grid card and token inventory.

Check for expiring grid cards

For planning purposes, it is useful to know how many assigned grid cards will expire in the near future. This lets you create, manufacture, and distribute new grid cards in a timely fashion.

The following shows how to check for grid cards that will expire soon:

```
// The following code returns the user cards
// that will expire in 10 days or less
UserFilter userFilter = new UserFilter();

// Set the expire end date to 10 days from now
Calendar expiryDate = Calendar.getInstance();
expiryDate.add(Calendar.DATE, 10);
userFilter.setExpireEndDate(expiryDate);

// Return 50 cards for this call. The default value for
// max return is 100.
userFilter.setMaxReturn(50);
UserCardListCallParms callParms = new UserCardListCallParms();
callParms.setFilter(userFilter);

// No next user to begin with
BigInteger nextUser = null;
System.out.println("Looking for cards that will expire in the next ten days...");
do {
    userFilter.setNextUser(nextUser);
    UserCardListResult result = adminBinding.userCardList(callParms);
    if (result == null) {
        break;
    }
    UserCardInfo[] cards = result.getCards();
    for (int i = 0; i < cards.length; i++) {
        UserCardInfo card = cards[i];
        System.out.println("Card with serial number " + card.getSerialNumber() + "
        belonging to user " + card.getUserName() + " in group " + card.getGroup() + " will
        expire on "
        +
        DateFormat.getDateInstance().format(card.getExpireDate().getTime()));
    }
}
```

```
    nextUser = result.getNextUser();
} while (nextUser.intValue() != 0);
```

Check grid card inventory

For planning purposes, it is useful to know how many unassigned preproduced grid cards are still available.

The following shows an easy way to count your remaining unassigned grid cards:

```
// The following code returns the number of
// preproduced cards for each group
PreproducedCardFilter preproducedCardFilter = new PreproducedCardFilter();
// If you want, you can restrict searching for preproduced
// cards to one or more specific groups
// preproducedCardFilter.setGroups(new String[] {"default"} );
PreproducedCardListCallParms listCallParms = new PreproducedCardListCallParms();
listCallParms.setFilter(preproducedCardFilter);
// No next card to begin with
String nextCard = null;
Map<String, Long> preproducedCardCounts = new HashMap();
System.out.println("Determining the number of preproduced cards for each group...");
do {
    preproducedCardFilter.setNextCard(nextCard);
    PreproducedCardListResult result =
adminBinding.preproducedCardList(listCallParms);
    if (result == null) {
        break;
    }
    PreproducedCardInfo[] cards = result.getCards();
    for (int i = 0; i < cards.length; i++) {
        PreproducedCardInfo card = cards[i];
        String group = card.getGroup();
        if (preproducedCardCounts.containsKey(group)) {
            preproducedCardCounts.put(group, preproducedCardCounts.get(group) + 1);
        } else {
            preproducedCardCounts.put(group, 1L);
        }
    }
    // get the new next card serial number
    nextCard = result.getNextCard();
} while (nextCard != null);
if (preproducedCardCounts.isEmpty()) {
```

```

        System.out.println("There are no preproduced cards in the system!");
    } else {
        Set<Entry<String, Long>> entries = preproducedCardCounts.entrySet();
        for (Iterator<Entry<String, Long>> it = entries.iterator(); it.hasNext();) {
            Entry<String, Long> entry = it.next();
            System.out.println("Group " + entry.getKey() + " has " + entry.getValue() + "
preproduced cards.");
        }
    }
}

```

Check for unused assigned grid cards

```

// The following code returns the assigned cards that have never been used,
// e.g., card state is PENDING or HOLD_PENDING
UserFilter userFilter = new UserFilter();
userFilter.setStates(new State[] { State.PENDING, State.HOLD_PENDING });
// Return 50 cards for this call. The default value for
// max return is 100.
userFilter.setMaxReturn(50);
UserCardListCallParms callParms = new UserCardListCallParms();
callParms.setFilter(userFilter);
// No next user to begin with
BigInteger nextUser = null;
do {
    userFilter.setNextUser(nextUser);
    UserCardListResult result = adminBinding.userCardList(callParms);
    if (result == null) {
        break;
    }
    UserCardInfo[] cards = result.getCards();
    for (int i = 0; i < cards.length; i++) {
        UserCardInfo card = cards[i];
        System.out.println("User " + card.getUserName() + " in group " +
card.getGroup() + " has a card with serial number " + card.getSerialNumber() + " in
state " + card.getState());
    }
    nextUser = result.getNextUser();
} while (nextUser.intValue() != 0);

```

Check token inventory

For planning purposes, it is useful to know how many unassigned tokens are still available.

The following shows an easy way to count your remaining unassigned tokens:

```
// The following code returns the number of unassigned tokens for each group
TokenFilter tokenFilter = new TokenFilter();
TokenListCallParms tokenListCallParms = new TokenListCallParms();
tokenListCallParms.setFilter(tokenFilter);
// Need to do this for each supported token vendor
TokenVendorInfo[] vendorInfo = adminBinding.tokenVendorList(new NameValue[0]);
for (int i = 0; i < vendorInfo.length; i++) {
    tokenFilter.setVendorId(vendorInfo[i].getVendorId());
    // If you want, you can restrict searching for
    // tokens to one or more specific groups
    // tokenFilter.setGroups(new String[] {"default"} );
    Map<String, Long> unassignedTokenCounts = new HashMap();
    // No next token to begin with
    String nextTokenSerialNumber = null;
    do {
        tokenFilter.setNextTokenSerialNumber(nextTokenSerialNumber);
        TokenListResult result = adminBinding.tokenList(tokenListCallParms);
        if (result == null) {
            break;
        }
        TokenInfo[] tokens = result.getTokens();
        for (int j = 0; j < tokens.length; j++) {
            TokenInfo token = tokens[j];
            String group = token.getGroup();
            if (unassignedTokenCounts.containsKey(group)) {
                unassignedTokenCounts.put(group, unassignedTokenCounts.get(group) +
1);
            } else {
                unassignedTokenCounts.put(group, 1L);
            }
        }
        nextTokenSerialNumber = result.getNextTokenSerialNumber();
    } while (nextTokenSerialNumber != null);
    if (unassignedTokenCounts.isEmpty()) {
        System.out.println("There are no unassigned tokens for vendor " +
vendorInfo[i].getVendorName() + " in the system");
    } else {
        System.out.println("For token vendor " + vendorInfo[i].getVendorName() + ":");
        Set<Entry<String, Long>> entries = unassignedTokenCounts.entrySet();
    }
}
```

```

        for (Iterator<Entry<String, Long>> it = entries.iterator(); it.hasNext();) {
            Entry<String, Long> entry = it.next();
            System.out.println("Group " + entry.getKey() + " has " + entry.getValue()
+ " unassigned tokens.");
        }
    }
}

```

Check for unused assigned grid cards or tokens

For planning and monitoring purposes, it may be useful to know which users have not yet used their assigned grid cards or tokens. If you issue new grid cards and tokens in the Pending state, their state changes to Current the first time a user successfully authenticates with them. You could also choose to issue grids in the Hold Pending state and force users to go through a registration process that moves their grid to the Current state.

The following shows how to list grid cards in the Pending and Hold Pending states.

Check for unused assigned grid cards

```

// The following code returns the assigned cards that have never been used,
// e.g., card state is PENDING or HOLD_PENDING
UserFilter userFilter = new UserFilter();
userFilter.setStates(new State[] { State.PENDING, State.HOLD_PENDING });
// Return 50 cards for this call. The default value for max return is 100.
userFilter.setMaxReturn(50);
UserCardListCallParams callParams = new UserCardListCallParams();
callParams.setFilter(userFilter);
// No next user to begin with
BigInteger nextUser = null;
do {
    userFilter.setNextUser(nextUser);
    UserCardListResult result = adminBinding.userCardList(callParams);
    if (result == null) {
        break;
    }
    UserCardInfo[] cards = result.getCards();
    for (int i = 0; i < cards.length; i++) {
        UserCardInfo card = cards[i];
        System.out.println("User " + card.getUserName() + " in group " +
card.getGroup() + " has a card with serial number " + card.getSerialNumber() + " in
state " + card.getState());
    }
    nextUser = result.getNextUser();
}

```

```
} while (nextUser.intValue() != 0);
```

Check for unused assigned tokens

Another approach, which is valid for tokens, is to search for those users who have not used their token for the past given number of days. For example, if you know your user community received tokens two weeks ago, then you can find those users who have not used their token since that time.

```
// The following code returns the assigned tokens
// that have not been used for the past 14 days
UserFilter userFilter = new UserFilter();
// Set the last used end date to 14 days ago
Calendar tokenLastUsedEndDate = Calendar.getInstance();
tokenLastUsedEndDate.add(Calendar.DATE, -14);
userFilter.setTokenLastUsedEndDate(tokenLastUsedEndDate);
// Return 50 tokens for this call. The default value for max return is 100.
userFilter.setMaxReturn(50);
UserTokenListCallParms callParms = new UserTokenListCallParms();
callParms.setFilter(userFilter);
// No next user to begin with
BigInteger nextUser = null;
do {
    userFilter.setNextUser(nextUser);
    // Return the tokens
    UserTokenListResult result = adminBinding.userTokenList(callParms);
    if (result == null) {
        break;
    }
    UserTokenInfo[] tokens = result.getTokens();
    for (int i = 0; i < tokens.length; i++) {
        UserTokenInfo token = tokens[i];
        System.out.println("User " + token.getUserName() + " in group " +
            token.getGroup() + " has a token with serial number " + token.getSerialNumber() + "
            from vendor "
                + token.getVendorId() + " that is in state " + token.getState() + "
            and has not been used in 14 days.");
    }
    nextUser = result.getNextUser();
} while (nextUser.intValue() != 0);
```

Administration of smart credentials

This section explains how to use the Administration API to manage your smart credential inventory.

Note: Before your application can administer smart credentials, you must configure your Entrust IdentityGuard system for creating, enrolling, and issuing smart credentials. Normally, this requires that you first install and configure the Print Module and Enrollment Module. You must also configure a managed CA in Entrust IdentityGuard to create PIV credentials. Refer to the *Entrust IdentityGuard Smart Credentials Guide* for the required steps.

Create smart credentials for a user

Before creating smart credentials for a user, make sure that both the user and a smart credential definition exist. In the following sample code, the definition is named "PIV".

```
/**
 * Create smart credentials for a user.
 *
 * @param binding
 * @throws AdminServiceFault
 * @throws RemoteException
 */
public static void userSmartCredentialCreate(AdminServiceBindingStub binding) throws
AdminServiceFault, RemoteException {

    // The name of the user who will receive the smart credential
    String userid = "userid";
    UserSmartCredentialParms parms = new UserSmartCredentialParms();

    // The name of the smart credential definition to use
    parms.setDefinitionId("PIV");

    String id = binding.userSmartCredentialCreate(new
UserSmartCredentialCreateCallParms(userid, parms));
    System.out.println("SmartCredential " + id + " has been created.");
}
```

Approve smart credentials

Some types of smart credentials must be approved by an administrator before being issued. The following code sample shows how to do this.

```
/**
 * Approve smart credentials.
 *
 * @param binding
 * @throws AdminServiceFault
 * @throws RemoteException
 */
public static void userSmartCredentialApprove(AdminServiceBindingStub binding) throws
AdminServiceFault, RemoteException {

    // The name of the user whose smart credentials will be approved.
    String userid = "userid";

    // The ID of the smart credentials to approve.
    String id = "ET9820551";

    UserSmartCredentialParms parms = new UserSmartCredentialParms();
    parms.setApproved(Boolean.TRUE);
    binding.userSmartCredentialSet(new UserSmartCredentialSetCallParms(userid, id,
parms));

    System.out.println("SmartCredential " + id + " approved.");
}
```

Issue smart credentials

The following code sample shows how to issue smart credentials for a user. Because a user might have more than one smart credential, the code must provide a smart credential ID to the API. In the sample, the smart card will be encoded by a smart card reader installed on the computer that hosts the Print Module. The Print Module was configured with the name "PrintOnly". Because a smart card reader will be used, the print operation is "EXTERNAL_ENCODE".

```
/**
 * Issue smart credentials
 *
 * @param binding
 * @throws AdminServiceFault
 * @throws RemoteException
```

```

*/

public static void userSmartCredentialIssue(AdminServiceBindingStub binding) throws
AdminServiceFault, RemoteException {

    // Set the name of the user whose smart credentials will be issued.
    String userid = "userid";

    // Set the ID of the smart credentials to issue.
    String id = "ET9820551";

    // Set the name of the Printer Module to use for encoding the card.
    // This is one of the names that are shown in Webadmin under
    // Smart Credentials, Configuration, Configuration Option, Print Modules.
    String printer = "PrintOnly";

    ArrayList<UserSmartCredentialIssueOp> ops = new
    ArrayList<UserSmartCredentialIssueOp>();

    // Set the print module to encode to a card using an external printer or // smart
    card
    // reader.
    ops.add(UserSmartCredentialIssueOp.EXTERNAL_ENCODE);

    // Issue the smart credentials
    binding.userSmartCredentialIssue(new UserSmartCredentialIssueCallParms(userid, id,
    printer, ops.toArray(new UserSmartCredentialIssueOp[ops.size()]));

    System.out.println("Sent issuance request to Print Module " + printer + " for " +
    userid + " " + id + " " + UserSmartCredentialIssueOp.EXTERNAL_ENCODE);

    // Refer to userSmartCredentialCheckStatus for how to check the status of the
    // issuance request.
}

```

Check the status of smart credentials issuance request

The above API (see “Issue smart credentials ” on page [110](#)) sends a request to the Print Module to issue a smart credential. That request might be queued initially and executed subsequently. Your application can check the status of its request by using the following code sample. You will find the possible responses in the Javadoc for the class `SmartCredentialIssueState`, and they include: `QUEUED`, `ENCODE_DONE`, `ENCODE_ERROR`, and others. Your application should respond to each of these states.

```

/**
 * Check the status of smart credentials issuance.
 *
 * @param binding
 * @throws AdminServiceFault
 * @throws RemoteException
 */
public static void userSmartCredentialCheckStatus(AdminServiceBindingStub binding)
throws AdminServiceFault, RemoteException {

    // Set the name of the user whose issuance status be checked.
    String userid = "userid";

    // Set the ID of the smart credentials whose issuance status be checked.
    String id = "ET9820551";

    UserSmartCredentialGetParms getParms = new UserSmartCredentialGetParms(true,
true);

    UserSmartCredentialGetCallParms callParms = new
UserSmartCredentialGetCallParms(userid, id, getParms);

    UserSmartCredentialInfo smartCredential =
binding.userSmartCredentialGet(callParms);

    SmartCredentialIssueState state = smartCredential.getIssueState();

    if (state != null) {
        System.out.println("Issue State: " + smartCredential.getIssueState());
    } else {
        System.out.println("Issue State could not be determined");
    }
}

```

Modify smart credentials

After they are created, smart credentials move through the states “Enrolling”, “Enrolled”, and “Approved”. After they are approved, the smart credentials can be issued. After enrollment and before being approved or issued, you might want your application to modify certain fields in the smart credentials; for example, your application could populate the first and last names and the email address of the enrolling user by retrieving that information from a database or LDAP directory. The following code sample shows how to set those enrollment values. After running the code, you can confirm that the enrollment values were populated by using the Entrust IdentityGuard Administration interface.

The code sample finds unassigned smart credentials of a given Entrust IdentityGuard user. In general, your application could search for enrolling smart credentials based on other criteria. Refer to the Javadocs for the `UserFilter` and the `UserSmartCredentialInfo` classes to see the available search criteria.

```
/**
 * Modify the enrollment values for smart credentials.
 *
 * @param binding
 * @throws AdminServiceFault
 * @throws RemoteException
 */
public static void userSmartCredentialModify(AdminServiceBindingStub binding) throws
AdminServiceFault, RemoteException {

    // The user whose smart credentials will be modified.
    String userid = "userid";

    // Get all smart credentials for this user.
    UserFilter filter = new UserFilter();
    filter.setUserId(userid);

    UserSmartCredentialListResult result = binding.userSmartCredentialList(new
UserSmartCredentialListCallParms(filter));

    UserSmartCredentialInfo[] smartCredentials = result.getSmartCredentials();

    if (smartCredentials == null || smartCredentials.length == 0) {
        System.out.println("Failed to find smart credentials for user: " + userid);
        return;
    }

    // Find smart credentials that were not assigned yet.
    // Note: This assumes the user has only one such smart credential.
    UserSmartCredentialInfo info = null;
    for (int i = 0; i < smartCredentials.length; i++) {
```

```

        UserSmartCredentialInfo tmp = smartCredentials[i];
        if (SmartCredentialState.UNASSIGNED.equals(tmp.getState())) {
            info = tmp;
            break;
        }
    }
    if (info == null) {
        System.out.println("User '" + userid + "' has no unassigned smart
credentials.");
        return;
    }

    System.out.println("Found smart credentials " + info.getId() + " for user " +
info.getUserid() + " in state: " + info.getState());

    // Modify the smart credential enrollment values
    userSmartCredentialSetEnrollmentValues(info, binding);
}

```

Modify enrollment values in smart credentials

To modify enrollment values of the smart credentials, create a list of enrollment name-value pairs as shown in the sample code in this section. The following example sets the first name, last name, and email address.

```

public static void userSmartCredentialSetEnrollmentValues(UserSmartCredentialInfo
scinfo, AdminServiceBindingStub binding) throws AdminServiceFault, RemoteException {
    // Create a list of enrollment values to set or modify
    List tplist = new ArrayList();

    NameValue tp = new NameValue();
    tp.setName("firstname");
    tp.setValue("John");
    tplist.add(tp);
    tp = new NameValue();
    tp.setName("lastname");
    tp.setValue("Smith");
    tplist.add(tp);
    tp = new NameValue();
    tp.setName("emailaddress");
    tp.setValue("jsmith@entrust.com");
    tplist.add(tp);
}

```

```

NameValue[] tpararray = new NameValue[tplist.size()];
tplist.toArray(tpararray);

UserSmartCredentialParms parms = new UserSmartCredentialParms();

// The above values were not encrypted here, so we request IG to encrypt.
parms.setRawEnrollmentValues(Boolean.TRUE);

// Set the enrollment values into the request
parms.setEnrollmentValues(tpararray);

// Send the request
binding.userSmartCredentialSet(new
UserSmartCredentialSetCallParms(scinfo.getUserid(), scinfo.getId(), parms));
}

```

Change the state of a smart credential

Administrators may often need to change the state of a smart credential.

Place a smart credential on hold

An administrator might need to suspend the use of a smart credential temporarily. For example, if a user has misplaced their smart card, it can be placed on hold until it is found. When the card is found, it can be restored to the active state. Administrators normally perform such operations manually. To do that, the administrator would find the smart credentials using the Entrust IdentityGuard Administration application, select **Edit Smart Credential**, and choose a state from the drop-down list. Refer to “Smart Credential States” in the *Entrust IdentityGuard Smart Credentials Guide* for information on this topic.

Cancel, delete, or unassign a smart credential

An administrator might need to cancel, delete or unassign a smart credential. Administrators normally perform such operations manually using the Entrust IdentityGuard Administration application. If the card was lost or damaged, or a replacement is required because of a name change or other change to user information, an administrator can change the smart credential state manually. Refer to “Smart Credential States” in the *Entrust IdentityGuard Smart Credentials Guide* for more information.

If you do need to create an application that changes the state of a smart credential programmatically, the following sample code demonstrates the API calls. It simply finds an arbitrary ACTIVE smart credential, sets it to HOLD, confirms its new state, and returns the smart credential to the ACTIVE state.

Sample code: Changing states

```

/**
 * Set an active smart credentials to HOLD.

```

```

*
* @param binding
* @throws AdminServiceFault
* @throws RemoteException
*/

public static void userSmartCredentialHold(AdminServiceBindingStub binding) throws
AdminServiceFault, RemoteException {

    // Return 100 users at most
    UserFilter filter = new UserFilter();
    filter.setNextUser(BigInteger.ONE);
    filter.setMaxReturn(new Integer(100));

    UserSmartCredentialListResult result = binding.userSmartCredentialList(new
UserSmartCredentialListCallParms(filter));
    UserSmartCredentialInfo[] smartCredentials = result.getSmartCredentials();

    // As an example, find any ACTIVE smart credential and set it to HOLD.
    SmartCredentialState state = SmartCredentialState.ACTIVE;
    UserSmartCredentialInfo scinfo = null;
    for (int i = 0; i < smartCredentials.length; i++) {
        scinfo = smartCredentials[i];
        if (state.equals(scinfo.getState())) {
            System.out.println("User " + scinfo.getGroup() + "/" +
scinfo.getUserName() + " has smart credentials " + scinfo.getId() + " in state " +
scinfo.getState());

            // set to HOLD

            UserSmartCredentialParms parms = new UserSmartCredentialParms();
            parms.setState(SmartCredentialState.HOLD);
            UserSmartCredentialSetCallParms p = new
UserSmartCredentialSetCallParms(scinfo.getUserid(), scinfo.getId(), parms);
            binding.userSmartCredentialSet(p);

            UserSmartCredentialGetParms getParms = new
UserSmartCredentialGetParms(true, true);

            // confirm the smart credential is on HOLD
            scinfo = binding.userSmartCredentialGet(new
UserSmartCredentialGetCallParms(scinfo.getUserid(), scinfo.getId(), getParms));

            System.out.println("User " + scinfo.getGroup() + "/" +
scinfo.getUserName() + " has smart credentials " + scinfo.getId() + " in state " +
scinfo.getState());

            // set back to ACTIVE before leaving
            parms.setState(SmartCredentialState.ACTIVE);
            p = new UserSmartCredentialSetCallParms(scinfo.getUserid(),
scinfo.getId(), parms);

```

```
        binding.userSmartCredentialSet(p);  
    }  
}  
}
```

Chapter 5:

Programming additional Entrust IdentityGuard functionality

This chapter explains how to extend the functionality of your Entrust IdentityGuard application.

Customizing out-of-band delivery of OTPs

Entrust IdentityGuard supports out-of-band (OOB) delivery methods for delivery of OTPs using Authenticate (telephone delivery), JavaMail, and SMS delivery to HTTP or HTTPS gateways.

You can program your own delivery methods, such as Instant Message delivery, using the Entrust IdentityGuard-provided Java interface.

Adding properties to `identityguard.properties` to support new OOB delivery methods

Entrust IdentityGuard includes a group of properties, all defined in the `identityguard.properties` file. The OOB delivery properties are contained in the

- Out-of-Band Email Delivery Configuration,
- Out-of-Band Voice Delivery Configuration, and the
- Out-of-Band SMS Delivery Configuration

sections of the Properties Editor interface.

You can configure and update the three built-in OOB delivery methods using the Entrust IdentityGuard Properties Editor, but to add new OOB delivery mechanisms, you can edit the `identityguard.properties` file directly or use **Add A New Custom Property** in the Properties Editor.

Note: For more information about Entrust IdentityGuard properties and the Properties Editor, see the *Entrust IdentityGuard Server Administration Guide*.

Programming a new delivery method requires that you add a new set of properties to the `identityguard.properties` file.

Attention: These properties must be in place before you can start testing your new OOB implementation.

To add new OOB properties to the `identityguard.properties` file

- 1 Choose a `type` and `name` for your new properties group. These are used in defining the new properties for your OOB mechanism.

All of your new properties begin with

```
identityguard.oobdelivery.<type>.<name>
```

where

- `<type>` is the type name of the OOB delivery mechanism you are adding.

Note: The `javamail`, `authenticate`, and `msggateway` types are reserved within Entrust IdentityGuard.

- `<name>` is the name of this particular instance of the new delivery mechanism

For example: If you are programming a new OOB delivery system for the fictional mobile phone company, “Sell Mobility,” you could choose to begin all your new property names with

```
identityguard.oobdelivery.sms.sell
```

- 2 Add the `impl` property. Using the Sell Mobility example above, your new property would be

```
identityguard.oobdelivery.sms.sell.impl
```

The value of your new `impl` property must be the full name of the new Java class you are creating which implements the interface.

```
com.entrust.identityguard.common.oobDelivery.OOBDeliveryV2Intf
```

- 3 Create a new `displayname` property of the form

```
identityguard.oobdelivery.<type>.<name>.displayname
```

This command sets the label that is shown to your users when presenting the OOB options. Continuing with the SMS example, this property would be

```
identityguard.oobdelivery.sms.sell.displayname = Sell Mobility
```

- 4 Note that you can also add any other properties you need at this stage. These new properties are available as parameters in the `initialize()` method defined by the Entrust IdentityGuard interface. What properties you need depends entirely upon your particular implementation.

The following examples show some of the properties you might choose to add for an XMPP implementation that communicates with Google Talk.

```
identityguard.oobdelivery.xmpp.google.host=talk.google.com
```

```
identityguard.oobdelivery.xmpp.google.port=5222
```

```
identityguard.oobdelivery.xmpp.google.userid=[Google Account Name]
```

```
&identityguard.oobdelivery.xmpp.google.password=[Encrypted Google Account Password]
```

- 5 After you have finished adding the new properties to the `identityguard.properties` file, you must restart the Entrust IdentityGuard services for them to take effect. See the *Entrust IdentityGuard Installation Guide* for instructions on stopping and starting the Entrust IdentityGuard services.

Creating the Java class to implement the Entrust IdentityGuard Java interface

You must create a Java class to add a new out-of-band delivery mechanism to Entrust IdentityGuard. Examples of OOB delivery methods you could add are instant messaging or SMS (text messaging) through an interface other than the one provided for HTTP or HTTPS gateways.

Note: The location of the JAR file that contains the definition of `OOBDeliveryIntf` (the 9.1 interface) and `OOBDeliveryV2Intf` (the 9.2 interface and up) is

`$IG_HOME/lib/IdentityGuardIntf.jar`.

Unless you are extending an existing OOB delivery mechanism developed against the 9.1 interface, you should use the native 9.2 `OOBDeliveryV2Intf` interface.

Creating your Java class

Attention: When creating your Java class, keep in mind that it must accommodate multiple threads concurrently running through a single instance of the class and delivering any number of OTPs simultaneously.

In other words, if there is any possibility that the class will be writing global (class or instance) variables after initialization, all reads and writes of these variables must be properly synchronized.

Create your Java class by implementing the 9.2 native `OOBDeliveryV2Intf` interface. The 9.1 version of the *Entrust IdentityGuard Programmer's Guide for the Java Platform* details the older interface.

OOBDeliveryV2Intf interface

```
/** The entry in the dataToDeliver map containing an OTP. */
public static final String OTP_TOKEN = "otp";

/** The entry in the dataToDeliver map containing a list of all OTPs. */
public static final String OTPS_TOKEN = "otps";

/** The entry in the dataToDeliver map containing a list of all new OTPs. */
public static final String NEW_OTPS_TOKEN = "newOTPs";

/** The entry in the dataToDeliver map containing a list of all old OTPs. */
public static final String OLD_OTPS_TOKEN = "oldOTPs";

/**
 * The entry in the dataToDeliver map containing the operation that resulted
 * in delivery being performed. The value will be one of CHALLENGE,
 * AUTHENTICATE or ADMIN.
 */
public static final String OPERATION_TOKEN = "operation";

/**
 * The entry in the dataToDeliver map indicating if dynamic refresh is
 * enabled.
 */
public static final String DYNAMIC_REFRESH_TOKEN = "dynamicRefresh";

/** The entry in the dataToDeliver map containing the OTP expiry date. */
public static final String EXPIRY_DATE_TOKEN = "expiryDate";
```

```

/**
 * The entry in the dataToDeliver map containing the transaction details.
 */
public static final String TRANSACTION_DETAILS_TOKEN = "transactionDetails";

/**
 * Deliver data OOB to an Entrust IdentityGuard user
 * @param userid the userid of the target user (used for display purposes
 * only)
 * @param fullName the full name of the target user (used for display
 * purposes only)
 * @param userPVN value of the target user's PVN if it's required to gain
 * access to the data being delivered
 * @param contactInfoValue the value (i.e., address/number/handle) to which
 * the data will be delivered
 * @param configLabel the name/label associated with this particular OOB
 * delivery mechanism
 * @param dataToDeliver a map of data to deliver. This map may contain
 * the following values:
 *
 * otp. A string containing an OTP.
 * otps. A Java List containing all OTPs for the user.
 * newOTPs. A Java List containing all new OTPs for the user.
 * oldOTPs. A Java List containing all old OTPs for the user.
 * operation. A string indicating what operation (CHALLENGE, AUTHENTICATE,
ADMIN) resulted in OTPs being delivered.
 * dynamicRefresh. A boolean indicating if OTP dynamic refresh is enabled.
 * expiryDate. A Java Date specifying the expiry date of the OTPs.
 * transactionDetails. A Java Map defining the transaction details.
 *
 * @throws Exception if delivery fails for any reason
 */
public void deliver(String userid,
String fullName,
String userPVN,
String contactInfoValue,
String configLabel,
Map dataToDeliver)
throws Exception;

/**
 * return flag indicating if this delivery config requires a PVN

```

```

*/
public boolean getRequiresPVN();

/**
 * Validate the correctness of the contact information value that will be
 * used as the delivery address.
 * @param contactinfoValue the value (i.e., address/number/handle) to which
 * data will be delivered
 * @return true if validation is successful, false otherwise
 */
public boolean validateContactInfoValue(String contactinfoValue);

/**
 * Initialize the delivery configuration instance
 * @param prop the properties that make up the initialization settings for
 * this delivery instance
 * @throws Exception if anything out of the ordinary happens when
 * attempting to initialize
 */
public void initialize(Properties prop) throws Exception;

/**
 * Close/terminate data delivery.
 * NOTE: Automatic calling of this method is not currently supported.
 * Delivery configurations are responsible for calling this
 * method themselves when they are finished delivering data.
 * If this functionality is not required, simply provide an empty method.
 * @throws Exception if anything out of the ordinary happens when
 * attempting to close/terminate data delivery
 */
public void close() throws Exception;

/**
 * Return the type of the OOB delivery implementation. Currently supported
 * types are EMAIL, PHONE, and OTHER. Custom implementations can return
 * anything they want for their own personal use, but Entrust IdentityGuard
 * itself will always consider custom implementations as being of type
 * OTHER.
 * @return the type of delivery implementation
 */
public String getType();

```

Setting up the Java class for use

Next, you must create your custom JAR file, composed of the Java classes that make up the OOB delivery method you are implementing.

To create the custom jar files

- 1 Compile the Java class you created in “Creating your Java class” on page [120](#) against the JAR file provided with Entrust IdentityGuard; `IdentityGuardIntf.jar`.
- 2 Create your own JAR file that contains your interface implementation class and any auxiliary classes that you create to perform your custom OOB OTP delivery.
- 3 Place the newly created JAR file into `$IG_HOME/lib`. This allows the Master user shell command, `deliveryconfig list`, to find and display your custom delivery configuration.
- 4 If you are using embedded Tomcat, you must also install the new JAR file in the `apache-tomcat<version>/lib` directory.
In this same directory, install any third-party JAR files you are using to perform your OOB delivery.
- 5 If you are using other supported application servers (WebSphere or WebLogic, for example), you must install your new JAR file and any third-party JAR files you are using to perform your OOB delivery in a directory that allows them to be accessible to all deployed Web applications, and particularly the Entrust IdentityGuard services.
- 6 After you have installed all of the JAR files, you must restart the Entrust IdentityGuard Services. See the *Entrust IdentityGuard Installation Guide* for instructions on stopping and starting the Entrust IdentityGuard services.

Integrating external Q&A providers

Entrust IdentityGuard allows you to define external providers of question and answer (Q&A) data that can be used instead of the Entrust IdentityGuard capabilities, to perform knowledge-based authentication.

You can implement your own external Q&A provider using the Java interface Java interface with Entrust IdentityGuard.

Creating the Java class to implement the Entrust IdentityGuard Java interface for external Q&A

You must create a Java class that implements the Java interface `ExternalQAIntf` to define your new external Q&A provider.

The location of the JAR file that contains the definition of `ExternalQAIntf` is `$IG_HOME/lib/IdentityGuardIntf.jar`.

When implementing an instance of the External Q&A interface, you must implement three methods: **`initialize()`**, **`getChallenge`**, and **`authenticate`**.

`initialize()`

This method is called when Entrust IdentityGuard initializes the instance of the external Q&A provider. The implementation should perform whatever initialization operations are required. The parameters passed to this method are the name of a configuration file (which may be null), and a logger instance. If specified, the configuration file will contain configuration values defined for the external Q&A provider. The logger instance can be used to log information to the Entrust IdentityGuard log files.

getChallenge

This method is called by Entrust IdentityGuard to retrieve a Q&A challenge from the external provider. This method has the following arguments:

- the userid of the user – This value consists of the actual userid of the user (not an alias) and includes both the group and user name.
- a locale – If specified, the external Q&A provider can use this value to select questions for a specific locale. This value may be null.
- the number of questions to be returned.

If the specified userid is unknown to the provider, it should throw the `UnknownUser` exception. If the user doesn't have enough questions defined to provide the required challenge it should throw the `NotEnoughQuestions` exception.

The method should return an array of questions.

authenticate

This method is called by Entrust IdentityGuard to authenticate the answer to a QA challenge. This method has the following arguments:

- the userid of the user
- a locale – If specified, the locale to be used when validating the answers
- a map of name/value pairs – Each name corresponds to a question from the previously requested challenge and each value is the answer for that question.

If the specified userid is unknown to the provider, it should throw the `UnknownUser` exception.

The method should return the number of question and answer pairs that were answered correctly.

Creating your Java class for external Q&A

Attention: When creating your Java class, keep in mind that it must accommodate multiple threads concurrently running through a single instance of the class performing any number of authentications simultaneously. In other words, if there is any possibility that the class will be writing global (class or instance) variables after initialization, all reads and writes of these variables must be properly synchronized.

The following code details the Java interface that defines an external Q&A provider.

ExternalQAIntf interface

```
/**
 * This interface defines the interface to a External QA Provider
 */
public interface ExternalQAIntf
{
```

```

/**
 * an exception thrown by the external QA provider if the specified
 * userid is unknown to the provider.
 */
public class UnknownUser extends Exception
{
}

/**
 * an exception thrown by the external QA provider if the user does
 * not have enough questions to generate the requested challenge
 */
public class NotEnoughQuestions extends Exception
{
}

/**
 * Initialize the external QA provider
 * @param fn the name of a file that contains provider specific settings.
 * The format of the file is up to the external QA provider.
 * @param logger a logger that the instance can use to log messages
 * @throws Exception if an error occurs when initializing the provider.
 */
public void initialize(String fn, Logger logger)
throws Exception;

/**
 * Return a challenge (i.e., list of questions) for the given user.
 * @param userid the userid of the user for which a challenge will be
 * generated
 * @param locale if specified, the locale of the user. The external QA
 * can use this value to return questions in the appropriate
 * locale. This value will be null if IdentityGuard does
 * not have the locale of the user.
 * @param numQuestions the number of questions to be returned in the
 * challenge.
 * @returns an array of questions for the specified user.
 * @throws Exception if an error occurs. If the specified userid s
 * not known to the provider, it should throw the UnkownUser
 * exception. If the provider does not have enough questions to
 * generate the requested challenge it should throw the
 * NotEnoughQuestions exception.

```

```

*/
public String[] getChallenge(String userid, Locale locale, int numQuestions)
throws Exception;

/**
 * Authenticate the given response for the given user.
 * @param userid the userid of the user against which the response will
 * be authenticated
 * @param locale if specified, the locale of the user.
 * This value will be null if IdentityGuard does
 * not have the locale of the user.
 * @param response a map of question/answer pairs. The key in the map
 * is the question. The value in the map is the answer.
 * @returns the number of provided answers that were correct.
 * @throws Exception if an error occurs. If the specified userid is
 * not known to the provider, it should throw the UnkownUser
 * exception.
 */
public int authenticate(String userid, Locale locale,
Map<String,String> response)
throws Exception;
}

```

Setting up the Java class for use for external Q&A

Next, you must create your custom JAR file, composed of the Java classes that make up the external provider you are implementing.

To create the custom jar files

- 1** Compile the Java class you created in “Creating the Java class to implement the Entrust IdentityGuard Java interface for external Q&A” on page [123](#) against the JAR file provided with Entrust IdentityGuard; `IdentityGuardIntf.jar`.
- 2** Create your own JAR file that contains your interface implementation class and any auxiliary classes that you create to perform your custom external Q&A provider.
- 3** Place the newly created JAR file into `$IG_HOME/lib`. This allows the Master user shell commands, `externalqa get` and `externalqa list`, to find and display your custom external Q&A configuration.
- 4** If you are using embedded Tomcat, you must also install the new JAR file (and any other required JAR files) in the `apache-tomcat<version>/lib` directory.
In this same directory, install any third-party JAR files you are using to perform external Q&A.
- 5** If you are using other supported application servers (WebSphere or WebLogic, for example), you must install your new JAR file and any third-party JAR files you are using to perform your external Q&A in a directory that allows them to be accessible to all deployed Web applications, and particularly the Entrust IdentityGuard services.
- 6** After you have installed all of the JAR files, you must restart the Entrust IdentityGuard Services. See the *Entrust IdentityGuard Installation Guide* for instructions on stopping and starting the Entrust IdentityGuard services.

Chapter 6:

Performing Identity Assured operations with smart credentials

The Entrust IdentityGuard Mobile Smart Credential provides features that use and manage IdentityGuard smart credentials that are issued by Entrust IdentityGuard and that interface with supported certificate authorities. These credentials can be used over-the-air to:

- Authenticate to Web sites and other applications
- Verify transaction details, sign the details if acceptable, or report possible fraud
- Sign documents for non-repudiation

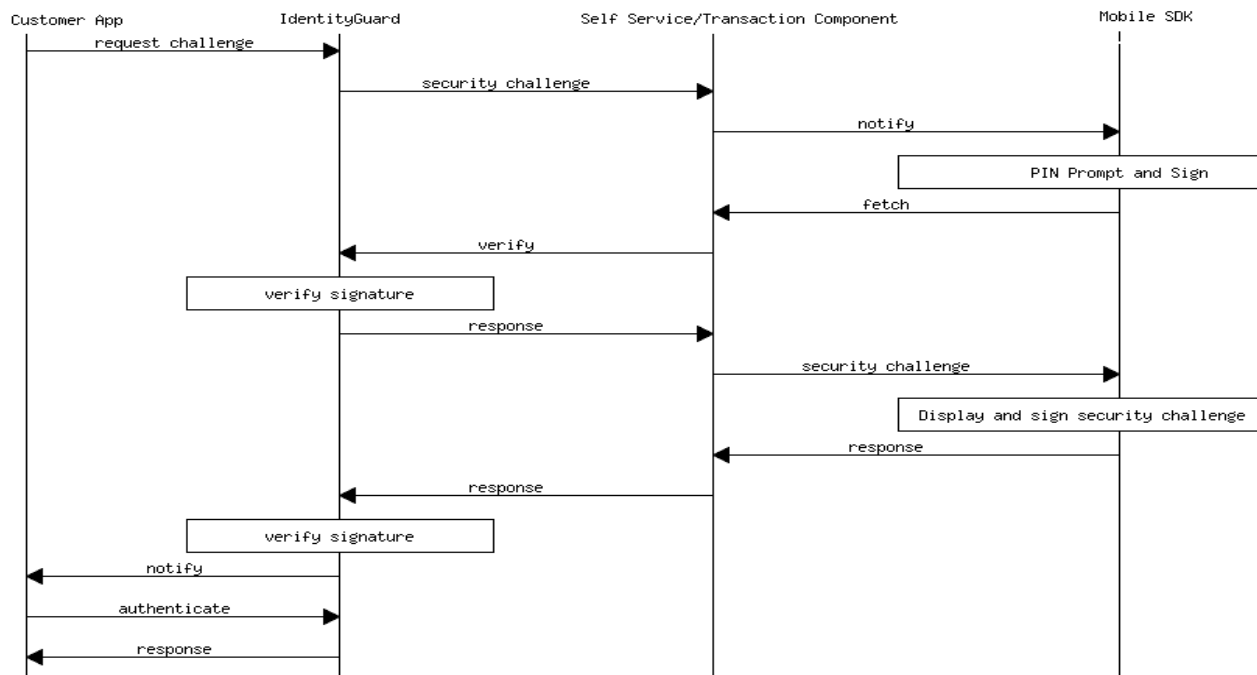
This chapter describes how the Entrust IdentityGuard Authentication and Administration APIs can be used to integrate this Entrust IdentityGuard Mobile Smart Credential functionality with existing applications.

Security challenges

Entrust IdentityGuard (release 10.2 and later) supports smart credential security challenges that work in conjunction with the Entrust Mobile Smart Credential app and customer applications that include the Mobile Smart Credential SDK. A security challenge works as follows:

- 1** A customer application submits a security challenge to Entrust IdentityGuard using the Entrust IdentityGuard APIs.
- 2** Entrust IdentityGuard forwards the security challenge to Entrust IdentityGuard Self-Service.
- 3** Optionally, a notification is sent to the user's mobile application to indicate that a security challenge is available.
- 4** The Entrust IdentityGuard Mobile Smart Credential app polls for security challenges. If a security challenge is available, the app fetches the security challenge. If required, the mobile application prompts the user for the smart credential PIN to authenticate that they can access the smart credential. The smart credential is used to authenticate the mobile application to Entrust IdentityGuard to prove that they have access to download the security challenge.
- 5** The mobile smart credential application displays the security challenge allowing the end user to respond with Confirm, Concern or Cancel. Confirm means the user agrees with the details presented in the challenge. Concern means the user sees something wrong with the details presented in the challenge. Cancel means the user no longer wishes to perform the action described in the challenge.
- 6** The end user's response (signed using one of the smart credential's private keys) is submitted to Entrust IdentityGuard Self-Service, and then to Entrust IdentityGuard.
- 7** The Customer application receives a signal that a response is ready.
- 8** The Customer application calls Entrust IdentityGuard using the Entrust IdentityGuard APIs to get the response.

The following sequence diagram provides details about these interactions.



Entrust IdentityGuard supports three kinds of smart credential security challenges: authentication, transaction verification, and digital signing.

Authentication

An authentication security challenge is a challenge that indicates that the end user has requested authentication. The mobile application displays the challenge indicating that a user has asked for authentication and returns a signed response.

Transaction verification

A transaction verification security challenge is similar to an authentication security challenge except that the challenge also includes transaction details. The transaction details are displayed to the end user by the mobile application and included in the signed response.

Digital signing

A signing security challenge works as follows:

- 1** The customer application calculates the hash on the object they wish to sign.
- 2** The customer application submits a signing security challenge that includes the calculated hash value and a description of what is to be signed.
- 3** The mobile application displays the security challenge including the description of the data that is to be signed.
- 4** If the user confirms the security challenge, the hash is signed and the signature of the hash is included in the response.
- 5** The response returned to the customer application includes the signature of the hash.

- 6 The customer application can generate a signature object as they wish. For example, they might generate a S/MIME message including the signature of the original object.

Security challenge integration

Within Entrust IdentityGuard, a security challenge is associated with certificate authentication. Both authentication and transaction verification security challenges are performed using the authentication API. Signing security challenges are performed using the administration API.

These security challenges are different from normal IdentityGuard authenticators, because the user responds to the challenges out-of-band from the application integrating the IdentityGuard APIs. Two mechanisms are available to application developers to account for the out-of-band nature of the authentication:

- HTTP callback: the application can register a callback URL that IdentityGuard will do an HTTP GET on when a response is received from the mobile device.
- Polling: the application can periodically poll to see if the authentication is complete.

See the details of these mechanisms in the sections that follow.

Configuration

To support security challenges, Entrust IdentityGuard and Entrust IdentityGuard Self-Service must be configured as follows:

- the Entrust IdentityGuard Self-Service Transaction Component must be enabled and configured. The transaction component is configured in the "Transaction Component Settings" section of the Entrust IdentityGuard Self-Service Configuration application.
- Entrust IdentityGuard configuration must include details of the Entrust IdentityGuard Self-Service Transaction Component. This configuration is accessed through the section "Transaction Component and Callback Settings" in the Entrust IdentityGuard Properties Editor.

The following Entrust IdentityGuard policies control various aspects of security challenge behavior. These policies are defined in the Certificate section of policies:

- challenge lifetime—specifies the lifetime of a security challenge. If the security challenge is not completed within this interval it will fail.
- challenge size—the security challenge sent to the mobile application contains a random NONCE. This policy defines its size.
- hash algorithm—this policy defines the hashing algorithm used when signing a security challenge.
- update lockout for replaced challenge—a user can only have one outstanding security challenge. If a security challenge is requested when another security challenge is pending completion, the original security challenge will be replaced. Depending on the setting of this policy, the certificate challenge lockout count will be updated when this happens.
- Allow identity assured authentication—must be set to true to allow any kind of security challenge
- Allow identity assured transaction verification—must be set to true to allow transaction verification security challenges
- Allow identity assured signing—must be set to true to allow signing security challenges.

- all user certs signed by managed CA—this policy must be set to true to allow certificate challenges using certificates signed by managed CAs including smart credentials. This policy must be set to true to perform security challenges.

By default all of the above policies are set to values that allow security challenges.

Security challenges for authentication and transaction verification

From a customer application perspective, both authentication and transaction security challenges are performed using the Entrust IdentityGuard authentication API. The steps required to perform a security challenge are:

- 1** The application calls `getGenericChallenge` requesting a certificate challenge. The parameters specify that a security challenge should be delivered to self-service. It may also specify a URL that Entrust IdentityGuard will call when a response is available for the security challenge.
- 2** Entrust IdentityGuard returns a certificate challenge to the application. A transaction Id is included in the challenge.
- 3** The application calls `authenticateGenericChallenge` providing the transaction Id. The application may make this call in response to a callback from Entrust IdentityGuard. Alternatively, it may periodically call Entrust IdentityGuard to see if a response is ready.
- 4** Entrust IdentityGuard either returns an error (the response is not ready, the security challenge failed) or returns a response indicating that the security challenge was confirmed.

The `getGenericChallenge` method is called first to initiate a security challenge.

The `GenericChallengeParms` structure defines parameters passed from the application to Entrust IdentityGuard in the `getGenericChallenge` call. Arguments specified to security challenges include:

- `AuthenticationType`—specify `CERTIFICATE` for smart credential security challenges.
- `requireCertificateDelivery`—when set to true only smart credentials that have registered to receive security challenges are considered
- `performCertificateDelivery`—when set to true the security challenge is sent to registered smart credentials
- `deliverySmartCredentials`—if specified, this argument specifies a list of smart credential Ids. Only the listed smart credentials are used. If not specified, all of the user's registered smart credentials are used.
- `smartCredentialDeliveryCallback`—if specified, this argument must specify an HTTP/HTTPS URL. When Entrust IdentityGuard receives a response for the security challenge it will perform an HTTP GET to the URL to notify it that a response is ready. If the URL contains the value `<STATUS>` it is replaced with the status of the security challenge (`CONFIRM`, `CONCERN`, `CANCEL`, `INVALID`). If the URL contains the value `<TRANSACTIONID>` it is replaced with the transaction id of the security challenge. If an HTTPS URL is specified, the SSL certificate for the URL must be loaded into the Entrust IdentityGuard keystore.
- `smartCredentialChallengeSummary`—when the Entrust IdentityGuard Mobile Application displays a security challenge, it includes a summary description. The `smartCredentialChallengeSummary` argument can be used by the customer application to specify this summary. If not specified, the mobile application will generate a default summary description.

- **transactionDetails**—specifies a list of transaction details. If specified, the application is performing a transaction verification security challenge. If not specified the application is performing an authentication security challenge.

A call to `getGenericChallenge` returns a `GenericChallenge`. When a smart credential security challenge has been requested, the `CertificateChallenge` field in the `GenericChallenge` will contain the information related to the security challenge:

- **smartCredentials**—a list of the smart credentials that can be used to answer the challenge. If `deliverySmartCredentials` was specified as true in the call to `getGenericChallenge`, this value will be the list of smart credentials to which the security challenge was sent.
- **transactionId**—the transaction Id of the security challenge
- **createDate**—the time at which the security challenge was created
- **lifetime**—the lifetime of the security challenge

The `authenticateGenericChallenge` method is called first to get the result of a security challenge.

The parameters for a call to `authenticateGenericChallenge` include:

- Response parameter to `authenticateGenericChallenge` normally includes the challenge response. When getting the response for a security challenge, the response should be null. If the response is null, the transaction Id must be specified in `GenericAuthenticateParms`.
- The `GenericAuthenticateParms` structure specifies parameters passed to the `authenticateGenericChallenge` call. Parameters specific to security challenges include:
 - **AuthenticationType**—specify `CERTIFICATE` for smart credential security challenges
 - **transactionId**—specify the value from the `CertificateChallenge` returned from the call to `getGenericChallenge`
 - **transactionDetails**—specify the same value passed to `getGenericChallenge`
 - **cancelTransaction**—an optional Boolean parameter. If specified as true, the current transaction is canceled if a response for the transaction is not available and a `TRANSACTION_CANCEL` error is returned.

The call to `authenticateGenericChallenge` returns an instance of `GenericAuthenticateResponse` structure if the challenge was validated. The parameters in the response that are specified to a security challenge are:

- **smartCredentialInfo**—a structure that specifies information about the smart credential that was used to authenticate the security challenge. This structure includes a PKCS#7 signed XML document that specifies information about the security challenge including the transaction details if they were specified. The XML document is signed by the user's smart credential.

The following error codes in exceptions thrown by a call to `authenticateGenericChallenge` are specific to security challenges:

- **NO_RESPONSE**—if the `cancelTransaction` parameter was not set to true and `Entrust IdentityGuard` does not have a response available for the specified transaction Id. The application should call `Entrust IdentityGuard` later
- **NO_CHALLENGE**—indicates that there is no pending security challenge. Either the application has not called `getGenericChallenge` to start a security challenge, `authenticateGenericChallenge` has already been called to retrieve the response for the security challenge or the security challenge has timed out.
- **TRANSACTION_CONCERN**—the mobile application replied to the security challenge with concern

- TRANSACTION_CANCEL—the mobile application replied to the security challenge with cancel or the cancelTransaction parameter was set to true and a response is not available for the transaction.
- TRANSACTION_INVALID—an error was encountered when validating the security challenge. An example cause may be where the user's smart credential has expired or the certificates may have expired.

The CERTIFICATE lockout count will be used when authenticating smart credential security challenges. A successful response will clear the lockout. Errors will update the lockout. Two exceptions are the NO_RESPONSE and TRANSACTION_CANCEL errors. These errors will not update the lockout.

If an application wishes to get a list of available smart credentials they can do so as follows:

- call getGenericChallenge with requireCertificateDelivery set to true and performCertificateDelivery set to false.
- the application will get a challenge listing all available smart credentials but a security challenge won't be delivered to the mobile applications.
- the application may wish to prompt the end user to select which smart credential they want to use.
- the application can then call getGenericChallenge against with requireCertificateDelivery and performCertificateDelivery both set to true. If the user has selected a specific smart credential to use the application can also set the deliverySmartCredentials parameter.

For the application developer there is a trade-off between using the delivery callback. If a delivery callback is not used, the application can poll Entrust IdentityGuard for a response by repeatedly calling authenticateGenericChallenge. This is simple to implement. However, this approach increases the load on Entrust IdentityGuard. The load can be alleviated by introducing a delay between each request. However, this may result in a delay when the response is actually available. Using the delivery callback addresses these issues at the cost of a more complicated implementation.

Authentication and transaction security challenges are identical except that a transaction security challenge includes transaction details which are parameters passed in both the getGenericChallenge and authenticateGenericChallenge calls. When transaction details are specified:

- they must be identical in both the getGenericChallenge and authenticateGenericChallenge calls.
- the transaction details are included in the signed transaction response

The following Java code sample shows how an application would call Entrust IdentityGuard to perform an authentication or transaction verification security challenge for a given user. This sample blocks until a response is available. It does not use the URL callback capability:

```
/*
 * perform an authentication or transaction verification security challenge for the
 * given user
 * depending on whether the transaction details parameter is null or not.
 */
public void authenticate(AuthenticationServiceBindingStub binding, String userid,
    NameValue[] transactionDetails) throws Exception {
    GenericChallengeParms gParms = new GenericChallengeParms();
```

```

gParms.setAuthenticationType(AuthenticationType.CERTIFICATE);
gParms.setRequireCertificateDelivery(Boolean.TRUE);
gParms.setPerformCertificateDelivery(Boolean.TRUE);
// if the transactionDetails are null then an authentication
// security challenge will be performed. Otherwise, a transaction
// verification security challenge will be performed.
gParms.setTransactionDetails(transactionDetails);

GenericChallenge challenge = binding.getGenericChallenge(new
GetGenericChallengeCallParms(userid, gParms));

GenericAuthenticateParms aParms = new GenericAuthenticateParms();
aParms.setAuthenticationType(AuthenticationType.CERTIFICATE);
aParms.setTransactionId(challenge.getCertificateChallenge().getTransactionId());
aParms.setTransactionDetails(transactionDetails);

// instead of looping forever we could look at the challenge
// lifetime in the returned challenge and quit after it has
// expired.
// Alternatively, Entrust IdentityGuard will return an error when
// the challenge has expired.
while (true) {
    try {
        GenericAuthenticateResponse response =
binding.authenticateGenericChallenge(new AuthenticateGenericChallengeCallParms(userid,
new Response(null, null, null), aParms));

        // we were successful. We could look at the signed security
        // challenge but for this sample return to the caller
    } catch (AuthenticationFault e) {
        // continue if we received NO_RESPONSE.
        // Otherwise return the error to the caller
        if (e.getErrorCode().equals(ErrorCode.NO_RESPONSE)) {
            // wait a few seconds before we try again
            Thread.sleep(2000);
            continue;
        }
        throw e;
    }
}
}

```

Security challenges for digital signing

A signing security challenge works as follows:

- 1** The customer application calculates the digest on the object it to be signed.
- 2** The customer application sends the digest value and a summary description of the data being signed to Entrust IdentityGuard.
- 3** Entrust IdentityGuard forwards the digest value in a signing security challenge to the Entrust Mobile application through Entrust Self-Service
- 4** The mobile application displays the signing security challenge, including the summary description, and asks the user to confirm it
- 5** If the user confirms the security challenge, the digest value is signed and a signed security challenge response (including the signed digest) is returned to Entrust IdentityGuard through Entrust IdentityGuard Self-Service.
- 6** If the user responds to the security challenge with Concern or Cancel, the digest value is not signed. A signed security challenge response (without the signed digest) is returned to Entrust IdentityGuard through Entrust IdentityGuard Self-Service,
- 7** The customer application receives the security challenge response, which may or may not include a signed digest.
- 8** The customer application uses the signed digest to generate a full signature of the object being signed.

When a user receives a signing security challenge request in their mobile application, the security challenge can include a summary description of what they are signing. However, there is no way for them to guarantee that the digest value that is actually being signed corresponds to the summary they are viewing or the data they are expecting to sign. It is the responsibility of the customer application to guarantee that the digest value and summary description that it submits to Entrust IdentityGuard correspond to the data the end user is expected to sign.

Entrust IdentityGuard helps to address this security issue by making the Entrust IdentityGuard signature signing capability available only using the Entrust IdentityGuard Administration API. This means that it is not possible for an application that has not authenticated to Entrust IdentityGuard using administration credentials to perform the signing security challenge operations. Furthermore, Entrust IdentityGuard has introduced a new role permission, `userSmartCredentialSign`, that is required to perform the signing operations. This allows customers to restrict the signing capabilities to specific administrators.

The method to submit a signing security challenge to Entrust IdentityGuard is `userSmartCredentialSignStart`. The parameters passed to this method are:

- `userid`—the `userid` that owns the smart credential to be used for signing
- `smartCredentialId`—the id of the smart credential to be used for signing
- `parms`—an instance of `UserSmartCredentialSignParms` specifying the object to be signed. The parameters in this structure include:
 - `digestHashAlg`—the name of the hashing algorithm used to generate the digest. Allowed values are SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512.
 - `digest`—the base-64 encoded digest of the data that the calling application is signing.

- **summary**—a summary description of the data that the calling application is signing. This value will be displayed by the mobile application when it receives the signing security challenge. The summary is required.
- **callback**—an optional HTTP/HTTPS URL. If specified, Entrust IdentityGuard will perform an HTTP GET to the URL when it receives a response to the security challenge. If the specified URL includes the value <STATUS> the value will be replaced with the status of the response (CONFIRM, CONCERN, CANCEL or INVALID). If the specified URL includes the value <TRANSACTIONID> it will be replaced with the transaction Id of the security challenge. If the URL is an HTTPS URL, the corresponding SSL certificate must be loaded into the Entrust IdentityGuard keystore.

UserSmartCredentialSignStartResult. The values in the response include:

- **transactionId**—the transaction Id of the security challenge. This value must be passed to subsequent calls to userSmartCredentialComplete.
- **createDate**—the time the security challenge was created
- **lifetime**—the lifetime of the security challenge. Along with the create date, these values can be used by the application to determine how long to wait for a response.
- **smartCredential**—details about the smart credential. These details are identical to those values returned by a call to userSmartCredentialGet.

The method to receive the result of a signing security challenge from Entrust IdentityGuard is userSmartCredentialSignComplete. The parameters passed to this method are:

- **userId**—the userId that owns the smart credential to be used for signing
- **smartCredentialId**—the id of the smart credential to be used for signing
- **transactionId**—the transaction id returned by a previous call to userSmartCredentialSignStart.
- **cancelTransaction**—an optional Boolean parameter. If specified with a value of true then the current transaction is canceled if a response is not available and a TRANSACTION_CANCEL error is returned.

A call to userSmartCredentialSignComplete returns an instance of UserSmartCredentialSignCompleteResult if the security challenge was confirmed by the mobile application. The values in the response include:

- **signature**—the base-64 encoded signature of the digest generated by the mobile smart credential.
- **certificateChain**—a list of base-64 encoded X509 certificates corresponding to the certificate chain from the root CA to the certificate corresponding to the private key used to generate the signature.
- **smartCredential**—details about the smart credential. These details are identical to those values returned by a call to userSmartCredentialGet.

The application can use the signature and certificateChain values to generate a full signature in whatever format it requires.

The following error codes are specific to security challenge behavior and may be returned in exceptions thrown by the userSmartCredentialSignComplete operation:

- **NO_RESPONSE**—if the cancelTransaction parameter was not specified as true and Entrust IdentityGuard does not have a response available for the specified transaction Id. The application should call Entrust IdentityGuard later.

- TRANSACTION_CONCERN—the mobile application replied to the security challenge with concern.
- TRANSACTION_CANCEL—the mobile application replied to the security challenge with cancel or the cancelTransaction parameter was specified with a value of true.
- TRANSACTION_INVALID—an error was encountered when validating the security challenge. An example cause may be where the user's smart credential has expired or the certificates may have expired.

For the application developer there is a trade-off between responsiveness and load on the system when using the delivery callback. If a delivery callback is not used, the application can poll Entrust IdentityGuard for a response by repeatedly calling `userSmartCredentialSignComplete`. This is simple to implement. However, this approach increases the load on Entrust IdentityGuard. The load can be alleviated by introducing a delay between each request. However, this may result in a delay when the response is actually available. Using the delivery callback addresses these issues at the cost of a more complicated implementation.

The following code sample shows how an application would call Entrust IdentityGuard to perform a signing security challenge for a given user. This sample stops until a response is available. It does not use the URL callback capability.

```
/**
 * sign the given base-64 encoded digest value using the smart credential specified by
 * the given
 *
 * userid and smart credential id. The digestAlg specifies the hashing algorithm used
 * to
 *
 * generate the digest. The summary specifies a description of the data that
 * corresponds to the
 *
 * digest.
 *
 *
 * The returned value includes the signature for the digest as well as the certificate
 * chain of
 *
 * the certificate corresponding to the private key used to sign the digest. These are
 * values
 *
 * that the caller can use to generate a full signature in whatever format the
 * application
 *
 * requires.
 */
public UserSmartCredentialSignCompleteResult sign(AdminServiceBindingStub binding,
String userid, String smartCredentialId, String digest, String digestAlg, String
summary) throws Exception {
    // for this sample, we assume that the caller has generated
    // the digest of data to be signed and a summary description.
    UserSmartCredentialSignParams startParams = new UserSmartCredentialSignParams();
    startParams.setDigest(digest);
    startParams.setDigestHashAlg(digestAlg);
    startParams.setSummary(summary);
}
```

```

        UserSmartCredentialSignStartResult startResult =
binding.userSmartCredentialSignStart(new UserSmartCredentialSignStartCallParms(userid,
smartCredentialId, startParms));

        // instead of looping forever we could look at the challenge
        // lifetime in the returned start result and quit after it has
        // expired.
        // Alternatively, Entrust IdentityGuard will return an error when
        // the challenge has expired.
        while (true) {
            try {
                UserSmartCredentialSignCompleteResult result =
binding.userSmartCredentialSignComplete(new
UserSmartCredentialSignCompleteCallParms(userid, smartCredentialId, startResult
                .getTransactionId(), null));

                // the result includes both a the signature of the digest
                // and the
                return result;
            } catch (AdminServiceFault e) {
                // continue if we received NO_RESPONSE.
                // Otherwise return the error to the caller
                if (e.getErrorCode().equals(ErrorCode.NO_RESPONSE)) {
                    // wait a bit before checking to see if a response is ready
                    Thread.sleep(2000);
                    continue;
                }
                throw e;
            }
        }
    }
}

```

Entrust IdentityGuard does not provide functionality to generate the digest value passed to a signing challenge or to generate a signature object like a PKCS#7 or XML digital signature from the signature information returned for a signing security challenge. The Entrust Security Toolkit for Java can be used for these purposes.

Anonymous smart credential security challenges

Use this approach if you want smart credential authentication to be performed without requiring users to enter their user Id. Users might not know their user Id in the IdentityGuard system, or it might be a lengthy string which is burdensome to enter. In this approach, the system does not know the identity of the user until they have authenticated successfully. Any user can respond to an anonymous smart credential security challenge by signing it with their assigned certificate or smart credential, and the IdentityGuard system determines which user responded, based on the certificate that signed the challenge.

An anonymous smart credential security challenge works as follows:

- 1 The customer application calls Entrust IdentityGuard to get an anonymous challenge.
- 2 The customer application displays a QR code that is contained in the anonymous challenge received from Entrust IdentityGuard. The QR code contains a challenge to be signed and a provider URL where the response should be sent. The provider URL is the transaction component of Entrust IdentityGuard Self-Service.

Note: In addition to the QR code, the security challenge and provider URL are also returned directly in the anonymous smart credential security challenge. The customer application could provide this to client applications that cannot scan QR codes, such as Entrust Intelligence Security Provider for Windows.

- 3 The user scans the QR code using a camera on a mobile device. The Entrust Mobile Smart Credential application displays a summary description of the challenge and asks the user to authenticate. The user may cancel instead of authenticating to the customer application.
- 4 If the user chooses to authenticate, the Entrust Mobile application signs the challenge and sends the response to Entrust IdentityGuard through Entrust IdentityGuard Self-Service.
- 5 The customer application can poll Entrust IdentityGuard to see whether any user has authenticated that challenge. If a user has authenticated the challenge, the polling result identifies authenticated user.
- 6 Entrust IdentityGuard validates the response and finds which user has authenticated by using the certificate in the response.
- 7 Optionally, the customer application might receive a callback from Entrust IdentityGuard, that contains the name of any user that authenticates a challenge successfully. This happens if the customer application set a callback URL when it requested the challenge from Entrust IdentityGuard.

The Java methods required to get an anonymous smart credential security challenge are:

- 1 The application calls `getAnonymousCertificateChallenge` to request the certificate challenge. It may specify a callback URL that Entrust IdentityGuard will call when a valid response was received and a user has successfully authenticated the security challenge.
- 2 Entrust IdentityGuard returns a certificate challenge to the application. A transaction Id is included in the challenge.
- 3 The application may periodically call `getAnonymousCertChallenge` while specifying that transaction Id, to see whether any user has authenticated that challenge. Entrust IdentityGuard either returns a CHALLENGE status, which means no user has authenticated that challenge yet, or an AUTHENTICATED status with the user Id.

- 4 Alternatively, The application may wait for a callback from Entrust IdentityGuard, which indicates that a user has authenticated successfully. The callback provides the user Id and the transaction Id of the challenge that they used for the authentication.

The `getAnonymousCertificateChallenge` method is called first to initiate an anonymous smart credential security challenge.

The `GetAnonymousCertChallengeCallParms` structure defines parameters passed from the application to Entrust IdentityGuard in the `getAnonymousCertificateChallenge` call.

- **Group**—specify the group whose policy will be used to create the anonymous smart credential security challenge. That policy determines the challenge length and the hash algorithm the client must use to sign the challenge. If no group is specified in `GetAnonymousCertChallengeCallParms`, the IG Authentication API creates a challenge with the default policy. Users from other groups can authenticate the anonymous challenge, if the challenge length is adequate for their group policy.

The `GetAnonymousCertChallengeCallParms` structure also contains a `GenericChallengeParms` structure, which includes:

- **authenticationType**—specify `CERTIFICATE` for smart credential security challenges.
- **applicationName**—if specified, a name that will be URL encoded and set into the summary query parameter in the URL
- **smartCredentialChallengeSummary**—if specified, a summary will be URL encoded and set into the summary query parameter in the URL
- **anonymousCertChallengeQRCodeSize**—specify a value between 100 and 1000. The default is 250.
- **setAnonymousCertChallengeCallback**—if specified, this argument must specify an HTTP/HTTPS URL. When Entrust IdentityGuard receives a response for the anonymous challenge and the response is a valid authentication, it will perform an HTTP GET to the URL to notify it that a response is ready.
 - If the URL contains the value `<USERID>` it is replaced with the user Id of the user who authenticated.
 - If the URL contains the value `<TRANSACTIONID>` it is replaced with the transaction id of the security challenge they used to authenticate.
 - If the URL contains the value `<STATUS>` it is replaced with `CONFIRM`, indicating the user has authenticated.
 - If an HTTPS URL is specified, the SSL certificate for the URL must be loaded into the Entrust IdentityGuard keystore.

A call to `getAnonymousCertificateChallenge` returns a `GenericChallenge`. It contains the following information:

- **QRCode**—the QR Code to display to the user who will authenticate. It encodes the `anonymousChallengeURL`.
- **anonymousChallengeURL**—a URL where a Mobile Smart Credential client should respond to authenticate the challenge. The URL scheme is `igmobilesc`. The URL contains the following query parameters: an `apiversion` parameter which is set to 5, an `action` parameter set to `anonchallenge`, a `txid` parameter set to the transaction Id of the challenge, a `provider` parameter set to the Self Service URL where the Mobile Smart Credential client should respond by an HTTP POST, a `hashalg` parameter set to the algorithm it must be use to hash

the challenge before signing, a challenge parameter set to the challenge string, and a summary parameter set to the summary that is displayed to the user.

- group—the group whose policy was used to create the challenge
- transactionId—the transaction Id of the anonymous challenge

The GenericChallenge contains a CertificateChallenge field with the following information related to the anonymous challenge

- challenge—the challenge string to the hashed and signed
- createDate—the time at which the security challenge was created
- lifetime—the lifetime of the security challenge. This lifetime determines how long the challenge will remain in the repository before the Anonymous Smart Credential Expiry process removes it, so that no user can authenticate it. Note that the time during which a user can authenticate the challenge is limited also by their group policy.
- transactionId—the transaction Id of the anonymous challenge, which is the same as the transactionId in the GenericChallenge)
- hashingAlgorithm—the hash algorithm that must be used to authenticate the challenge

After generating the anonymous challenge, the application may poll Entrust IdentityGuard to check whether any user has authenticated it. To poll, call the getAnonymousCertChallenge method while specifying the transaction Id of a challenge.

The GetAnonymousCertChallengeCallParms parameters should contain this:

- transactionId—the transaction Id of the anonymous challenge to check for authentication

The polling call to getAnonymousCertificateChallenge returns a GenericChallenge. It contains all the information returned in the original call that created the challenge. In addition, it contains a ChallengeRequestResult with the authentication status:

- CHALLENGE—indicates that no user has authenticated the challenge successfully.

Note: One of more clients might have tried and failed to authenticate it. By definition, their user Id is not known to IdentityGuard, because they did not provide proof-of-possession of a valid certificate.

- AUTHENTICATED—indicates that a user has authenticated the challenge successfully.

If a user has authenticated, the GenericChallenge also contains this information:

- Username—indicates that a user has authenticated the challenge successfully.
- Group—indicates the group of the user who authenticated.

Error code

The following error code is found in exceptions thrown by a call to getAnonymousCertChallenge to poll for an authentication result:

- INVALID_PARAMETER —if there is no anonymous certificate challenge with the provided transaction ID.

Example

The following Java code shows how an application would call Entrust IdentityGuard to authenticate users by anonymous smart credential security challenges. This sample polls until a user authenticates successfully. It does not use the URL callback capability:

```
/**
 * Authenticate a user by an anonymous smart credential security challenge.
 */
public void authenticateAnonymousCertificate() throws Exception
{
    GenericChallengeParms gcParms = new GenericChallengeParms();
    gcParms.setAuthenticationType(AuthenticationType.CERTIFICATE);

    // IdentityGuard will set these into the URL in the QR code.
    // The Entrust Mobile Smart Credential application displays them.
    gcParms.setApplicationName("AnyBank");
    String smartCredentialChallengeSummary =
        "Please authenticate to access your account on AnyBank";
    gcParms.setSmartCredentialChallengeSummary(smartCredentialChallengeSummary);

    // The default QR size is 250.
    gcParms.setAnonymousCertChallengeQRCodeSize(250);

    // Optional call back that returns the user ID of an authenticated user.
    // This will make fewer calls to the Authentication API than polling.
    String callbackURL = "https://myhost.entrust.com/";
    final String CALLBACK_QUERY =
        "?action=anonchallenge&txnid=<TRANSACTIONID>&status=<STATUS>&userid=<USERID>";
    gcParms.setAnonymousCertChallengeCallback(callbackURL + CALLBACK_QUERY);

    GetAnonymousCertChallengeCallParms gAnonParms =
        new GetAnonymousCertChallengeCallParms();

    // Get anonymous challenge with hash algorithm, lifetime, and
    // challenge length determined by policy for the specified group.
    // Users in other groups with consistent policy can authenticate too.
    // Null will use default group policy.
    gAnonParms.setGroup(null);

    gAnonParms.setParms(gcParms);
    GenericChallenge anonchall =
authBinding.getAnonymousCertChallenge(gAnonParms);

    // Get the QR code to the user to scan
    byte[] qrCode = anonchall.getQRCode();
}
```

```

// Optional. display this link to launch Entrust Entelligence
// Security Provider for Windows, for the user to authenticate with.
String url = anonchall.getAnonymousChallengeURL().replaceFirst(
    "igmobilesc", "igespcapi");

// Call the Auth API with the transaction ID to get the name of the
// user who authenticated.
// Instead of looping forever we could get the challenge create date
// and lifetime and quit after it expires.
Date created = anonchall.getCertificateChallenge().getCreateDate().getTime();
int lifetime = anonchall.getCertificateChallenge().getLifetime();

while (true) {
    String transactionId = anonchall.getTransactionId();
    GetAnonymousCertChallengeCallParms aparms =
        new GetAnonymousCertChallengeCallParms();
    aparms.setTransactionID(transactionId);
    GenericChallenge challenge =
authBinding.getAnonymousCertChallenge(aparms);
    if (ChallengeRequestResult.AUTHENTICATED.equals(
        challenge.getChallengeRequestResult())) {
        System.out.println("authenticated a user " + challenge.getUsername());
        System.out.println("authenticated a user in group " +
            challenge.getGroup());

        break;
    }
    Thread.sleep(5000L);
}
}

```

The `authenticateAnonymousCertChallenge` method is called to authenticate an anonymous smart credential security challenge. Normally this method is called by Entrust IdentityGuard Self-Service when it receives a response from the Entrust Mobile Application. The customer application does not have to use `authenticateAnonymousCertChallenge` because the response is sent to Entrust IdentityGuard Self-Service.

The parameters for a call to `authenticateAnonymousCertChallenge` are set into an `AuthenticateAnonymousCertChallengeCallParms`, which contains response and a `GenericAuthenticateParms`. The following parameters are set.

- Response—this is challenge response that will authenticate and identity the user
- The `GenericAuthenticateParms` structure specifies parameters passed to the `authenticateAnonymousCertChallenge` call. Parameters specific to anonymous smart credential challenges include:
- `AuthenticationType`—specify `CERTIFICATE` for anonymous security challenges

- `transactionId`—specify the value from the `CertificateChallenge` returned from the call to `getGenericChallenge`

The call to `authenticateAnonymousCertChallenge` returns an instance of `GenericAuthenticateResponse` structure if the challenge was validated. The parameters in the response that are specified to anonymous smart credential security challenge are:

- `UserName`—identifies the name of the user who was authenticated by the response
- `Group`—identifies the group of the user who was authenticated by the response
- `FullName`—identifies the full name of the user who was authenticated by the response
- `LastAuth`—the last time this user authenticated and the type of authentication
- `LastFailedAuth`—the last time this user failed authentication and the type of authentication that failed

The `GenericAuthenticateResponse` contains a `CertificateData`—the field with the following information related to the certificate that authenticated the anonymous smart credential security challenge.

- `subjectDN`—the subject distinguished name of the certificate that authenticated the response
- `issuerDN`—the issuer distinguished name of the certificate that authenticated the response
- `serialNumber`—the serial number of the certificate that authenticated the response
- `expiryDate`—the expiry date of the the certificate that authenticated the response
- `issueDate`—the issue date of the certificate that authenticated the response

Chapter 7:

Integrating Entrust mobile soft tokens

The Entrust IdentityGuard Mobile Soft Token app provides features that use and manage IdentityGuard soft tokens that are issued by Entrust IdentityGuard. These soft tokens can be used over-the-air

- to authenticate to Web sites and other applications
- to verify transaction details, sign the details if acceptable, or report possible fraud.

This chapter describes how the Entrust IdentityGuard Authentication and Administration APIs can be used to integrate the Entrust IdentityGuard Mobile Soft Token app functionality with existing applications.

Soft token activation

To activate a mobile soft token, the Entrust IdentityGuard Server generates an activation code. Along with the token serial number, this value is provided to the mobile soft token. The mobile soft token generates a registration code which is provided to the server. The registration code can either be submitted over the network to the server (if the server address is provided with the activation information) or manually entered. These values are used by the mobile soft token and server to generate a common token seed.

The mobile soft token application supports three versions of activation:

- online (quick) activation – An application-specific URL is provided to the mobile application either as a link displayed in the mobile browser or in an email. When this link is selected, the mobile application is launched and communicates with the Entrust IdentityGuard to retrieve the activation information and complete activation
- offline (QR Code) activation – The activation information is encoded in a QR Code that is displayed to the end user. The mobile application is used to scan the QR Code, which then retrieves the activation information from the QR Code. The content of the QR Code is protected with a random password generated by Entrust IdentityGuard. This random password must be entered into the mobile application.
- manual activation – the activation information is displayed to the end user who manually enters it into the mobile application.

Token activation methods

The following methods in the Entrust IdentityGuard Administration API can be used to perform token activation:

`userTokenActivate` – Creates the activation code for a specified token and returns the information required for manual activation.

`userTokenActivateComplete`

This method allows the registration code to be submitted to the Entrust IdentityGuard server to complete activation.

Token activation integration

The following code snippets show how an application can use the APIs to perform soft token activation:

```
UserTokenActivateResult startActivation(AdminServiceBindingStub binding, String
userid, String serialNumber) throws Exception {
    UserTokenFilter filter = new UserTokenFilter();
    filter.setSerialNumber(serialNumber);

    UserTokenParms parms = new UserTokenParms();
    parms.setActivationDelivery("tokenemaildelivery"); // this delivery has
    // to be defined in identityguard.properties
    parms.setActivationContact("work email"); // the user has to have a
    // contact of this type
    parms.setActivationAddress("ssm.yourcorp.com:8445/igst"); // this is the
    // address of the IG SSM server
    parms.setGenerateQRCode(Boolean.TRUE);

    UserTokenActivateResult result = binding.userTokenActivate(new
    UserTokenActivateCallParms(userid, filter, parms));

    // result contains the information the application needs to provide to the //soft
    token for
    // activation to proceed
    return result;
}

// check if the specified token is activated. This will be the case if the
//registration code
// was submitted from the mobile application automatically.
boolean isActivationComplete(AdminServiceBindingStub binding, String userid, String
serialNumber) throws Exception {
    UserTokenFilter filter = new UserTokenFilter();
    filter.setSerialNumber(serialNumber);

    UserTokenInfo[] tokens = binding.userTokenGet(new UserTokenGetCallParms(userid,
    filter));

    if (tokens.length != 1)
        throw new Exception("Unexpected number of tokens returned.");
}
```

```

        return tokens[0].getActivationState().equals("ACTIVATED");
    }

    void completeActivation(AdminServiceBindingStub binding, String userid, String
serialNumber, String registrationCode) throws Exception {
        UserTokenFilter filter = new UserTokenFilter();
        filter.setSerialNumber(serialNumber);

        NameValue regCode = new NameValue("registrationCode", registrationCode);

        binding.userTokenActivateComplete(new UserTokenActivateCompleteCallParms(userid,
filter, new NameValue[] { regCode }));
    }

```

Soft token transaction verification

In previous releases users had to manually enter security codes to verify transactions. With Entrust IdentityGuard 10.2 FP1 or later, transactions can be verified by selecting a button in the Entrust IdentityGuard Mobile Soft Token app.

With Entrust IdentityGuard Server 10.2 FP1 Patch 194310 or later and version 3.0 or later of the Entrust IdentityGuard Mobile Soft Token app, users can also perform transaction verification when their mobile device has no Internet connection (offline). This is achieved by use of the QR (Quick Response) Code feature implemented in Entrust IdentityGuard Server 10.2 FP1 Patch 194310 or later.

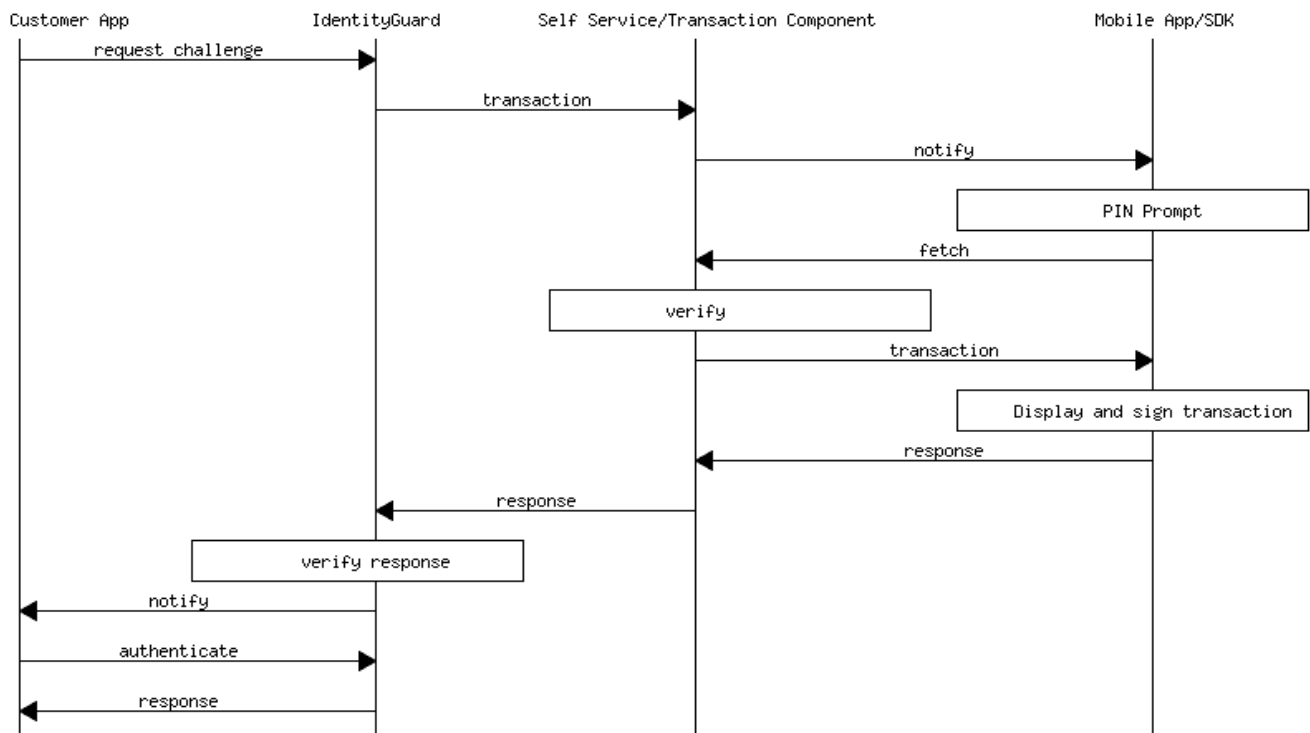
Online transaction verification

Entrust IdentityGuard release 10.2FP1 introduced support for online authentication and transaction verification using the Entrust IdentityGuard Mobile Soft Token app. This means that, unlike transaction verification in earlier releases, users select a button in the app to respond to a transaction notification. They no longer need to manually enter a security code displayed by the app. Online transaction verification works in conjunction with the Entrust Mobile Soft Token app and customer applications that include the Mobile Soft Token SDK. A transaction verification request works as follows:

- 1** A customer application submits a transaction request to Entrust IdentityGuard using the Entrust IdentityGuard APIs.
- 2** Entrust IdentityGuard forwards the request to Entrust IdentityGuard Self-Service.
- 3** Optionally, a notification is sent to the user's Mobile Soft Token app to indicate that a transaction is available.
- 4** The Entrust IdentityGuard Mobile Soft Token app polls for notifications. If a notification is available, the app fetches the transaction. If required, the app prompts the user for the PIN to authenticate that they can access the soft token. The soft token is used to authenticate the app to Entrust IdentityGuard to prove that they have access to download the transaction details.

- 5 The Mobile Soft Token app displays the transaction details and prompts the end user to respond with Confirm, Concern or Cancel. Confirm means the user agrees with the details presented in the transaction. Concern means the user sees something wrong with the details presented in the transaction. Cancel means the user no longer wishes to perform the action described in the transaction notification.
- 6 The token seed is used to sign the transaction details. The OTP security code is the end user's signature. It is submitted to Entrust IdentityGuard Self-Service and then to Entrust IdentityGuard.
- 7 The Customer application receives a signal that a response is ready.
- 8 The Customer application calls Entrust IdentityGuard using the Entrust IdentityGuard APIs to get the response.

The following sequence diagram provides details about these interactions.



Entrust IdentityGuard supports two kinds of transaction notifications:

Authentication

An authentication notification indicates that the end user has requested authentication. The app displays the notification indicating that a user has asked for authentication and returns a authenticated response.

Transaction verification

A transaction verification notification is similar to an authentication notification except that it also includes transaction details. The transaction details are displayed to the end user by the mobile app and included in the authenticated response when the user selects **Confirm**, **Cancel**, or **Concern**.

Online transaction verification integration

Both authentication and transaction verification are performed using the authentication API.

Two mechanisms are available to application developers to account for the out-of-band nature of the authentication:

- HTTP callback: the application can register a callback URL that IdentityGuard will do an HTTP GET on when a response is received from the mobile device.
- Polling: the application can periodically poll to see if the authentication is complete.

See the details of these mechanisms in the sections that follow.

Configuration

To support transaction notifications, Entrust IdentityGuard and Entrust IdentityGuard Self-Service must be configured as follows:

- The Entrust IdentityGuard Self-Service Transaction Component must be enabled and configured. The transaction component is configured in the "Transaction Component Settings" section of the Entrust IdentityGuard Self-Service Configuration application.
- Entrust IdentityGuard configuration must include details of the Entrust IdentityGuard Self-Service Transaction Component. This configuration is accessed through the section "Transaction Component and Callback Settings" in the Entrust IdentityGuard Properties Editor.

The following Entrust IdentityGuard policies control various aspects of transaction verification behavior. These policies are defined in the Token section of policies:

- Token challenge lifetime - specifies the lifetime of a transaction notification. If the transaction is not completed within this interval, it will fail.
- Token transaction lifetime - specifies the amount of time, in seconds, that a transaction is available for a response (Confirm, Cancel, or Concern) from the user. Perform transaction delivery to soft tokens - specifies whether transaction details are sent to the Entrust IdentityGuard Mobile Soft Token app. By default all of the above policies are set to values that allow online transaction verification.

Online authentication and transaction verification

From a customer application perspective, both authentication and transaction verification are performed using the Entrust IdentityGuard authentication API. The steps required to perform online transaction verification are:

- 1 The application calls `getGenericChallenge` requesting a token challenge. The parameters specify that a transaction verification challenge should be delivered to self-service. It may also specify a URL that Entrust IdentityGuard will call when a response is available for the transaction verification challenge.
- 2 Entrust IdentityGuard returns a token challenge to the application. A transaction Id is included in the challenge.
- 3 The application calls `authenticateGenericChallenge` providing the transaction Id. The application may make this call in response to a callback from Entrust IdentityGuard. Alternatively, it may periodically call Entrust IdentityGuard to see if a response is ready.

- 4** Entrust IdentityGuard either returns an error (the response is not ready, the transaction verification challenge failed) or returns a response indicating that the transaction verification challenge was confirmed.

The `getGenericChallenge` method is called first to initiate a transaction verification challenge.

The `GenericChallengeParms` structure defines parameters passed from the application to Entrust IdentityGuard in the `getGenericChallenge` call. Arguments specified to transaction verification challenges include:

- `AuthenticationType` - specify `TOKENRO` for token challenges.
- `setRequireDeliveryAndSignatureIfAvailable` - when set to true, only tokens that have registered for transaction verification challenges are considered.
- `setPerformDeliveryAndSignature` - when set to true the transaction notification is sent to registered tokens.
- `deliveryTokens` - if specified, this argument specifies a list of token Ids. Only the listed tokens are used. If not specified, all of the user's registered tokens are used.
- `tokenDeliveryCallback` - if specified, this argument must specify an HTTP/HTTPS URL. When Entrust IdentityGuard receives a response for the transaction verification challenge it will perform an HTTP GET to the URL to notify it that a response is ready. If the URL contains the value `<STATUS>` it is replaced with the status of the transaction verification challenge (`CONFIRM`, `CONCERN`, `CANCEL`, `INVALID`). If the URL contains the value `<TRANSACTIONID>`, it is replaced with the transaction id of the transaction verification challenge. If an HTTPS URL is specified, the SSL certificate for the URL must be loaded into the Entrust IdentityGuard keystore.
- `tokenChallengeSummary` - when the Entrust IdentityGuard Mobile Application displays a transaction verification challenge, it includes a summary description. The `tokenChallengeSummary` argument can be used by the customer application to specify this summary. If not specified, the mobile application will generate a default summary description.
- `transactionDetails` - specifies a list of transaction details. If specified, the application is performing a transaction verification challenge. If not specified the application is performing an authentication transaction.

Note: When transaction queueing is enabled, the transaction details can be used optionally to set a transaction priority. The Mobile Soft Token client will display that priority in its list of queued transactions. To set the priority, add a transaction detail whose name is `ENT_IDG_TRANSACTION_QUEUE_PRIORITY` and give it a value of 1 through 9. For more information about queueing multiple transactions, see "Enable transaction queueing" in the *Entrust IdentityGuard Server Administration Guide*.

A call to `getGenericChallenge` returns a `GenericChallenge`. When a token transaction verification challenge has been requested, the `TokenChallenge` field in the `GenericChallenge` will contain the information related to the transaction:

- `transactionId` - the transaction Id of the transaction
- `createDate` - the time at which the transaction was created
- `lifetime` - the lifetime of the transaction

The `authenticateGenericChallenge` method is called first to get the result of a token challenge.

The parameters for a call to `authenticateGenericChallenge` include:

- The Response parameter to `authenticateGenericChallenge` normally includes the challenge response. When getting the response for a token transaction, the response should be null. If the response is null, the transaction Id must be specified in `GenericAuthenticateParms`.
- The `GenericAuthenticateParms` structure specifies parameters passed to the `authenticateGenericChallenge` call. Parameters specific to token challenges include:
 - `AuthenticationType` - specify `TOKENRO` for token transactions
 - `transactionId` - specify the value from the `TokenChallenge` returned from the call to `getGenericChallenge`
 - `transactionDetails` - specify the same value passed to `getGenericChallenge`
 - `cancelTransaction` - an optional Boolean parameter. If specified as true, the current transaction is canceled if a response for the transaction is not available and a `TRANSACTION_CANCEL` error is returned.

The following error codes in exceptions thrown by a call to `authenticateGenericChallenge` are specific to token challenges:

- `NO_RESPONSE` - if the `cancelTransaction` parameter was not set to true and `Entrust IdentityGuard` does not have a response available for the specified transaction Id. The application should call `Entrust IdentityGuard` later
- `USER_NO_CHALLENGE` - indicates that there is no pending transaction. Either the application has not called `getGenericChallenge` to start a transaction, `authenticateGenericChallenge` has already been called to retrieve the response for the transaction or the transaction has timed out.
- `TRANSACTION_CONCERN` - the mobile application replied to the transaction with concern
- `TRANSACTION_CANCEL` - the mobile application replied to the transaction with cancel or the `cancelTransaction` parameter was set to true and a response is not available for the transaction.
- `TRANSACTION_INVALID` - an error was encountered when validating the transaction.

The `TOKENRO` logout count will be used when authenticating token transaction. A successful response will clear the logout. Errors will update the logout. Two exceptions are the `NO_RESPONSE` and `TRANSACTION_CANCEL` errors. These errors will not update the logout.

For the application developer there is a trade-off between using the delivery callback. If a delivery callback is not used, the application can poll `Entrust IdentityGuard` for a response by repeatedly calling `authenticateGenericChallenge`. This is simple to implement. However, this approach increases the load on `Entrust IdentityGuard`. The load can be alleviated by introducing a delay between each request. However, this may result in a delay when the response is actually available. Using the delivery callback addresses these issues at the cost of a more complicated implementation.

Authentication and transaction verification are identical except that a transaction verification includes transaction details which are parameters passed in both the `getGenericChallenge` and `authenticateGenericChallenge` calls. When transaction details are specified:

- they must be identical in both the `getGenericChallenge` and `authenticateGenericChallenge` calls.
- the transaction details are included in the signed transaction response

The following Java code sample shows how an application would call `Entrust IdentityGuard` to perform an online authentication or transaction verification for a given user. This sample stops until a response is available. It does not use the URL callback capability:

```

/*
 * perform an authentication or transaction verification online token challenge for
the given
 * user depending on whether the transaction details parameter is null or not.
 */
public void authenticate(AuthenticationServiceBindingStub binding, String userid,
NameValue[] transactionDetails, String summary) throws Exception {
    GenericChallengeParms gParms = new GenericChallengeParms();
    gParms.setAuthenticationType(AuthenticationType.TOKENRO);
    gParms.setPerformDeliveryAndSignature(Boolean.TRUE);
    gParms.setRequireDeliveryAndSignatureIfAvailable(Boolean.TRUE);
    gParms.setTokenTransactionMode(TokenTransactionMode.ONLINE);
    gParms.setTokenChallengeSummary(summary);

    // if the transactionDetails are null then an authentication
    // transaction verification challenge will be performed.
    // Otherwise, a transaction verification challenge will be performed.
    gParms.setTransactionDetails(transactionDetails);

    GenericChallenge challenge = binding.getGenericChallenge(new
GetGenericChallengeCallParms(userid, gParms));

    GenericAuthenticateParms aParms = new GenericAuthenticateParms();
    aParms.setAuthenticationType(AuthenticationType.TOKENRO);
    aParms.setTransactionId(challenge.getTokenChallenge().getTransactionId());
    aParms.setTransactionDetails(transactionDetails);

    // instead of looping forever we could look at the challenge
    // lifetime in the returned challenge and quit after it has
    // expired.
    // Alternatively, Entrust IdentityGuard will return an error when
    // the challenge has expired.
    while (true) {
        try {
            GenericAuthenticateResponse response =
binding.authenticateGenericChallenge(new AuthenticateGenericChallengeCallParms(userid,
new Response(null, null, null), aParms));
        } catch (AuthenticationFault e) {
            // continue if we received NO_RESPONSE.
            // Otherwise return the error to the caller
            if (e.getErrorCode().equals(ErrorCode.NO_RESPONSE)) {
                // wait a few seconds before we try again
                Thread.sleep(2000);
            }
        }
    }
}

```

```

        continue;
    }
    throw e;
}
}
}

```

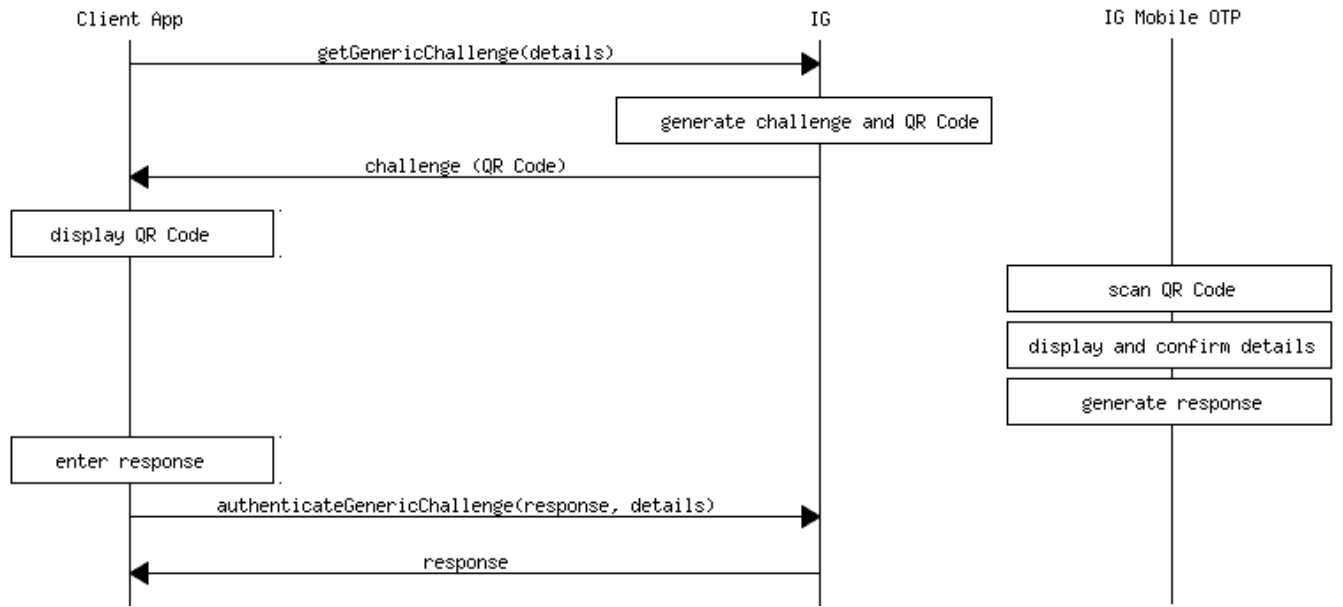
Offline transaction verification

Offline transaction verification is a variation of soft token transaction verification where the transaction details are communicated to the mobile soft token application using a QR Code. This allows a customer to use transaction verification where the mobile soft token application does not have network connectivity.

An offline transaction verification works as follows:

- 1** A customer application submits a transaction request to Entrust IdentityGuard using the Entrust IdentityGuard APIs.
- 2** Entrust IdentityGuard returns a QR Code that encodes the transaction details.
- 3** The customer application displays the QR Code and prompts for an OTP security code response for the transaction.
- 4** The end user uses the Mobile Soft Token application to scan the QR Code
- 5** The Mobile Soft Token application displays the transaction details and prompts the end user to respond with **Confirm**, **Concern** or **Cancel**. If the user selects **Confirm**, the mobile application displays an OTP security code for the transaction.
- 6** The end user enters the OTP security code into the customer application.
- 7** The customer application submits the OTP security code to Entrust IdentityGuard
- 8** Entrust IdentityGuard confirms the OTP security code.

The following sequence diagram shows the events in offline transaction verification.



Offline transaction verification integration

From a customer application perspective, offline transaction verification is performed using the Entrust IdentityGuard authentication API. The steps required are:

- 1** The application calls `getGenericChallenge` requesting a token challenge. The request will include transaction details and an optional transaction summary. The request must specify that an offline transaction is required.
- 2** Entrust IdentityGuard returns a token challenge. For each token in the challenge, a QR Code will be returned.
- 3** The application calls `authenticateGenericChallenge` or `authenticateGenericChallenge` to authenticate the transaction. The request will include the transaction details and the OTP security code provided by the mobile soft token application.

The parameters passed to `getGenericChallenge` in the `GenericChallengeParms` structure includes the following arguments:

- `authenticationType` – specify `TOKENRO` for token challenges
- `tokenTransactionMode` – specify `TokenTransactionMode.OFFLINE`
- `tokenChallengeSummary` – an optional parameter specifying a summary of the transaction
- `transactionDetails` – specifies a list of transaction details. For offline transaction verification, this argument is required.
- `tokenTransactionReturnQRCode` – this argument indicates that a QR Code should be returned when `tokenTransactionMode` is set to `TokenTransactionMode.CLASSIC`. This allows the application to perform a classic transaction (where the transaction is delivered to the mobile application over the network) and also display the QR Code in case the transaction delivery fails.
- `tokenTransactionQRCodeSize` – this argument allows the application to specify the size of the QR Code in pixels. The default is 250. If specified, the value must be between 100 and 1000 pixels.
- `tokenTransactionAppScheme` – the URL scheme of the application specific URL in which the transaction information is encoded. If not specified, the value used during token activation is used and if that is not available the default value which matches the Entrust IdentityGuard Mobile application is used.

For each token, the offline transaction challenge is an application specific URL including the transaction details and transaction challenge summary. This information is encrypted and MACed using keys derived from the token seed. This means that the corresponding token is the only entity that can access the transaction information. The application specific URL is encoded as a QR Code.

The token challenge included in the response from `getGenericChallenge` includes a list of `TokenData` values for each token. The QR Code related values in `TokenData` include:

- `offlineChallenge` – the application specific URL that contains the transaction information encoded in the QR Code.
- `offlineChallengeQRCode` – the binary JPEG encoding of the QR Code

The customer's application will display the `offlineChallengeQRCode` value so that it can be scanned by the mobile application. However, if the application displaying the QR Code is running on the user's mobile device, this won't be possible. In that case, the application can make the

offlineChallenge an HTML link. Clicking on this link should launch the mobile application and pass the transaction information to it.

```
// this code snippet shows the authentication API code needed to request
// a QR challenge

//
// return an offline QR Code transaction challenge for each token of the
// specified user
//
public TokenData[] getQRCodeChallenge(AuthenticationServiceBindingStub binding, String
userid, NameValue[] transactionDetails, String transactionSummary) throws Exception {
    GenericChallengeParms gParms = new GenericChallengeParms();
    gParms.setAuthenticationType(AuthenticationType.TOKENRO);
    gParms.setTokenTransactionMode(TokenTransactionMode.OFFLINE);
    gParms.setTokenChallengeSummary(transactionSummary);
    gParms.setTransactionDetails(transactionDetails);

    GenericChallenge challenge = binding.getGenericChallenge(new
GetGenericChallengeCallParms(userid, gParms));

    if ((challenge.getTokenChallenge().getTokens() == null) ||
(challenge.getTokenChallenge().getTokens().length == 0)) {
        throw new Exception("No tokens.");
    }

    // each token returned will include the QR Code information which must be
    // displayed by the application
    return challenge.getTokenChallenge().getTokens();
}

//
// authenticate the transaction verification response for the given user
//
public void authenticateQRCodeChallenge(AuthenticationServiceBindingStub binding,
String userid, NameValue[] transactionDetails, String[] response) throws Exception {
    GenericAuthenticateParms aParms = new GenericAuthenticateParms();
    aParms.setAuthenticationType(AuthenticationType.TOKENRO);
    aParms.setTransactionDetails(transactionDetails);

    binding.authenticateGenericChallenge(new
AuthenticateGenericChallengeCallParms(userid, new Response(null, response, null),
aParms));
}
```

Chapter 8:

API exceptions

This chapter explains how to use error and fault classes included with the Entrust IdentityGuard APIs.

System and service faults

System and service faults provide helpful feedback from the Entrust IdentityGuard system. The `AuthenticationSystemFault` or the `AuthenticationServiceFault` classes in the Authentication API, and the `AdminServiceFault` class in the Administration API, provide the access methods listed in the following table to use this feedback.

Access methods

Method name	Returns	Description
<code>getErrorMessage</code>	String	Returns the error message supplied by the Entrust IdentityGuard Authentication or Administration service describing the failure.
<code>getMessage</code>	String	overrides the method defined in the Exception class and returns the same information as the <code>getErrorMessage</code> method.
<code>getErrorCode</code>	ErrorCode	Returns an <code>ErrorCode</code> object that can be used for error handling. This is the preferred method for detecting and handling error situations. Use the <code>getInternalCode</code> method only when <code>ErrorCode</code> does not provide enough information to identify and resolve the error condition.
<code>getInternalCode</code>	String	Returns the Entrust IdentityGuard internal code. This code corresponds to the codes documented in the error messages file and the values logged by the Entrust IdentityGuard server.
<code>getId</code>	String	Returns an ID that can be used to identify the error in the Entrust IdentityGuard server log files. It is an instance-specific value. This ID is useful in helping an administrator identify an exact error in a large log file.

Error output sample

In a Web application, you can use these methods to output information directly to the screen using a method similar to the following examples. It is up to the client developers to decide whether the errors being returned to the client application can be considered useful to the end-user.

Error output sample for Authentication API

```
try {
    an_API_Service_That_Fails.that_fails();
} catch (AuthenticationServiceFault asf) {
    out.write("<p><b>Error message:</b>");
    out.write(asf.getMessage());
    out.write("<p><b>Error Code:</b>");
    out.write(asf.getErrorCode().toString());
    out.write("<p><b>Reference #:</b>");
    out.write(asf.getID());
}
```

Error output sample for Administration API

```
try {
    an_API_Service_That_Fails.that_fails();
} catch (AdminstrationServiceFault asf) {
    out.write("<p><b>Error message:</b>");
    out.write(asf.getMessage());
    out.write("<p><b>Error Code:</b>");
    out.write(asf.getErrorCode().toString());
    out.write("<p><b>Reference #:</b>");
    out.write(asf.getID());
}
```

ErrorCode class

ErrorCode is a container class for an enumerated set of error codes that enable the client to perform cleaner error checking. Many internal error codes may result in a more general external error code that the client can use for general error detection. See the Javadocs included with the release for a description of the errors.

Error checking using the ErrorCode is the recommended method of detecting errors. Rather than doing direct string comparisons of error codes, which results in application source code that is difficult to maintain, you can perform comparisons using the ErrorCode enumeration. See the following example code.

Error checking sample for Authentication API

```

try {
    AuthenticateResponse ar = authenticate(userid, response);
} catch (AuthenticationFault af) {
    if (af.getErrorCode().equals(ErrorCode.INVALID_RESPONSE)) {
        // Do something
    } else if (af.getErrorCode().equals(ErrorCode.NO_ACTIVE_CARDS)) {
        // Do something else
    } else {
        // Do something more
    }
}

```

Authentication faults

Authentication Web service failures are either “business rule” failures or system failures. In the case of business rule failures, error conditions are returned to the client application in the form of `AuthenticationServiceFaults`. The user is told that an error has occurred and that they should try again. For error details, see “`AuthenticationServiceFault` class”.

`AuthenticationSystemFaults` are conditions that occur outside the normal scope of service operation. They require administrator intervention. For error details, see “`AuthenticationSystemFault` class”.

For the complete list of error messages and explanations, see the *Entrust IdentityGuard Error Messages*.

AuthenticationFault base class

`AuthenticationServiceFault`, `AuthenticationSystemFault`, and the `warningFault` contained in the `AuthenticateResponse` all have a base class of `AuthenticationFault`. Although the code samples illustrate catching service and system faults separately, both types of fault are caught using only the base class. See “System and service faults” on page [157](#) for a list of the access methods common to all fault classes.

For more information about `warningFault`, see the section “Authentication warning faults” on page [161](#).

AuthenticationServiceFault class

You should expect service faults to occur as a standard part of the day-to-day operation of Entrust IdentityGuard. In general, authentication service faults are conditions that are corrected by providing alternate input to the Web service or by having an administrator resolve a data issue.

For the complete list of error codes, see the *Entrust IdentityGuard Error Messages*.

AuthenticationSystemFault class

As a general rule, `AuthenticationSystemFaults` are not a part of a normal functioning system. They indicate that a serious system failure of some type has occurred, for example, that Entrust IdentityGuard is not initialized, or that a communication error has occurred that may require administrator intervention to resolve.

For the complete list of error codes, see the *Entrust IdentityGuard Error Messages*.

The following example illustrates how to use the `AuthenticationServiceFault` and `AuthenticationSystemFault` classes to catch errors in an `authenticateAnonymousChallenge` operation:

```
// The following assumes the challengeSet was stored from a previous
// getAnonymousChallenge request. The userId, and challengeResponse values are
// expected to be returned as form input elements.
try {
    String userId = request.getParameter("userId");
    // Should check to make sure that userId is not null.
    String[] challengeResponse = request.getParameterValues("challengeResponse");
    // Similarly, should make sure that challengeResponse is not null!
    GenericAuthenticateParms parms = new GenericAuthenticateParms();
    Response response = new Response();
    response.setResponse(challengeResponse);
    AuthenticateAnonymousChallengeCallParms authenticateAnonymousChallengeCallParms =
new AuthenticateAnonymousChallengeCallParms();
    authenticateAnonymousChallengeCallParms.setUserId(userId);

    authenticateAnonymousChallengeCallParms.setChallengeSet(cachedAnonymousChallengeSet);
    authenticateAnonymousChallengeCallParms.setResponse(response);
    GenericAuthenticateResponse resp =
authBinding.authenticateAnonymousChallenge(authenticateAnonymousChallengeCallParms);
    // If no exceptions are thrown, then authentication was successful.
    System.out.println("Authenticated!!!");
} catch (AuthenticationServiceFault e) {
    // Authentication failure occurred, so catch and display the error
    System.err.println(e.getMessage());
} catch (AuthenticationSystemFault se) {
    // Handle a system failure gracefully, possibly redirecting
    // to an error page.
    System.err.println(se.getMessage());
}
```

Authentication warning faults

Warning faults are `AuthenticationFault` objects returned as part of an `AuthenticateResponse`. Your application can retrieve the fault using `getWarningFault`. Only authentication problems with shared secrets generate warning faults.

A warning fault does not stop processing or cause an authentication to fail, because the client application has supplied a correct response to a challenge. Nevertheless, the error is reported to the client application and is logged by the system to further diagnose the situation, if necessary.

All exceptions relating to authentication secret and shared secret functionality are returned as warning faults. The client application must be aware that a failure occurred while setting secret information, but the failure most likely would not result in an authentication failure. Only failures that occur after a successful authentication are returned as warning faults. Any failure that occurs before authentication results in an immediate authentication failure being returned to the client.

Authentication operation exceptions

The following table explains faults returned as part of specific authentication operations.

In addition, the error `java.rmi.RemoteException` is thrown if there is an error while performing the RMI communication with the application server. This error relates to a communication error, or possibly indicates that the target server does not have the Entrust IdentityGuard Authentication service installed.

Operations and exceptions

Operation	Expected result	Exceptions
<code>authenticateAnonymousChallenge</code>	Returns the <code>GenericAuthenticateResponse</code> object.	<code>AuthenticationServiceFault</code> is thrown if a challenge response fails during the processing of various failure conditions. <code>AuthenticationSystemFault</code> is thrown by any other failure during the processing of this operation.
<code>authenticateGenericChallenge</code>	Returns the <code>GenericAuthenticateResponse</code> object.	<code>AuthenticationServiceFault</code> is thrown if a challenge response fails during the processing of various failure conditions. <code>AuthenticationSystemFault</code> is thrown by any other failure during the processing of this operation.
<code>getAllowedAuthenticationTypes</code>	Returns the <code>AllowedAuthenticationTypes</code> object for the user for generic authentication and machine registration.	<code>AuthenticationServiceFault</code> is thrown if the user is empty or null, or does not exist. <code>AuthenticationSystemFault</code> is thrown by any failure other than an empty or null, or non-existent user or group ID.

Operation	Expected result	Exceptions
getAllowedAuthenticationTypesForGroup	Returns the AllowedAuthenticationTypes object for the group for generic authentication and machine registration.	AuthenticationServiceFault is thrown if the group ID is empty or null, or does not exist. AuthenticationSystemFault is thrown by any failure other than an empty or null, or non-existent user or group ID.
getAnonymousChallenge	Returns the GenericChallenge object, which contains the Entrust IdentityGuard grid challenge, and information pertaining to the temporary PIN and PVN specifications.	AuthenticationServiceFault is thrown if anonymous authentication is disabled. AuthenticationSystemFault is thrown by any other failure during the processing of this operation.
getAnonymousChallengeForGroup	Returns the GenericChallenge object, which contains the Entrust IdentityGuard grid challenge, and information pertaining to the temporary PIN and PVN specifications.	AuthenticationServiceFault is thrown if anonymous authentication is disabled. AuthenticationSystemFault is thrown by any other failure during the processing of this operation.
getGenericChallenge	Returns the GenericChallenge object.	AuthenticationServiceFault is thrown if the Entrust IdentityGuard service failed to generate a challenge. AuthenticationSystemFault is thrown if an IdentityGuard server fails.
ping	Does not return an exception if the Entrust IdentityGuard Authentication Service is alive.	RemoteException is thrown if the Entrust IdentityGuard Authentication Service is not available.

Administration faults

Administration Web service failures occur when users fail to properly complete administrative functions, such as registration. They also occur when administrative functions, like grid card creation, fail to complete successfully.

AdminServiceFault base class

This is the class used to catch all exceptions created when running the Administration API. See “System and service faults” on page [157](#) for a list of the access methods common to all fault classes.

AdminPasswordChangeRequiredFault

The Administration API returns this exception when a login operation fails because the administrator requires a password change. In addition to the values included in the `AdminServiceFault`, it also includes the current password rules.

Besides the access methods common to all fault classes, this fault class also uses the `PasswordRulesInfo` class.

Access methods

Method name	Description
<code>getPasswordRules</code>	Defines the password policy used for administrators. It specifies the number of uppercase characters, lowercase characters, special characters, and numbers a password must have, and the password's minimum length.

AdminServiceFault and AdminPasswordChangeRequiredFault code sample

For an example of how to handle the `AdminPasswordChangeRequiredFault` that can occur during an administrative login, see “Administration setup and login” on page [84](#).

Appendix A:

UtilityMethods class

This appendix presents the source code for the UtilityMethods class. This class includes helper methods.

```
package com.entrust.identityGuard.programmersGuide;

import java.io.BufferedReader;
import java.io.Console;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PushbackInputStream;
import java.net.URL;
import java.rmi.RemoteException;
import java.util.Arrays;

import javax.xml.rpc.ServiceException;

import com.entrust.identityGuard.authenticationManagement.wsv10.AuthenticationServiceBindingsStub;
import com.entrust.identityGuard.authenticationManagement.wsv10.AuthenticationService_ServiceLocator;
import com.entrust.identityGuard.authenticationManagement.wsv10.AuthenticationType;
import com.entrust.identityGuard.authenticationManagement.wsv10.Challenge;
import com.entrust.identityGuard.authenticationManagement.wsv10.ChallengeSet;
import com.entrust.identityGuard.authenticationManagement.wsv10.TokenChallenge;
import com.entrust.identityGuard.userManagement.wsv10.AdminPasswordChangeRequiredFault;
import com.entrust.identityGuard.userManagement.wsv10.AdminServiceBindingStub;
import com.entrust.identityGuard.userManagement.wsv10.AdminServiceFault;
import com.entrust.identityGuard.userManagement.wsv10.AdminService_ServiceLocator;
import com.entrust.identityGuard.userManagement.wsv10.ChangePasswordCallParms;
import com.entrust.identityGuard.userManagement.wsv10.LoginCallParms;
import com.entrust.identityGuard.userManagement.wsv10.LoginChallenge;
import com.entrust.identityGuard.userManagement.wsv10.LoginParms;
import com.entrust.identityGuard.userManagement.wsv10.LoginResult;
import com.entrust.identityGuard.userManagement.wsv10.LoginState;

public class UtilityMethods {
```

```

// Fix <hostname> placeholders below to reflect your environment.
// You may also need to change the default port numbers as well.

private static final String AUTH_SERVICE_URL =
"http://<hostname>:8080/IdentityGuardAuthService/services/AuthenticationServiceV9";
private static final String ADMIN_SERVICE_URL =
"https://<hostname>:8444/IdentityGuardAdminService/services/AdminServiceV9";

private static AuthenticationServiceBindingStub authBinding = null;
private static AdminServiceBindingStub adminBinding = null;

public static AuthenticationServiceBindingStub getAuthBinding() throws
ServiceException, IOException {

    if (authBinding != null) {
        return authBinding;
    }
    // Create the URL where the authentication service is located
    URL authServiceUrl = new URL(AUTH_SERVICE_URL);
    // Create a new binding using the URL just created
    try {
        AuthenticationService_ServiceLocator locator = new
AuthenticationService_ServiceLocator();
        authBinding = (AuthenticationServiceBindingStub)
locator.getAuthenticationService(authServiceUrl);
        return authBinding;
    } catch (ServiceException se) {
        // Handle exception
        if (se.getLinkedCause() != null) {
            se.getLinkedCause().printStackTrace();
        }
        throw se;
    }
}

private static String[] getSecondFactorChallengeResponse(BufferedReader in,
LoginChallenge loginChallenge) throws IOException {

    String[] response = null;
    AuthenticationType type = loginChallenge.getType();
    if (type.equals(AuthenticationType.GRID)) {
        ChallengeSet challengeSet = loginChallenge.getGridChallenge();
        // Code to construct a challenge prompt

```

```

        String prompt =
UtilityMethods.getGridChallengePrompt(challengeSet.getChallenge());
        System.out.print("Please answer the grid challenge " + prompt + ": ");
        response = in.readLine().split(" ");
    } else if (type.equals(AuthenticationType.TOKENRO)) {
        TokenChallenge tokenChallenge = loginChallenge.getTokenChallenge();
        // For simplicity, we assume user only has one token
        System.out.print("Enter your response for the token with serial # " +
tokenChallenge.getTokens()[0].getSerialNumber() + ": ");
        response = new String[] { in.readLine() };
    } else if (type.equals(AuthenticationType.TOKENCR)) {
        TokenChallenge tokenChallenge = loginChallenge.getTokenChallenge();
        // Get the token challenge
        // For simplicity, we assume the user only has one token
        System.out.println("For the challenge " + tokenChallenge.getChallenge() +
", enter your response for the token with serial # " +
tokenChallenge.getTokens()[0].getSerialNumber() + ": ");
        response = new String[] { in.readLine() };
    }
    return response;
}

public static AdminServiceBindingStub getAdminBinding() throws ServiceException,
IOException {
    // The following sample code generates an administration service
    // binding object that connects to the Administration services
    // URL, and logs in the given administrator with the given
    // password.
    if (adminBinding != null) {
        return adminBinding;
    }
    URL adminServiceUrl = new URL(ADMIN_SERVICE_URL);
    // Create a new binding using the URL just created
    try {
        AdminService_ServiceLocator locator = new AdminService_ServiceLocator();
        adminBinding = (AdminServiceBindingStub)
locator.getAdminService(adminServiceUrl);
        adminBinding.setMaintainSession(true);
    } catch (ServiceException se) {
        // Handle exception
        if (se.getLinkedCause() != null) {
            se.getLinkedCause().printStackTrace();
        }
        throw se;
    }
}

```

```

// login an administrative id
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
LoginParms loginParms = new LoginParms();
LoginCallParms callParms = new LoginCallParms();
callParms.setParms(loginParms);
LoginResult loginResult = null;
try {
    char[] password;

    System.out.print("Admin Id: ");
    loginParms.setAdminId(in.readLine());
    password = getPassword("Password: ");
    loginParms.setPassword(new String(password));
    Arrays.fill(password, ' ');
    loginResult = adminBinding.login(callParms);
} catch (AdminPasswordChangeRequiredFault apcrf) {
    // Optionally, add code to display the password rules
    // PasswordRulesInfo pwdinfo = apcrf.getPasswordRules();
    char[] newpwd, confirmpwd;
    do {
        newpwd = UtilityMethods.getPassword("Please enter your new password:
");
        confirmpwd = UtilityMethods.getPassword("Please confirm your new
password: ");
    } while (!Arrays.equals(newpwd, confirmpwd));
    ChangePasswordCallParms changePasswordCallParms = new
ChangePasswordCallParms();
    changePasswordCallParms.setNewpassword(new String(newpwd));
    changePasswordCallParms.setParms(loginParms);
    Arrays.fill(newpwd, ' ');
    Arrays.fill(confirmpwd, ' ');
    try {
        loginResult = adminBinding.changePassword(changePasswordCallParms);
        if (loginResult.getState().equals(LoginState.NEED_SECOND_FACTOR)) {
            // Need to answer second-factor authentication
            loginParms.setPassword(null);
            loginParms.setResponse(getSecondFactorChallengeResponse(in,
loginResult.getLoginChallenge()));
            loginResult =
adminBinding.changePassword(changePasswordCallParms);
            if (!loginResult.getState().equals(LoginState.COMPLETE)) {
                // Cannot complete second-factor authentication
                throw new Exception("Cannot complete second-factor auth");
            }
        }
    }
}

```

```

        }
    } catch (AdminServiceFault fault) {
        // Handle password change failed
        System.err.println("Password Change Failed: " +
fault.getErrorMessage());
        return null;
    } catch (Exception e) {
        System.err.println("Password Change Failed: " + e.getMessage());
        return null;
    }
} catch (AdminServiceFault fault) {
    // Handle login failed
    System.err.println("Login Failed: " + fault.getErrorMessage());
    return null;
} catch (RemoteException re) {
    // Handle other types of exceptions
    System.err.println("Login Failed: " + re.getMessage());
    return null;
}

// We could get here and still require second-factor authentication
if (loginResult.getState().equals(LoginState.NEED_SECOND_FACTOR)) {
    loginParms.setPassword(null);
    loginParms.setResponse(getSecondFactorChallengeResponse(in,
loginResult.getLoginChallenge()));
    try {
        loginResult = adminBinding.login(callParms);
        if (!loginResult.getState().equals(LoginState.COMPLETE)) {
            // Cannot complete second-factor authentication
            throw new Exception("Cannot complete second-factor auth");
        }
    } catch (AdminServiceFault fault) {
        System.err.println("Second Factor Authentication Failed: " +
fault.getErrorMessage());
        return null;
    } catch (Exception re) {
        System.err.println("Second Factor Authentication Failed: " +
re.getMessage());
        return null;
    }
}

System.out.println("Successfully authenticated to admin service with user " +
loginResult.getUserName() + " in group " + loginResult.getGroupName());
return adminBinding;

```



```

    }

    public static String getGridChallengePrompt(Challenge[] challArr) {
        StringBuilder challengeString = new StringBuilder();
        for (int i = 0; i < challArr.length; i++) {
            if (i != 0) {
                challengeString.append(' ');
            }
            Challenge chall = challArr[i];
            challengeString.append('[');
            // Cards with more than 26 columns will need a more
            // sophisticated algorithm
            challengeString.append((char) (chall.getColumn() + (int) 'A'));
            challengeString.append(',');
            challengeString.append(chall.getRow() + 1);
            challengeString.append(']');
        }
        return challengeString.toString();
    }

    public static char[] getPassword(String prompt) throws IOException {
        Console console = System.console();
        if (console != null) {
            return console.readPassword(prompt);
        } else {
            // Can't get a console, so password will be echoed (oh well)
            char[] buf, lineBuffer;

            buf = lineBuffer = new char[64];

            int room = buf.length;
            int offset = 0;
            int c;
            PushbackInputStream in = new PushbackInputStream(System.in);
            System.out.print(prompt);
            while (true) {
                c = in.read();
                if (c == -1 || c == '\n') {
                    break;
                }
                if (c == '\r') {
                    int c2 = in.read();

```

```

        if (!(c2 == -1 || c2 == '\n')) {
            in.unread(c2);
        } else {
            break;
        }
    } else {
        if (--room < 0) {
            buf = new char[offset + 64];
            room = buf.length - offset - 1;
            System.arraycopy(lineBuffer, 0, buf, 0, offset);
            Arrays.fill(lineBuffer, ' ');
            lineBuffer = buf;
        }
        buf[offset++] = (char) c;
    }
}

// See if anything was read
if (offset == 0) {
    return null;
}

char[] pwd = new char[offset];
System.arraycopy(buf, 0, pwd, 0, offset);
Arrays.fill(buf, ' ');
return pwd;
}

}

```

Index

A

Access methoSds

- getErrorCode, 143
- getErrorMessage, 143
- getId, 143
- getInternalCode, 143
- getMessage, 143
- getPasswordRules, 149

activate grid card, 82

ActiveX, 67

Administration

- activate grid card, 82
- cards, 95
- create grid cards, 82
- create one-time password, 85
- grid cards, 95
- knowledge-based questions, 93
- one-time password, 85
- preproduced cards, 84
- PVN, 92
- retrieve one-time password, 85
- send one-time password, 86
- service binding object, 77
- temporary PIN, 91
- token, 89, 90
- unlock users, 93
- user contact information, 87

Administration commands

- cardCreate, 84
- userCardCreate, 82

AdminService_Service, 25

AdminService_ServiceLocator, 26

AdminServiceBindingStub, 26

AdminServiceFault, 142

alerts

- see transaction notifications, 74

Apache Axis, 20

API overview

- Administration API, 5
- Administration API, 77
- Administration API
 - error output, 144
- Authentication API, 5, 7
- Authentication API
 - error output, 143

Authentication

- first-factor, 37
- generic, 40
- grid, 61
- knowledge-based, 61
- knowledge-based questions, 54
- machine, 61, 68
- multifactor, 84
- one-step (anonymous), 37
- one-time password, 42, 61
- Operations and exceptions
 - authenticateAnonymousChallenge, 147
 - authenticateGenericChallenge, 147
 - getAllowedAuthenticationTypes, 148
 - getAllowedAuthenticationTypesForGroup, 148
 - getAnonymousChallenge, 148
 - getAnonymousChallengeForGroup, 148
 - getGenericChallenge, 148
- questions and answers, 61
- risk-based, 71
- second-factor, 37
- token, 42, 61

Authentication commands

- authenticateAnonymousChallenge, 37
- authenticateGenericChallenge, 61
- getAnonymousChallenge, 37, 39
- getAnonymousChallengeForGroup, 39
- getChallenge, 39
- getGenericChallenge, 39, 61, 85

AuthenticationService_ServiceLocator, 25

AuthenticationServiceBindingStub, 25

AuthenticationServiceFault, 142, 145

AuthenticationSystemFault, 142

B

binding object

- creating, 24, 25

C

cards

- create, 95
- distribute, 95
- manufacture, 95

challenge requests, 10

Client-side processing for machine authentication, 69

code samples, 102

confirmation

- transaction, 74

create grid card, 82

Customer support, 4

E

Entrust IdentityGuard

- Java JAR files, 18
- replica servers, 23
- Web services, 18

Entrust IdentityGuard Mobile, 74

- see also soft tokens, 74

Entrust IdentityGuard repository, 81

error checking

- using an ErrorCode, 144

ErrorCode, 144

export

- keystore, 22

F

failures, 145, 149

faults, 142

- administration, 149
- AdminServiceFault, 142
- AuthenticationServiceFault, 142
- AuthenticationSystemFault, 142
- warning, 147

fingerprint

- machine fingerprint, 67

firewall rules, 10

first-factor authentication, 37

Flash, 68

G

generic authentication, 40

- code sample, 47
- knowledge-based questions, 45
- one-time password, 42
- token, 42

Getting help

- Technical Support, 4

Grid

- authenticate, 61

grid cards

- check inventory, 95
- create, 95
- distribute, 95
- manufacture, 95

I

IGAdminServiceFailoverFactory, 28
IGAuthServiceFailoverFactory, 26
iPhone, 74

J

Java

- classpath, 20
- errors, 20
- JAR files, 18
- Javadocs, 9
- Javascript, 66
- libraries, 8
- settings, 23

Java API toolkits, 9

Java class

- creating, 111, 114
- setting up, 114

Javadocs, 9

K

keystore, 22

- export, 22

keytool, 22

- documentation, 22

knowledge-based questions, 54

- authenticate, 61
- set up, 93

M

machine authentication, 61, 68

- application data, 68

Machine authentication Web sample, 69

machine information

- Get, 64, 67

machine nonce, 68

man-in-the-browser, 74

mitb attack, see man-in-the-browser, 74

multifactor authentication, 84

mutual authentication

- grid, 54
- token, 54

Mutual authentication

- knowledge-based questions, 54

N

nonce

- machine, 68
- optional sequence, 68

notification

- transaction, 74

notifications

- of transactions, 74

O

one-step (anonymous) authentication, 37

one-time password, 42, 85

- authenticate, 61
- create, 85
- retrieve, 85
- send, 86

OOB delivery

- adding identityguard properties, 109
- creating jar files, 114
- creating Java class, 111, 114
- setting up Java class, 114

OOB delivery of OTP, 109

optional application data:, 68

optional sequence nonce, 68

OTP

- Retrieve delivery configuration, 86

out of band delivery of OTP, 109

P

personal verification number - see PVN, 92

preproduced cards, 84

- assign, 84
- create, 84

Professional Services, 4

programming OOB, 109

PVN

- assign, 92
- create, 92
- modify, 92

R

replay

- grid values, 54
- image and message, 54
- serial number, 54

replica servers, 23

risk-based authentication, 71

S

sample admin tasks, 81

sample application, 9

sample smart credential code, 102

second-factor authentication, 37

serial numbers

- replay, 54

Server-side processing for machine authentication, 70

service binding object, 77

SOAP, 18, 19, 24

string conversion, 39

T

Technical Support, 4

temporary PIN

- assign, 91
- create, 91
- modify, 91

third-party libraries, 8

token

- assign, 89
- authenticate, 61
- check inventory, 95
- modify, 89
- reset, 89
- unlock, 90

Tomcat, 18, 22

transaction

- component, 74
- notifications, 74

Transaction notifications, 74

typographic conventions, 2

U

unlock users, 93

user contact information

- assign, 87
- create, 87
- modify, 87

W

warning faults, 147

Web service

- Administration, 18
- Authentication, 18

WebLogic, 20

WebSphere, 20

WSDL, 18

- files, 7

