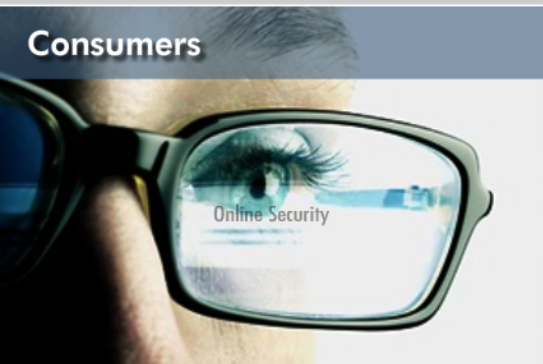


A Layered Security Approach to Enable...

Consumers



Enterprises



Governments



Entrust IdentityGuard 12.0

Programming Guide for the .NET Platform

Document Issue: 5.0
August 2018

Entrust is a trademark or a registered trademark of Entrust, Inc. in certain countries. Entrust is a registered trademark of Entrust Datacard Limited in Canada. All Entrust product names and logos are reademarks or registered trademarks of Entrust, Inc. or Entrust Datacard in certain countries. All other company and product names are trademarks or registered trademarks of their respective owners in certain countries. 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.

© 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 .NET class files..... | 16 |
| V11 .NET API documentation | 16 |
| Sample applications | 18 |
| Sample command-line applications..... | 18 |
| Running the sample authentication client..... | 18 |
| Running the sample administration client | 20 |
| Using the administration commands | 21 |
| Chapter 2: Client application setup | 23 |
| Setting up your application | 23 |
| Using SSL communication..... | 24 |
| Configuring trust | 24 |
| Configuring SSL with Entrust IdentityGuard replicas | 25 |
| Create a binding object..... | 27 |
| Create an authentication binding object | 27 |
| Create an administration binding object | 27 |
| Create a failover authentication binding object | 28 |
| Create a failover administration binding object | 28 |
| Performance optimizations..... | 29 |
| Reuse a single binding object | 29 |
| Disable the Default Web Proxy | 29 |
| Migrating V9, and V10 services to V11..... | 29 |
| Update service URLs..... | 29 |
| Update proxy class library..... | 30 |
| Update proxy class namespace | 30 |

| | |
|---|-----------|
| Changes to the Authentication API | 30 |
| Changes to the Administration API..... | 32 |
| Chapter 3: Authentication approaches..... | 36 |
| Anonymous grid authentication..... | 36 |
| Anonymous grid authentication method | 36 |
| Security considerations..... | 37 |
| Anonymous grid authentication sample | 37 |
| Challenge retention | 39 |
| String conversion sample..... | 39 |
| Generic authentication | 40 |
| Generic API methods..... | 40 |
| Grids | 41 |
| Tokens | 41 |
| One time password (OTP) | 42 |
| Knowledge-based questions and answers | 45 |
| External authentication | 45 |
| Password authentication..... | 45 |
| Certificate challenge response authentication | 45 |
| Generic API code sample | 46 |
| Mutual authentication | 54 |
| Grid and token serial number and location replay..... | 54 |
| Knowledge-based authentication | 54 |
| Image and caption replay..... | 54 |
| Image and caption replay samples..... | 55 |
| Image management..... | 58 |
| Step-up authentication | 59 |
| Machine authentication | 59 |
| Machine authentication API methods | 60 |
| Machine authentication API code example..... | 60 |
| Sources of machine information..... | 62 |
| Storing and retrieving machine information | 66 |
| Machine authentication Web sample..... | 67 |
| Risk-based authentication (RBA) | 69 |
| Transaction authentication..... | 72 |
| Chapter 4: Administration tasks | 77 |
| Administration setup and login..... | 77 |
| Basic administration tasks | 78 |
| Create and register a user | 78 |
| Rename a user | 79 |
| Get grid contents for a user | 79 |
| Create and activate a user's grid card..... | 80 |
| Create and assign preproduced grid cards..... | 81 |
| Create and send an OTP | 82 |

| | |
|---|------------|
| Retrieve delivery configuration of an OTP | 84 |
| Create and modify user contact information | 85 |
| Assign and modify a token..... | 86 |
| Create and modify a temporary PIN..... | 88 |
| Create and modify a personal verification number (PVN) | 90 |
| Set up a user's questions and answers..... | 91 |
| Unlock users | 91 |
| Administer machine secrets..... | 92 |
| Administrative monitoring tasks..... | 93 |
| Check for expiring grid cards | 93 |
| Check grid card inventory | 94 |
| Check token inventory | 94 |
| Check for unused assigned grid cards or tokens | 95 |
| Administration of smart credentials..... | 97 |
| Create smart credentials for a user | 97 |
| Approve smart credentials | 98 |
| Issue smart credentials..... | 99 |
| Check the status of smart credentials issuance request | 100 |
| Modify smart credentials | 101 |
| Modify enrollment values in smart credentials | 102 |
| Change the state of a smart credential..... | 103 |
| Chapter 5: Performing Identity Assured operations with smart credentials | 106 |
| Security challenges | 106 |
| Security challenge integration | 108 |
| Configuration..... | 108 |
| Security challenges for authentication and transaction verification..... | 109 |
| Security challenges for digital signing | 113 |
| Anonymous smart credential security challenges | 117 |
| Chapter 6: Integrating Entrust mobile soft tokens..... | 123 |
| Soft token activation..... | 123 |
| Token activation methods | 123 |
| userTokenActivateComplete | 124 |
| Token activation integration..... | 124 |
| Soft token transaction verification..... | 125 |
| Online transaction verification | 125 |
| Authentication notifications | 127 |
| Transaction verification notifications..... | 127 |
| Online transaction verification integration | 127 |
| Configuration..... | 127 |
| Online authentication and transaction verification..... | 128 |
| Offline transaction verification..... | 131 |
| Offline transaction verification integration..... | 132 |
| Chapter 7: API exceptions..... | 135 |

| | |
|--|------------|
| SoapException returned by proxy classes | 135 |
| ErrorCode class | 136 |
| Authentication warning faults | 137 |
| Authentication operation exceptions..... | 137 |
| Administration Password Change | 139 |
| Index | 141 |

About this guide

The *Entrust IdentityGuard Programming Guide* provides detailed information about how to use the .NET version of the Entrust IdentityGuard Authentication API and the Administration APIs 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 |
|-------------------------|--|--|
| 5.0 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. |
| 4.0 July 2018 | Various | Updated version of .NET required to 4.0 |
| 3.0 March 2018 | Security challenges Online transaction verification | Improved readability of sequence diagrams |
| 2.0 July 2017 | Challenge retention | Additional paragraph and list |
| | Create a failover administration binding object | Changed "authentication" to "administration" in first paragraph. |
| 1.0 March 2017 | All sections | First release of this guide for Entrust IdentityGuard Server 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 Server Administration Guide</i> |
| Blue text | Used for hyperlinks to other sections in the document | For more information about initialization see, Initializing IdentityGuard . |
| Underlined blue text | 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 configuring and using smart credentials, 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. A patch to the V8 APIs introduced support for multiple passwords |

| API version | Entrust IdentityGuard version | API features |
|-------------|-------------------------------|--|
| 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, and 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 (also referred to as the Authentication API) 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 information about programming on the Java Platform, see the *Entrust IdentityGuard Programming Guide for the Java Platform*.

Administration V11 API

The Entrust IdentityGuard Administration service (also referred to as the Administration API) is a servlet 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 Java Platform, see the *Entrust IdentityGuard Programming Guide for the Java 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 .NET class files

Entrust IdentityGuard 12.0 supports Microsoft .NET Framework 4.0, and includes the following files used for .NET development:

- `IdentityGuardAuthServiceV11API.cs`, which is the .NET proxy class code in C# for Authentication V11 service.
- `IdentityGuardAuthServiceV11API.dll`, which is the .NET class library for the proxy class of the Authentication V11 service.
- `IdentityGuardAdminServiceV11API.cs`, which is the .NET proxy class code in C# for the Administration V11 service.
- `IdentityGuardAdminServiceV11API.dll`, which is the .NET class library for the proxy class of the Administration V11 service.
- `IdentityGuardCommonFailoverAPI.dll`, which is the .NET class library for enabling Entrust IdentityGuard failover functionality.
- `IdentityGuardAuthServiceV11API.XmlSerializer.dll`, which is the .NET precompiled XML Serialization library for the Authentication V11 service
- `IdentityGuardAdminServiceV11API.XmlSerializer.dll`, which is the .NET precompiled XML Serialization library for the Administration V11 service. You can locate these:
 - on UNIX, in `$IG_HOME/client/C#/lib`
 - on Windows, in `<IG_HOME>\client\C#\lib`

V11 .NET API documentation

Your Entrust IdentityGuard installation also includes a set of HTML files generated from the C# proxy class code, `IdentityGuardAdminServiceV11API.html` and `IdentityGuardAuthServiceV11API.html`, that explain the .NET API toolkits.

You can locate these:

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

You can also find the API documentation in the Authentication and Administration WSDL files.

Sample applications

Entrust provides two command-line sample applications to help guide your Entrust IdentityGuard implementation.

Sample command-line applications

Entrust IdentityGuard includes sample applications that you can use to test the APIs that perform challenge, authentication, and administration requests. The sample application code is written in C#, and supports Microsoft .NET Framework 4.0.

- For the Authentication service, the file `IdentityGuardAuthServiceClient.cs` contains the C# source code for a sample command-line application.

The compiled version is `IdentityGuardAuthServiceClient.exe`. For instructions for running this sample, see [Running the sample administration client](#).

- For the Administration service, the file `IdentityGuardAdminServiceClient.cs` contains the C# source code for a sample command-line application.

The compiled version is `IdentityGuardAdminServiceClient.exe`. For instructions for running this sample, see [Running the sample administration client](#).

Running the sample authentication client

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

Before running the samples, you may want to create sample users with grids, tokens, or other types of 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.

You must install Microsoft .NET Framework Version 4.0 on the Windows computer running the C# samples.

Note: Run the C# Administration sample client on a Windows computer with Microsoft .NET Framework 4.0 installed.

To run the C# authentication sample on Microsoft Windows

- 1 If your Entrust IdentityGuard server is installed on UNIX or Linux, copy all .NET files to a Microsoft Windows-based computer.
- 2 Import the Entrust IdentityGuard SSL certificate to the Windows computer running the sample client application.
Ensure that the Windows user running the application has proper permissions to access the certificate.
- 3 Open in a text editor:
`<IG_HOME>\client\C#\sample\auth\runAuthClient.bat`
- 4 Change the Authentication service URL to your Entrust IdentityGuard Authentication V11 Service URL.
- 5 Save `runAuthClient.bat`.

6 Run `runAuthClient.bat` from the command prompt.

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 C# Sample App
=====

Welcome to Entrust IdentityGuard Authentication Service sample
application.

This application gives samples of the usage of the IdentityGuard
Authentication API, which is a set of Web services used for retrieving
challenge requests and authenticating 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 IdentityGuard authentication service URL:

http://localhost:8080/IdentityGuardAuthService/services/AuthenticationServiceV11
```

For more information about the authentication mechanisms listed, see [Chapter 3: Authentication approaches](#), or the HTML help document for the .NET proxy class.

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, and you may include more than one occurrence of an attribute.
- Many commands require a user ID.

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

See Documentation conventions 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.

Note: Run the C# Administration sample client on a Windows computer with Microsoft .NET Framework 4.0 installed.

To run the C# administration sample client application

- 1** Ensure that the Entrust IdentityGuard server is running.
- 2** Ensure that Microsoft .NET Framework 4.0 is installed on the computer to run the sample application.
- 3** Import the Entrust IdentityGuard SSL certificate to the Windows computer running the sample client application.
Ensure that the Windows user running the application has proper permissions to access the certificate.
- 4** If you are running the sample client from a Windows computer that is different from the computer on which Entrust IdentityGuard installed, copy all files from `<IG_HOME>\client\C#\sample\admin`.
- 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 service.
 - b** Edit the administration credentials for an administrator with the required permissions.
In the following example, `sample-group/superadmin` has the superuser role, so it can invoke any administration function.

```
# the URL of the entrust 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=sample-group/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
```

- c Open the `runAdminClient.bat` file in a text editor.
The `runAdminClient.bat` file contains the commands and parameter to run the administration sample client application.
- d Update the URL to ensure that it points to your Entrust IdentityGuard Administration service.
- e Save and close the file.

6 Run `runAdminClient.bat` from the command prompt.

```
=====
Entrust IdentityGuard AdminService Client C# Sample App
=====

Welcome to Entrust IdentityGuard Administration Service sample client
application.

This application gives the sample usage of the Entrust IdentityGuard
Administration service API, which is a set of Web services that allow an
administrator to perform administration tasks to users, PINs, cards and
tokens.

To perform the administration tasks, you must first login with a valid
admin id and password.

To display all available commands, type 'help'.

Connected to Entrust IdentityGuard admin service URL:
https://localhost:8444/IdentityGuardAdminService/services/AdminServiceV11
```

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`, though 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, and may include more than one occurrence of an attribute.
- Many commands require a user ID.

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

Chapter 2:

Client application setup

The Entrust IdentityGuard .NET Framework 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 must set up your application to use the Entrust IdentityGuard .NET proxy class libraries, which provide a convenient API for your client application. Your client application makes API calls to the Entrust IdentityGuard .NET proxy classes. The details for conforming to the WSDL definition and communicating with the Entrust IdentityGuard Web services are handled automatically.

To use the Entrust IdentityGuard APIs, copy the following files from the `<IG_HOME>\client\C#\lib` directory and make them available to your client application:

- `IdentityGuardAuthServiceV11API.dll` (if you are using the Authentication API)
- `IdentityGuardAdminServiceV11API.dll` (if you are using the Administration API)
- `IdentityGuardCommonFailoverAPI.dll` (required for both the Authentication API and Administration API)
- `IdentityGuardAuthServiceV11API.XmlSerializer.dll`, which is the .NET precompiled XML Serialization library for the Authentication V11 service.
- `IdentityGuardAdminServiceV11API.XmlSerializer.dll`, which is the .NET precompiled XML Serialization library for the Administration V11 service.

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.

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

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.

Running the .NET sample applications and running ASP.NET applications both require you to configure your machine to trust the Entrust IdentityGuard certificate.

If you just need to run the command line .NET sample application, just install the SSL certificate in the certificate store for your Windows user.

To run ASP.NET applications, however, this is not sufficient, because by default, ASP.NET runs using the Network Services account (on Windows Server 2003) rather than your user account.

The best way around this is to install the certificate in the local machine store so that the account used to run ASP.NET can access it. Then, you must either configure its ACL to grant access to your application's account, or if you have more than just the SSL certificate and a trusted root certificate in the certification path, create a certificate chain.

To configure your client application to trust the Entrust IdentityGuard certificate

- 1 Open Internet Explorer and point it to your Entrust IdentityGuard administration service URL.
- 2 In Internet Explorer, locate the lock icon. Depending on your version, the icon may be on the lower right corner of the browser window or in the URL field.
- 3 Perform the applicable operation for your version, which may be a double-click or single click. A pop-up window appears.
- 4 From the pop-up **Certificate** window, click **Install Certificate**. (If there is a view option instead of an install option, this indicates your browser already has a stored certificate.)

- 5 Follow the instructions of the **Certificate Import Wizard** to install the certificate.

Note: If you have upgraded Entrust IdentityGuard, your client application can continue using the same certificate. A new SSL certificate is not issued during upgrade from an earlier version of Entrust IdentityGuard.

Install the SSL certificate in the local machine store

If you are developing an ASP.NET application, use the following procedures to configure your application to trust the SSL certificate. This procedure describes creating a certificate chain; you can also configure the ACL to grant access to your application's account.

To locate the SSL certificate used by Entrust IdentityGuard

- 1 Locate the certificate:
 - To use the Entrust IdentityGuard self-signed certificate, find `identityguard.cer` on the server hosting Entrust IdentityGuard.
 - To use a CA certificate, locate the certificate used by Entrust IdentityGuard and make sure you know the full list of certificates in the path to the root certificate.
- 2 On the Internet Explorer **Tools** menu, click **Internet Options**.
- 3 Click the **Content** tab.
- 4 Under **Certificates**, click the **Certificates** button to view a list of certificates.
- 5 Click the **Trusted Root Certification Authorities** tab and select the certificate you want to import.
- 6 Click **Import**.
The **Certificate Import Wizard** appears.
- 7 In the **Certificate Import Wizard**, click **Next**.
- 8 Browse to the certificate to import and click **Next**.
- 9 Select **Place all certificates in the following store** and click **Browse**.
- 10 Select **Show physical stores** on the **Select Certificate Store** dialog box.
- 11 Select **Trusted Root Certification Authorities > Local computer** and then click **OK**.
- 12 Click **Next** on the **Certificate Store** dialog box, then click **Finish**.
- 13 For an SSL certificate issued by a CA certificate, repeat this process for each certificate in the certification path.

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-based 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 trusted certificate store 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's trusted certificate store or your local machine store.
 - 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's **trusted certificate store**, or your **local machine store**.

Create a binding object

By using specific .NET proxy 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` interface class:

```
// the URL where the authentication service is located:
string urlString =
"http://localhost:8080/IdentityGuardAuthService/services/AuthenticationServiceV11";

// Create a new binding using the URL just created:
AuthenticationService authBinding = new AuthenticationService();
authBinding.Url = urlString;
```

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:

```
AdminService adminBinding = null;
string urlString =
"https://localhost:8444/IdentityGuardAdminService/services/AdminServiceV11";

// Create the URL where the administration service is located:
adminBinding = new AdminService();
adminBinding.CookieContainer = new CookieContainer();
adminBinding.Url = urlString;
```

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.

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:

```
String[] serviceUrls = {  
    "http://localhost:8080/IdentityGuardAuthService/services/AuthenticationServiceV11",  
    "http://identityguard.example.com:8080/IdentityGuardAuthService/services/AuthenticationServiceV11"  
};  
  
IGAAuthServiceFailoverFactory factory = new  
IGAAuthServiceFailoverFactory(serviceUrls);  
  
AuthenticationService binding = factory.getService();
```

Create a failover administration binding object

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

```
String[] serviceUrls = {  
    "https://localhost:8444/IdentityGuardAdminService/services/AdminServiceV11",  
    "https://identityguard.example.com:8444/IdentityGuardAdminService/services/AdminServiceV11"  
};  
  
String adminid = "admin";  
String adminpassword = "TopSecret123";  
  
IGAdminServiceFailoverFactory failoverFactory =  
    new IGAdminServiceFailoverFactory(  
        serviceUrls,  
        adminid,  
        adminpassword);  
  
AdminService binding = failoverFactory.getService();
```

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.

Disable the Default Web Proxy

If your application is not making use of the Windows Web proxy configuration, you can disable automatic discovery of web proxy configuration settings. Automatic discovery of Web proxy configuration settings has been known to take up to 2 seconds to complete the first time you connect to Entrust IdentityGuard APIs. To disable the Web proxy, your application must run the following command before it initializes the Administration or Authentication binding objects.

```
WebRequest.DefaultWebProxy = null;
```

Note: This command affects all Web requests sent by your application.

Migrating V9, and V10 services to V11

After upgrading to Entrust IdentityGuard release 12.0, your .NET 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 or 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 V9 and V10 services are the same Entrust IdentityGuard URLs with v11 rather than v9 or v10 appended on the end. You must update the URLs to access V11 services.

For example, instead of accessing the Authentication service from

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

use

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

To access the Administration V10 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 `IdentityGuardAuthServiceV10API.dll` to `IdentityGuardAuthServiceV11API.dll`.

Change the administration proxy class library from `IdentityGuardAdminServiceV10API.dll` to `IdentityGuardAdminServiceV11API.dll`.

Add the failover class library `IdentityGuardCommonFailoverAPI.dll`.

Update proxy class namespace

In your application code, update the namespace of the V10 proxy class to V11. Instead of using namespace `IdentityGuardAuthServiceV10API`, update it to `IdentityGuardAuthServiceV11API`.

For the Administration service, update `IdentityGuardAdminServiceV10API` to `IdentityGuardAdminServiceV11API`.

Changes to the Authentication API

For full descriptions of these new and changed classes, see the HTML documentation for .NET 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 <code>authenticateAnonymousCertChallenge</code> <code>getAnonymousCertChallenge</code> |
| AuthenticationServiceBindingSkeleton | Removed usage of extended methods from V10 Added <code>authenticateAnonymousCertChallenge</code> <code>getAnonymousCertChallenge</code> |
| AuthenticationServiceBindingStub | Removed usage of extended methods from V10 Added <code>authenticateAnonymousCertChallenge</code> |

| Class | Changes |
|---|--|
| | getAnonymousCertChallenge |
| AuthenticationService_PortType | Removed usage of extended methods from V10 Added authenticateAnonymousCertChallenge 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 .NET 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 |
| DigitalIdConfigEntrustSpecificParams | Added Fields changeDNMode supportsDNChange |
| MachineSecretInfo | Added Field machineExternalId |
| MachineSecretParams | 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 |
| 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 APIs.

Anonymous grid authentication

Note: This approach applies to 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 in 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 password input field.
- Entrust IdentityGuard:** A section containing three small square icons labeled [C3], [D2], and [E4].
- Link:** A blue hyperlink 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.GridChallenge;
```

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.group = group;

GenericChallenge challenge =
authBinding.getAnonymousChallengeForGroup(callParms);
ChallengeSet challengeSet = challenge.GridChallenge;
```

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

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.response = challengeResponse;
AuthenticateAnonymousChallengeCallParms
    authenticateAnonymousChallengeCallParms
    = new AuthenticateAnonymousChallengeCallParms();
authenticateAnonymousChallengeCallParms.userId = userid;
authenticateAnonymousChallengeCallParms.challengeSet =
    ms_cachedAnonymousChallengeSet;
authenticateAnonymousChallengeCallParms.response = response;
try
{
    GenericAuthenticateResponse resp =
        authBinding.authenticateAnonymousChallenge(
            authenticateAnonymousChallengeCallParms);
}
catch (SoapException soapEx)
{
    AuthenticationFault fault =
        AuthenticationService.getFault(soapEx);
    if (fault != null)
    {
        // handle the IdentityGuard fault
        System.Console.WriteLine(fault.errorMessage);
        return;
    }
    else
    {
        // not IG fault, handle the generic SoapException
        System.Console.WriteLine(soapEx.Message);
        return;
    }
}
catch (Exception ex)
{
    // handle non-soap exception, e.g., connection error
    System.Console.WriteLine(ex.Message);
    return;
```

}

Challenge retention

By default, 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.

However, you can change various policy settings to alter the default challenge retention behavior. Grid card, certificate, knowledge-based Q&A, and token challenge-response policy categories all have the ability to disable challenge retention or set a time period for which the current challenge is valid. In conjunction with these settings, there is also the ability to update the lockout count when a challenge needs to be replaced. For more information, see the “Policies quick reference” section in the Entrust IdentityGuard Server Administration Guide. There are descriptions of the following policy settings for grid (card), certificate, knowledge-based (Q&A), and token authentication types:

- Disable Challenge Retention
- <authentication type> Challenge Lifetime
- Update Lockout Count for Replaced Challenge

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. For more information, see [Security considerations](#).

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.

```
StringBuilder builder = new StringBuilder("");
Challenge[] challArr = challengeSet.challenge;
for (int i = 0; i < challArr.Length; i++)
{
    if (i != 0)
    {
        builder.Append(" ");
    }
    Challenge chall = challArr[i];
    builder.Append('[');
    builder.Append((char)(chall.Column + (int)'A'));
```

```
builder.Append(' ');
builder.Append(chall.Row + 1);
builder.Append(']');
}
string challengeString = builder.ToString();
```

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](#).

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](#) for a detailed example.

Grid challenges also accept a temporary PIN as a response.

Two-step authentication with a grid

The diagram illustrates a two-step authentication process. In the first step, a user logs into 'Any Bank' with their username 'John Smith' and a masked password. In the second step, the user is presented with an 'Entrust IdentityGuard Challenge' form. This form includes fields for IdentityGuard values [B4], [E4], and [G5], and a 'Submit Form' button. A grid card is shown next to the challenge form. The grid card features a 5x10 grid of numbers. Red arrows indicate the lookup path for coordinates B3 and H1. The grid card also displays the serial number #1234567.

| | A | B | C | D | E | F | G | H | I | J |
|---|---|---|---|---|---|---|---|---|---|---|
| 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 | 6 | 3 | 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](#) for a more detailed example.

Entrust soft tokens can also be used to sign transactions details (currently only supported by the iPhone). To perform transaction delivery, simply include the transaction details in the `GenericChallengeParms` when requesting a `TOKENRO` authentication from Entrust IdentityGuard.

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

One time password (OTP)

You can authenticate users with an 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.

An example of code showing how to use out-of-band authentication using OTPs is included 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.

// Create the generic challenge parameters as follows
// without specifying useDefaultDelivery or contactInfoLabel
GenericChallengeParms genericChallengeParms = new GenericChallengeParms();
genericChallengeParms.AuthenticationType = AuthenticationType.OTP;

// Get the generic challenge
GetGenericChallengeCallParms callParms = new GetGenericChallengeCallParms();
callParms.userId = userid;
callParms.parms = genericChallengeParms;
GenericChallenge genericChallenge = null;
// call getGenericChallenge, the OTP will be generated but not delivered
try
```

```

{
    genericChallenge =
        authBinding.getGenericChallenge(callParms);
}
catch (SoapException soapEx)
{
    AuthenticationFault fault =
        AuthenticationService.getFault(soapEx);
    if (fault != null)
    {
        // handle the IdentityGuard fault
        System.Console.WriteLine(fault.errorMessage);
        return;
    }
    else
    {
        // not IG fault, handle the generic SoapException
        System.Console.WriteLine(soapEx.Message);
        return;
    }
}
catch (Exception ex)
{
    // handle non-soap exception, e.g., connection error
    System.Console.WriteLine(ex.Message);
    return;
}

// get the available contactinfo from the response
OTPChallenge otpChallenge = genericChallenge.OTPChallenge;
// if manualDeliveryRequired is true, client application
// must retrieve and deliver the OTP
string contactinfoLabelToUse = null;
if (otpChallenge.manualDeliveryRequired != null &&
    (bool)otpChallenge.manualDeliveryRequired)
{
    // client app code to retrieve (using admin API) and deliver the OTP
}
else
{
    DeliveryMechanism[] deliveryConfigList = otpChallenge.deliveryMechanism;
    // just get the first available
    if (deliveryConfigList.Length == 0)
    {

```

```

// no delivery mechanism available
// app code to handle the situation
}
else
{
    contactinfoLabelToUse =
    deliveryConfigList[0].contactInfoLabel;
}
}
genericChallengeParms = new GenericChallengeParms();
genericChallengeParms.AuthenticationType = AuthenticationType.OTP;
genericChallengeParms.contactInfoLabel = new string[1]{contactinfoLabelToUse};
callParms = new GetGenericChallengeCallParms();
callParms.userId = userid;
callParms.parms = genericChallengeParms;
// call getGenericChallenge again, the OTP
// is delivered to the specified contactinfo
try
{
    genericChallenge = authBinding.getGenericChallenge(callParms);
}
catch (SoapException soapEx)
{
    AuthenticationFault fault = AuthenticationService.getFault(soapEx);
    if (fault != null)
    {
        // handle the IdentityGuard fault
        System.Console.WriteLine(fault.errorMessage);
        return;
    }
    else
    {
        // not IG fault, handle the generic SoapException
        System.Console.WriteLine(soapEx.Message);
        return;
    }
}
catch (Exception ex)
{
    // handle non-soap exception, e.g., connection error
    System.Console.WriteLine(ex.Message);
    return;
}

```

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](#) 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.

Question-and-answer challenge

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.

For a code sample, see [Generic API code sample](#).

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.

For a code sample, see [Generic API code sample](#).

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.

For a code sample, see [Generic API code sample](#).

Certificate challenge response authentication

There are two forms of certificate authentication: one is integrated with Risk-Based Authentication (see [Risk-based authentication \(RBA\)](#)), 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 available in C#.

Generic API code sample

The sample shows how to create or retrieve a generic challenge method for a given user, how to issue the generic challenge, and how to authenticate it.

```
//Create the generic challenge parameters as follows:
GenericChallengeParms genericChallengeParms = new GenericChallengeParms();
// Optionally, retrieve the allowed authentication types for
// the generic authentication operations. You can also retrieve
// the allowed types for the specified group.
GetAllowedAuthenticationTypesCallParms getTypeCallParms
    = new GetAllowedAuthenticationTypesCallParms();
getTypeCallParms.userId = userid;
AllowedAuthenticationTypes types = null;
try
{
    types =
        authBinding.getAllowedAuthenticationTypes(getTypeCallParms);
}
catch (SoapException soapEx)
{
    AuthenticationFault fault = AuthenticationService.getFault(soapEx);
    if (fault != null)
    {
        // handle the IdentityGuard fault
        System.Console.WriteLine(fault.errorMessage);
    }
}
```

```

        return;
    }
    else
    {
        // not IG fault, handle the generic SoapException
        System.Console.WriteLine(soupEx.Message);
        return;
    }
}
catch (Exception ex)
{
    // handle non-soap exception, e.g., connection error
    System.Console.WriteLine(ex.Message);
    return;
}

// Next, set the authentication method to use. You can set
// the authentication type to one of the allowed authentication
// types. If one is not set, the default type for the user is used.
// The following code sets QA as the auth type if it is an allowed
// generic auth type; otherwise it uses the first allowed generic auth type
AuthenticationType? authType = null;
// Use the first type in the generic auth list
AuthenticationType[] genericAuthTypes = types.genericAuth;
for (int i = 0; i < genericAuthTypes.Length; i++)
{
    if (genericAuthTypes[i].Equals(AuthenticationType.QA))
    {
        authType = AuthenticationType.QA;
        break;
    }
}
if (authType == null && genericAuthTypes.Length >= 1)
{
    authType = genericAuthTypes[0];
}
genericChallengeParms.AuthenticationType = authType;
// Get the generic challenge
GetGenericChallengeCallParms callParms = new GetGenericChallengeCallParms();
callParms.userId = userid;
callParms.parms = genericChallengeParms;
GenericChallenge genericChallenge = null;
try

```

```

{
    genericChallenge = authBinding.getGenericChallenge(callParms);
}
catch (SoapException soapEx)
{
    AuthenticationFault fault = AuthenticationService.getFault(soapEx);
    if (fault != null)
    {
        // handle the IdentityGuard fault
        System.Console.WriteLine(fault.errorMessage);
        return;
    }
    else
    {
        // not IG fault, handle the generic SoapException
        System.Console.WriteLine(soapEx.Message);
        return;
    }
}
catch (Exception ex)
{
    // handle non-soap exception, e.g., connection error
    System.Console.WriteLine(ex.Message);
    return;
}

// process the challenge
Response response = new Response();
GenericAuthenticateParms authParms = new GenericAuthenticateParms();

// If the authentication method is a grid:
if (genericChallenge.type.Equals(AuthenticationType.GRID))
{
    Challenge[] challArr = genericChallenge.GridChallenge.challenge;
    StringBuilder builder = new StringBuilder("");
    for (int i = 0; i < challArr.Length; i++)
    {
        if (i != 0)
        {
            builder.Append(" ");
        }
        Challenge chall = challArr[i];
        builder.Append('[');
    }
}

```



```

        builder.Append((char)(chall.Column + (int)'A'));
        builder.Append(',');
        builder.Append(chall.Row + 1);
        builder.Append('');
    }
    String prompt = builder.ToString();
    Console.Write("Please answer the grid challenge: " + prompt + ": ");
    response.response = Console.ReadLine().Split(' ');
}

// If the authentication method is a read-only token:
else if (genericChallenge.type.Equals(AuthenticationType.TOKENRO))
{
    TokenChallenge tokenChallenge = genericChallenge.TokenChallenge;
    // For simplicity, we assume the user only has one response only token
    Console.Write("Enter your response for the token with serial number "
        + tokenChallenge.tokens[0].SerialNumber + ": ");
    String[] userResponse = { Console.ReadLine() };
    response.response = userResponse;
}

// If the authentication method is a challenge-response token
else if (genericChallenge.type.Equals(AuthenticationType.TOKENCR))
{
    TokenChallenge tokenChallenge =
        genericChallenge.TokenChallenge;
    // Get the token challenge
    // For simplicity, we assume the user only has one CR token
    Console.Write("For the challenge " + tokenChallenge.challenge
        + ", enter your response for the token with serial number "
        + tokenChallenge.tokens[0].SerialNumber + ": ");
    String[] userResponse = { Console.ReadLine() };
    response.response = userResponse;
}

// If the authentication method is knowledge-based (questions and answers):
else if (genericChallenge.type.Equals(AuthenticationType.QA))
{
    String[] questions = genericChallenge.QAChallenge;
    String[] answers = new String[questions.Length];
    Console.WriteLine(
        "Please provide answers for the following questions:");
    for (int i = 0; i < questions.Length; i++)
    {
        Console.Write(questions[i] + " ");
        answers[i] = Console.ReadLine();
    }
}

```

```

    }
    response.response = answers;
}
// If the authentication method is one-time password:
else if( genericChallenge.type.Equals(AuthenticationType.OTP))
{
    // depending on the delivery configuration setup, IG may deliver the OTP
    // to user via out-of-band mechanism
    Console.WriteLine("Please enter your OTP: ");
    String[] userResponse = { Console.ReadLine() };
    response.response = userResponse;
}
else if(genericChallenge.type.Equals(AuthenticationType.PASSWORD))
{
    //Display the password challenge
    Console.WriteLine("Please enter your password: ");
    String[] userResponse = { Console.ReadLine() };
    response.response = userResponse;

    // add code to handle the password change requirement
    PasswordChallenge pswdChall = genericChallenge.PasswordChallenge;
    bool? changeRequired = pswdChall.changeRequired;
    if (changeRequired.HasValue && changeRequired.Value)
    {
        // optionally, add code to display the password rules
        // PasswordRules pswdRules = pswdChall.passwordRules;
        String newpwd, confirmpwd;
        do
        {
            Console.WriteLine("Please enter your new password: ");
            newpwd = Console.ReadLine();

            Console.WriteLine("Please confirm your new password: ");
            confirmpwd = Console.ReadLine();

        } while (newpwd != confirmpwd);
        authParms.newPassword = newpwd;
    }
}
else if (genericChallenge.type.Equals(
    AuthenticationType.EXTERNAL))
{

```

```

// Add code to handle the external authentication
// Assume external authentication is password based.
Console.WriteLine("Please enter your external password: ");
String[] userResponse = { Console.ReadLine() };
response.response = userResponse;
}
else if (genericChallenge.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..NET in the
    // client/sample folder.
    Console.Error.WriteLine(
        "This program does not support certificate authentication!");
    return;
}
// If the authentication method is NONE, do the following:
else if (genericChallenge.type.Equals(AuthenticationType.NONE))
{
    // For NONE, do not call authenticatGenericChallenge --
    // the user is not authenticated in any way
    Console.WriteLine("User does not require authentication.");
    return;
}

// Determine if the authentication method has a pvn
PVNInfo pvnInfo = genericChallenge.PVNInfo;
if (pvnInfo != null)
{
    // Determine if the pvn is required
    if (pvnInfo.required)
    {
        Boolean changeRequired = false;
        // If the pvn is available. prompt the user to enter it

```

```

if (pvnInfo.available.HasValue && pvnInfo.available.Value)
{
    // Prompt the user to enter their PVN
    Console.WriteLine("Please enter your Personal Verification Number: ");
    response.PVN = Console.ReadLine();

    // If the pvn requires updating prompt the user to update pvn
    if (pvnInfo.changeRequired.HasValue &&
        pvnInfo.changeRequired.Value)
    {
        changeRequired = true;
    }
}
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.
    String newpvn, confirmpvn;
    do
    {
        Console.WriteLine("Please enter a new " + pvnInfo.length
            + "digit Personal Verification Number: ");
        newpvn = Console.ReadLine();
        Console.WriteLine("Please confirm your new PVN: ");
        confirmpvn = Console.ReadLine();
    } while (newpvn != confirmpvn);
    authParms.newPVN = newpvn;
}

}
}
// Once the client application gets a challenge response from the
// user, authenticate the response
try
{
    authParms.AuthenticationType = authType;
    AuthenticateGenericChallengeCallParms authCallParms
        = new AuthenticateGenericChallengeCallParms();
    authCallParms.userId = userid;
}

```

```

authCallParms.parms = authParms;
authCallParms.response = response;
GenericAuthenticateResponse authResponse
= authBinding.authenticateGenericChallenge(authCallParms);
String name = authResponse.FullName;
if (name == null) name = userid;
Console.WriteLine("The user " + name
+ " has successfully authenticated!");
}
catch (SoapException soapEx)
{
    AuthenticationFault fault
    = AuthenticationService.getFault(soapEx);
    if (fault != null)
    {
        // handle the IdentityGuard fault
        System.Console.WriteLine(fault.errorMessage);
        return;
    }
    else
    {
        // not IG fault, handle the generic SoapException
        System.Console.WriteLine(soapEx.Message);
        return;
    }
}
catch (Exception ex)
{
    // handle non-soap exception, e.g., connection error
    System.Console.WriteLine(ex.Message);
    return;
}

```

Mutual authentication

Your organization needs to have confidence in the identity of the user trying to log in. Likewise, users need confidence that they are initiating a transaction with the intended organization. Entrust IdentityGuard provides ways for both parties to authenticate each other; this is called “mutual authentication”.

Mutual authentication can be combined with any other authentication type.

Grid and token serial number and location replay

Grid and token authentication include built-in mechanisms for mutual authentication.

One mechanism 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. Use code like the following:

```
// Get the card serial number
string[] sernum = challengeSet.cardSerialNumbers;
```

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:

```
TokenData[] tokens = tokenChallenge.tokens;

String[] tokenSernum = new String[tokens.Length];
for (int i = 0; i < tokens.Length; i++)
{
    tokenSernum[i] = tokens[i].SerialNumber;
}
```

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](#) 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

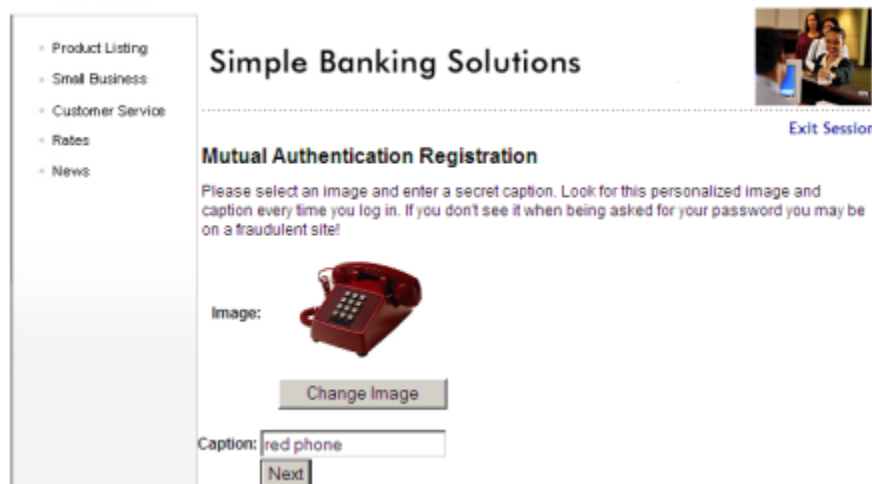


Image and caption replay samples

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

To retrieve authentication secrets, you must do one of the following:

- 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.GetAllSecrets = true;
```

- If you want to retrieve the secrets by specifying their names, use a line like this:

```
authSecParms.GetSecrets = auth_sec_name_array;
```

- Add the `AuthenticationSecretParms` object to `GenericChallengeParms`, which is used in `getGenericChallenge` operation.
- Add the `AuthenticationSecretParms` object to `AuthenticateGenericChallengeCallParms`, which is used in `authenticateGenericChallenge` operation.
- Retrieve the authentication secrets from the generic challenge object.

To retrieve authentication secrets using `getGenericChallenge`

The following sample code retrieves authentication secrets using the `getGenericChallenge` operation:

```
/// the following code retrieves authentication secrets
// from a get generic challenge operation
// Set AuthenticationSecretsParms to get secrets
```



```

AuthenticationSecretParms authSecParms = new AuthenticationSecretParms();
authSecParms.GetAllSecrets = true;
// or specifically retrieve the authentication secret names
// using authSecParms.GetSecrets = new string[] { "IMAGE_SECRET", "CAPTION_SECRET" };
// Add authSecParm to genericChallengeParms
GenericChallengeParms genChallparms = new GenericChallengeParms();
genChallparms.authSecretParms = authSecParms;

// Get a challenge
GetGenericChallengeCallParms callParms = new GetGenericChallengeCallParms();
callParms.userId = userid;
callParms.parms = genChallparms;
GenericChallenge genericChallenge = authBinding.getGenericChallenge(callParms);

// Separate the authentication secrets from the challenge
NameValue[] secrets = genericChallenge.AuthenticationSecrets;
// Add code to display the authentication secrets to the user

```

To retrieve authentication secrets using `authenticateGenericChallenge`

The following sample code retrieves authentication secrets using the `authenticateGenericChallenge` operation:

```

// The following code retrieves the authentication secrets from the authenticate
// generic challenge operation. Set AuthenticationSecretsParms to get secrets
AuthenticationSecretParms authSecParms =
    new AuthenticationSecretParms();
authSecParms.GetAllSecrets = true;
// authenticate challenge response
GenericAuthenticateParms authParms = new GenericAuthenticateParms();
authParms.authSecretParms = authSecParms;
AuthenticateGenericChallengeCallParms callParms =
    new AuthenticateGenericChallengeCallParms();
callParms.userId = userid;
callParms.response = response;
callParms.parms = authParms;

GenericAuthenticateResponse authResponse =
    authBinding.authenticateGenericChallenge(callParms);
//Separate the authentication secrets from the challenge
NameValue[] secrets = authResponse.AuthenticationSecrets;
//Add code to display the authentication secrets to the user

```

Position the code fragments above in your application code. See the sample under [Generic API code sample](#) for a more comprehensive example.

To set or change mutual authentication secrets

The following code sample sets authentication secrets:

```
// The following code adds an authentication secret for a user before
NameValue[] authSecretsToAdd = new NameValue[] {authSecret};
AuthenticationSecretParms authSecretParms = new AuthenticationSecretParms();
authSecretParms.MergeSecrets = true;
authSecretParms.SetSecrets = authSecretsToAdd;

GenericAuthenticateParms genericAuthenticateParms = new GenericAuthenticateParms();
genericAuthenticateParms.authSecretParms = authSecretParms;

Response response = new Response();
AuthenticateGenericChallengeCallParms authenticateGenericChallengeCallParms = new
AuthenticateGenericChallengeCallParms();
authenticateGenericChallengeCallParms.userId = userid;
authenticateGenericChallengeCallParms.response = response;
authenticateGenericChallengeCallParms.parms = genericAuthenticateParms;
// code to authenticate 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](#). 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 [56](#).

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 `GenericAuthenticateParams`.

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

The result of calling `getGenericChallenge` includes a `MachineSecretPolicy`. The `MachineSecretPolicy` structure includes details on 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 this method of authentication goes as follows:

- 1** 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 machine secret
MachineSecret machineSecret = buildMachineSecret();
GenericChallengeParms parms = new GenericChallengeParms();
parms.machineSecret = machineSecret;
GetGenericChallengeCallParms getGenericChallengeCallParms
    = new GetGenericChallengeCallParms();
getGenericChallengeCallParms.userId = userid;
getGenericChallengeCallParms.parms = parms;
GenericChallenge genericChallenge =
    authBinding.getGenericChallenge(getGenericChallengeCallParms);
if (genericChallenge.challengeRequestResult.Equals(
    ChallengeRequestResult.AUTHENTICATED))
{
    // 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.machineSecret;
}
else if (genericChallenge.challengeRequestResult.Equals(
    ChallengeRequestResult.CHALLENGE))
{
    // require second-factor challenge
    // add code to display challenge and retrieve user response
}
else if (genericChallenge.challengeRequestResult.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
GenericAuthenticateParms genAuthParms = new GenericAuthenticateParms();
genAuthParms.registerMachineSecret = registerMachine;
genAuthParms.machineSecret = machineSecret;
```

```

Response response = new Response();
response.response = challengeResponse;
AuthenticateGenericChallengeCallParms authenticateGenericChallengeCallParms
    = new AuthenticateGenericChallengeCallParms();
authenticateGenericChallengeCallParms.userId = userId;
authenticateGenericChallengeCallParms.response= response;
authenticateGenericChallengeCallParms.parms = genAuthParms;
GenericAuthenticateResponse resp = authBinding.authenticateGenericChallenge(
    authenticateGenericChallengeCallParms);

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

```

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 a Web browser only. 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 code sample shows a `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 |
|--|--|
| <code>navigator.appCodeName</code> | Mozilla |
| <code>navigator.appName</code> | Microsoft Internet Explorer |
| <code>navigator.appMinorVersion</code> | ;SP2; |
| <code>navigator.cpuClass</code> | x86 |
| <code>navigator.platform</code> | Win32 |
| <code>navigator.systemLanguage</code> | en-us |
| <code>navigator.userLanguage</code> | en-us |
| <code>navigator.appVersion</code> | 4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322; .NET CLR 2.0.50727) |
| <code>navigator.userAgent</code> | Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322; .NET CLR 2.0.50727) |
| <code>navigator.onLine</code> | true |
| <code>navigator.cookieEnabled</code> | true |
| <code>screen.availHeight</code> | 1170 |
| <code>screen.availWidth</code> | 1600 |
| <code>screen.bufferDepth</code> | 0 |
| <code>screen.colorDepth</code> | 32 |
| <code>screen.deviceXDPI</code> | 96 |

| Property | Value |
|-----------------------------|-------|
| screen.deviceXDPI | 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 |

| Property | Value |
|----------------------------------|--|
| 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

You can deploy signed Java applets or ActiveX controls that leave the browser sandbox and allow the applet to access the system directly. 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)

You can combine these elements with other available elements to create the machine fingerprint.

Web application (server-side)

You can augment the information available through JavaScript and client-side software with data available from the Web application. The following code 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
```

Much of this information is derived from the HTTP headers in the Get request (see page [62](#)). This list includes a port and IP address for the user. Port information may change each time and is not a useful property for a machine fingerprint. You can also use a user's IP address to look up geolocation information.

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 the IP address of the PC.)

Storing and retrieving machine information

For machine authentication, you need to modify your application to first gather the information available as described in “Sources of machine information” starting on page 62. 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.

You must use your application to store the sequence nonce on the client machine, 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" starting on page 62.

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

You can decide how many properties must successfully match 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 ASP.NET and C# 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 IdentityGuard installation under `/IdentityGuard/identityguard<release>/client/sample`.

The relevant files are:

- `/scripts/FlashInterface.js` (Client Side Flash Cookie interface)
- `/scripts/MachineInfo.js` (Client Side Machine Fingerprinting interface)
- `/scripts/LocalStorage.swf` (Flash Object to read/write Flash cookies. Requires Adobe Macromedia Flash Player 9 or greater)

- /C#/* (All the components required to establish a Machine Authentication ASP.NET 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.

```
// Call to machineInfo.js
// Create a MachineSecret object providing a boolean
// parameter set to true which indicates this MachineSecret
// is intended for reading only (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** Find out 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 the HTML/ASP.NET

See `/C#/MachineAuthentication.aspx` 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 collecting of the machine fingerprint and the machine and sequence nonces (if Flash is used). Once the form data has been 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 perform the authentication step. This can be done through an ASP.NET Web Form used to encapsulate the form data and a Page Handler to perform the authentication.

See `/C#/MachineAuthentication.aspx.cs` for implementation details with respect to authenticating the machine information.

- 2 When attempting to return a challenge, if machine authentication is successful and persistent storage is supported, the machine label, nonce, and sequence nonce must 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 `/C#/ApplyResults.aspx` and `/C#/ApplyResults.aspx.cs` for implementation details with respect to creating an HTML form object based on the Machine and Sequence Nonce set within the HTTP Session from {hyperlink}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 {hyperlink}“Step-up authentication” on page 59.

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.

```
// Create the generic challenge parameters:
GenericChallengeParms genericChallengeParms = new
GenericChallengeParms();
string ipAddress = "x.x.x.x";
// Set the ip address of the requesting client.
// This value is used to perform risk based analysis in
// addition to IP information. Machine secrets can also be
// included for risk based analysis.
genericChallengeParms.IPAddress = ipAddress;
// set the security level used for risk based authentication
```

```

genericChallengeParms.SecurityLevel = SecurityLevel.NORMAL;

GetGenericChallengeCallParms callParms
    = new GetGenericChallengeCallParms();
callParms.userId = userid;
callParms.parms = genericChallengeParms;
// NOTE: If special-use IP addresses 0.*.*.* or 127.*.*.* are
// specified within the IPAddress field of the
// GenericChallengeParms, the following call to getGenericChallenge
// results in a INVALID_IP_ADDRESS error
GenericChallenge genericChallenge
    = authBinding.getGenericChallenge(callParms);
ChallengeRequestResult challengeResult
    = genericChallenge.challengeRequestResult;
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. See // 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`.

During the `authenticateGenericChallenge` call, the `challengehistory` value is not used. However, if the given response passes, the current authentication type is added to the input `challengehistory` and returned to the application.

To set the challenge history

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

```
AuthenticationType[] challHistory =  
new AuthenticationType[] { AuthenticationType.GRID };  
genericChallengeParms.ChallengeHistory = 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:

```
genericAuthenticateParms.ChallengeHistory = challHistory;
```

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

```
AuthenticationType[] newChallHistory =  
genericAuthenticateResponse.ChallengeHistory;
```

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.  
  
// Define transaction details  
NameValue[] transactionDetails = new NameValue[2];
```



```

transactionDetails[0] = new NameValue();
transactionDetails[0].Name = "Detail 1";
transactionDetails[0].Value = "Value 1";
transactionDetails[1] = new NameValue();
transactionDetails[1].Name = "Detail 2";
transactionDetails[1].Value = "Value 2";

// Prompt for the userid
Console.Write("Enter UserId: ");
String userid = System.Console.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.AuthenticationType = AuthenticationType.TOKENRO;
// Include the transaction details to be signed.
genericChallengeParms.transactionDetails = transactionDetails;
callParms.parms = genericChallengeParms;
callParms.userId = userid;

// call getGenericChallenge, the transaction details will attempt to be
// delivered
GenericChallenge genericChallenge = null;
try
{
    genericChallenge =
        authBinding.getGenericChallenge(callParms);
}
catch (SoapException soapEx)
{
    AuthenticationFault fault =
        AuthenticationService.getFault(soapEx);
    if (fault != null)
    {
        // handle the IdentityGuard fault
        System.Console.WriteLine(fault.errorMessage);
        return;
    }
    else

```

```

    {
        // not IG fault, handle the generic SoapException
        System.Console.WriteLine(soapEx.Message);
        return;
    }
}
catch (Exception ex)
{
    // handle non-soap exception, e.g., connection error
    System.Console.WriteLine(ex.Message);
    return;
}

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

    TokenSetParser parser =
    new TokenSetParser(genericChallenge.TokenChallenge.tokens);
    foreach (string t in parser.getTokenSets())
    {
        foreach (TokenData tok in parser.getTokensForSet(t))
        {
            if (genericChallenge.type.Equals(AuthenticationType.TOKENRO)
                && tok.DeliveryStatus == 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.
    Console.WriteLine("The transaction could not be delivered to any tokens.");
    return;
}

```

```

Console.WriteLine("Please enter the confirmation code generated by your token: ");
String[] userResponse = { Console.ReadLine() };

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

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

    AuthenticateGenericChallengeCallParms authCallParms
    = new AuthenticateGenericChallengeCallParms();
    authCallParms.userId = userid;
    authCallParms.parms = authParms;
    authCallParms.response = response;
    GenericAuthenticateResponse authResponse
    = authBinding.authenticateGenericChallenge(authCallParms);

    String name = authResponse.FullName;
    if (name == null) name = userid;
    Console.WriteLine("The user " + name
    + " has successfully confirmed the transaction details!");
}
catch (SoapException soapEx)
{
    AuthenticationFault fault
    = AuthenticationService.getFault(soapEx);
    if (fault != null)
    {
        // handle the IdentityGuard fault
        System.Console.WriteLine(fault.errorMessage);
        return;
    }
    else
    {
        // not IG fault, handle the generic SoapException
        System.Console.WriteLine(soapEx.Message);
        return;
    }
}

```

```
    }  
    }  
    catch (Exception ex)  
    {  
        // handle non-soap exception, e.g., connection error  
        System.Console.WriteLine(ex.Message);  
        return;  
    }  
}
```

Chapter 4:

Administration tasks

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

Administration setup and login

For a client application to use the Administration API, the client must have administrator credentials. Its administrator role must include any 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 login the admin ID
// Create the URL where the administration service is located:
try
{
    AdminService adminBinding = new AdminService();
    adminBinding.CookieContainer = new CookieContainer();
    adminBinding.Url = adminServiceURL;
    LoginParms loginParms = new LoginParms();
    loginParms.adminId = admin_id;
    loginParms.password = admin_pswd;
    LoginCallParms callParms = new LoginCallParms();
    callParms.parms = loginParms;
    LoginResult loginResult = adminBinding.login(callParms);
    if (loginResult.state.Equals(LoginState.COMPLETE))
    {
        // login complete
    }
    else if
        (loginResult.state.Equals(LoginState.NEED_SECOND_FACTOR))
    {
```

```
LoginChallenge loginChallenge = loginResult.loginChallenge;
// add code to handle second-factor authentication
// using GRID or TOKENCR/TOKENRO

}
else
{
// login failed, add code to handle login failure
}
}
catch (Exception e)
{
// add code to handle login failure
}
```

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.Group = groupid;
userParms.Aliases = aliases;
userParms.ContactInfoList = contactinfoList;
userParms.AuthenticationSecrets = authSecrets;
userParms.QaSecrets = qaSecrets;
userParms.SharedSecrets = sharedSecrets;
UserCreateCallParms callParms = new UserCreateCallParms();
callParms.userid = userid;
callParms.parms = 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.Userid = olduserid;
userParms.Group = "default";
userParms.Aliases = aliases;
userParms.ContactInfoList = contactInfoList;
userParms.AuthenticationSecrets = authenticationSecrets;
userParms.QaSecrets = qaSecrets;
userParms.SharedSecrets = sharedSecrets;

UserSetCallParms callParms = new UserSetCallParms();
callParms.userid = userid;
callParms.parms = userParms;

adminBinding.userSet(callParms);
```

Get grid contents for a user

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

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

// set the flag to get grid content
CardGetParms cardGetParms = new CardGetParms();
cardGetParms.getGrid = true;
callParms.parms = cardGetParms;

UserCardFilter userCardFilter = new UserCardFilter();
userCardFilter.SerialNumber = serialNumber;
callParms.filter = userCardFilter;

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

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

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 reproduced grid cards” on page 81](#), 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

```
UserCardParms userCardParms = new UserCardParms();

userCardParms.Comment = comment;
// set card state to HOLD_PENDING
userCardParms.State = State.HOLD_PENDING;

UserCardCreateCallParms callParms = new UserCardCreateCallParms();
callParms.userid = userid;
callParms.parms = 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.SerialNumber = cardSerialNumber;

UserCardParms userCardParms = new UserCardParms();
userCardParms.State = State.PENDING;

UserCardSetCallParms callParms = new UserCardSetCallParms();
callParms.userid = userid;
callParms.filter = userCardFilter;
callParms.parms = userCardParms;
int numCardsUpdated = adminBinding.userCardSet(callParms);

try
{
    // authenticate the user using authentication service
    // see grid authentication; auth service API example
```



```

}
catch (Exception e)
{
    // if authentication failed, change the card state back
    // to HOLD_PENDING
    userCardParms.State = State.HOLD_PENDING;
    callParms.parms = userCardParms;
    adminBinding.userCardSet(callParms);
}

```

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.Lifetime = 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 cards 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.Supersede = 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 preproduced 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.NumCards = numcards;
preproducedCardParms.Group = group;

PreproducedCardCreateCallParms callParms = new PreproducedCardCreateCallParms();

```

```
callParms.parms = preproducedCardParms;
PreproducedCardCreateResult result =
    adminBinding.preproducedCardCreate(callParms);
```

Assign preproduced grid card

```
// The following code assigns a preproduced card to a user.
UserCardParms userCardParms = new UserCardParms();

userCardParms.SerialNumber = sernum;
userCardParms.Lifetime = lifetime;
// set the card state to HOLD_PENDING
userCardParms.State = State.HOLD_PENDING;

UserCardCreateCallParms callParms = new UserCardCreateCallParms();
callParms.userid = userid;
callParms.parms = userCardParms;
adminBinding.userCardCreate(callParms);
```

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 can 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 an OTP. 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
UserOTPParms userOTPParms = new UserOTPParms();

// The lifetime of the OTP in milliseconds
userOTPParms.Lifetime = lifetime;

UserOTPCreateCallParms callParms = new UserOTPCreateCallParms();
callParms.userid = userid;
```

```
callParms.parms = userOTPParms;
```

```
UserOTPInfo[] otp = adminBinding.userOTPCreate(callParms);
```

Retrieve an OTP

```
// The following code retrieves the OTP for a given user ID
```

```
UserOTPGetCallParms callParms = new UserOTPGetCallParms();
```

```
callParms.userid = userid;
```

```
callParms.filter = new UserOTPFilter();
```

```
UserOTPInfo[] otp = adminBinding.userOTPGet(callParms);
```

```
// If the user did not have at least one OTP, a fault would have been thrown,
```

```
// so it is guaranteed that the array is non-null and non-zero
```

```
string firstOtpString = otp[0].OTP;
```

```
// 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.userid = userid;
```

```
ContactInfo[] infos = adminBinding.userContactInfoGet(contactInfoParms);
```

```
string contactInfoLabel = null;
```

```
if (infos != null)
```

```
{
```

```
    foreach (ContactInfo info in infos)
```

```
    {
```

```
        // find the valid delivery config
```

```
        if (!info.deliveryConfigLabel.ToLower().Equals("none"))
```

```
        {
```

```
            contactInfoLabel = info.contactInfoLabel;
```

```
            break;
```

```
        }
```

```
    }
```

```
}
```

```
if (contactInfoLabel == null)
```

```
{
```

```
    // Return some kind of error detailing the fact that no
```

```

// contact information had been configured for the user, and
// therefore the OTP could not be delivered
Console.WriteLine("No delivery mechanisms found.");
return;
}
// Deliver the OTP using one of the delivery mechanisms assigned
// within the user's contact info list.
UserOTPParms otpdeliverparms = new UserOTPParms();
otpdeliverparms.contactInfoLabel = new string[1] {contactInfoLabel };

UserOTPDeliverCallParms callparms = new UserOTPDeliverCallParms();
callparms.userid = userid;
callparms.parms = otpdeliverparms;
adminBinding.userOTPDeliver(callparms);

```

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 retrieve the delivery configuration list:

```

// this sample retrieves the delivery config list
NameValue[] deliveryConfigListCallParms = new NameValue[0];
DeliveryConfigInfoList configList =
    adminBinding.deliveryConfigList(deliveryConfigListCallParms);

// retrieve the delivery config
DeliveryConfigInfo[] configs = configList.deliveryConfigInfo;
string deliveryConfigLabel = null;
if (configs.Length > 0)
{
    // take the first one
    deliveryConfigLabel = configs[0].configLabel;
}

```

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 contactinfo with a specified label,  
// value and config label  
UserContactInfoParms userContactInfoParms = new UserContactInfoParms();  
userContactInfoParms.contactInfoLabel = contactInfoLabel;  
userContactInfoParms.value = contactInfoValue;  
userContactInfoParms.deliveryConfigLabel = deliveryConfigLabel;  
// instead of specifying particular contactinfo,  
// you can use the default one  
// parms.defaultContactInfo = true;  
UserContactInfoCreateCallParms userContactInfoCreateCallParms  
    = new UserContactInfoCreateCallParms();  
userContactInfoCreateCallParms.userid = userid;  
userContactInfoCreateCallParms.parms = 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.value = contactInfoValue;  
userContactInfoParms.defaultContactInfo = true;  
// instead of specifying particular contactinfo,  
// you can use the default one  
// parms.defaultContactInfo = true;  
UserContactInfoSetCallParms userContactInfoSetCallParms =  
    new UserContactInfoSetCallParms();  
userContactInfoSetCallParms.userid = userid;  
userContactInfoSetCallParms.label = contactInfoLabel;  
userContactInfoSetCallParms.parms = userContactInfoParms;  
adminBinding.userContactInfoSet(userContactInfoSetCallParms);
```

Delete contact information

```
// The following code deletes user contactinfo
```

```
UserContactInfoDeleteCallParms userContactInfoDeleteCallParms =  
    new UserContactInfoDeleteCallParms();  
userContactInfoDeleteCallParms.userid = userid;  
userContactInfoDeleteCallParms.labels = new string[]{contactinfoLabel};  
adminBinding.userContactInfoDelete(userContactInfoDeleteCallParms);
```

Retrieve contact information

```
// This sample retrieves a user contactinfo  
UserContactInfoGetCallParms userContactInfoGetCallParms  
    = new UserContactInfoGetCallParms();  
userContactInfoGetCallParms.userid = userid;  
userContactInfoGetCallParms.labels  
    = new string[] { contactinfoLabel };  
ContactInfo[] userContactInfo = adminBinding.userContactInfoGet(  
    userContactInfoGetCallParms);  
if (userContactInfo != null && userContactInfo.Length == 1)  
{  
    Console.WriteLine("Address for Work Email is: " + userContactInfo[0].value);  
}
```

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 you 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.State = State.HOLD_PENDING;  
UserTokenAssignCallParms callParms = new UserTokenAssignCallParms();  
callParms.userid = userid;  
callParms.serialNumber = sernum;  
// if not specified, the token vendor defaults to the default token vendor
```

```
callParms.vendorId = tokenVendor;
callParms.parms = userTokenParms;
adminBinding.userTokenAssign(callParms);
```

Modify a token

```
// The following code changes the state of a user token to PENDING
UserTokenFilter userTokenFilter = new UserTokenFilter();
userTokenFilter.SerialNumber = sernum;
// if not specified, the token vendor defaults to the default token vendor
userTokenFilter.VendorId = tokenVendor;
UserTokenParms userTokenParms = new UserTokenParms();
userTokenParms.State = State.PENDING;

UserTokenSetCallParms callParms = new UserTokenSetCallParms();
callParms.userid = userid;
callParms.filter = userTokenFilter;
callParms.parms = userTokenParms;
adminBinding.userTokenSet(callParms);
```

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.resetToken = true;
// Some token implementations require one or two token responses.
// In this case, the resetToken flag is ignored.
userTokenParms.resetTokenResponse1 = resetResps1;
userTokenParms.resetTokenResponse2 = resetResps2;

UserTokenSetCallParms userTokenSetCallParms = new UserTokenSetCallParms();
userTokenSetCallParms.userid = userid;
UserTokenFilter filter = new UserTokenFilter();
// This filter indicates which token to unlock.
// This filter must specify a single token.
filter.SerialNumber = sernum;
// if not specified, the token vendor defaults to the default token vendor
```

```
filter.VendorId(tokenVendor);
userTokenSetCallParms.filter = filter;

userTokenSetCallParms.parms = 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.userid = userid;
UserTokenFilter filter = new UserTokenFilter();
// The filter indicates which token is to be unlocked.
// This filter must specify a single token.
filter.SerialNumber = sernum;
// if not specified, the token vendor defaults to the default token vendor
filter.VendorId = tokenVendor;
userTokenUnlockCallParms.filter = filter;
// The unlock challenge is generated by the token.
userTokenUnlockCallParms.challenge = challenge;

UserTokenUnlockResult result =
    adminBinding.userTokenUnlock(userTokenUnlockCallParms);
// Then communicate the unlock code to the user.
// The user must enter this number to unlock the token.
string unlockCode = result.unlockCode;
```

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.

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

UserPINParms userPINParms = new UserPINParms();

// The lifetime of the PIN in milliseconds
userPINParms.Lifetime = lifetime;

// replace the existing temporary PIN
userPINParms.Force = true;

UserPINCreateCallParms callParms = new UserPINCreateCallParms();
callParms.userid = userid;
callParms.parms = userPINParms;

adminBinding.userPINCreate(callParms);
```

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
UserPINGetCallParms callParms = new UserPINGetCallParms();
callParms.userid = userid;
UserPINInfo pin = adminBinding.userPINGet(callParms);
string[] pinValues = pin.PIN;
```

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.Lifetime = lifetime;
userPINParms.MaxUses = max;

UserPINSetCallParms callParms = new UserPINSetCallParms();
callParms.userid = userid;
callParms.parms = 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 show you how 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.AutoGenerate = true;
// Optionally, you can also specify the PVN value
// userPVNCreateParms.PVN = new string("<PVN value>");

// Require the user to change the PVN value
parms.ChangeRequired = true;
UserPVNCreateCallParms userPVNCreateCallParms =new UserPVNCreateCallParms();
userPVNCreateCallParms.userid = userid;
userPVNCreateCallParms.parms = 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.userid = userid;
UserPVNInfo pvnInfo = adminBinding.userPVNGet(userPVNGetCallParms);
string pvnString = pvnInfo.PVN;
```

Update a user PVN

```
// The following code updates a PVN for a given user
UserPVNSetParms userPVNSetParms = new UserPVNSetParms();
userPVNSetParms.PVN = newPVN;
UserPVNSetCallParms userPVNSetCallParms =new UserPVNSetCallParms();
```

```
userPVNSetCallParms.userid = userid;
userPVNSetCallParms.parms = userPVNSetParms;
adminBinding.userPVNSet(userPVNSetCallParms);
```

Delete a user PVN

```
// The following code deletes a PVN for a given user
UserPVNDeleteCallParms userPVNDeleteCallParms = new UserPVNDeleteCallParms();
userPVNDeleteCallParms.userid = 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].Name = questionArray[i];
    qaPairs[i].Value = answerArray[i];
}

UserParms userParms = new UserParms();
userParms.QaSecrets = qaPairs;

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

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:

```
UserLockoutParms userLockoutParms = new UserLockoutParms();
userLockoutParms.Clearlockout = true;
userLockoutParms.IncreaseLockout = false;
userLockoutParms.AuthenticatorLockoutIds = null;

// The following code unlocks the user
UserParms userParms = new UserParms();
userParms.lockoutParms = userLockoutParms;

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

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 attempt to access the Web site.

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

UserSetCallParms callParms = new UserSetCallParms();
callParms.userid = userid;
callParms.parms = 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.userid = userid;
// you can only delete the machine secret by specifying the machine nonce
userMachineSecretDeleteCallParms.machineNonce = 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 soon expire.

```
// Return the cards that will expire in 10 days
UserFilter userFilter = new UserFilter();

// set the expiry end date to 10 days from now
DateTime expiryDate = DateTime.Today;
expiryDate = expiryDate.AddDays(10);
userFilter.expireEndDate = expiryDate;

// return 50 cards for this call. The default value for max return is 100.
userFilter.maxReturn = 50;

// optionally, starting from the nextUser value of the previous search
userFilter.nextUser = userToStart;

UserCardListCallParms callParms = new UserCardListCallParms();
callParms.filter = userFilter;
UserCardListResult result = adminBinding.userCardList(callParms);
```

```
// the nextUser value can be used at subsequent search
string nextUser = cards.nextUser;
```

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:

```
// Returns the number of preproduced cards for the given groups
int numberOfCards = 0;
string nextCard = "-1";

while(nextCard != null)
{
    PreproducedCardFilter preproducedCardFilter = new PreproducedCardFilter();
    preproducedCardFilter.groups = groupids;

    // start from the next card of previous search
    if (!nextCard.Equals("-1"))
    {
        preproducedCardFilter.nextCard = nextCard;
    }

    PreproducedCardListCallParms callParms
    = new PreproducedCardListCallParms();
    callParms.filter = preproducedCardFilter;

    PreproducedCardListResult result =
    adminBinding.preproducedCardList(callParms);
    // get the new next card serial number
    nextCard = result.nextCard;
    numberOfCards = numberOfCards + result.cards.Length;
}
```

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:

```
// This code returns the number of unassigned tokens for the given groups
```

```

int tokenNumber = 0;
string nextTokenSerialNumber = "-1";
string nextTokenVendorId = null;
while (nextTokenSerialNumber != null)
{
    TokenFilter tokenFilter = new TokenFilter();
    tokenFilter.groups = groupid;
    // start from the next token of previous search
    if (!nextTokenSerialNumber.Equals("-1"))
    {
        tokenFilter.nextTokenSerialNumber = nextTokenSerialNumber;
        tokenFilter.nextTokenVendorId = nextTokenVendorId;
    }
    TokenListCallParms callParms = new TokenListCallParms();
    callParms.filter = tokenFilter;
    TokenListResult result = adminBinding.tokenList(callParms);
    // get the new next token serial number
    nextTokenSerialNumber = result.nextTokenSerialNumber;
    nextTokenVendorId = result.nextTokenVendorId;
    tokenNumber = tokenNumber + result.tokens.Length;
}

```

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

```

// Returns the assigned cards that have never been used,
// e.g., card state is PENDING or HOLD_PENDING
UserFilter userFilter = new UserFilter();

State[] states = new State[2];
states[0] = State.PENDING;
states[1] = State.HOLD_PENDING;
userFilter.states = states;

// return 50 cards for this call. The default value for max return is 100.
userFilter.maxReturn = 50;

```

```
// optionally, starting from the nextUser value of the previous search
userFilter.nextUser = userToStart;

UserCardListCallParms callParms = new UserCardListCallParms();
callParms.filter = userFilter;
UserCardListResult result = adminBinding.userCardList(callParms);

// the nextUser value can be used at subsequent search
String nextUser = result.nextUser;
UserCardInfo[] cardInfoArray = result.cards;
```

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
DateTime tokenLastUsedEndDate = DateTime.Today;
tokenLastUsedEndDate = tokenLastUsedEndDate.AddDays(-14);
userFilter.tokenLastUsedEndDate = tokenLastUsedEndDate;

// return 50 tokens for this call. The default value for max return is 100.
userFilter.maxReturn = 50;

// optionally, start from the nextUser value of the previous search
userFilter.nextUser = userToStart;

UserTokenListCallParms callParms = new UserTokenListCallParms();
callParms.filter = userFilter;

// return the tokens
UserTokenListResult result = adminBinding.userTokenList(callParms);

// the nextUser value can be used at subsequent search
string nextUser = result.nextUser;
UserTokenInfo[] tokenInfoArray = result.tokens;
```

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 Enterprise".

```
public void userSmartCredentialCreate()
{
    try
    {
        // The smart credential parameter
        UserSmartCredentialParms parms = new UserSmartCredentialParms();

        // The name of the smart credential definition to use
        parms.DefinitionId = "PIV Enterprise";

        // The smart credential creation parameter
        UserSmartCredentialCreateCallParms createParms =
new        UserSmartCredentialCreateCallParms();

        // The name of the user who will receive the smart credential
        createParms.userid = "userid";
        createParms.parms = parms;

        smartid = adminBinding.userSmartCredentialCreate(createParms);

        System.Console.WriteLine("SmartCredential for user " + userid + " has been
created. The SmartCredential ID is" + smartid);
    }
    catch (SoapException soapEx)
    {
        AdminServiceFault fault = AdminService.getFault(soapEx);
        if (fault != null)
```

```

    {
        // handle the IdentityGuard fault
        System.Console.WriteLine(fault.errorMessage);
        return;
    }
    else
    {
        // not IG fault, handle the generic SoapException
        System.Console.WriteLine(soapEx.Message);
        return;
    }
}
catch (Exception ex)
{
    // handle non-soap exception, e.g., connection error
    System.Console.WriteLine(ex.Message);
    return;
}
}

```

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.

```

public void userSmartCredentialApprove()
{
    // The name of the user whose smart credentials will be approved.
    String userID = "userid";

    // The ID of the smart credentials to approve.
    String id = "ET5080888";

    UserSmartCredentialParms parms = new UserSmartCredentialParms();

    parms.Approved = true;

    UserSmartCredentialSetCallParms setparms =
new    UserSmartCredentialSetCallParms();
    setparms.userid =userID;
    setparms.Id = id;
    setparms.parms = parms;
}

```

```

adminBinding.userSmartCredentialSet(setparms);

System.Console.WriteLine("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, a smart credential ID must be provided to the API. In this example, 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 "IGPM1". Because a smart card reader will be used, the print operation is "EXTERNAL_ENCODE".

```

public void userSmartCredentialIssue()
{
    // The name of the user whose smart credentials will be approved.
    String userID = "userid";

    // The ID of the smart credentials to issue.
    String id = "ET5080888";

    // 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.
    UserSmartCredentialIssueOp[] issueop = new UserSmartCredentialIssueOp[1];

    // Set the print module to encode to a card using an external printer
    // or smart card reader.

    issueop[0] = UserSmartCredentialIssueOp.EXTERNAL_ENCODE;

    UserSmartCredentialIssueCallParms issueparms =
new    UserSmartCredentialIssueCallParms();
    issueparms.userid = userID;
    issueparms.id = id;
    issueparms.printModule = "IGPM1";
    issueparms.operation = issueop;

    // Issue the smart credentials
    adminBinding.userSmartCredentialIssue(issueparms);

    System.Console.WriteLine("Sent issuance request to Print Module " +
issueparms.printModule + " 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 preceding API (see “Issue smart credentials ” on page 99) 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 `IdentityGuardAdminServiceV9API.html` under the `<IG_HOME>\client\C#\doc` folder. They include `QUEUED`, `ENCODE_DONE`, `ENCODE_ERROR`, and others. Your application should respond to each of these states.

```

public void userSmartCredentialCheckStatus()
{
    UserSmartCredentialGetParms getParms = new UserSmartCredentialGetParms();
    getParms.getDetails = true;
    getParms.getRevocationDetails = true;

    UserSmartCredentialGetCallParms parms = new UserSmartCredentialGetCallParms();
    parms.Id = "ET0580888";
    parms.userid = "userid";
    parms.parms = getParms;

    UserSmartCredentialInfo smartCredential
    =    adminBinding.userSmartCredentialGet(parms);

    SmartCredentialState state = smartCredential.State;

    object issuestate = smartCredential.IssueState;

    if (issuestate != null)
    {
        System.Console.WriteLine("Issue State: " + issuestate);
    }
    else
    {
        System.Console.WriteLine("Issue State could not be determined");
    }

    System.Console.WriteLine("User " + userid + " has smart credentials " + parms.Id +
    " in state " + state);
}

```

}

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 [IdentityGuardAdminServiceV9API.html](#) under the `<IG_HOME>\client\C#\doc` folder to see the available search criteria for the `UserFilter` and the `UserSmartCredentialInfo` classes.

```
public void userSmartCredentialModify()
{
    // Get all smart credentials for this user.
    UserFilter filter = new UserFilter();
    filter.userid = "userid";

    UserSmartCredentialListCallParms parms = new UserSmartCredentialListCallParms();
    parms.filter = filter;

    UserSmartCredentialListResult result =
adminBinding.userSmartCredentialList(parms);

    UserSmartCredentialInfo[] smartCredentials = result.SmartCredentials;

    if (smartCredentials == null || smartCredentials.Length == 0) {
System.Console.WriteLine("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 == tmp.State ) {
            info = tmp;
        }
    }
}
```

```

        break;
    }
}

if (info == null) {
    System.Console.WriteLine("User '" + userid + "' has no unassigned smart
credentials.");
    return;
}

System.Console.WriteLine("Found smart credentials " + info.Id +
    " for user " + info.Userid +
    " in state: " + info.State);

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

```

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 void userSmartCredentialSetEnrollmentValues(UserSmartCredentialInfo scinfo)
{
    // Create a list of enrollment values to set or modify
    ArrayList tplist = new ArrayList();

    NameValue tp = new NameValue();
    tp.Name = "firstname";
    tp.Value = "John";
    tplist.Add(tp);
    tp = new NameValue();
    tp.Name = "lastname";
    tp.Value = "Smith";
    tplist.Add(tp);
    tp = new NameValue();
    tp.Name = "emailaddress";
    tp.Value = "jsmith@entrust.com";
    tplist.Add(tp);

    NameValue[] tpparray = new NameValue[tplist.Count];
}

```

```

tplist.CopyTo(tparray,0);

UserSmartCredentialParms parms = new UserSmartCredentialParms();

// The above values were not encrypted here, so we request IG to encrypt.
parms.RawEnrollmentValues = true;

// Set the enrollment values into the request
parms.EnrollmentValues = tparray;

// Send the request
UserSmartCredentialSetCallParms setparms =
new UserSmartCredentialSetCallParms();
setparms.Id = scinfo.Id;
setparms.userid = scinfo.Userid;
setparms.parms = parms;
adminBinding.userSmartCredentialSet(setparms);
}

```

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

```
public void userSmartCredentialHold()
{
    // Return 100 users at most
    UserFilter filter = new UserFilter();
    filter.nextUser = "1";
    filter.maxReturn= 100;

    UserSmartCredentialListCallParms parms = new UserSmartCredentialListCallParms();
    parms.filter = filter;

    UserSmartCredentialListResult result
=    adminBinding.userSmartCredentialList(parms);
    UserSmartCredentialInfo[] smartCredentials = result.SmartCredentials;

    // As an example, find any ACTIVE smart credential and set it to HOLD.
    SmartCredentialState state = SmartCredentialState.ACTIVE;

    foreach( UserSmartCredentialInfo scinfo in smartCredentials )
    {
        if(state == scinfo.State)
        {
            System.Console.WriteLine("User " + scinfo.Group + "/" + scinfo.UserName +
                " has smart credentials " + scinfo.Id +
                " in state " + scinfo.State);

            // set to HOLD
            UserSmartCredentialParms userparms = new UserSmartCredentialParms();
            userparms.State = SmartCredentialState.HOLD;
            UserSmartCredentialSetCallParms p = new
            UserSmartCredentialSetCallParms();
            p.userid = scinfo.Userid;
            p.Id = scinfo.Id;
            p.parms = userparms;
            adminBinding.userSmartCredentialSet(p);

            // confirm the smart credential is on HOLD
            UserSmartCredentialGetCallParms parmsget = new
            UserSmartCredentialGetCallParms();
            parmsget.userid = scinfo.Userid;
            parmsget.Id = scinfo.Id;

            UserSmartCredentialInfo scinfoNew = adminBinding.userSmartCredentialGet(
            parmsget);
        }
    }
}
```



```
System.Console.WriteLine("User " + scinfonew.Group + "/" +
    scinfonew.UserName +
        " has smart credentials " + scinfonew.Id +
        " in state " + scinfonew.State);

// set back to ACTIVE before leaving
userparms.State = SmartCredentialState.ACTIVE;
UserSmartCredentialSetCallParms p2 = new
UserSmartCredentialSetCallParms();
    p2.userid = scinfonew.UserId;
    p2.Id = scinfonew.Id;
    p2.parms = userparms;
    adminBinding.userSmartCredentialSet(p2);
}
}
```

Chapter 5:

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

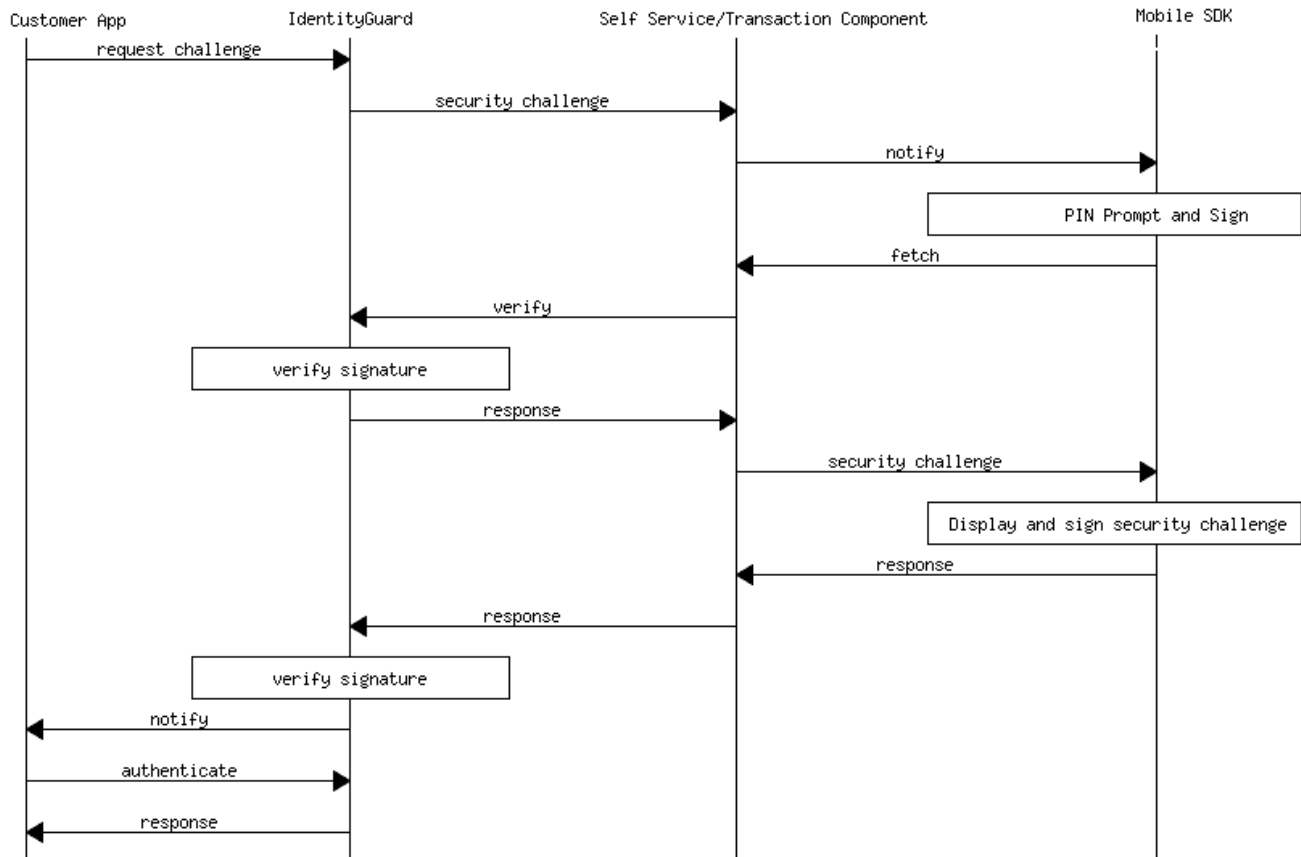
This chapter describes how the Entrust IdentityGuard Authentication and Administration APIs can be used to integrate this Entrust IdentityGuard Mobile Smart Credential functionality with existing applications.

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:

- The **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 .NET 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(AuthenticationService binding, string userid, NameValue[]
transactionDetails)
    {
        GenericChallengeParms gParms = new GenericChallengeParms();
        gParms.AuthenticationType = AuthenticationType.CERTIFICATE;
        gParms.requireCertificateDelivery = true;
        gParms.performCertificateDelivery = true;

        // if the transactionDetails are null then an authentication
        // security challenge will be performed. Otherwise, a transaction
        // verification security challenge will be performed.
        gParms.transactionDetails = transactionDetails;

        GetGenericChallengeCallParms getGenericChallengeParms = new
GetGenericChallengeCallParms();
        getGenericChallengeParms.userId = userid;
        getGenericChallengeParms.parms = gParms;

        GenericChallenge challenge =
binding.getGenericChallenge(getGenericChallengeParms);

        GenericAuthenticateParms aParms = new GenericAuthenticateParms();
        aParms.AuthenticationType = AuthenticationType.CERTIFICATE;
        aParms.transactionId = challenge.CertificateChallenge.TransactionId;
        aParms.transactionDetails = 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
            {
                AuthenticateGenericChallengeCallParms callParms = new
AuthenticateGenericChallengeCallParms();
                callParms.userId = userid;
                callParms.response = new Response();
                GenericAuthenticateResponse response =
binding.authenticateGenericChallenge(callParms);

                // we were successful. We could look at the signed security
                // challenge but for this sample return to the caller
            }
            catch (SoapException soapEx)
            {
                AuthenticationFault fault = AuthenticationService.getFault(soapEx);

```



```

        if (fault != null && fault.ErrorCode == ErrorCode.NO_RESPONSE)
        {
            // continue if we received NO_RESPONSE.
            // Otherwise return the error to the caller
            System.Console.WriteLine(fault.errorMessage);
            Thread.sleep(1000);
            continue;
        }
        else
        {
            // not IG fault, handle the generic SoapException
            System.Console.WriteLine(soapEx.Message);
            throw;
        }
    }
}
}
}

```

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(AdminService binding, string
userid, string smartCredentialId, string digest, string digestAlg, string summary)
{
    // for this sample, we assume that the caller has generated
    // the digest of data to be signed and a summary description.
    UserSmartCredentialSignParms startParms = new UserSmartCredentialSignParms();
    startParms.digest = digest;
    startParms.digestHashAlg = digestAlg;
    startParms.summary = summary;

    UserSmartCredentialSignStartCallParms startCallParms = new
UserSmartCredentialSignStartCallParms();
    startCallParms.userid = userid;
    startCallParms.id = smartCredentialId;
    startCallParms.parms = startParms;

    UserSmartCredentialSignStartResult startResult =
binding.userSmartCredentialSignStart(startCallParms);

    // 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
        {
            UserSmartCredentialSignCompleteCallParms completeCallParms = new
UserSmartCredentialSignCompleteCallParms();
            completeCallParms.userid = userid;
            completeCallParms.id = smartCredentialId;
            completeCallParms.transactionId = startResult.transactionId;
            completeCallParms.cancelTransaction = null;

            UserSmartCredentialSignCompleteResult result =
binding.userSmartCredentialSignComplete(completeCallParms);

            // the result includes both a the signature of the digest
            // and the
            return result;
        }
        catch (SoapException soapEx)
        {
            AdminServiceFault fault = AdminService.getFault(soapEx);
            if (fault != null && fault.ErrorCode == ErrorCode.NO_RESPONSE)
            {
                // continue if we received NO_RESPONSE.
                // Otherwise return the error to the caller
            }
        }
    }
}

```

```

        System.Console.WriteLine(fault.errorMessage);
        Thread.Sleep(1000);
        continue;
    }
    else
    {
        // not IG fault, handle the generic SoapException
        System.Console.WriteLine(soapEx.Message);
        throw;
    }
}
}
}
}

```

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 Entelligence 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 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 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 anonymous security challenge.
 */
public void authenticateAnonymousCertificate() throws Exception
{
    GenericChallengeParms gcParms = new GenericChallengeParms();
    gcParms.authenticationType = AuthenticationType.CERTIFICATE;
    // IdentityGuard will set these into the URL in the QR code.
    // The Entrust Mobile Smart Credential application displays them.
    gcParms.ApplicationName = "AnyBank";
    string smartCredentialChallengeSummary =
    "Please authenticate to access your account on AnyBank";
    gcParms.smartCredentialChallengeSummary = smartCredentialChallengeSummary;

    // The default QR code size is 250.
    gcParms.anonymousCertChallengeQRCodeSize = 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/";
    string CALLBACK_QUERY =
    "?action=anonchallenge&txnid=<TRANSACTIONID>&status=<STATUS>&userid=<USERID>";
    gcParms.anonymousCertChallengeCallback = 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.group = null;

gAnonParms.parms = gcParms;
GenericChallenge anonchall =
authBinding.getAnonymousCertChallenge(gAnonParms);

// Get the QR code to send to the user for them to scan
byte[] qrCode = anonchall.QRCode;

// 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.
CertificateChallenge cc = anonchall.CertificateChallenge;
DateTime created = cc.CreateDate;
int lifetime = cc.Lifetime;

while (true) {
    String transactionId = anonchall.TransactionId;
    GetAnonymousCertChallengeCallParms aparms =
        new GetAnonymousCertChallengeCallParms();
    aparms.transactionID = transactionId;
    GenericChallenge challenge =
authBinding.getAnonymousCertChallenge(aparms);
    ChallengeRequestResult result = challenge.ChallengeRequestResult;
    if
(result.getValue().Equals(ChallengeRequestResult.AUTHENTICATED.getValue())) {
        Console.Out.WriteLine("authenticated a user " + challenge.username);
        Console.Out.WriteLine("authenticated a user in group " +
challenge.group);
        break;
    }
}
}

```

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 normally 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 identify 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 6:

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 this 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(AdminService binding, String userid,
String serialNumber)
{
    UserTokenFilter filter = new UserTokenFilter();
    filter.SerialNumber = serialNumber;

    UserTokenParms parms = new UserTokenParms();
    parms.activationDelivery = "tokenemaildelivery"; // this delivery has
    // to be defined in identityguard.properties
    parms.activationContact = "work email"; // the user has to have a
    // contact of this type
    parms.activationAddress = "ssm.yourcorp.com:8445/igst"; // this is the
    // address of the IG SSM server
    parms.generateQRCode = true;

    UserTokenActivateCallParms callParms = new UserTokenActivateCallParms();
    callParms.userid = userid;
    callParms.filter = filter;
    callParms.parms = parms;

    UserTokenActivateResult result = binding.userTokenActivate(callParms);

    // 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.
bool isActivationComplete(AdminService binding, String userid, String
serialNumber)
{
    UserTokenFilter filter = new UserTokenFilter();
    filter.SerialNumber = serialNumber;

    UserTokenGetCallParms callParms = new UserTokenGetCallParms();
    callParms.userid = userid;
    callParms.filter = filter;

    UserTokenInfo[] tokens = binding.userTokenGet(callParms);

    if (tokens.Length != 1)
    {
```

```

        throw new Exception("Unexpected number of tokens returned.");
    }

    return tokens[0].ActivationState == "ACTIVATED";
}

void completeActivation(AdminService binding, String userid, String serialNumber,
String registrationCode)
{
    UserTokenFilter filter = new UserTokenFilter();
    filter.SerialNumber = serialNumber;

    IdentityGuardAdminServiceV11API.NameValue regCode = new
IdentityGuardAdminServiceV11API.NameValue();
    regCode.Name = "registrationCode";
    regCode.Value = registrationCode;

    UserTokenActivateCompleteCallParms callParms = new
UserTokenActivateCompleteCallParms();

    callParms.userid = userid;
    callParms.filter = filter;
    callParms.activationParms = new IdentityGuardAdminServiceV11API.NameValue[] {
regCode };

    binding.userTokenActivateComplete(callParms);
}

```

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.

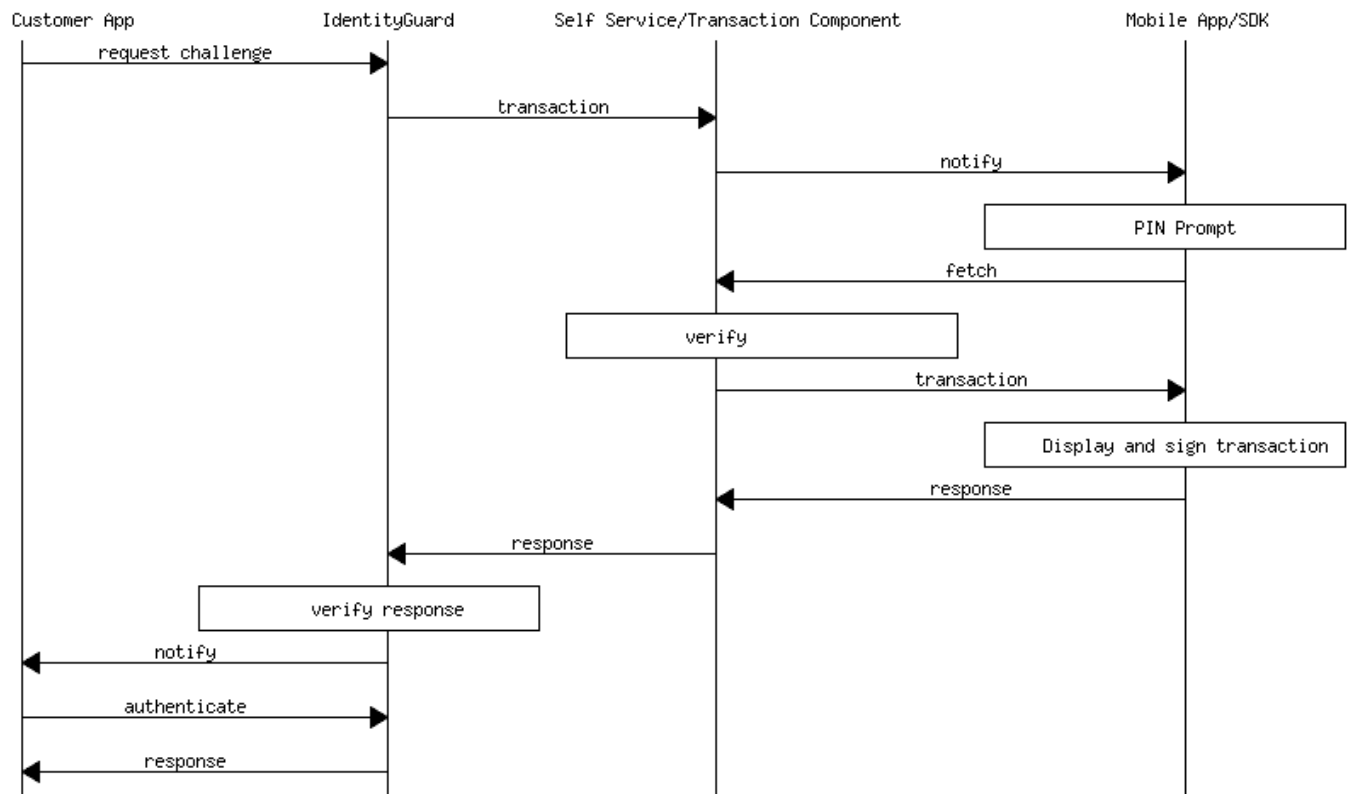
Online transaction verification

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 Entrust IdentityGuard 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. For example, either the details presented in the transaction are incorrect, or the user did not initiate the transaction that triggered the challenge. 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 and transaction verification.

Authentication notifications

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 notifications

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.
- `requireCertificateDelivery` - when set to true, only tokens that have registered for transaction verification challenges are considered.
- `performCertificateDelivery` - 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 `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 challenge. If not specified the application is performing an authentication transaction.

Note: When transaction queuing 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 queuing multiple transactions, see “Enable transaction queuing” 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` lockout count will be used when authenticating token transaction. 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.

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 .NET 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(AuthenticationService binding, string userid, NameValue[]
transactionDetails, string summary)
{
    GenericChallengeParms gParms = new GenericChallengeParms();

    gParms.AuthenticationType = AuthenticationType.TOKENRO;
    gParms.performDeliveryAndSignature = true;
    gParms.requireDeliveryAndSignatureIfAvailable = true;
    gParms.tokenTransactionMode = TokenTransactionMode.ONLINE;
    gParms.tokenChallengeSummary = summary;

    // if the transactionDetails are null then an authentication
    // transaction verification challenge will be performed.
    // Otherwise, a transaction verification challenge will be performed.
    gParms.transactionDetails = transactionDetails;

    GetGenericChallengeCallParms challengeCallParms = new
GetGenericChallengeCallParms();
    challengeCallParms.userId = userid;
    challengeCallParms.parms = gParms;

    GenericChallenge challenge = binding.getGenericChallenge(challengeCallParms);

    GenericAuthenticateParms aParms = new GenericAuthenticateParms();

    aParms.AuthenticationType = AuthenticationType.TOKENRO;
    aParms.transactionId = challenge.TokenChallenge.TransactionId;
    aParms.transactionDetails = 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
    {
        AuthenticateGenericChallengeCallParms genChallengeCallParms = new
AuthenticateGenericChallengeCallParms();
        genChallengeCallParms.userId = userid;
        genChallengeCallParms.response = new Response();
        genChallengeCallParms.parms = aParms;

        GenericAuthenticateResponse response =
binding.authenticateGenericChallenge(genChallengeCallParms);
    }
    catch (SoapException soapEx)
    {
        AuthenticationFault fault = AuthenticationService.getFault(soapEx);
        if (fault != null && fault.ErrorCode == ErrorCode.NO_RESPONSE)
        {
            // continue if we received NO_RESPONSE.
            // Otherwise return the error to the caller
            System.Console.WriteLine(fault.errorMessage);
            Thread.Sleep(2000);
            continue;
        }
        else
        {
            // not IG fault, handle the generic SoapException
            System.Console.WriteLine(soapEx.Message);
            throw;
        }
    }
}
}

```

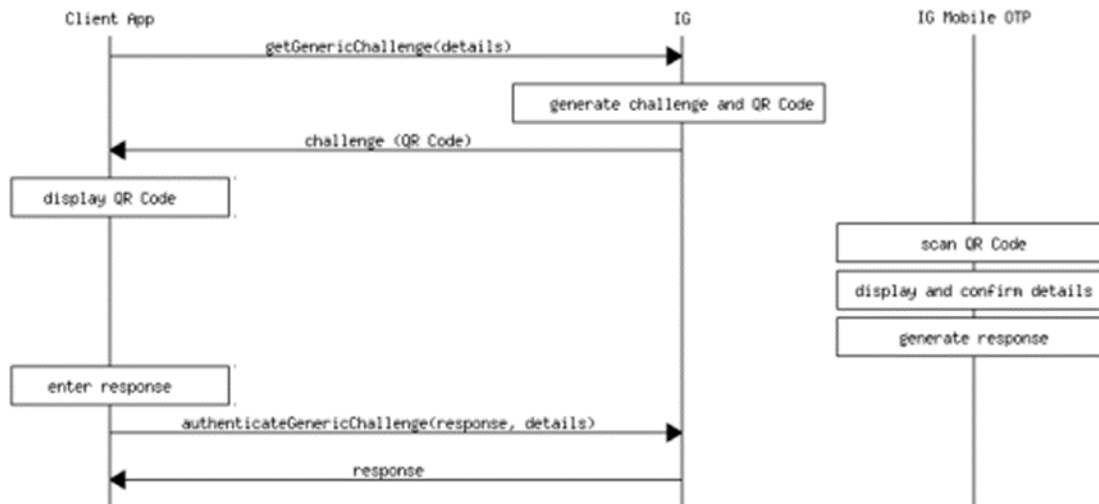
Offline transaction verification

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` 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(AuthenticationService binding, String
userid,
    NameValue[] transactionDetails, String transactionSummary)
    {
        GenericChallengeParms gParms = new GenericChallengeParms();
        gParms.AuthenticationType = AuthenticationType.TOKENRO;
```

```

gParms.tokenTransactionMode = TokenTransactionMode.OFFLINE;
gParms.tokenChallengeSummary = transactionSummary;
gParms.transactionDetails = transactionDetails;

GetGenericChallengeCallParms callParms = new GetGenericChallengeCallParms();
callParms.userId = userid;
callParms.parms = gParms;

GenericChallenge challenge = binding.getGenericChallenge(callParms);

if ((challenge.TokenChallenge.tokens == null)
    || (challenge.TokenChallenge.tokens.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.TokenChallenge.tokens;
}

```

Chapter 7:

API exceptions

This chapter explains how to handle the exceptions returned by the Entrust IdentityGuard APIs.

SoapException returned by proxy classes

The Authentication and Administration .NET proxy classes return a `SoapException` to indicate error conditions. The `SoapException` contains a element called “detail,” in which you can find the serialized XML string for `AuthenticationFault` or `AdminServiceFault` returned by the Entrust IdentityGuard server.

To simplify the client application code, a static method `getFault(SoapException)` is provided by .NET proxy classes to map the soap exceptions to IdentityGuard faults. This method parses the Fault XML string to a `AuthenticationFault` or `AdminServiceFault` object.

To use the `getFault` method:

```
try
{
    an_API_Service_That_Fails.that_fails();
}
catch (SoapException soapEx)
{
    AuthenticationFault fault = AuthenticationService.getFault(soapEx);
    if (fault != null)
    {
        // handle the IdentityGuard fault
    }
    else
    {
        // not IG fault, handle the generic SoapException
    }
}
catch(Exception ex)
{
    // handle non-soap exception, e.g., connection error
}
```

Note: This method is only added for .NET proxy classes. Java APIs throw `AuthenticationServiceFault` or `AdminServiceFault`, which are extended from the `Exception` object.

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 more general external error codes that the client can use for general error detection. See the .NET proxy class documents for a complete description of the error messages. For the list of documents, see [V9 .NET API documentation](#).

Error properties

| Property name | Type | Description |
|---------------|-----------|---|
| errorMessage | string | Returns the error message supplied by the Entrust IdentityGuard Authentication or Administration service describing the failure. |
| ErrorCode | ErrorCode | Returns an ErrorCode 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 ErrorCode does not provide enough information to identify and resolve the error condition. |
| InternalCode | 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. |
| id | 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. |

Using the ErrorCode information to detect errors

The `SoapException` returned by the .NET proxy classes contain the `ErrorCode` information. The following is an example string comparison of error code.

```
try
{
    GenericAuthenticateResponse resp =
        binding.authenticateGenericChallenge(callParms);
}
catch (System.Web.Services.Protocols.SoapException soapException)
{
    System.Xml.XmlNode detailNode = soapException.Detail;
    System.Xml.XmlNodeList nodeList = detailNode.ChildNodes;
```



```

for (int i = 0; nodeList != null && i < nodeList.Count; i++)
{
    System.Xml.XmlNode xmlNode = nodeList.Item(i);
    if (xmlNode.Name.CompareTo("ErrorCode") == 0)
    {
        if (xmlNode.InnerText.CompareTo("INVALID_RESPONSE") == 0)
        {
            // Do something
        }
        else if (xmlNode.InnerText.CompareTo("NO_ACTIVE_CARDS") == 0)
        {
            // Do something else
        }
        else
        {
            // Do something more
        }
    }
}

```

Authentication warning faults

Warning faults are `AuthenticationFault` objects returned as part of an `AuthenticateResponse`. Your application can retrieve the `WarningFault` property.

A warning fault does not stop processing or cause an authentication to fail, because the client application supplied a correct response to a challenge. Nevertheless, the client application receives a report of the error and the system logs it to further diagnose the situation, if necessary.

All exceptions relating to authentication secret and shared secret functionality are returned as warning faults. Ensure that your client application is aware that a failure occurred while setting secret information; however, it is unlikely that the failure results 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, an error is thrown if there is an error while communicating 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 |
|---------------------------------------|---|--|
| authenticateAnonymousChallenge | Returns the GenericAuthenticateResponse object. | AuthenticationServiceFault is thrown if a challenge response fails during the processing of various failure conditions. AuthenticationSystemFault is thrown by any other failure during the processing of this operation. |
| authenticateGenericChallenge | Returns the GenericAuthenticateResponse object. | AuthenticationServiceFault is thrown if a challenge response fails during the processing of various failure conditions. AuthenticationSystemFault is thrown by any other failure during the processing of this operation. |
| getAllowedAuthenticationTypes | Returns the AllowedAuthenticationTypes object for the user for generic authentication and machine registration. | AuthenticationServiceFault is thrown if the user 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. |
| 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. |

| Operation | Expected result | Exceptions |
|-----------|--|---|
| | | AuthenticationSystemFault is thrown if an IdentityGuard server fails. |
| ping | Does not return an exception if the Entrust IdentityGuard Authentication Service is alive. | System.Net.WebException is thrown if the Entrust IdentityGuard Authentication Service is not available. |

Administration Password Change

When a login operation fails because the administrator requires a password change, the Administration proxy class returns a `SoapException` containing the message stating that the password must be changed.

This following is an example of handling the Administration password change request:

```
try
{
    LoginParms loginParms = new LoginParms();
    loginParms.adminId = admin_id;
    loginParms.password = admin_pswd;
    loginParms.response = resps;
    LoginCallParms callParms = new LoginCallParms();
    callParms.parms = loginParms;
    LoginResult loginResult = binding.login(callParms);
    if (loginResult.state.Equals(LoginState.COMPLETE))
    {
        // login success
    }
    else if (loginResult.state.Equals(LoginState.NEED_SECOND_FACTOR))
    {
        // need second-factor challenge response
    }
    else
    {
        // login failed
    }
}
catch (SoapException soapException)
{
    AdminServiceFault fault = AdminService.getFault(soapException);
    if (fault != null)
    {

```

```
if (fault is AdminPasswordChangeRequiredFault)
{
    // code to change password
}
else
{
    // code to handle other IdentityGuard error
}
}
else
{
    // code to handle the non-IdentityGuard error
}
}
catch (Exception ex)
{
    // code to handle other exception, e.g., connection error
}
```

Index

.

.NET proxy classes [NET proxy classes], 142

A

activate grid card, 85

ActiveX, 70

Administration

- activate grid card, 85
- cards, 100
- create grid cards, 85
- create one-time password, 87
- grid cards, 99
- knowledge-based questions, 97
- one-time password, 87
- preproduced grid cards, 86
- PVN, 96
- retrieve one-time password, 88
- send one-time password, 88
- service binding object, 82
- temporary PIN, 94
- token, 92, 93
- unlock users, 98
- user contact information, 91

Administration commands

- cardCreate, 86
- userCardCreate, 85

AdminService_Service, 26

alerts

see transaction notifications, 77

anonymous grid authentication, 41

API overview

- Administration API, 12
- Administration API, 82
- Authentication API, 12
- Authentication V2 API, 14

Authentication

- first-factor, 41
- generic, 45
- grid, 64
- knowledge-based, 64
- knowledge-based questions, 59
- machine, 64, 71, 72
- multifactor, 87
- one-step (anonymous), 41
- one-time password, 47, 64
- Operations and exceptions
 - authenticateAnonymousChallenge, 144
 - authenticateGenericChallenge, 144
 - getAllowedAuthenticationTypes, 144
 - getAllowedAuthenticationTypesForGroup, 144
 - getAnonymousChallenge, 144
 - getAnonymousChallengeForGroup, 144
 - getGenericChallenge, 145
- questions and answers, 64
- risk-based, 51, 74
- second-factor, 41
- token, 46, 64

Authentication commands

- authenticateAnonymousChallenge, 41
- authenticateGenericChallenge, 65

getAnonymousChallenge, 41, 44
getAnonymousChallengeForGroup, 44
getChallenge, 44
getGenericChallenge, 44, 65, 87

B

binding object

creating, 26

C

cards

create, 99
distribute, 99
manufacture, 99

Client-Side Processing for Machine Authentication, 73

code samples, 103

confirmation

transaction, 77

create grid card, 85

Customer support, 10

E

Entrust IdentityGuard

replica servers, 24

Entrust IdentityGuard Mobile, 77

see also soft tokens, 77

Entrust IdentityGuard repository, 83

Error properties

ErrorCode, 142
errorMessage, 142
id, 142
InternalCode, 142

ErrorCode, 142

F

fingerprint

machine fingerprint, 70

first-factor authentication, 41

Flash, 71, 72

G

generic authentication, 45

code sample, 51
knowledge-based questions, 50
one-time password, 47
token, 46

Getting help

Technical Support, 10

Grid

authenticate, 64

grid cards

check inventory, 100
create, 99
distribute, 99
manufacture, 99

I

IGAdminServiceFailoverFactory, 27

IGAAuthServiceFailoverFactory, 27

iPhone, 78

J

Java, 70

K

keystore, 23

knowledge-based questions, 59

- authenticate, 64
- set up, 97

M

machine authentication, 64, 71

- nonce, 72

Machine authentication Web sample, 72

machine information

- Get, 67, 71

machine nonce, 71

man-in-the-browser, 78

mitb attack, see man-in-the-browser, 78

multifactor authentication, 87

mutual authentication

- grid, 59
- token, 59

Mutual authentication

- knowledge-based questions, 59

N

namespace, 29

nonce

- machine, 71
- optional sequence, 72

notification

- transaction, 77

notifications

- of transactions, 77

O

one-step (anonymous) authentication, 41

one-time password, 47, 87

- authenticate, 64
- create, 87
- retrieve, 88
- send, 88

optional application data:, 72

optional sequence nonce, 72

OTP

- Retrieve delivery configuration, 89

P

personal verification number - see PVN, 96

preproduced grid cards, 86

assign, 86
create, 86

Professional Services, 11

PVN

assign, 96
create, 96
modify, 96

R

replay

grid values, 59
image and message, 60
serial number, 59

replica servers, 24

risk-based authentication, 51, 74

S

sample smart credential code, 103

second-factor authentication, 41

serial numbers

replay, 59

Server-side processing for machine authentication, 74

service binding object, 82

shared secret, 143

SOAP, 26

SoapException, 141

string conversion, 44

T

Technical Support, 10

temporary PIN

assign, 94
create, 94
modify, 94

token

assign, 92
authenticate, 64
check inventory, 100
modify, 93
reset, 93

Tomcat, 23

transaction

component, 77
notifications, 77

Transaction notifications, 77

typographic conventions, 8

U

unlock users, 98

user contact information

assign, 91
create, 91
modify, 91

W

warning faults, 143

WarningFault property, 143

WSDL

files, 14