digitalPersona

Access Management API

Developer Guide

Published/Revised: May 3, 2019 (Software version 3.0.0)

Contents

SECTION TWO - WEB AUTHENTICATION

SECTION THREE - WINDOWS AUTHENTICATION

SECTION FOUR - APPENDIX

Introduction

The DigitalPersona Access Management API provides the APIs, tools, documentation and sample code that developers can use to take advantage of the powerful features built-in to the DigitalPersona software solution and implement them in their applications.

This developer guide is divided into four sections, not counting this chapter, that align with the specific uses of the various API subsets for supported features and platforms. These sections are

Section Title	Page	Purpose
Section Two - Web Authentication	56	Provides authentication for web-based applications. User enrollment must be handled through a DigitalPersona client or the Web Enrollment API.
Section Two - Web enrollment	69	Provides web-based enrollment and management of DigitalPersona credentials as well as capturing and saving additional user biographic information (when used with a DigitalPersona LDS Server) such as user surname, date of birth, address, etc.
Section Three - Windows Authentication	141	Provides authentication and identification for Windows applications. User enrollment must be handled through a DigitalPersona client or through the Web Enrollment API.
Section Four - Appendix	149	Provides information applicable to the entire API or to more than one of the previous sections.

Through the Windows and Web Authentication APIs, you can authenticate DigitalPersona users quickly and easily against authentication policies as defined by the DigitalPersona administrator or custom policies defined by your application, and subsequently release their users' protected data (secrets).

Through the Web Enrollment API, you can enroll DigitalPersona users and access most of the enrollment features provided through the DigitalPersona clients.

All of the authentication credentials provided in the DigitalPersona solution are supported through the corresponding APIs except for the Face credential (for web APIs) and the Bluetooth credential (web and Windows APIs).

You may notice that some methods in this API will include reference to 'Altus' in their names. This is due to terminology that has been changed recently but has not been updated at the code level. The term 'Altus" should be understood as referring to Non AD users in the DigitalPersona LDS solution.

Working environment

Use of the included APIs assumes that an appropriate DigitalPersona solution has been installed, configured and verified. Some features of the Windows API can be used in a minimal DigitalPersona environment consisting of the DigitalPersona Workstation or DigitalPersona Kiosk and a single DigitalPersona AD or LDS Server.

In most cases, it is highly recommended to have the DigitalPersona environment set up and verified as part of a separately purchased Installation package.

Target Audience

Developers should have an understanding of the core components of the DigitalPersona solution and its terminology and concepts. They should also be knowledgeable in the specific target platform and the relevant development language.

Additional Resources

You can refer to the additional resources described in this section to assist you in using the API.

Subject	Document
Concepts, features, processes and terminology used in DigitalPersona solutions	DigitalPersona AD and LDS Administrator Guides, Client Guide and supporting documentation is available at: https://www.crossmatch.com/company/support/documentation
DigitalPersona Developer Connection Forum for DigitalPersona Developers	http://devportal.digitalpersona.com/ (Requires free registration.)
Latest patches, updates and utilities for DigitalPersona software products	http://www.crossmatch.com/support/downloads/

System Requirements

Development system

Windows API

The recommended minimum software requirements needed to develop applications with the DigitalPersona Windows API are:

- Development workstation running Windows 7 or later and DigitalPersona Workstation or Kiosk
- To compile the sample code: Visual Studio 2008 or later.
- DigitalPersona Server running Windows Server 2012 and DigitalPersona AD or LDS Server.

See the topic *Supported DigitalPersona Products* below for a complete list of compatible DigitalPersona clients and Servers.

Web AUTH and Web Enrollment APIs

In addition to the requirements listed above, the following are required for use of the Web AUTH and Web Enrollment APIs.

- Windows Web Server (IIS)
- DigitalPersona Web Management Components
- An SSL certificate

See the DigitalPersona Administrator and Client Guides for instructions on installing and configuring the above components.

Target system

Recommended minimum software requirements are the same as for the development system with the following exceptions:

- Visual Studio is not required.

- If the logon and Password Manager features are not needed, the DigitalPersona client can be installed without these applications. This installs the DigitalPersona Access Management API runtime only.

Supported DigitalPersona Products

The DigitalPersona Access Management API is compatible with the following DigitalPersona products:

- DigitalPersona AD or LDS Workstation 2.1 or later.
- DigitalPersona AD or LDS Kiosk 2.1 or later.
- DigitalPersona AD or LDS AD Server, version 2.1 or later.
- DigitalPersona Access Management API 2.1 or later.

Section One - Web enrollment

This section includes the following chapters:

Introduction 2

THIS CHAPTER PROVIDES AN OVERVIEW OF THE FUNCTIONALITY AND FEATURES OF THE DIGITALPERSONA WEB ENROLLMENT API.

The *DigitalPersona Web Enrollment API* allows you to add management of DigitalPersona credentials to your web-based application. Significant features of the Altus WES SDK include:

- Creation and deletion of a user account in the DigitalPersona database
- Enrollment, unenrollment and re-enrollment of DigitalPersona credentials*
- Capturing and saving additional user biographic information such as user surname, date of birth, email address, etc.

This version of the Altus WES SDK supports the following DigitalPersona credentials:

- Password
- Fingerprint
- PIN
- Proximity Cards
- Live Questions

Support for web-based management of additional credentials such as Smart Card and Bluetooth devices is planned, but these additional credentials are only supported locally at this time.

The API uses the Microsoft Web Server (Internet Information Services (IIS)) as a hosting environment and Microsoft WCF as a framework for the building environment.

Note: Use of this API requires previous installation and cnfiguration of the DigitalPersona Web Management Components. For details, see the DigitalPersona Administrator Guide.

Target Audience

This guide is for developers who have a working knowledge of the C++ programming language. In addition, readers must have an understanding of the DigitalPersona product and its authentication terminology and concepts.

Chapter Overview

Introduction (this chapter), provides an overview of the features of the API.

Chapter 3, *DP Web Enrollment API*, provides full details on the methods provided through the Web Enrollment API.

Chapter 4, *WES Credentials Data Format*, definesthe data members of the Credential class and the methodsfor credential enrollment which have as an input parameter an object of the Credential class.

Chapter 5, *Web Enrollment Sample application*, describes the included sample program (DPWebDemo.exe), which creates a GUI displaying fields and buttons that showcase the primary features of the DigitalPersona Web Enrollment API.

DP Web Enrollment API 3

THIS CHAPTER PROVIDES A DESCRIPTION OF THE DIGITALPERSONA WEB EROLLMENT API.

The DigitalPersona Web Enrollment API provides the methods necessary for creating users in the DigitalPersona LDS database, enrolling both DigitalPersona AD and DigitalPersona LDS users, and managing credentials and other data relating to those users.

Web Enrollment is accomplished through the DigitalPersona Web Enrollment Service (DP WES), implemented as a GUI-less Web Service. It should be located inside the enterprise's firewall, and will direct enrollment requests to an existing DigitalPersona Server. DP WES uses a simple RESTful API with HTTP GET/POST/PU/DELETE for stateless communication. It also requires one or more open web endpoints listening on the HTTPS protocol for intranet and/or internet use.

The included sample program described in Chapter 4 provides a GUI-based application that illustrates the main features of the service provided through this API.

IDPWebEnroll interface

The IDPWebEnroll interface is a Windows Foundation Class (WCF) interface, and is described below.

```
namespace WebServices.DPWebEnroll
{
  [ServiceContract]
  public interface IDPWebEnroll
  {
    /*
    * Return information which credentials are enrolled for specific user
    */
    [OperationContract()]
    [WebInvoke(Method = "GET",
      ResponseFormat = WebMessageFormat.Json,
      BodyStyle = WebMessageBodyStyle.Wrapped,
      UriTemplate ="GetUserCredentials?user={userName}&type={userNameType}")]
    Object GetUserCredentials(String userName, UInt16 userNameType);
    /*
    * Return credential specific public enrollment information
    */
    [OperationContract()]
    [WebInvoke(Method = "GET",
      ResponseFormat = WebMessageFormat.Json,
      BodyStyle = WebMessageBodyStyle.Wrapped,
      UriTemplate = "GetEnrollmentData?user={userName}&type={userNameType}&cred_id={credentialId}")]
    String GetEnrollmentData(String userName, UInt16 userNameType, String credentialId);
    /*
    * Creates specific user
    */
    [OperationContract()]
    [WebInvoke(Method = "PUT",
      ResponseFormat = WebMessageFormat.Json,
      BodyStyle = WebMessageBodyStyle.Wrapped,
      UriTemplate = "CreateUser")]
    void CreateUser(Ticket secOfficer, User user, String password);
    /*
    * Deletes specific user
    */
```

```
[OperationContract()]
[WebInvoke(Method = "DELETE",
  ResponseFormat = WebMessageFormat.Json,
  BodyStyle = WebMessageBodyStyle.Wrapped,
  UriTemplate = "DeleteUser")]
void DeleteUser(Ticket secOfficer, User user);
/*
* Enroll specific credentials for specific user
*/
[OperationContract()]
[WebInvoke(Method = "PUT",
    ResponseFormat = WebMessageFormat.Json,
    BodyStyle = WebMessageBodyStyle.Wrapped,
    UriTemplate = "EnrollUserCredentials")]
void EnrollUserCredentials(Ticket secOfficer, Ticket owner, Credential credential);
/*
* Delete specific credentials for specific user
*/
[OperationContract()]
[WebInvoke(Method = "DELETE",
  ResponseFormat = WebMessageFormat.Json,
  BodyStyle = WebMessageBodyStyle.Wrapped,
  UriTemplate = "DeleteUserCredentials")]
void DeleteUserCredentials(Ticket secOfficer, Ticket owner, Credential credential);

/*
* Enroll specific credentials for Non AD user without user authentication
*/
[OperationContract()]
[WebInvoke(Method = "PUT",
  ResponseFormat = WebMessageFormat.Json,
  BodyStyle = WebMessageBodyStyle.Wrapped,
  UriTemplate = "EnrollAltusUserCredentials")]
void EnrollAltusUserCredentials(Ticket secOfficer, User user, Credential credential);
/*
* Delete specific credentials for Non AD user without user authentication
*/
[OperationContract()]
[WebInvoke(Method = "DELETE",
  ResponseFormat = WebMessageFormat.Json,
  BodyStyle = WebMessageBodyStyle.Wrapped,
  UriTemplate = "DeleteAltusUserCredentials")]
void DeleteAltusUserCredentials(Ticket secOfficer, User user, Credential credential);
/*
* Get specific attribute for specific user
*/
[OperationContract()]
[WebInvoke(Method = "POST",
  ResponseFormat = WebMessageFormat.Json,
  BodyStyle = WebMessageBodyStyle.Wrapped,
  UriTemplate = "GetUserAttribute")]
Attribute GetUserAttribute(Ticket ticket, User user, String attributeName);
/*
* Put specific attribute to specific user
```

```
    */
    [OperationContract()]
    [WebInvoke(Method = "PUT",
      ResponseFormat = WebMessageFormat.Json,
      BodyStyle = WebMessageBodyStyle.Wrapped,
      UriTemplate = "PutUserAttribute")]
  void PutUserAttribute(Ticket ticket, User user, String attributeName,
       AttributeAction action, Attribute attributeData);
    /*
    * Self Unlock user account
    */
    [OperationContract()]
    [WebInvoke(Method = "POST",
      ResponseFormat = WebMessageFormat.Json,
      BodyStyle = WebMessageBodyStyle.Wrapped,
      UriTemplate = "UnlockUser")]
    void UnlockUser(User user, Credential credential);
    /*
    * Call for credential specific Custom Action.
    */
    [OperationContract()]
    [WebInvoke(Method = "POST",
      ResponseFormat = WebMessageFormat.Json,
      BodyStyle = WebMessageBodyStyle.Wrapped,
      UriTemplate = "CustomAction")]
    String CustomAction(Ticket ticket, User user, Credential credential, UInt16 actionId);
    /*
    *   Check if enrollment allowed for specific user
    */
    [OperationContract()]
    [WebInvoke(Method = "POST",
      ResponseFormat = WebMessageFormat.Json,
      BodyStyle = WebMessageBodyStyle.Wrapped,
      UriTemplate = "IsEnrollmentAllowed")]
    void IsEnrollmentAllowed(Ticket secOfficer, User user, String credentialId);
    }
}
```

Methods

The methods available through the WES API are described in the following pages.

GetUserCredentials method

The *GetUserCredentials* method allows the caller to request information about the credentials that have been enrolled by a specified user. GetUserCredentials should be implemented as HTTP GET using JSON as the response format.

Syntax

```
List<String> GetUserCredentials(String userName, UInt16 userNameType);
```

Parameter	Description	
userName	Name of the user whose credential needs to be verified.	
userNameType	The specific format of the user name is provided in the first parameter. Valid formats are:	
	3	Windows NT® 4.0 account name (for example, digital_persona\klozin). The domainonly version includes trailing backslashes (\\).
	4	Account name format used in Microsoft® Windows NT® 4.0. For example, "someone".
	5	GUID string that the IIDFromString function returns (for example, {4fa050f0-f561-11cfbdd9-00aa003a77b6}).
	7	User Principal Name (UPN). For example, someone@mycompany.com.
	8	User SID string (for example, S-1-5-21-1004).
	9	DigitalPersona user name format (user name associated with DigitalPersona identity database).

Return values

List of credential IDs of all credentials enrolled by a user. For full details on all supported credential IDs, see the chapter *WES Credentials Data Format* beginning on page *33*.

Examples

```
https://www.somecompany.com/DPWebEnrollervice.svc/GetUserCredentials?
user=john.doe@somecompany.com&type=6
```

Here user name is john.doe@somecompany.com, user name type is 6 which means UPN.

The example response would be the following:

```
{"GetUserCredentialsResult":
  [
    "D1A1F561-E14A-4699-9138-2EB523E132CC",
    "AC184A13-60AB-40e5-A514-E10F777EC2F9",
    "8A6FCEC3-3C8A-40c2-8AC0-A039EC01BA05"
  ]
}
```

This response means that the user john.doe@somecompany.com has password, fingerprint and PIN credentials enrolled.

GetEnrollmentData method

The GetEnrollmentData method is a utility method which allows the caller to get credential enrollment specific data. For example, Live Question authentication may require knowing which particular questions were enrolled, for fingerprint - fingerprint positions enrolled, etc.

GetEnrollmentData should be implemented as HTTP GET using JSON as response format.

Syntax

```
String GetEnrollmentData(String userName, UInt16 userNameType, String credentialId);
```

Parameter	Description	
userName	Name of the user whose enrollment data needs to be obtained.	
userNameType	The specific format of the user name is provided in the first parameter.	
	3	Windows NT® 4.0 account name (for example, digital_persona\klozin). The domainonly version includes trailing backslashes (\\).
	4	Account name format used in Microsoft® Windows NT® 4.0. For example, "someone".
	5	GUID string that the IIDFromString function returns (for example, {4fa050f0-f561-11cfbdd9-00aa003a77b6}).
	7	User Principal Name (UPN). For example, someone@mycompany.com.
	8	User SID string (for example, S-1-5-21-1004).
	9	DigitalPersona user name format (user name associated with DigitalPersona identity database).
credentialId		Unique ID of credential whose data needs to be returned. For full details on all supported credential IDs, see the chapter *WES Credentials Data Format* beginning on page *33*.

Return values

Base64Url encoded enrollment data. The format of such enrollment data is credential-specific and will be detailed in separate document(s).

Example

```
https://www.somecompany.com/DPWebEnrollService.svc/GetEnrollmentData?
user=john.doe@somecompany.com&type=6&cred_id=AC184A13-60AB-40e5-A514-E10F777EC2F9
```

Here the user name is john.doe@somecompany.com, the user name type is 6 which means UPN name and the credential ID is {AC184A13-60AB-40e5-A514-E10F777EC2F9}, which means information about a fingerprint credential was requested.

NOTE: The use of braces {} is considered unsafe in URLs (see RFC 1738), which is why "braceless" GUID representation is used in the API.

The example response would be the following:

```
{"GetEnrollmentDataResult":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"}
```

This response shows fingerprint enrollment data for the user john.doe@somecompany.com.

CreateUser method

CreateUser method creates user account in DigitalPersona database. This method makes sense only if DigitalPersona is used as backend server. In DigitalPersona AD user account is created in Active Directory and Administrator must use standard Active Directory tools to create it.

CreateUser should be implemented as HTTP PUT using JSON as the response format.

Syntax

```
void CreateUser(Ticket secOfficer, User user, String password);
```

Parameter	Description
secOfficer	JSON Web Token of Security Officer. Security Officer should use the DigitalPersona Web AUTH Service to authenticate himself and acquire this token. Token must be valid to call succeeded. To be valid token must be: 1) issued no longer than 10 minutes before the operation, 2) one of the Primary credentials must be used to acquire this token and 3) token owner must have a rights to create user account in DigitalPersona (AD LDS) database.
user	User account that needs to be created. See the definition of the User class on page *68*.
password	String which represents the initial password for newly created user account. We cannot create user account without setting initial. Password must satisfy password complexity policy set for AD LDS database otherwise call will fail.

Example

```
https://www.somecompany.com/DPWebEnrollService.svc/CreateUser
```

Below is example of an HTTP BODY of CreateUser request.

```
{
  "secOfficer":{"jwt":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"},
  "user":
  {
    "name":"john.doe@somecompany.com",
    "type":6
  },
  "password":"aaaAAA123"
}
```

The call above creates an account in DigitalPersona (AD LDS) database for Active Directory user with UPN name john.doe@somecompany.com.

DeleteUser method

The DeleteUser method deletes a user account from the DigitalPersona database. This method makes sense only if DigitalPersona is used as the backend server. In DigitalPersona AD, the user account is deleted in Active Directory and an Administrator should use the standard Active Directory tools to do so.

DeleteUser should be implemented as HTTP DELETE using JSON as the response format.

Syntax

```
void DeleteUser(Ticket secOfficer, User user);
```

Parameter	Description
secOfficer	JSON Web Token of Security Officer. Security Officer should use the DigitalPersona Web AUTH Service to authenticate himself and acquire this token. Token must be valid to call succeeded. To be valid, a token must be: • Issued no longer than 10 minutes before the operation • One of the Primary credentials must be used to acquire this token and • The token owner must have a rights to create user account in DigitalPersona (AD LDS) database.
user	User account that needs to be deleted. See the definition of the User class on page *68*.

Examples

Below is example of a URL that can be used to activate a DeleteUser request:

```
https://www.somecompany.com/DPWebEnrollService.svc/DeleteUser
```

Below is example of HTTP BODY of DeleteUser request:

```
{
  "secOfficer":{"jwt":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"},
  "user":
  {
    "name":"john.doe@somecompany.com",
    "type":6
  }
}
```

The call above deletes an account from DigitalPersona (AD LDS) database for Active Directory user with UPN name john.doe@somecompany.com.

EnrollUserCredentials method

The EnrollUserCredentials method enrolls (or re-enrolls) specific credentials for a named user and stores their credential data in the DigitalPersona AD database. This method will work for both DigitalPersona AD and DigitalPersona LDS backend servers.

EnrollUserCredentials should be implemented as HTTP PUT using JSON as response format.

Syntax

```
void EnrollUserCredentials(Ticket secOfficer, Ticket owner, Credential credential);
```

Parameter	Description
secOfficer	JSON Web Token of Security Officer. Security Officer should use the DigitalPersona Web AUTH Service to authenticate himself and acquire this token. Token must be valid to call succeeded. To be valid, a token must be: • Issued no longer than 10 minutes before the operation • One of the Primary credentials must be used to acquire this token and • The token owner must have a rights to create user account in the DigitalPersona LDS or DigitalPersona AD database. NOTE: This parameter is optional. If user has rights to enroll himself (self-enrollment allowed), caller may provide "null" to this parameter.
owner	JSON Web Token of the owner of credentials. User should use DigitalPersona Web AUTH Service to authenticate itself and acquire this token. Token must be valid to call succeeded. To be valid, a token must be: • Issued no longer than 10 minutes before the operation • One of the Primary credentials (or same credentials) must be used to acquire this token.
credential	Credential to be enrolled. Note that the Data field of this parameter is specific to each credential. See the definition of the Credential class on page *33* and following.

Examples

Below is an example of a URL which can be used to PUT EnrollUserCredentials request:

```
https://www.somecompany.com/DPWebEnrollService.svc/EnrollUserCredentials
```

Below is example of HTTP BODY of EnrollUserCredentials request:

```
{
  "secOfficer":{"jwt":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"},
  "owner":{"jwt":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"},
  "credential":
  {
    "id":"AC184A13-60AB-40e5-A514-E10F777EC2F9",
    "data":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"
  }
}
```

The call above enrolls fingerprint credentials for the user specified in the "owner" token.

DeleteUserCredentials method

`DeleteUserCredentials` method deletes (un-enrolls) specific credentials for a user and remove credential data from DigitalPersona database. This method will work for both DigitalPersona AD and DigitalPersona LDS backend servers.

`DeleteUserCredentials` should be implemented as HTTP DELETE using JSON as response format.

Syntax

```
void DeleteUserCredentials(Ticket secOfficer, Ticket owner, Credential credential);
```

Parameter	Description
secOfficer	JSON Web Token of Security Officer. Security Officer should use the DigitalPersona Web AUTH Service to authenticate himself and acquire this token. Token must be valid to call succeeded. To be valid token must be: • Issued no longer than 10 minutes before the operation • One of the Primary credentials must be used to acquire this token • The token owner must have a rights to create user account in the DigitalPersona LDS or DigitalPersona AD database. NOTE: This parameter is optional. If user has rights to enroll himself (self-enrollment allowed), caller may provide "null" to this parameter.
owner	JSON Web Token of the owner of credentials. User should use DigitalPersona Web AUTH Service to authenticate itself and acquire this token. Token must be valid to call succeeded. To be valid token must be: • Issued no longer than 10 minutes before the operation • One of the Primary credentials (or same credentials) must be used to acquire this token.
credential	Credential to be deleted. Note that the Data field of this parameter is specific to each credential. See the definition of the Credential class on page *33* and following.

Example

Below is example of a URL which can be used to DELETE **`DeleteUserCredentials`** request:

https://www.somecompany.com/DPWebEnrollService.svc/DeleteUserCredentials

Below is example of HTTP BODY of **`DeleteUserCredentials`** request:

```
{
  "secOfficer":{"jwt":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"},
  "owner":{"jwt":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"},
  "credential":
  {
    "id":"AC184A13-60AB-40e5-A514-E10F777EC2F9",
    "data":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"
  }
}
```

The call above deletes any fingerprint credentials for the user specified in the "owner" token.

EnrollAltusUserCredentials method

The EnrollAltusUserCredentials method enrolls (or re-enrolls) specific credentials for specific user and store credential data in the DigitalPersona database. This method will work only for Non AD users and the DigitalPersona LDS Server backend. For AD users, the function will return Access Denied.

This method is different from EnrollUserCredentials because it allows enroll user credentials without user being previously authenticated. Only authentication of Security Officer is required. By default the DigitalPersona Server requires the user to be authenticated to enroll credentials so we introduce new GPO setting: AllowSecurityOfficerEnrollment. If this GPO is not configured or set to 0, EnrollAltusUserCredentials function will always return Access Denied error. If this GPO set to 1 and Security Officer has rights to enroll this particular user, enrollment will be performed.

Syntax

```
void EnrollAltusUserCredentials(Ticket secOfficer, User user, Credential credential);
```

Parameter	Description
secOfficer	JSON Web Token of Security Officer. Security Officer should use DigitalPersona Web AUTH Service to authenticate itself and acquire this token. Token must be valid to call succeeded. To be valid, a token must be: • Issued no longer than 10 minutes before the operation • One of the Primary credentials must be used to acquire this token • The token owner must have a rights to enroll user in the DigitalPersona LDS database.
user	User which credentials needs to be enrolled. Only Non AD users can be accepted by this function. See the definition of the User class on page *68*.
credential	Credential to be deleted. Note that the Data field of this parameter is specific to each credential. See the definition of the Credential class on page *33* and following.

EnrollAltusUserCredentials should be implemented as HTTP PUT using JSON as response format.

Below is example of URL which can be used to PUT EnrollAltusUserCredentials request:

```
https://www.digitalpersona.com/DPWebEnrollService.svc/EnrollAltusUserCredentials
```

Below is example of HTTP BODY of EnrollAltusUserCredentials request:

```
{
  "secOfficer":{"jwt":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"},
  "user":
    {
      "name":"someone",
      "type":9
    },
    "credential":
    {
      "id":"AC184A13-60AB-40e5-A514-E10F777EC2F9",
      "data":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"
    }
}
```

The call above enrolls fingerprint credentials for the Non AD user "someone".

NOTE: This method can be used to Reset a Non AD user’s password without user intervention.

NOTE: This method also can be used to Randomize user Password. To randomize user password caller must provide "null" in data parameter in "credential" class. Below is an example of Password Randomization request:

```
{
  "secOfficer":{"jwt":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"},
  "user":
    {
      "name":"someone ",
      "type":9
```

```
    },
     "credential":
    {
      "id":" D1A1F561-E14A-4699-9138-2EB523E132CC ",
      "data":null
    }
}
```

DeleteAltusUserCredentials method

DeleteAltusUserCredentials method deletes (un-enrolls) specific credentials for a specific Non AD user and removes credential data from the DigitalPersona database. This method will work only for Non AD users. For AD users it will return Access Denied.

This method is different from DeleteUserCredentials because it allows delete user credentials without user being previously authenticated. Only authentication of Security Officer is required. By default the DigitalPersona Server requires the user to be authenticated to delete credentials so we introduce new GPO setting: AllowSecurityOfficerEnrollment. If this GPO is not configured or set to 0, DeleteAltusUserCredentials function will always return Access Denied error. If this GPO set to 1 and Security Officer has rights to enroll this particular user, credential deletion will be performed.

Syntax

```
void DeleteAltusUserCredentials(Ticket secOfficer, User user, Credential credential);
```

Parameter	Description
secOfficer	JSON Web Token of Security Officer. Security Officer should use DigitalPersona Web AUTH Service to authenticate itself and acquire this token. Token must be valid to call succeed. To be valid, a token must be: • Issued no longer than 10 minutes before the operation • One of the Primary credentials must be used to acquire this token • The token owner must have a rights to enroll user in the DigitalPersona database (LDS version) or in Active Directory (AD version). NOTE: This parameter is optional. If user has rights to enroll himself (self-enrollment allowed), caller may provide "null" to this parameter.
user	User whose credentials needs to be deleted. See the definition of the User class on page *68*.
credential	Credential to be deleted. Note that the Data field of this parameter is specific to each credential. See the definition of the Credential class on page *33* and following.

DeleteAltusUserCredentials should be implemented as HTTP DELETE using JSON as response format.

Below is example of URL which can be used to DELETE DeleteAltusUserCredentials request:

```
https://www.digitalpersona.com/DPWebEnrollService.svc/DeleteAltusUserCredentials
```

Below is example of HTTP BODY of DeleteAltusUserCredentials request:

```
{
  "secOfficer":{"jwt":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"},
  "user":
  {
    "name":"someone ",
```

```
    "type":9
  },
  "credential":
  {
    "id":"AC184A13-60AB-40e5-A514-E10F777EC2F9",
    "data":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"
  }
}
```

The call above delete fingerprint credentials for Non AD user "someone".

GetUserAttribute method

GetUserAttribute method request some public (biographic) information about user, like user surname, date of birth, e-mail address, etc.

GetUserAttribute should be implemented as HTTP POST using JSON as response format.

Syntax

```
Attribute GetUserAttribute(Ticket ticket, User user, String attributeName);
```

Parameter	Description
ticket	JSON Web Token of user who requests this information. This could be an attribute owner, Security Officer, Administrator or any user who has rights to read this information. Token must be valid to call succeed. To be valid token must be: 1) issued no longer than 10 minutes before the operation, 2) one of the Primary credentials must be used to acquire this token and 3) token owner must have a rights to read this attribute user in DigitalPersona (AD LDS) or DigitalPersona AD (Active Directory) database.
user	User which information requested. See the definition of the User class on page *68*.
attributeName	Name of the information requested. Both AD and AD LDS are LDAP databases so this name must be Ldap-Display-Name of Attribute Schema in User object in LDAP database.

Return values

JSON representation of object of Attribute class will be returned if the call succeeds. For details on the Attribute class, see the topic *Attribute class on page 29.*

Examples

Below is an example of a URL which can be used to POST GetUserAttribute request.

```
https://www.somecompany.com/DPWebEnrollService.svc/GetUserAttribute
```

Below is example of HTTP BODY of GetUserAttribute request:

```
{
  "ticket":{"jwt":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"},
  "user":
  {
    "name":"john.doe@somecompany.com",
    "type":6
```

```
  },
  "attributeName":"sn"
}
```

The call above requests "Surname" attribute for Active Directory user with UPN name john.doe@somecompany.com.

The example of return value of this call could be:

```
{"GetUserAttributeResult":
{
"type":3,
"values":["Lozin"]
}
}
```

PutUserAttribute method

The PutUserAttribute method writes, updates or cleara specific public data (attribute) for the named user. This method makes sense only for DigitalPersona as backend server (AD LDS), for Active Directory Administrator must use standard AD tools to manage attributes (with exception of DP specific attributes).

PutUserAttribute should be implemented as HTTP PUT using JSON as response format.

Syntax

```
void PutUserAttribute(Ticket ticket, User user, String attributeName,
     AttributeAction action, Attribute attributeData);
```

Parameter	Description
ticket	JSON Web Token of user who requests attribute modification. This could be an attribute owner, Security Officer, Administrator or any user who has rights to write this information. Token must be valid to call succeed. To be valid token must be: 1) issued no longer than 10 minutes before the operation, 2) one of the Primary credentials must be used to acquire this token and 3) token owner must have a rights to write this attribute user in DigitalPersona (AD LDS) or DigitalPersona AD (Active Directory) database.
user	User whose attribute needs to be modified. See the definition of the User class on page *68*.
attributeName	Name of the information requested. Both AD and AD LDS are LDAP databases so this name must be Ldap-Display-Name of Attribute Schema in User object in LDAP database.
action	Action that needs to be taken. It could be Append, Update, Delete or Clear. For additional information, see page *28*.
attributeData	Attribute data that needs to be written. For details on the Attribute class, see the topic *Attribute class on page 29.*

Examples

Below is an example of URL which can be used to PUT PutUserAttribute request:

```
https://www.somecompany.com/DPWebEnrollService.svc/PutUserAttribute
```

Below is example of HTTP BODY of PutUserAttribute request:

```
{
  "ticket":{"jwt":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"},
  "user":
  {
    "name":"john.doe@somecompany.com",
    "type":6
  },
  "attributeName":"sn",
  "action":2,
  "attributeData":
  {
    "type":3,
    "values":["Lozin"]
  }
}
```

The call above requests update of "Surname" attribute for Active Directory user with UPN name john.doe@somecompany.com. New value should be "Lozin".

Data Contracts

Below are the Data Contracts used in the Web Enrollment API. Only the data specific to Web Enrollment is described below. For a description of additional data contracts used in the ,Web Authentication Service API, see page *68* and following.

AttributeAction enumeration

AttributeAction enumeration specifies action which could be taken with attribute in PutUserAttribute call.

```
[DataContract]
public enum AttributeAction
{
  [EnumMember]
  Clear = 1,
  [EnumMember]
  Update = 2,
  [EnumMember]
  Append = 3,
  [EnumMember]
  Delete = 4,
}
```

Parameter	Description
clear	Attribute must be cleared. The attributeData argument of the PutUserAttribute method will be ignored and can be "null".
Update	Attribute will be updated. All previous data in the attribute will be cleared.
Append	The data will be appended to data which already exists in the attribute. Makes sense only for multivalued attributes.
Delete	The data provided in the attributeData argument of the PutUserAttribute method will be deleted from the attribute. Makes sense only for multivalued attributes.

AttributeType enumeration

AttributeType enumeration specifies type of attribute value(s).

```
DataContract]
public enum AttributeType
{
  [EnumMember]
  Boolean = 1,
  [EnumMember]
  Integer = 2,
  [EnumMember]
  String = 3,
  [EnumMember]
  Blob = 4,
}
```

Value	Description
Boolean	Attribute has Boolean value(s).
Integer	Attribute has Integer value(s).
String	Attribute has String value(s).
Blob	Attribute has Blob value(s).

Attribute class

Attribute class is Attribute representation in the Web Enrollment API.

```
[DataContract]
  public class Attribute
  {
    [DataMember]
    public AttributeType type { get; set; }
    [DataMember]
    public List<Object> values { get; set; }
  }
```

Parameters	Description
type	Type of Attribute value(s).
values	Values of attribute. We assume all attributes are multivalued because singlevalued attributes are a subsystem of multivalued attributes. Below we give details of Json representation of attributes of different types.

Boolean attributes

For Boolean attributes Json representation would be Json Boolean. Below is an example of "isDeleted" attribute in Active Directory:.

```
{
  "type":1,
  "values":[true]
}
```

The attribute above claims user is deleted from Active Directory.

Integer attributes

For Integer attributes Json representation would be Json Integer. We will use Json integer for all types of integers like Uin8, Uint16, Uint32 and Uint64. Timestamps will be also represented as long integers (Uint64). Below is an example of "userAccountControl" attribute in Active Directory.

```
{
  "type":2,
  "values":[65536]
}
```

The attribute above claims users' password never expires.

String attributes

For String attributes Json representation would be Json String. Below is an example of "otherMailbox" (users' e-mail addresses) attribute in Active Directory.

```
{
  "type":3,
  "values":["john.doe@somecompany.com","john.doe@mycompany.com"]
}
```

The attribute above has all users e-mail addresses.

Blob attributes

For Blob attributes Json representation would be Json String. To convert Blob to a string, we should use Base64UrlEncoding. Below is an example of "thumbnailPhoto" attribute in Active Directory:

```
{
  "type":4,
  "values":["Z3NhZGhhc2Rma0FTREZLYWZyZGtB"]
}
```

The attribute above has Base64UrlEncoded users' thumbnail photo.

CustomAction method

CustomAction method performs credential specific operation (custom action) for specific user. CustomAction should be implemented as HTTP POST using JSON as response format. For further details, see *CustomAction method* on page *64*.

Syntax

```
void CustomAction(Ticket ticket, User user, Credential credential, UInt16 actionId);
```

Parameters	Description
ticket	JSON Web Token of person who wants to perform CustomAction operation. This parameter is optional because not all CustomAction require Access Control so caller may provide "null" to this parameter.
user	User to whom CustomAction needs to be performed. This parameter is optional because not all CustomAction operations are performed to specific user so caller may provide "null" to this parameter. credential Credential to which CustomAction needs to be performed. Id attribute of Credential class should point to valid DigitalPersona Credential. Data attribute of Credential class should point to Base64Url encoded credential specific data and could be set to "null".
credential	Credential to which CustomAction needs to be performed. Id attribute of Credential class should point to valid DigitalPersona Credential. Data attribute of Credential class should point to Base64Url encoded credential specific data and could be set to "null".

Below is example of URL which can be used to POST CustomAction request:

```
https://www.digitalpersona.com/DPWebEnrollService.svc/CustomAction
```

Below is example of HTTP BODY of CustomAction request:

```
{
  "ticket":{"jwt":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"},
  "user":
  {
    "name":"someone",
    "type":9
  },
  "credential":
  {
    "id":"AC184A13-60AB-40e5-A514-E10F777EC2F9",
    "data":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"
  },
  "actionId":9
}
```

The call above send CustomAction request to fingerprint credentials with actionId 6 for DigitalPersona user "someone".

This call should return Base64Encoded output data of CustomAction call. The returned information is credential specific and should be provided "Web Authentication Service Credentials Data Format" document.

UnlockUser method

UnlockUser method performs self unlocking currently locked user.

Syntax

```
void UnlockUser(User user, Credential credential);
```

Parameters	Description
user	User whose account needs to be unlocked.
credential	A valid user credentials (we consider Live Questions credentials here) to be verified before the account gets unlocked.

UnlockUser should be implemented as HTTP POST using JSON as response format.

Below is example of URL which can be used to POST UnlockUser request:

`https://www.digitalpersona.com/DPWebEnrollService.svc/UnlockUser`

NOTE: To be able to self-unlock own account the following GPO must be enabled on AD (or LDS) server:

Allow users to unlock their Windows account using DigitalPersona Recovery Questions.

WES Credentials Data Format 4

THIS CHAPTER DESCRIBES THEDATA MEMBERS FOR VARIOUS SUPPORTED CREDENTIALS.

In the **IDPWebEnroll** interface (see *IDPWebEnroll interface on page 15*) we define a number of methods for credential enrollment which have as an input parameter an object of the **Credential** class. This chapter defines and describes the format of the **data** member for each **Credential** supported by DigitalPersona.

Credential class

The **Credential** class is defined as follows:

```
[DataContract]
public class Credential
{
  [DataMember]
  public String id { get; set; } // unique id (Guid) of credential
  [DataMember]
  public String data { get; set; } //  credential data
}
```

The **data** member of the **Credential** class is different for each credential. For example, the **data** member of the Fingerprint **Credential** is different from the same member of the Password **Credential**.

Fingerprint Credential

The following ID is defined for Fingerprint Credentials.

```
{AC184A13-60AB-40e5-A514-E10F777EC2F9}
```

We described BioSample class will be used in description below in the chapter *WAS Credentials Data Format* on page *99*.

GetEnrollmentDataResult

To ask information about enrolled fingerprints, the client should send an IDPWebEnroll-> GetEnrollmentData request to the server. Below is an example of such a request:

```
https://www.mycompany.com/DPWebEnrollService.svc/GetEnrollmentData?
```

```
user=john.doe@yourdomain.com&type=6&cred_id=AC184A13-60AB-40e5-A514-E10F777EC2F9
```

The result of a successful GetEnrollmentData request should be a Base64url encoded UTF-8 representation of a list (array) of fingerprints (fingerprint positions) enrolled by the user. Below is the ANSI 381 list of valid fingerprint positions.

Fingerprint position	Value		Fingerprint position	Value
Unknown	0		Left thumb	6
Right thumb	1		Left index finger	7
Right index finger	2		Left middle finger	8
Right middle finger	3		Left ring finger	9
Right ring finger	4		Left little finger	10
Right little finger	5			

The following GetEnrollmentDataResult indicates that the user has their right thumb, right index finger and left middle fingers enrolled.

```
[{"position":1},{"position":2},{"position":8}]
```

EnrollUserCredentials

The following class will represent enrollment sample for fingerprint credentials:

```
[DataContract]
public class BioEnrollment
{
  [DataMember]
  Byte position { get; set; }            // Bio sample (fingerprint) position
  [DataMember]
  List<BioSample> samples { get; set; } // Bio samples to enroll
}
```

Parameter	Description
position	Fingerprint position to enroll. For a list of valid fingerprint positions, see the table on the previous page.
samples	List BioSample with fingerprint data to enroll for such position. List could include from one BioSample to any BioSamples (see BioSample class on page *99*.

The following data members for BioSample could be provided for enrollment:

BioSampleData members	Description
BioFactor	Must be set to 0x0008 (Fingerprints).
BioSampleFormat->FormatOwner	Could be 51 (DigitalPersona) or 49 Neurotec. This parameter will be ignored if Fingerprint image provided as BioSample.

BioSampleData members	Description
BioSampleType	Could be 0x01 (fingerprint image) or 0x04 (Fingerprint enrollment template).
BioSamplePurpose	Could be 0 (Any purpose) or 3 (purpose Enrollment).
BioSampleEncryption	Could be 0 (not encrypted) or 1 (XTEA encryption).

Below is an example of JSON representation of BioEnrollment:

```
{"position":7,              // Left index finger
  "samples":
  [{
    "Version":1,
    "Header":
    {
      "Factor":8,           // Fingerprint
      "Format":
      {
        "FormatOwner":51,   // DigitalPersona engine
        "FormatID":0
      },
      "Type":4,             // Fingerprint template
      "Purpose":3,          // Enrollment purpose
      "Quality":-1,
      "Encryption":0        // Unencrypted
    },
    "Data":"eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9" // Base64url encoded
Fingerprint Template
  }]
}
```

The sample below represent enrollment request to enroll Left index finger and provides processed Fingerprint Template as enrollment data.

The following steps will be needed to create Fingerprint Enrollment packet:

1 Create JSON representation of every BioSample.

2 Combine those JSON representations in a JSON array ([]).

3 Add fingerprint position to create JSON representation of BioEnrollment class as shown above.

4 Base64Url encode the string you created in step #3.

5 Set Fingerprint Credential ID as id member and string we created in step #4 as data member.

NOTE: If user already has fingerprint data enrolled at this position, old data will be replaced with new one. If user does not have fingerprint data enrolled at this position, data will be added.

DeleteUserCredentials

To delete user fingerprint credentials you will need send Base64Url encoded array of fingerprint positions you would like to delete. For example if you want to delete "Right thumb", "Right index finger" and "Left middle finger" fingers, you need contrast the following JSON string:

```
[{"position":1},{"position":2},{"position":8}]
```

Then Base64Url encode it and present as a data member of credential argument of DeleteUserCredentials call.

If you would like to clear all users' fingerprint credentials, you need provide null as member of credential argument of DeleteUserCredentials call. For example:

```
{"id":"AC184A13-60AB-40e5-A514-E10F777EC2F9",
"data":null}
```

CustomAction

CustomAction information is detailed in "Web Authentication Service Credentials Data Format" document.

Password Credential

The following ID is defined for Password Credential.

```
{D1A1F561-E14A-4699-9138-2EB523E132CC}
```

GetEnrollmentDataResult

This call is not supported for the Password credential.

EnrollUserCredentials

The following class used to represent Password Enrollment credentials:

```
[DataContract]
  public class PasswordEnrollment
  {
    [DataMember]
    String oldPassword { get; set; } // Old password
    [DataMember]
    String newPassword { get; set; } // New password
  }
```

Parameter	Description
oldPassword	User's old password. This parameter is optional and can be null, in this case Password Reset operation would be used. If old password is presented, Password Change operation will be used and owner argument of EnrollUserCredentials could be null.
newPassword	Users new password. Password must satisfy password policy otherwise this call will fail. NOTE: Password Reset operation is not supported for Active Directory users, so caller must provide old password to call succeed.

The following example of JSON representation of Password Enrollment request:

```
{"oldPassword":"aaaAAA111","newPassword":"aaaAAA123"}
```

To send this enrollment request you need Base64Url encode JSON representation of Password Enrollment credentials above and provide obtained string as *data* member of *credential* argument of *EnrollUserCredentials* call.

DeleteUserCredentials

This call is not supported for password credential.

CustomAction

CustomAction information is detailed in "Web Authentication Service Credentials Data Format" document.

PIN Credential

The following ID is defined for the PIN Credential.

```
{8A6FCEC3-3C8A-40c2-8AC0-A039EC01BA05}
```

GetEnrollmentDataResult

This call is not supported for the PIN credential.

EnrollUserCredentials

The following steps need to be done to create PIN Enrollment Credential:

1 Base64url encode UTF-8 representation of PIN. NOTE: It is not necessary to include null terminating character to UTF-8 representation.

2 Create JSON representation to PIN credential setting PIN Credential ID as id member and string we created in step #1 as data member.

For example we need to create a JSON representation of the following PIN: 1234. The Base64url encoded UTF-8 representation of such PIN is:

```
MTIzNA
```

Finally we can create a JSON representation of Credential class which we can send for PIN authentication to the DigitalPersona Server:

```
{"id":"8A6FCEC3-3C8A-40c2-8AC0-A039EC01BA05",
"data":"MTIzNA"}
```

NOTE: If PIN is already enrolled, it will be replaced with new data.

DeleteUserCredentials

The data member of credential argument of *DeleteUserCredential* should be *null* and will be ignored.

CustomAction

CustomAction information is detailed in "Web Authentication Service Credentials Data Format" document.

Live Questions Credential

The following ID is defined for the Live Questions Credential.

```
{B49E99C6-6C94-42DE-ACD7-FD6B415DF503}
```

GetEnrollmentData

The result of a successful GetEnrollmentData call is detailed in "Web Authentication Service Credentials Data Format" document.

EnrollUserCredentials

The following class is used to represent Live Questions Enrollment data:

```
[DataContract]
  public class LiveQuestionEnrollment
  {
    [DataMember]
```

```
    LiveQuestion question { get; set; } // Question information
    [DataMember]
    LiveAnswer answer { get; set; } // Answer information
  }
```

Below is an example of JSON representation of Live Questions Enrollment data:

```
{
  "question":
  {
    "version":1,
    "number":6,
    "type":0,
    "lang_id":9,
    "sublang_id":1,
    "keyboard_layout":1033,
    "text":"Who was your first employer?"
  },
  "answer":
  {
    "version":1,
    "number":6,
    "text":"DigitalPersona"
  }
}
```

The following steps will be needed to create Live Questions Enrollment packet:

1 Create JSON representation of every Question/Answer as it was shown above.

2 Combine question/answers pairs JSON representation in a JSON array ([]).

3 Base64Url encode the string you created in step #2.

4 Set Live Question Credential ID as id member and string we created in step #3 as data member.

NOTE: We always replace Live Question data enrolled previously on enrollment request. The merging data is not supported.

DeleteUserCredentials

The data member of credential argument of DeleteUserCredential should be null and will be ignored. All Live Questions will be deleted by this request.

CustomAction

CustomAction information is detailed in "Web Authentication Service Credentials Data Format" document.

Proximity Card Credential

The following ID is defined for the Proximity Card Credential:

```
{1F31360C-81C0-4EE0-9ACD-5A4400F66CC2}
```

GetEnrollmentData

We treat the Proximity Card Data (Prox Card ID) as an opaque blob. Follow these steps to create the Prox Card Credential.

1 Base64url encode Prox Card ID. The Card ID data must be a 64 bytes long byte array, padded with zeroes if the Card ID is shorter than 64 byte (for ex., for iClass or MiFare cards). This 64 bytes array must be Base64url encoded.

2 Create the JSON representation of the **`Credential`** class, setting the Proximity Card Credential ID as the **`id`** member and the string we created in step #1 as the **`data`** member; **.**

For example, the client gets Prox Card ID and it is represented by the following byte array:

```
[123,34,116,121,112,34,58,34,74,87,84,34,44,10,32,34,97,108,103,34,58,
34,32,82,83,50,53,54,34,125]
```

The Base64url encoded value of Prox Card ID is:

```
eyJ0eXAiOiJKV1QiLAogImFsZyI6IiBSUzI1NiJ9
```

Then we create a JSON representation of the **`Credential`** class which we can send for Prox Card enrollment to the DigitalPersona Server:.

```
{"id":"1F31360C-81C0-4EE0-9ACD-5A4400F66CC2",
"data":"eyJ0eXAiOiJKV1QiLAogImFsZyI6IiBSUzI1NiJ9"}
```

NOTE: If the Proximity Card is already enrolled, it will be replaced with new data.

DeleteUserCredentials

The data member of credential argument of DeleteUserCredential should be null and will be ignored.

CustomAction

CustomAction information is specific to each credential. See *CustomAction* under each credential in the *WAS Credentials Data Format* chapter beginning on page *96*.

Time-Based OTP (TOTP) Credential

The following ID is defined for the TOTP Credential:

```
{324C38BD-0B51-4E4D-BD75-200DA0C8177F}
```

GetEnrollmentData

The result of *GetEnrollmentData* call is specific to each credential. See *GetEnrollmentData* under each credential in the *WAS Credentials Data Format* chapter beginning on page *96*.

EnrollUserCredentials

There are two types of OTP tokens supported: 1) software and 2) hardware. Software tokens are running on smart phones or other mobile devices. Hardware tokens running on spe4cificaly manufactured hardware. Enrollment data is different for software and hardware tokens.

Software OTP token enrollment

The following class will represent enrollment sample for TOTP credentials:

```
[DataContract]
public class OTPEnrollData
{
  [DataMember]
  public String otp{ get; set; }      // String with OTP verification code
  [DataMember]
  public String key { get; set; }     // String with Base64Encoded OTO key.
  [DataMember]
```

```
  public String phoneNumber { get; set; }     // User phone number..
}
```

Parameter	Description
otp	String with OTP verification code. If OTP code cannot be verified, enrollment operation will fail with the following error: "The operation being requested was not performed because the user has not been authenticated.". NOTE: verification can fail only for two reasons: • The user mistyped the OTP code • The clocks on the phone and the DigitalPersona Server are not synchronized.
key	Bease64Url Encoded TOTP key.
phoneNumber	User's phone number. This parameter is required to support SMS OTP logon option and if SMS OTP support is not required can be omitted or set to "null".

The following steps are needed to create TOTP Enrollment Credential:

1 Base64Url encode TOTP Key. We support TOTP Key of any length but at least 160bit length TOTP Key is suggested for security reason.

2 Create JSON representation of OTPEnrollData class where to otp assign OTP verification code typed by user and key is string created in step #1.

3 Base64Url encode string created in step #2.

4 Create JSON representation to Credential class setting TOTP Credential ID as id member and string we created in step #3 as data member.

For example client gets TOTP Key and it could be represented by following byte array:

```
[123,34,116,121,112,34,58,34,74,87,84,34,44,10,32,34,97,108,103,34,58,
34,32,82,83,50,53,54,34,125]
```

The Base64url encoded value of TOTP Key is:

```
eyJ0eXAiOiJKV1QiLAogImFsZyI6IiBSUzI1NiJ9
```

User types the following verification code:

```
123456
```

JSON representation of TOTP enrolment data would be the following:

```
{
"otp":":"123456",
"key":":"eyJ0eXAiOiJKV1QiLAogImFsZyI6IiBSUzI1NiJ9"
}
```

Finally by Base64Url encoding string above we can create a JSON representation of Credential class which we can send for TOTP enrollment to the DigitalPersona Server:

```
{"id":"324C38BD-0B51-4E4D-BD75-200DA0C8177F",
"data":"eyahsadHKJjahdsdjlaJHKHLhalkdhsj1298475JHLHGLglagd"}
```

NOTE: If TOTP is already enrolled, it will be replaced with new data.

Hardware OTP token enrollment

The following class will represent enrollment sample for TOTP credentials:

```
[DataContract]
public class OTPHDEnrollData
{
  [DataMember]
  public String otp{ get; set; }      // String with OTP verification code
  [DataMember]
  public String serialNumber{ get; set; } // String with hardware token
                                          manufacturing serial number.
}
```

Parameter	Description
otp	String with OTP verification code. If OTP code cannot be verified, enrollment operation will fail with the following error: "The operation being requested was not performed because the user has not been authenticated.". NOTE: verification can fail only for two reasons: • The user mistyped OTP code • The clocks on the phone and the DigitalPersona Server are not synchronized.
serialNumber	Hardware token serial number. This serial number was assign to the token during manufacturing and usually written on the token itself.

The following steps are needed to create TOTP Enrollment Credential:

1 Create JSON representation of OTPHDEnrollData class where to otp assign OTP verification code typed by user and serialNumber is token serial number also typed by user.

2 Base64Url encode string created in step #1.

3 Create JSON representation to Credential class setting TOTP Credential ID as id member and string we created in step #2 as data member.

For example user types the following verification code:

```
123456
```

And the following serial number:

```
2608513503936
```

JSON representation of TOTP enrolment data would be the following:

```
{
"otp":":"123456",
"serialNumber":":"2608513503936"
}
```

Finally by Base64Url encoding string above we can create a JSON representation of Credential class which we can send for TOTP enrollment to the DigitalPersona Server:

```
{"id":"324C38BD-0B51-4E4D-BD75-200DA0C8177F",
"data":"eyahsadHKJjahdsdjlaJHKHLhalkdhsj1298475JHLHGLglagd"}
```

NOTE: If TOTP is already enrolled, it will be replaced with new data.

DeleteUserCredentials

The data member of credential argument of DeleteUserCredential should be null and will be ignored.

CustomAction

CustomAction informationis specific to each credential. See *CustomAction* under each credential in the *WAS Credentials Data Format* chapter beginning on page *96*.

Smart Card Credential

The following ID is defined for Smart Card Credential:

```
{D66CC98D-4153-4987-8EBE-FB46E848EA98}
```

GetEnrollmentData

This method is not supported for the Smart Card credential.

EnrollUserCredntials

The data for smart card credential is a Base64url encoded UTF-8 representation of the JSON representation of the CDPJsonSCEnrollData class. CDPJsonSCEnrollData class is defined as following:

```
[DataContract]
public class CDPJsonSCEnrollData
{
  [DataMember]
  public Byte version { get; set; }     // version
  [DataMember]
  public String key { get; set; }       // public key
  [DataMember]
  public String nickname { get; set; } // token's nickname
}
```

where:

Parameter	Description
Byte *version*	Version of CDPJsonSCEnrollData object, must be set to 1 for current implementation
String *key*	Public key, Base64url UTF-8 encoded string. The public key from the Smart Card must be imported in the PUBLICKEYBLOB format. After that, it must be Base64url encoded to the string as shown below. `Key = Base64urlEncode( (PUBLICKEYBLOB (PuK) )`
String *nickname*	The nickname of the smart card token. It's suggested to use the card name, like "SmartCafe Expert 72K DI v3.2", as a part of the nickname. The server limits the length of the nickname to 255 symbols, the exceeding symbols will be cut off.

To create the Smart Card Credential for enrollment, following steps must be performed on the client:

1 Enumerate the asymmetric key pairs on the Smart Card and select the exact key pair to use. WASSC supports RSA keys of any length but at least 1024 bit length key is suggested for security reason.

2 Create a JSON representation of the CDPJsonSCEnrollData class for the public key selected in step #1 above.

3 Base64Url encode string created in step #2 above.

4 Finally, create a JSON representation of the Credential class using Smart Card Credential ID as id member and string created in step #3 as a data member.

DeleteUserCredentials

To delete the particular Smart Card credential, the input data for this call must be a string presenting the Smart Card Public key's hash, calculated or received from the server as described in the chapter *Web Smart Card support* on page *153*.

Parameter	Description
String *keyHash*	Public key's hash, Base64url UTF-8 encoded string. The public key from the Smart Card must be imported in the PUBLICKEYBLOB format. After that, the RSA256 hash of the key must be calculated. The resulting 32 bytes must be Base64url encoded to the string:
KeyHash	Base64urlEncode(RSA256 Hash (PUBLICKEYBLOB (PuK))

To delete all the Smart Card credentials enrolled for particular user, the input data for this call must be an empty string (not a NULL pointer).

CustomAction

CustomAction information is specific to each credential. See *CustomAction* under each credential in the *WAS Credentials Data Format* chapter beginning on page *96*.

Face Credential

The following ID is defined for Face Credentials.

`{85AEAA44-413B-4DC1-AF09-ADE15892730A}`

One of the following third-party SDKs can be used to support the Face Credential in your application.

- Cognitec FVSDK ver. 9.1.0
- Innovatrics IFace SDK ver. 3.1.0

All the DigitalPersona Servers and Clients in the environment must use the same SDK. *Enrollment data is not compatible between the SDKs!*

The BioSample class used in description below is declared in "Altus 1 1 WAS Credentials Format.docx"

EnrollUserCredentials

The data for Face credential enrollment is a Base64url encoded UTF-8 representation of the JSON array, containing *BioSample* objects(s).

The following data members for BioSample should be provided for authentication.

Data member	Description
BioFactor	Must be set to 2 (DP_BIO_FACTOR::FACIAL_FEATURES)
BioSamplePurpose	Must be set to 0 (Any purpose) or 3 (purpose Enroll)
BioSampleEncryption	Must be 0 (not encrypted) or 1 (XTEA encryption)
BioSampleType	Must be one of the following: DP_BIO_SAMPLE_TYPE::PROCESSED

Data member	Description
	DP_BIO_SAMPLE_TYPE::RAW

The following types of *BioSampleType* are supported for Face credential enrollment.

DP_BIO_SAMPLE_TYPE::PROCESSED image

Indicates a face template in one of the following internal SDK formats.

- Cognitec FIR format;
- Innovatrics Template format.

When the input credential data is already a face template (type 4), then only one BioSample object is expected in the array.

Data member	Description
BioSampleType	Must be 4.
BioSampleFormat->FormatOwner	"Organization Identifier" number from the International Biometrics Identity Association. • 0x63 (99) for Cognitec • 0x35 (53) for Innovatrics
BioSampleEncryption	Must be 0 (not encrypted) or 1 (XTEA encryption)
Data	Base64url encoded JSON representation of the *CDPJsonFIR* class described below:

```
[DataContract]
public class CDPJsonFIR
{
  [DataMember]
  public Byte Version { get; set; }    // version
  [DataMember]
  public long SDKVersion { get; set; } // SDK version
  [DataMember]
  public String Data { get; set; }     // FIR object
}
```

where:

Data member	Description
Byte *Version*	Specifies the version of the CDPJsonFIR object. It must be set to 1.
ULONG *SDKVersion*	Specifies the version of the SDK: • Cognitec FVSDK, must be not less than 0x90100 (ver. 9.1.0). • Innovatrics IFace SDK, must be not less than 0x30100 (ver. 3.1.0).

Data member	Description
String *Data*	Contains a Base64url encoded BYTE array. • Cognitec SDK: this BYTE array is a serialized Cognitec FIR object, created using *FIR::writeTo()* method. • Innovatrics SDK: this BYTE array is a Template object, created using *IFACE_CreateTemplate* function.

If *BioSampleEncryption* is set to 1 (XTEA encryption), then this data is encrypted.

Below is an example of JSON representation of the *CDPJsonFIR* object containing the Cognitec FIR object.

```
{
  "Version":1,
  "SDKVersion":?590080??, //? ?0x90100?, ver. 9.1.0
  "Data":"2lvbiI6MSwibnVtYmVyIjoyLCJ0ZXh0IjoiTmV3IFlvcmsifSx7InZlcnNpb24iOj
  EsIm51bWJlciI6NiwidGV4dCI6IkNhbnlvbiBNaWRkbGUifSx7InZlcnNpb24iOjEsIm51bW
  JlciI6MTAyLCJ0ZXh0IjoiMDQvMjQvMjAwOSJ9XQ" // Base64url encoded serialized
FIR object
       }
```

Example of JSON representation of the enrollment array containing the face FIR template:

```
{
  [{
  "Version":1,
  "Header":
  {
    "Factor":2,           // Facial features
    "Format":
    {
      "FormatOwner":99,   // Cognitec
      "FormatID":0
    },
    "Type":4,             // Face template
    "Purpose":3,          // Enrollment
    "Quality":-1,
    "Encryption":0        // Unencrypted
  },
    "Data":"eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9" // Base64url encoded
    CDPJsonFIR object
  }]
}
```

DP_BIO_SAMPLE_TYPE::RAW image

Indicates a raw face image. It's recommended to have a minimum of ten *BioSample* objects containing raw images in the authentication array for successful verification.

The only type of raw image supported by the current version is a jpeg file.

Data member	Description
BioSampleType	Must be 1.

Data member	Description
BioSampleFormat->FormatOwner	Not used, must be 0.
Data	Base64url encoded JSON representation of the CDPJsonFaceImage class described below.

```
[DataContract]
public class CDPJsonFaceImage
{
  [DataMember]
  public Byte Version   { get; set; }  // version
  [DataMember]
  public DP_FACE_IMAGE_TYPE ImageType  { get; set; } // type of image
  [DataMember]
  public String ImageData { get; set; } // face image
}
```

where:

Data member	Description
Byte *Version*	version of CDPJsonFaceImage object, must be set to 1.
DP_FACE_IMAGE_TYPE *ImageType*	type of image. Must be set to 1, JPEG_FILE:

```
typedef enum DP_FACE_IMAGE_TYPE
{
   JPEG_FILE = 1, // Base64Url encoded JPEG file
}
DP_FACE_IMAGE_TYPE;
```

String *ImageData* - contains Base64Url encoded raw image data, according to the *ImageType*. In the current version, it's a jpeg file. If BioSampleEncryption is set to 1 (XTEA encryption), this data is encrypted.

Below is an example of JSON representation of the CDPJsonFaceImage object containing the raw face image (jpeg file):

```
{
  "Version":1,
  "ImageType":1,        // JPEG_FILE
  "ImageData":"W3sidmVyc2lvbiI6MSwibnVtYmVyIjoyLCJ0ZXh0IjoiTmV3IFlvcmsifSx7
  InZlcnNpb24iOjEsIm51bWJlciI6NiwidGV4dCI6IkNhbnlvbiBNaWRkbGUifSx7InZlcnNp
  b24iOjEsIm51bWJlciI6MTAyLCJ0ZXh0IjoiMDQvMjQvMjAwOSJ9XQ" // Base64url
  encoded jpeg file
       }
```

Example of JSON representation of the authentication array containing the raw face images (jpeg files).

```
{
  [{
    "Version":1,
    "Header":
    {
      "Factor":2, // Facial features
      "Format":
      {
```

```
      "FormatOwner":0, // Not used
      "FormatID":0
      },
      "Type":1,// Raw image
      "Purpose":1, // Enrollment
      "Quality":-1,
      "Encryption":0 // Unencrypted
    },
"Data":"WF0T3duZXIiOjUxLA0KIkZvcm1hdElEIjowDQp9LA0KIlR5cGUiOjIsDQoiUHVycG9z
…
…
    ZSI6MCwNCiJRdWFsaX" // Base64url encoded CDPJsonFaceImage object
    },
…
…
    {
    "Version":1,
    "Header":
    {
      "Factor":2, // Facial features
      "Format":
      {
      "FormatOwner":0, // Not used
      "FormatID":0
      },
      "Type":1,// Raw image
      "Purpose":1, // Enrollment
      "Quality":-1,
      "Encryption":0 // Unencrypted
    },
  "Data":"SGVhZGVyIjp7IkRldmljZUlkIjowLCJEZXZpY2VUeXBlIjo0OTI2NDQxNzM0NzI3Mjc
  …
  …
  wNCwiaURhdGFBY3F1a" // Base64url encoded CDPJsonFaceImage object
}]
}
```

The following steps will be needed to create the Face credential enrollment packet.

1 Create JSON representation of BioSample(s).

2 Combine those JSON representations in a JSON array ([]).

3 Base64Url encode the string created in step #2.

4 Create a JSON representation of the Credential class using Face Credential ID as *id* member and a string created in step #3 as a *data* member.

NOTE: If user already has the face credentials enrolled at this position, then the old credential data will be replaced with new one. If user does not have the face credentials enrolled at this position, then the face credentials will be added to the user record.

DeleteUserCredentials

The data member of credential argument of DeleteUserCredential should be null and will be ignored. For example:

```
{"id":"85AEAA44-413B-4DC1-AF09-ADE15892730A","data":null}
```

GetEnrollmentDataResult

This method is not supported.

CustomAction

CustomAction information is specific to each credential. See *CustomAction* under each credential in the *WAS Credentials Data Format* chapter beginning on page *96*.

Contactless Card Credential

The following ID is defined for Contactless Card Credential:

```
{F674862D-AC70-48ca-B73E-64A22F3BAC44}
```

GetEnrollmentData

This method is not supported by the Contactless Card credential.

EnrollUserCredentials

The data for the Contactless Card credential is a Base64url encoded UTF-8 representation of the JSON CDPJsonCLCEnrollData class, which is defined as the following:

```
[DataContract]
  public class CDPJsonCLCEnrollData
  {
    [DataMember]
    public Byte version { get; set; }    // version
    [DataMember]
    public String key { get; set; }      // symmetric key
    [DataMember]
    public String nickname { get; set; } // token's nickname
    [DataMember]
    public String UID { get; set; }      // card UID
    [DataMember]
    public int32 address { get; set; }   // address of card record
  }
```

where:

Data member	Description
Byte *Version*	version of CDPJsonCLCEnrollData object, must be set to 1.
String *key*	Card's symmetric key, presented as a Base64Url UTF-8 encoded string. The key must be imported in the PLAINTEXTKEYBLOB format. After that, it must be Base64url encoded to the string: `Key = Base64urlEncode( (PLAINTEXTKEYBLOB (Ps) )`
String *nickname*	The nickname of the smart card token. It's suggested to use the card name, like "iClass ISO 14443 A", as a part of the nickname. The server limits the length of the nickname to 255 symbols, the exceeding symbols will be cut off.
String *UID*	Card's unique ID, array of 64 bytes, presented as Base64url UTF-8 encoded string.

Data member	Description
int32 *address*	A DWORD containing the information on the physical address of the DP CA data on the card. This information will be used to speed up the card access.

To create the Contactless Card Credential for enrollment, following steps must be performed on the client:

1 Read the card UID and the symmetric key stored on the Contactless card, create the SHA 256 hash of this key.

2 Use the key hash as a TOTP seed, generate the OTP.

3 Create a JSON representation of the CDPJsonCLCAuthToken class.

- Use the OTP string created in step #2;
- Base64url UTF-8 encode the card UID;

4 Base64Url encode the JSON representation of the CDPJsonCLCAuthToken class;

5 Create a JSON representation of the Credential class using Contactless Card Credential ID as id member and string created in step #4 as a data member.

DeleteUserCredentials

The data member of credential argument of DeleteUserCredential should be *null* and will be ignored.

CustomAction

CustomAction is not currently supported for the Contactless Card credential.

U2F Device Credential

The following ID is defined for the U2F Credential.

```
{5D5F73AF-BCE5-4161-9584-42A61AED0E48}
```

GetEnrollmentData

This method is not supported.

EnrollUserCredentials

The data for U2F Device credential is a Base64url encoded UTF-8 representation of the JSON representation of the CDPJsonU2FEnrollData class. CDPJsonU2FEnrollData class is defined as following:

```
[DataContract]
  public class CDPJsonU2FEnrollData
  {
    [DataMember]
    public String version { get; set; }
    [DataMember]
    public String appId { get; set; }
    [DataMember]
    public String serialNum { get; set; }
    [DataMember]
    public String registrationData { get; set; }
    [DataMember]
    public String clientData { get; set; }
  }
```

where:

Data member	Description
String *version*	- version of supported FIDO standard, must be set to "U2F_V2" for current implementation;
String *appId*	- facet ID of the caller.
String *serialNum*	- U2F Device serial number.
String *registrationData*	- registration response from U2F device, Base64url UTF-8 encoded string.
String *clientData*	- Base64Url encoded client data (see https://fidoalliance.org/specs/fido-u2f-v1.0-nfc-bt-amendment-20150514/fido-u2f-raw-message-formats.html for details). Client data contains challenged.

DeleteUserCredentials

The data member of credential argument of DeleteUserCredential should be *null* and will be ignored.

CustomAction

None? No entry in document for this cred.

Web Enrollment Sample application 5

THIS CHAPTER DESCRIBES THE SAMPLE PROGRAM, DPWEBDEMO.EXE AND THE API FEATURES THAT IT DEMONSTRATES.

Introduction

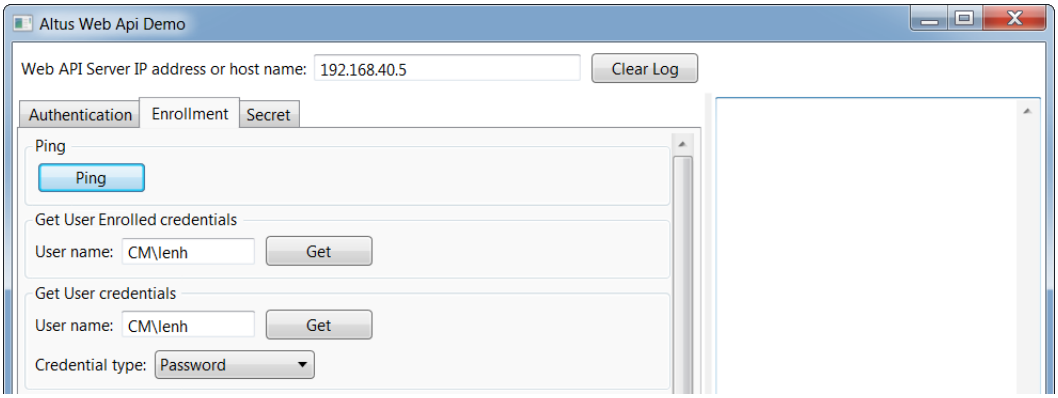
The sample program (DPWebDemo.exe) creates a GUI displaying fields and buttons that showcase the primary features of the DigitalPersona Web Enrollment API. In order to conserve space in this document and make it more readable, illustrations of the window are shown in sections as various features are dsicussed in the text.

We will begin with the second tab, *Enrollment*, since in most cases you will need to first enroll credentials before you can use them on the first tab for authentication or identificaiton.

Although the sample program will function when the associated DigitalPersona Web Enrollment components are installed on either a DigitalPersona LDS Server or DigitalPersona AD Server, the *Create User* and *Delete User* elements only work for a DigitalPersona LDS Server.

Note: By default, most of the user name fields in the sample application are prepopulated with the current (logged on) Windows User name.

Enrollment tab



IP Address or host name

Purpose: Specifies the IP Address or host name where the DigitalPersona Web Enrollment components have been installed and where the service is currently running.

Actions: Enter the IP Address or host name for the DigitalPersona Web Enrollment service.

Results: There are no obvious UI changes in the sample application. However, the application will not function until the IP Address or host name has been entered.

Ping

Purpose: Pings the specified IP Address or host name specified at the top of the sample application window.

Actions: Click *Ping*.

Results: If successful, displays *True*.

Get Enrolled User credentials

Purpose: Gets a list of DigitalPersona credentials that have been enrolled by the user.

Actions: Enter the *domain\User name*. For example, *MyDomain/JohnDoe*. Then click *Get*.

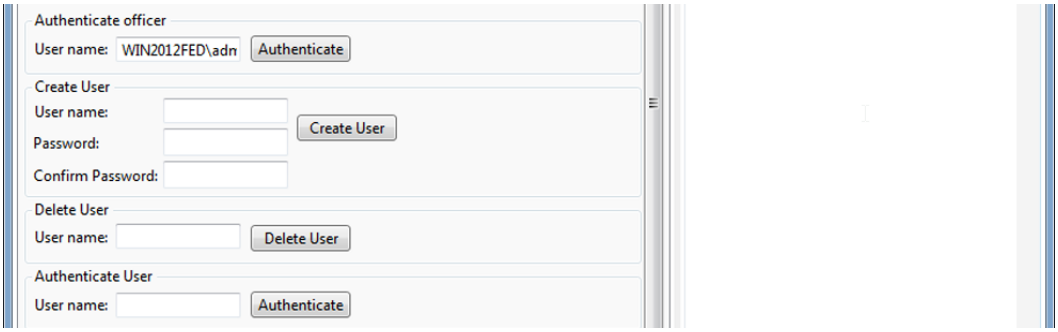
Results: If successful, lists the credentials that the specified user has enrolled.

Get User credential data

Purpose: Displays the value of certain credentials. For example, show the list of specified questions for the LiveQuestions credential and a list of enrolled fingers for the FingerPrint credential.

Actions: Select credential and click *Get*.

Results: If successful, displays the value of the selected credential.



Authenticate officer

Purpose: Authenticates a member of the Security Officers group through their User name and an enrolled credential.

Actions: Click *Authenticate*. Then, in the Credentials window, authenticate with the specified Security Officer's previously enrolled credential.

Results: If successful, displays *Officer token accepted.*

Create User

Purpose: Creates a DigitalPersona (non-domain) user and password.

Actions: Enter a user name and password for the new DigitalPersona LDS user. Then confirm the password and click *Create User*.

Results: If successful, displays *OK.*

Notes: This function in the sample program does not work when the connected web services is running on a DigitalPersona AD server, since it cannot create domain users.

Delete User

Purpose: Deletes a DigitalPersona (non-domain) user.

Actions: Enter the user name of a previously created DigitalPersona LDS user. Then click *Delete User*.

Results: If successful, displays *OK.*

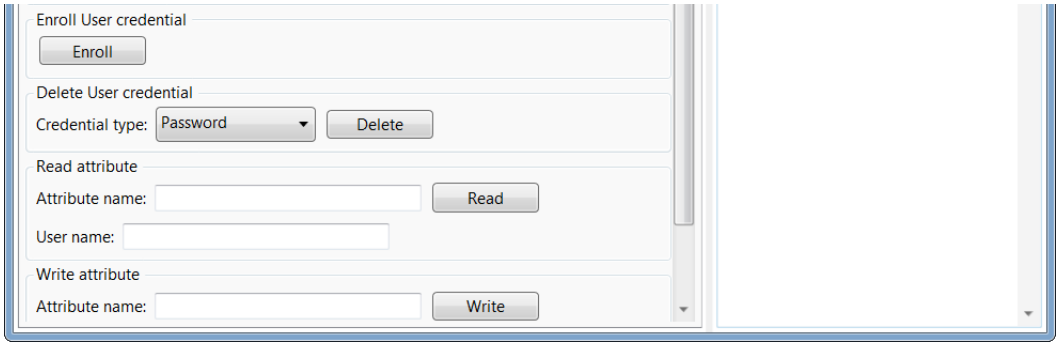
Notes: This function in the sample program does not work when the connected web services is running on a DigitalPersona AD server, since it cannot delete domain users.

Authenticate User

Purpose: Authenticates a DigitalPersona AD (Windows account) or DigitalPersona LDS user.

Actions: Enter the User name to be authenticated and click *Authenticate*. In the Credentials window, perform authentication with any credential previously enrolled for the user.

Results: If successful, displays *User token accepted*. If unsuccessful, displays *The user name or password is incorrect.*



Enroll User credential

Purpose: Provides a demonstration of enrolling the various types of DigitalPersona credentials.

Actions: Click *Enroll* and follow the onscreen instructions for enrolling each specific credential. For additional detailed instructions on enrolling credentials, see your DigitalPersona Client Guide.

Results: If successful, displays *OK*.

Delete User credential

Purpose: Deletes the selected credential.

Actions: Select a credential from the *Password* dropdown menu. Then click *Delete*.

Results: If successful, displays *OK*.

Read attribute

Purpose: Reads the value of a defined attribute for a specified user.

Actions: Enter an attribute name and User name. Then click *Read*.

Results: If successful, displays the value of the attribute.

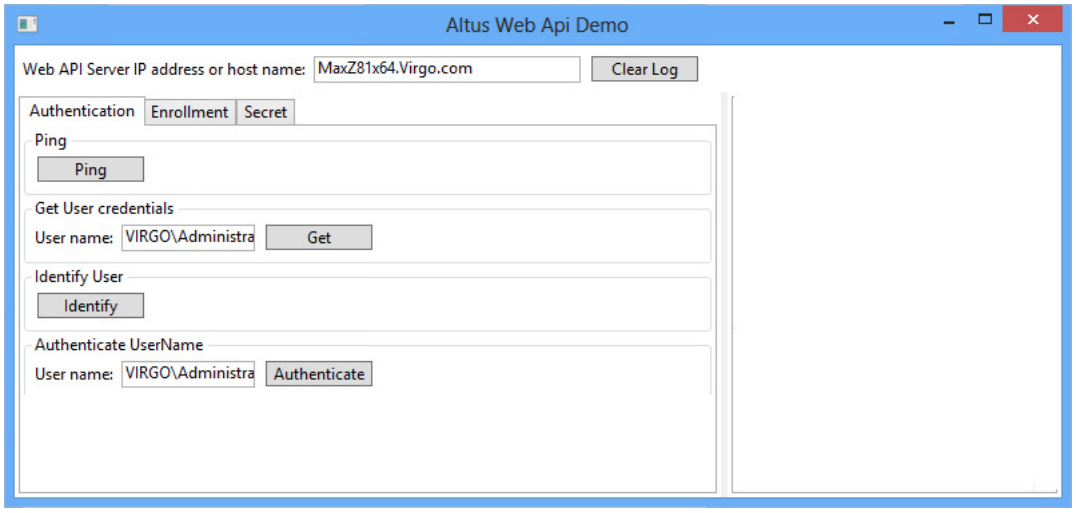
Write attribute

Purpose: Writes a value to a defined attribute for a specified user.

Actions: Enter an *Attribute name*, an *Attribute value* and a valid *User name*. Then click *Write*.

Results: If successful, displays *OK*.

Authentication tab



Ping

Purpose: Pings the specified IP Address or host name specified at the top of the sample application window.

Actions: Click *Ping*.

Results: If successful, displays *OK*.

Get Enrolled User credentials

Purpose: Gets a list of DigitalPersona credentials that have been enrolled by the user.

Actions: Enter the *domain\User name*. For example, *MyDomain/JohnDoe*. Then click *Get*.

Results: If successful, lists the credentials that the specified user has enrolled.

Identify User

Purpose: Identifies the specified user.

Actions: Click *Identify*.

Results: Displays a CredentialsWindow where you can submit a DigitalPersona credential to verify the identity of the specified user. If successful, displays the name of the identified user.

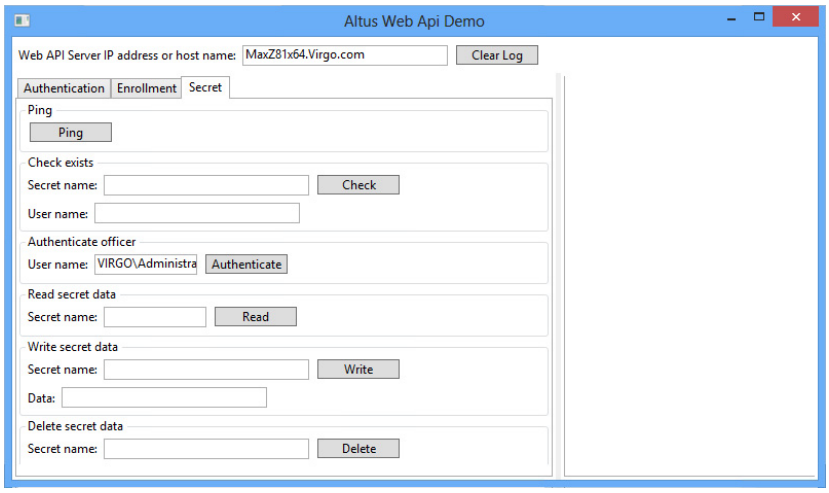
Authenticate UserName

Purpose: Authenticates the specfied UserName

Actions: Click *Authenticate*.

Results: If successful, displays *Ticket acepted*.

Secret tab



Ping

Purpose: Pings the specified IP Address or host name specified at the top of the sample application window.

Actions: Click *Ping*.

Results: If successful, displays *True*.

Check exists

Purpose: Checks whether or not the named secret exists for the specified user.

Actions: Enter Secret name and User name. Then click *Check*.

Results: If successful, displays *True*. Otherwise displays *False*.

Authenticate officer

Purpose: Authenticates a member of the Security Officers group through their User name and password.

Actions: Click *Authenticate*. Then enter the Security Officer's password.

Results: If successful, displays *Officer token accepted.*

Read secret data

Purpose: Reads the data stored in a specified secret for the designated user.

Actions: Enter the secret name. Then click *Read*.

Results: If successful, displays *the value of the secret.*

Write secret data

Purpose: Writes specified secret data to the named secret for a specified user.

Actions: Enter the secret name and data. Then click *Write*.

Results: If successful, displays *OK*.

Delete secret data

Purpose: Deletes specified secret data to the named secret for a specified user.

Actions: Enter the secret name. Then click *Delete*.

Results: If successful, displays *True*.

Section Two - Web Authentication

This section includes the following chapters:

Chapter Number and Title	Purpose	Page
6 - Introduction	Introduces the Web AUTH API.	57
4 - Installation		22
7 - DP Web Authentication Service API		59
8 - DP Web Secret Management Service API		78
9 - DP Web Authentication Policy API		86
6 - Credentials Data Format		40

Introduction 6

THIS CHAPTER PROVIDES AN OVERVIEW OF THE FUNCTIONALITY AND FEATURES OF THE DIGITALPERSONA WEB AUTH API.

The purpose of the DigitalPersona AUTH API is to allow you to add web-based authentication to your web applications. The AUTH API lets you authenticate DigitalPersona users quickly and easily using a DigitalPersona Server and the authentication policy defined by the DigitalPersona administrator.

The following authentication methods provided in DigitalPersona can be accessed through the API.

- Password
- Fingerprint
- Live Questions
- PIN

The DigitalPersona AUTH API provides authentication only. User enrollment must be handled through a DigitalPersona client.

Authentication can be provided to cover the following scenarios.

- Users need to be authenticated by the DigitalPersona Server from trusted clients located outside the enterprise firewall.
- Users need to be authenticated by the DigitalPersona Server from clients located inside the firewall, but “not trusted” i.e. clients which are not part of Active Directory domains.
- Users need to be authenticated by the DigitalPersona Server from clients located outside the firewall and “not trusted”.

When you install a DigitalPersona Workstation or Kiosk client, the DigitalPersona AUTH API runtime is installed as well. As shown in the diagram below, your application runs on workstations that are also running one of the DigitalPersona clients.

Additional features of the API include:

- Authenticating users with the authentication policy used by DigitalPersona Workstation/Kiosk and optionally reading a user secret.
- Retrieving and saving user secrets. Secrets are cryptographically protected and are released to an application only after successful authentication of the user. Secrets are stored in the DigitalPersona database and roam with the rest of the user data.
- Using custom authentication policies which extend the DigitalPersona administrator’s policies or create new policies.

The DigitalPersona AUTH API observes all of the settings in DigitalPersona regarding its communications with the server, supported credentials, policies, etc. Your application can require additional credentials (i.e., you can create a custom authentication policy), but if secret release is required, your application must meet the requirements of the policy set by the DigitalPersona administrator. For additional information about custom policies, see *“Custom Authentication Policies” on page 150*.

Target Audience

This guide is for developers who have a working knowledge of the C++ programming language. In addition, readers should have an understanding of the DigitalPersona DigitalPersona product and its authentication terminology and concepts.

Chapter Overview

Chapter 6, Introduction (this chapter), gives an overview of the API's purpose, describes its audience, cites resources that may assist you in using the API, identifies the minimum system requirements needed to run it and lists the DigitalPersona products supported by the AUTH API.

Chapter 4, *Installation*, contains instructions for installing the API on your development system.

Chapter 7, *DP Web Authentication Service API*, contains descriptions of the DigitalPersona Web Authentication Service API.

Chapter 8, *DP Web Secret Management Service API*, contains descriptions of the DigitalPersona Web Secrets Management Service API.

Chapter 9, *DP Web Authentication Policy API*, contains descriptions of the DigitalPersona Web Authentication Policy Service API.

Chapter 6, *Credentials Data Format*, contains descriptions of the DigitalPersona objects included in the DigitalPersona Credentials class.

Chapter 14, *Custom Authentication Policies*, describes how authentication policies are stored, how to extend an authentication policy and how to define a new authentication policy.

Development System

Minimum software and hardware requirements - Needed to develop applications with the AUTH API.

- Development system running Windows 7 or higher
- To run the sample code and test applications: DigitalPersona DigitalPersona Workstation or Kiosk (see "Supported DigitalPersona products" below for a complete list of compatible clients)
- To compile sample code: Microsoft Visual Studio 2008 or higher

Supported DigitalPersona products

The AUTH API is compatible with the following DigitalPersona products:

- DigitalPersona and DigitalPersona AD Server 1.1 and higher.
- DigitalPersona and DigitalPersona AD Workstation 1.1 and higher.
- DigitalPersona and DigitalPersona AD Kiosk 1.1 and higher.

As of the publication date of this guide, the DigitalPersona Pro 4.3 SDK is not supported and applications written for it cannot be used with the DigitalPersona AUTH API. Check our website or contact tech support for up-to-date information on available upgrade paths and the compatibility patches/upgrades for other DigitalPersona clients.

Target system

Requirements for the target system are the same as specified for the DigitalPersona client being used, i.e. DigitalPersona Workstation or DigitalPersona AD Workstation, or DigitalPersona Kiosk or DigitalPersona AD Kiosk. For these requirements, see the DigitalPersona Client Guide. I assume client is not required on the target system for Web AUTH or Enrollment. What about development system?

DP Web Authentication Service API 7

THIS CHAPTER DESCRIBES THE API FOR THE DIGITALPERSONA WEB AUTHENTICATION SERVICE (DP WAS).

The purpose of the DigitalPersona Web Authentication Service (DP WAS) is user authentication and identification. DPWAS is a GUI-less Web Service which should be located inside the enterprise firewall, and directs authentication (identification) requests to the DigitalPersona Server over DCOM. The DigitalPersona Server is the only one who can process authentication requests.

The result of successful authentication is a JSON token signed by the Web Service.

IDPWebAuth interface

DigitalPersona Web Authentication Service is a WCF(Windows Communication Foundation) service. IDPWebAuth interface is the WCF interface.

```
namespace DPWcfAuthService
{
  [ServiceContract]
  public interface IDPWebAuth
  {
    [OperationContract()]
    /*
    * Return list of credentials enrolled by user
    */
    [WebInvoke(Method = "GET",
      ResponseFormat = WebMessageFormat.Json,
      BodyStyle = WebMessageBodyStyle.Wrapped,
      UriTemplate = "GetUserCredentials?user={userName}&type={userNameType}")]
      List<String> GetUserCredentials(String userName, UInt16 userNameType);
    /*
    * Return credential specific public enrollment information
    */
    [WebInvoke(Method = "GET",
      ResponseFormat = WebMessageFormat.Json,
      BodyStyle = WebMessageBodyStyle.Wrapped,
      UriTemplate = "GetEnrollmentData?user={userName}&type={userNameType}&cred_id={credentialId}")]
      String GetEnrollmentData(String userName, UInt16 userNameType, String credentialId);
    /*
    * Identify user based on credential provided
    */
    [WebInvoke(Method = "POST",
      ResponseFormat = WebMessageFormat.Json,
      BodyStyle = WebMessageBodyStyle.Wrapped,
      UriTemplate = "IdentifyUser")]
      Ticket IdentifyUser(Credential credential);
    /*
    * Authenticate user based on user name and credential provided
    */
    [WebInvoke(Method = "POST",
      ResponseFormat = WebMessageFormat.Json,
      BodyStyle = WebMessageBodyStyle.Wrapped,
      UriTemplate = "AuthenticateUser")]
      Ticket AuthenticateUser(User user, Credential credential);
    /*
```

```
    * Authenticate user based on previously issued ticket and credential provided
    */
    [WebInvoke(Method = "POST",
      ResponseFormat = WebMessageFormat.Json,
      BodyStyle = WebMessageBodyStyle.Wrapped,
      UriTemplate = "AuthenticateUserTicket")]
      Ticket AuthenticateUserTicket(Ticket ticket, Credential credential);
  }
  [DataContract]
  public class User
  {
    [DataMember]
    public String name { get; set; }   // name of the user
    [DataMember]
    public UInt16 type { get; set; }   // form (or type) of user name
  }
  [DataContract]
  public class Credential
  {
    [DataMember]
    public String id { get; set; }     // unique id (Guid) of credential
    [DataMember]
    public String data { get; set; }   // Base64 encoded credential data
  }
  [DataContract]
  public class Ticket
  {
    [DataMember]
    public String jwt { get; set; }    // JSON Web Token
  }
}
```

Methods

The IDPWebAuth interface has 5 methods. Below we give a detailed description of each method.

GetUserCredentials method

GetUserCredentials method is a utility method which allows the caller to determine which particular credentials were enrolled by a user.

Syntax

```
List<String> GetUserCredentials(String userName, UInt16 userNameType);
```

Parameter	Description
userName	Name of the user whose credential needs to be verified.
userNameType	Format of the user name provided in the userName parameter. See **"User class" on page 68** for a detailed description of user name formats.

Return values

List of credential IDs specifying the credentials enrolled by a user. The format of credential IDs is described below in **"Credential class" on page 69**.

Notes

GetUserCredentials should be implemented as HTTP GET using JSON as response format.

The following is an example of the use of GetUserCredentials.

```
https://www.mydomain.com/DPWebAuthService.svc/
GetUserCredentials?user=someone@mycompany.com&type=6
```

Here the user name is someone@mycompany.com and the user name type is 6 which means UPN.

The example response would be the following:

```
{"GetUserCredentialsResult":
  [
  "D1A1F561-E14A-4699-9138-2EB523E132CC",
  "AC184A13-60AB-40e5-A514-E10F777EC2F9",
  "8A6FCEC3-3C8A-40c2-8AC0-A039EC01BA05"
  ]
}
```

This response means that the user someone@mycompany.com has password, fingerprint and PIN credentials enrolled.

GetEnrollmentData method

GetEnrollmentData method is a utility method which allows the caller to get some credential enrollment specific data. For example Live Question authentication may require knowing which particular questions were enrolled.

Syntax

```
String GetEnrollmentData(String userName, UInt16 userNameType, String credentialId);
```

Parameter	Description
userName	Name of the user whose enrollment data needs to be returned.
userNameType	Format of the user name provided in the userName parameter. See **"User class" on page 68** for a detailed description of user name formats.
credentialId	Unique ID of credential whose data needs to be returned. The format of credential IDs is described below in **"Credential class" on page 69**.

Return values

Base64 encoded enrollment data. The format of this enrollment data is credential-specific.

Notes

GetEnrollmentData should be implemented as HTTP GET using JSON as the response format.

This is an example of using GetEnrollmentData:

```
https://www.mydomain.com/DPWebAuthService.svc/GetEnrollmentData?
user=someone@mycompany.com&type=6&cred_id=AC184A13-60AB-40e5-A514-E10F777EC2F9
```

Here the user name is someone@mycompany.com, the user name type is 6 which means UPN name and the credential ID is {AC184A13-60AB-40e5-A514-E10F777EC2F9} which means that information about the fingerprint credential is being requested.

The use of braces { } is considered unsafe in URLs (see RFC 1738), which is why we use “braceless” GUID representations in our API.

The example response would be the following:

```
{"GetEnrollmentDataResult":”Z3NhZGhhc2Rma0FTREZLYWZyZGtB”}
```

This response has fingerprint enrollment data for the user someone@mycompany.com.

IdentifyUser method

The IdentifyUser method allows identification of the user based on a provided credential. Note that not all credentials support identification. For example, fingerprints support identification but password doesn’t.

Syntax

```
Ticket IdentifyUser(Credential credential);
```

Return values

Parameter	Description
credential	Credential ID and data which needs to be used for identification.

As a result of successful identification a Ticket will be returned. The format of this ticket is described in the section “JSON Web Token (JWT)” beginning on page *71*.

Notes

IdentifyUser should be implemented as HTTP POST using JSON as the response format. Below is an example of URL which can be used to POST IdentifyUser request:

```
https://www.mydomain.com/DPWebAuthService.svc/IdentifyUser
```

Below is an example of HTTP BODY of an IdentifyUser request.

```
{
  “credential”:
  {
  “id":”AC184A13-60AB-40e5-A514-E10F777EC2F9”,
  “data”:”Z3NhZGhhc2Rma0FTREZLYWZyZGtB”
  }
}
```

This is a request to identify the user with a provided fingerprint credential.

The following is an example of an IdentifyUser response.

```
{"IdentifyUserResult":{“jwt”:”eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9.eyJpc3
MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb29
0Ijp0cnVlfQ.dBjftJeZ4CVP-B92K27uhbUJU1p1rwW1gFWFOEjXk”}}
```

AuthenticateUser method

The AuthenticateUser method allows authentication of a user based on a provided credential.

Syntax

```
Ticket AuthenticateUser(User user,Credential credential);
```

Parameter	Description
user	User identity which needs to be authenticated.
credential	Credential ID and data to be used for identification.

Return values

As a result of successful authentication a Ticket will be returned. The format of this ticket will be described below in details.

Notes

AuthenticateUser should be implemented as HTTP POST using JSON as the response format. Below is an example of a URL which can be used to POST the AuthenticateUser request.

```
https://www.mydomain.com/DPWebAuthService.svc/AuthenticateUser
```

Below is example of the HTTP BODY of an AuthenticateUser request. This is a request to authenticate the user “someone@mycompany.com” with a provided fingerprint credential.

```
{
  “user”:
  {
    “name”:”someone@mycompany.com”,
    “type”:6
  },
  “credential”:
  {
    “id":”AC184A13-60AB-40e5-A514-E10F777EC2F9”,
    “data”:”Z3NhZGhhc2Rma0FTREZLYWZyZGtB”
  }
}
```

The following is an example of an AuthenticateUser response.

```
{"AuthenticateUserResult":{“jwt”:”eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ
9.eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmN
vbS9pc19yb290Ijp0cnVlfQ.dBjftJeZ4CVP-B92K27uhbUJU1p1rwW1gFWFOEjXk”}}
```

AuthenticateUserTicket method

The AuthenticateUserTicket method allows authentication of a user through a previously issued ticket and the provided credential.

Syntax

```
Ticket AuthenticateUserTicket(Ticket ticket, Credential credential);
```

Parameter	Description
ticket	Ticket previously issued by the authentication authority.
credential	Credential ID and data to be used for identification.

Return values

As a result of successful authentication a Ticket will be returned.

AuthenticateUserTicket should be implemented as HTTP POST using JSON as the response format. Below is an example of a URL which can be used to POST the AuthenticateUserTicket request.

```
https://www.mydomain.com/DPWebAuthService.svc/AuthenticateUserTicket
```

Below is example of the HTTP BODY for an AuthenticateUserTicket request. This is a request to authenticate the user whose name is encoded in JWT with the provided fingerprint credential.

```
{
  "ticket":
  {
    "jwt":
    "eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJqb2UiLA0KICJleHAiOjE
    zMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ.dBjftJeZ4
    CVP-B92K27uhbUJU1p1rwW1gFWFOEjXk"
  },
  "credential":
  {
    "id":"AC184A13-60AB-40e5-A514-E10F777EC2F9",
    "data":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"
  }
}
```

Below is an example of the AuthenticateUserTicket response.

```
{"AuthenticateUserTicketResult":{"jwt":"eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1
NiJ9.eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmN
vbS9pc19yb290Ijp0cnVlfQ.dBjftJeZ4CVP-B92K27uhbUJU1p1rwW1gFWFOEjXk"}}
```

CustomAction method

The CustomAction method allows to call some specific custom action for specific authentication token.

Syntax

```
String CustomAction(Ticket ticket, User user, Credential credential, UInt16 actionId);
```

Parameter	Description
ticket	Ticket previously issued by the authentication authority.
user	User to which credential custom action would need to apply.
credential	Credential ID and data to be used for identification.
actionId	ID of specific action needs to be apply. This ID is specific for each authentication token.

Return values

CustomAction may or may not return some data. This information is specific for particular authentication token.

CustomAction should be implemented as HTTP POST using JSON as response format.

Below is example of URL which can be used to POST CustomAction request:

```
https://www.digitalpersona.com/DPWebAuthService.svc/CustomAction
```

Below is example of HTTP BODY of CustomAction request:

```
{
  "ticket":
  {
```

```
    "jwt":
    "eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJqb2UiLA0KICJleHAiOjE
    zMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ.dBjftJeZ4
    CVP-B92K27uhbUJU1p1rwW1gFWFOEjXk"
  },
  "user":
  {
    "name":"someone@mycompany.com",
    "type":6
  },
  "credential":
  {
    "id":"AC184A13-60AB-40e5-A514-E10F777EC2F9",
    "data":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"
  },
  "actionId":6
}
```

This is request to perform custom action #6 for fingerprint authentication token for user someone@mycompany.com.

CreateUserAuthentication method

CreateUserAuthentication creates Extended Authentication for specific user and specific authentication token (credential).

Syntax

```
UInt32 CreateUserAuthentication(User user, String credentialId);
```

Parameter	Description
user	User for whom extended authentication needs to be created.
credentialId	Credential ID of authentication token (credential) for which extended authentication needs to be created.

Return values

As a result unique extended authentication handle will be returned.

CreateUserAuthentication should be implemented as HTTP POST using JSON as response format.

Below is example of URL which can be used to POST CreateUserAuthentication request:

```
https://www.mycompany.com/DPWebAuthService.svc/CreateUserAuthentication
```

Below is example of HTTP BODY of CreateUserAuthentication request:

```
{
 "user":
{
"name":"klozin@digitalpersona.com",
"type":6
},
 "credentialId":"AC184A13-60AB-40e5-A514-E10F777EC2F9"
}
```

This is request to create extended authentication for user someone@mycompany.com and for fingerprint authentication token.

The following is an example of CreateUserAuthentication response:

```
{"CreateUserAuthenticationResult":52346}
```

CreateTicketAuthentication method

The CreateTicketAuthentication method creates Extended Authentication based on existent Ticket and specific authentication token (credential).

Syntax

```
UInt32 CreateTicketAuthentication(Ticket ticket, String credentialId);
```

Parameter	Description
ticket	Ticket previously issued by authentication authority.
credentialId	Credential ID of authentication token (credential) for which extended authentication needs to be created.

Return values

As a result unique extended authentication handle will be returned.

CreateTicketAuthentication should be implemented as HTTP POST using JSON as response format.

Below is example of URL which can be used to POST CreateTicketAuthenticationrequest:

```
https://www.mycompany.com/DPWebAuthService.svc/CreateTicketAuthentication
```

Below is example of HTTP BODY of CreateTicketAuthentication request:

```
{
  "ticket":
  {
  "jwt":
  "eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJqb2UiLA0KICJleHAiOjE
  zMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ.dBjftJeZ4
  CVP-B92K27uhbUJU1p1rwW1gFWFOEjXk"
  },
  "credentialId":"AC184A13-60AB-40e5-A514-E10F777EC2F9"
}
```

This is request to create extended authentication based on Ticket provided and for fingerprint authentication token.

The following is an example of CreateTicketAuthentication response:

```
{"CreateTicketAuthenticationResult":52346}
```

ContinueAuthentication method

ContinueAuthentication perform one step of Extended Authentication. Extended Authentication would require handshake with two or even more steps so ContinueAuthentication could be called multiple times.

Syntax

```
ExtendedAUTHResult ContinueAuthentication(UInt32 authId, String authData);
```

Parameter	Description
authId	Extended Authentication handle returned by CreateUserAuthentication or CreateTicketAuthentication call.
authData	Authentication token specific authentication data.

Return values

Extended Authentication Result will be returned (see 4.10.12 ExtendedAUTHResult data stricture for details).

ContinueAuthentication should be implemented as HTTP POST using JSON as response format.

Below is example of URL which can be used to POST ContinueAuthentication request:

```
https://www.mycompany.com/DPWebAuthService.svc/ContinueAuthentication
```

Below is example of HTTP BODY of ContinueAuthentication request:

```
{
  "authId": 52346,
  "authData":
    "eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJqb2UiLA0KICJleHAiOj
    EzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ.dBjftJe
    Z4CVP-B92K27uhbUJU1p1rwW1gFWFOEjXk"
}
```

This is request for step in extended authentication.

The following is an example of ContinueAuthentication response:

```
{"ContinueAuthenticationResult":
  {
    "status":2,
    "authData":null,
    "jwt":"eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJqb2UiLA0KICJl
    eHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ.dB
    jftJeZ4CVP-B92K27uhbUJU1p1rwW1gFWFOEjXk"
  }
}
```

In this example status 2 means Extended Authentication successfully completed and JSON Web Token is returned as result of successful authentication in "jwt" parameter.

DestroyAuthentication method

DestroyAuthentication destroys previously created Extended Authentication and clear all associated resources. NOTE: it is not necessary to call DestroyAuthentication, it will be automatically destroyed upon successful (or unsuccessful) completion. Call DestroyAuthentication only if you want to stop authentication which is not completed yet.

Syntax

```
void DestroyAuthentication(UInt32 authId);
```

Parameter	Description
authId	Extended Authentication handle returned by CreateUserAuthentication or CreateTicketAuthentication call.

Return values

None.

DestroyAuthentication should be implemented as HTTP DELETE using JSON as response format.

Below is example of URL which can be used to DELETE DestroyAuthentication request:

```
https://www.mycompany.com/DPWebAuthService.svc/DestroyAuthentication
```

Below is example of HTTP BODY of DestroyAuthentication request:

```
{
"authId": 52346
}
```

This is request destroy extended authentication with handle 52346.

Data Contracts

Below are Data Contracts used in this API.

User class

User class is user representation in this API.

```
public class User
  {
  [DataMember]
  public String name { get; set; } // name of the user
  [DataMember]
  public UInt16 type { get; set; } // form (or type) of user name
  }
```

Parameter	Description
name	String which represents the name of the user.
type	Integer which represents the format of the user name string.

The following user name formats are currently supported.

Type	Description
3	Windows NT® 4.0 account name (for example, digital_persona\klozin). The domain-only version includes trailing backslashes (\\).
4	Account name format used in Microsoft® Windows NT® 4.0. For example, "someone";
5	GUID string that the IIDFromString function returns (for example, {4fa050f0-f561-11cf-bdd9-00aa003a77b6}).
7	User principal name (for example, someone@mycompany.com).
8	User SID string (for example, S-1-5-21-1004).
9	DigitalPersona user name format (user name associated with DigitalPersona identity database).

Below is an example of the JSON representation of a user.

```
{
  "user":
  {
  "name":"someone@mycompany.com",
  "type":6
  }
}
```

Credential class

Credential class is credential representation in this API.

```
public class Credential
  {
  [DataMember]
  public String id { get; set; }     // unique id (Guid) of credential
  [DataMember]
  public String data { get; set; }   // Base64 encoded credential data
  }
```

Parameter	Description
id	Unique Id of credential.
data	Base64url encoded credential data. The format of this data depends on the credential and is explained in the chapter **"Credentials Data Format" on page 44**.

The following credentials are currently supported.

```
{D1A1F561-E14A-4699-9138-2EB523E132CC} - User password;
{AC184A13-60AB-40e5-A514-E10F777EC2F9} - Fingerprints;
{8A6FCEC3-3C8A-40c2-8AC0-A039EC01BA05} - PIN;
{D66CC98D-4153-4987-8EBE-FB46E848EA98} - Smart Cards;
{1F31360C-81C0-4EE0-9ACD-5A4400F66CC2} - Proximity cards;
{7BF3E290-5BA5-4C2D-AA33-24B48C189399} - Contactless cards;
{B49E99C6-6C94-42DE-ACD7-FD6B415DF503} - Live Questions;
{E750A180-577B-47f7-ACD9-F89A7E27FA49} - Bluetooth;
[MISSING CREDENTIALS]
```

Note

You must use "braceless" GUID representation in our API in URLs and JSON representation. Below is an example of the JSON representation of a credential.

```
{
  "credential":
  {
    "id":"AC184A13-60AB-40e5-A514-E10F777EC2F9",
    "data":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"
  }
}
```

Ticket class

The result of successful authentication is a Ticket.

```
public class Ticket
  {
```

```
  [DataMember]
  public String jwt { get; set; }    // JSON Web Token
  }
```

The format of the Ticket is JSON Web Token (JWT). See the next section for further details on JWT.

AuthenticationStatus enumeration

AuthenticationStatus is enumeration for status of extended authentication operation.

```
public enum AuthenticationStatus
{
  /// <summary>
  /// Error happened. Authentication attempt failed.
  /// </summary>
  [EnumMember]
  Error = 0,

  /// <summary>
  /// Authentication step succeeded but authentication continuation is
required.
  /// </summary>
  [EnumMember]
  Continue = 1,

  /// <summary>
  /// Authentication successfully completed.
  /// </summary>
  [EnumMember]
  Completed = 2,
}
```

Status	Description
Error	Extended Authentication operation is failed.
Continue	Extended Authentication operation is succeeded but continuation is required.
Completed	Extended Authentication operation is successfully completed.

ExtendedAUTHResult class

ExtendedAUTHResult class represents result of Extended Authentication operation.

```
public class ExtendedAUTHResult
{
  [DataMember]
  public AuthenticationStatus status { get; set; } // Status of extended
    authentication operation.
  [DataMember]
  public String authData { get; set; } // Authentication data returned by the
    Server.
  [DataMember]
  public String jwt{ get; set; }       // JWT of successful authentication
    operation.
    }
```

Parameter	Description
status	Status of Extended Authentication operation (see 4.11.4 for details).
authData	Authentication data returned by server. This data would be required to proceed with authentication handshake. This is optional parameter and could be returned if operation status is Continue (handshake continuation is required).
jwt	JSON Web Token returned by server upon successful authentication. This is optional parameter and should be returned if operation status is Completed (authentication handshake is suucessfuly completed).

JSON Web Token (JWT)

The format of a returned Ticket upon successful authentication is a JSON Web Token. The official specification of JWT is detailed here: https://tools.ietf.org/html/draft-jones-json-web-token-10.

The JWT token returned by the DigitalPersona Web Authentication Service is constructed as follows.

The standard JWT has three parts: 1) Header, 2) Claims and 3) Signature.

All three parts are created separately as UTF-8 strings, then Base64url encoded and finally concatenated in the order above with periods between the parts, yielding the complete JWT.

Base64url Encoding - for the purposes of this specification, this term always refers to the URL- and filename-safe Base64 encoding described in RFC 4648 [RFC4648], Section 5, with the (non URL-safe) '=' padding characters omitted, as permitted by Section 3.2.

JWT Header

In the current version of the API, we use the following JWT header.

```
{"typ":"JWT",
 "alg":" RS256"} –
```

Value	Description
"typ":"JWT"	means this is a JSON Web Token.
"alg":" RS256"	means we will use RSA with a SHA-256 hash algorithm for the signature.

JWT Claims

The following claims must be included in our JWT (claims order is not defined so they can be presented in any order.

"jti" (JWT ID) Claim

The "jti" (JWT ID) claim provides a unique identifier for the JWT. The identifier value must be assigned in a manner that ensures that there is a negligible probability that the same value will be accidentally assigned to a different data object. The "jti" claim can be used to prevent the JWT from being replayed. The "jti" value is case sensitive. Its value must be a string.

The “jti” claim should be a string representation of the GUID randomly generated by the DigitalPersona Server during JWT creation.

Below is an example of this claim.

```
"jti":"AC184A13-60AB-40e5-A514-E10F777EC2F9"
```

"iss" (Issuer) Claim

The "iss" (issuer) claim identifies the principal that issued the JWT. For this API, the issuer will always be the DigitalPersona Server, so we include the DigitalPersona Server DNS name in this claim.

Below is an example of this claim.

```
"iss":"altus-01.mydomain.com"
```

"dom" (Issuer Domain) Claim

The "dom" (issuer domain) claim identifies the domain that issued the JWT. For DigitalPersona (AD LDS) users, the issuer domain is the DigitalPersona (AD LDS) instance. For DigitalPersona AD users, the issuer domain is the NETBIOS Domain name that the user belong to.

Below is an example of this claim.

```
"dom":"Software_Department"
```

"iat" (Issued At) Claim

The "iat" (issued at) claim identifies the date and time when the JWT was issued. This claim can be used to determine the age of the token. Its value must be a number containing an IntDate value.

The time is always in UTC Unix time format, an integral value representing the number of seconds elapsed since 00:00 hours, i.e. Jan 1, 1970 UTC.

Below is an example of this claim.

```
"iat”:1300819380
```

"exp" (Expiration Time) Claim

The exp (expiration time) claim identifies the expiration time on or after which the JWT ***must not*** be accepted for processing. The processing of the exp claim requires that the current date/time MUST be before the expiration date/time listed in the exp claim. Implementers ***may*** provide for some small leeway, usually no more than a few minutes, to account for clock skew. Its value ***must*** be a number containing a IntDate value. Use of this claim is optional.

The time is always in UTC Unix time format - it is generally implemented as an integral value representing the number of seconds elapsed since 00:00 hours, Jan 1, 1970 UTC.

Below is example of this claim:

```
"exp":1300819380
```

sub" (Subject) Claim

The "sub" (subject) claim identifies the principal that the JWT was issued to. This should be a readable string specifying the identity of the authenticated user; for example, the user’s display name or account name.

Below is an example of this claim.

```
"sub":”Tom Jones”
```

"uid" (Subject Unique ID) Claim

The "uid" (subject UID) claim uniquely identifies the principal (inside a particular DigitalPersona instance) that the JWT is issued to. This is the objectGUID of the user record in the DigitalPersona (AD LDS) database.

Below is an example of this claim.

```
"uid":”6E2A0211-59E0-4EFA-89C5-68F75E6CE8B7”
```

"crd" (Credentials) Claim

The "crd" (Credentials) claim must list all the credentials used for the authentication, providing the credential id and the authentication timestamp in the Unix time format.

Below is an example of this claim.

```
"crd":[
      {“id”:"D1A1F561-E14A-4699-9138-2EB523E132CC",”time”: 1300819380},
      {“id”:”AC184A13-60AB-40e5-A514-E10F777EC2F9",”time”: 7865233679}
      ]
```

The claim above means that two types of credentials were used for authentication at two separate times.

NOTE: The JWT claim does not include an expiration time, providing the flexibility for the Service Provider to decide to accept this ticket or reject it based on their own logic. It is assumed that the SP clock is in sync with the DigitalPersona Server clock.

"wan" (Windows Account Name) Claim

The "wan" (Windows Account Name) claim uniquely identifies the user that the JWT is issued to. This claim is optional and will be issued only for Active Directory users and provides user SAM account name of the user (name format <NETBIOS Domain Name\user account name>. For DigitalPersona users this claim will be omitted.

Below is an example of this claim:

```
"wan":"mydomain\someone"
```

"t24" (T24 Name) Claim

The "t24" (Windows Account Name) claim uniquely identifies the principal to whom JWT is issued inside the T24 database. This claim is optional and will be issued only for T24 users.

Below is an example of this claim:

```
"t24":"someone"
```

Example of JWT claims section

Below is an example of the claims part of a JWT:

```
{"jti":"AC184A13-60AB-40e5-A514-E10F777EC2F9",
"dom":"Software_Department",
"iss":"altus-01.mycompany.com",
"iat":1300819380,
“exp”:1300819623
"sub":”Kirill Lozin”,
"uid":”6E2A0211-59E0-4EFA-89C5-68F75E6CE8B7”
"crd":[
      {"id":"D1A1F561-E14A-4699-9138-2EB523E132CC","time": 1300819380},
      {"id":"AC184A13-60AB-40e5-A514-E10F777EC2F9","time": 7865233679}
      ]}
```

The token above claims user “someone@mycompany.com” was authenticated using password and fingerprint credentials on server “altus-01.mycompany.com”.

JSON Web Signature (JWS)

The detailed description for the JWS (JSON Web Signature) is available at: https://tools.ietf.org/html/draft-jones-json-web-signature-04.

Rules for Creating and Validating a JWS

To create a JWS, you need to perform the following steps.

1 Create the content to be used as the JWS Payload (JWT Claims in our case).

2 Base64url encode the bytes of the JWS Payload. This encoding becomes the Encoded JWS Payload.

3 Create a JWS Header containing the desired set of header parameters (JWT Header in our case). Note that white space is explicitly allowed in the representation and no canonicalization is performed before encoding.

4 Base64url encode the bytes of the UTF-8 representation of the JWS Header to create the Encoded JWS Header.

5 Compute the JWS Signature in the manner defined for the particular algorithm being used. The JWS Signing Input is always the concatenation of the Encoded JWS Header, a period ('.') character, and the Encoded JWS Payload. The "alg" header parameter must be present in the JSON Header, with the algorithm value accurately representing the algorithm used to construct the JWS Signature.

6 Base64url encode the representation of the JWS Signature to create the Encoded JWS Signature.

When validating a JWS, the following steps must be taken. If any of the listed steps fails, then the signed content must be rejected.

1 The Encoded JWS Header must be successfully base64url decoded following the restriction given in this specification that no padding characters have been used.

2 The JWS Header must be completely valid JSON syntax conforming to RFC 4627 [RFC4627].

3 The JWS Header must be validated to only include parameters and values whose syntax and semantics are both understood and supported.

4 The Encoded JWS Payload must be successfully base64url decoded following the restriction given in this specification that no padding characters have been used.

5 The Encoded JWS Signature must be successfully base64url decoded following the restriction given in this specification that no padding characters have been used.

6 The JWS Signature must be successfully validated against the JWS Header and JWS Payload in the manner defined for the algorithm being used, which must be accurately represented by the value of the "alg" header parameter, which must be present.

Processing a JWS inevitably requires comparing known strings to values in the header. For example, in checking what the algorithm is, the Unicode string encoding "alg" will be checked against the member names in the JWS Header to see if there is a matching header parameter name. A similar process occurs when determining if the value of the "alg" header parameter represents a supported algorithm.

Comparisons between JSON strings and other Unicode strings must be performed as specified below.

1 Remove any JSON applied escaping to produce an array of Unicode code points.

2 Unicode Normalization [USA15] must NOT be applied at any point to either the JSON string or to the string it is to be compared against.

3 Comparisons between the two strings must be performed as a Unicode code point to code point equality comparison.

Creating a JWS with RSA SHA-256

This section defines the use of the RSASSA-PKCS1-v1_5 signature algorithm as defined in RFC 3447 [RFC3447], Section 8.2 (commonly known as PKCS#1), using SHA-256, SHA-384, or SHA-512 as the hash function. The RSASSA-PKCS1-v1_5 algorithm is described in FIPS 186-3 [FIPS.186-3], Section 5.5, and the SHA-256, SHA-384, and SHA-512 cryptographic hash functions are defined in FIPS 180-3 [FIPS.180-3].

The "alg" header parameter values "RS256" are used in the JWS Header to indicate that the Encoded JWS Signature contains a base64url encoded RSA signature using the respective hash function.

The public keys employed will be distributed using methods that are outside the scope of this document.

A 2048-bit or longer key length must be used with this algorithm.

The RSA SHA-256 signature is generated as follows.

1 Generate a digital signature of the UTF-8 representation of the JWS Signing Input using RSASSA-PKCS1-V1_5-SIGN and the SHA-256 hash function with the desired private key. The output will be a byte array.

2 Base64url encode the resulting byte array. The output is the Encoded JWS Signature for that JWS.

The RSA SHA-256 signature for a JWS is validated as follows.

1 Take the Encoded JWS Signature and base64url decode it into a byte array. If decoding fails, the signed content must be rejected.

2 Submit the UTF-8 representation of the JWS Signing Input and the public key corresponding to the private key used by the signer to the RSASSA-PKCS1-V1_5-VERIFY algorithm using SHA-256 as the hash function.

3 If the validation fails, the signed content must be rejected.

JWT Example

The following example JWS Header declares that the data structure is a JSON Web Token (JWT) [JWT] and the JWS Signing Input is signed using the RSA SHA-256 algorithm. Note that white space is explicitly allowed in JWS Header strings and no canonicalization is performed before encoding.

```
{"typ":"JWT",
  "alg":" RS256"}
```

The following byte array contains the UTF-8 characters for the JWS Header:

```
[123,34,116,121,112,34,58,34,74,87,84,34,44,10,32,34,97,108,103,34,58,34,32,
82,83,50,53,54,34,125]
```

Base64url encoding this UTF-8 representation yields this Encoded JWS Header value.

```
eyJ0eXAiOiJKV1QiLAogImFsZyI6IiBSUzI1NiJ9
```

The JWS Payload used in this example follows. Note that the payload can be any base64url encoded content, and need not be a base64url encoded JSON object.

```
{"jti":"AC184A13-60AB-40e5-A514-E10F777EC2F9",
"dom":"Software_Department",
"iss":"altus-01.mycompany.com",
"iat":1300819380,
"sub":”Kirill Lozin”,
"uid":”6E2A0211-59E0-4EFA-89C5-68F75E6CE8B7”
"crd":[
      {"id":"D1A1F561-E14A-4699-9138-2EB523E132CC","time": 1300819380},
      {"id":"AC184A13-60AB-40e5-A514-E10F777EC2F9","time": 7865233679}
      ]}
```

The following byte array contains the UTF-8 characters for the JWS Payload:

```
[123,34,106,116,105,34,58,34,123,65,67,49,56,52,65,49,51,45,54,48,65,66,45,52
,48,101,53,45,65,53,49,52,45,69,49,48,70,55,55,55,69,67,50,70,57,125,34,44,10
,34,105,115,115,34,58,34,97,108,116,117,115,45,48,49,46,100,105,103,105,116,9
7,108,112,101,114,115,111,110,97,46,99,111,109,34,44,10,34,105,97,116,34,58,4
9,51,48,48,56,49,57,51,56,48,44,10,34,115,117,98,34,58,123,34,110,97,109,101,
34,58,34,107,108,111,122,105,110,64,100,105,103,105,116,97,108,112,101,114,11
5,111,110,97,46,99,111,109,34,44,34,116,121,112,101,34,58,54,125,44,10,34,99,
114,100,34,58,91,10,123,34,105,100,34,58,34,123,68,49,65,49,70,53,54,49,45,69
,49,52,65,45,52,54,57,57,45,57,49,51,56,45,50,69,66,53,50,51,69,49,51,50,67,6
7,125,34,44,34,116,105,109,101,34,58,32,49,51,48,48,56,49,57,51,56,48,125,44,
10,123,34,105,100,34,58,34,123,65,67,49,56,52,65,49,51,45,54,48,65,66,45,52,4
8,101,53,45,65,53,49,52,45,69,49,48,70,55,55,55,69,67,50,70,57,125,34,44,34,1
16,105,109,101,34,58,32,55,56,54,53,50,51,51,54,55,57,125,10,93,125]
```

Base64url encoding the above yields the Encoded JWS Payload value (with line breaks for display purposes only):

```
eyJqdGkiOiJ7QUMxODRBMTMtNjBBQi00MGU1LUE1MTQtRTEwRjc3N0VDMkY5fSIsCiJpc3MiOiJh
bHR1cy0wMS5kaWdpdGFscGVyc29uYS5jb20iLAoiaWF0IjoxMzAwODE5MzgwLAoic3ViIjp7Im5h
bWUiOiJrbG96aW5AZGlnaXRhbHBlcnNvbmEuY29tIiwidHlwZSI6Nn0sCiJjcmQiOlsKeyJpZCI6
IntEMUExRjU2MS1FMTRBLTQ2OTktOTEzOC0yRUI1MjNFMTMyQ0N9IiwidGltZSI6IDEzMDA4MTkz
ODB9LAp7ImlkIjoie0FDMTg0QTEzLTYwQUItNDBlNS1BNTE0LUUxMEY3NzdFQzJGOX0iLCJ0aW1l
IjogNzg2NTIzMzY3OX0KXX0
```

Concatenating the Encoded JWS Header, a period character, and the Encoded JWS Payload yields this JWS Signing Input value (with line breaks for display purposes only):

```
eyJ0eXAiOiJKV1QiLAogImFsZyI6IiBSUzI1NiJ9
.
eyJqdGkiOiJ7QUMxODRBMTMtNjBBQi00MGU1LUE1MTQtRTEwRjc3N0VDMkY5fSIsCiJpc3MiOiJh
bHR1cy0wMS5kaWdpdGFscGVyc29uYS5jb20iLAoiaWF0IjoxMzAwODE5MzgwLAoic3ViIjp7Im5h
bWUiOiJrbG96aW5AZGlnaXRhbHBlcnNvbmEuY29tIiwidHlwZSI6Nn0sCiJjcmQiOlsKeyJpZCI6
IntEMUExRjU2MS1FMTRBLTQ2OTktOTEzOC0yRUI1MjNFMTMyQ0N9IiwidGltZSI6IDEzMDA4MTkz
ODB9LAp7ImlkIjoie0FDMTg0QTEzLTYwQUItNDBlNS1BNTE0LUUxMEY3NzdFQzJGOX0iLCJ0aW1l
IjogNzg2NTIzMzY3OX0KXX0
```

Running the RSA SHA-256 algorithm on the UTF-8 representation of the JWS Signing Input with this key yields the following byte array.

```
[116, 24, 223, 180, 151, 153, 224, 37, 79, 250, 96, 125, 216, 173, 187, 186,
22, 212, 37, 77, 105, 214, 191, 240, 91, 88, 5, 88, 83, 132, 141, 121]
```

Base64url encoding the above output yields the Encoded JWS Signature value:

```
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

The following string represents the final JWT.

```
eyJ0eXAiOiJKV1QiLAogImFsZyI6IiBSUzI1NiJ9
.
eyJqdGkiOiJ7QUMxODRBMTMtNjBBQi00MGU1LUE1MTQtRTEwRjc3N0VDMkY5fSIsCiJpc3MiOiJh
bHR1cy0wMS5kaWdpdGFscGVyc29uYS5jb20iLAoiaWF0IjoxMzAwODE5MzgwLAoic3ViIjp7Im5h
bWUiOiJrbG96aW5AZGlnaXRhbHBlcnNvbmEuY29tIiwidHlwZSI6Nn0sCiJjcmQiOlsKeyJpZCI6
IntEMUExRjU2MS1FMTRBLTQ2OTktOTEzOC0yRUI1MjNFMTMyQ0N9IiwidGltZSI6IDEzMDA4MTkz
ODB9LAp7ImlkIjoie0FDMTg0QTEzLTYwQUItNDBlNS1BNTE0LUUxMEY3NzdFQzJGOX0iLCJ0aW1l
IjogNzg2NTIzMzY3OX0KXX0
.
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

Error Handling

If the DigitalPersona Server failed process authentication request, HTTP status of the request would be set to 404 (Not found) and WebFaultException of the WebFault class will be fired so the HTTP response will have Json representation of WebFault class:

For further details on the WebFaultException, see the following Microsoft article.

https://msdn.microsoft.com/en-us/library/system.servicemodel.web.webfaultexception(v=vs.110).aspx

```
[DataContract]
public class WebFault
  {
  public WebFault(int err, String desc)
  {
    error_code = err;
    description = desc;
```

```
  }
[DataMember]
public int error_code { get; set; }
[DataMember]
public String description { get; set; }
}
```

Value	Description
error_code	Error code returned by the DigitalPersona Server, for example -2147024891 (0x80070005) for access denied error.
description	Detailed description of the error, for example "Access is denied" for access denied error.

Below is an example of error returned by WAS:

```
{
  "error_code"=-2147024891,
  "description"="Access is denied."
}
```

DP Web Secret Management Service API 8

THIS CHAPTER DESCRIBES THE API FOR THE DIGITALPERSONA WEB SECRET MANAGEMENT SERVICE.

The Web Secret Management Service (WSMS) is part of the DigitalPersona Web Secret Service Provider (WSSP), a set of Web Services whose goal is to provide Secret Management functionality such as writing, reading and deleting protected data (Secrets). Generally speaking the main goal is to give access to Secret Management to applications running on computers outside the firewall. The actual secrets are stored protected in the DigitalPersona Server Database and to access them WSSP needs to send a request to the DigitalPersona Server.

The goal of the DigitalPersona Web Secret Management Service (WSMS) is to provide Secret Management. By Secret management we mean the following operations:

- Read Secret content
- Modify or Create Secret content
- Delete Secret.

It is implemented as a GUIless Web Service.

WSMS should be located inside the firewall and direct secret management requests to DigitalPersona Server over DCOM. DigitalPersona Server is the only one who can process secret management requests.

WSMS allows secret management only to authenticated users whose authenticated credentials satisfy the authentication policy provided by the Web Authentication Policy Service.

WSMS opens one or several Web endpoints listening on the HTTPS protocol. These endpoints may be designated as accessible from an intranet only or available from the internet as well.

HTTPS (Transport Layer Security) is used for our Web Service to simplify client side implementation. HTTPS protects user Secrets from man-in-the-middle attacks during transactions. This requires the Web Service hosting environment to have a Certificate which is accepted by the Client.

IDPWebSecretMgr interface

We define the IDPWebSecretMgr interface as a WCF interface.

```
namespace DPWcfSecretService
{
  [ServiceContract]
  public interface IDPWebSecretMgr
    {
    /*
    * Return policy for specific secret, specific user and specific action
    */
    [OperationContract()]
    [WebInvoke(Method = "GET",
      ResponseFormat = WebMessageFormat.Json,
      BodyStyle = WebMessageBodyStyle.Wrapped,
      UriTemplate = "GetAuthPolicy?user={userName}&type={nameType}&secret=
        {secretName}&action={action}")]
      List<Policy> GetAuthPolicy(String userName, UInt16 nameType,
        String secretName, ResourceActions action);
    /*
    * Return policy for specific secret, specific user and specific action
    */
    [OperationContract()]
    [WebInvoke(Method = "GET",
      ResponseFormat = WebMessageFormat.Json,
```

```
    BodyStyle = WebMessageBodyStyle.Wrapped,
    UriTemplate = "DoesSecretExist?user={userName}&type={nameType}&secret=
      {secretName}")]
     bool DoesSecretExist(String userName, UInt16 nameType, String secretName);
    /*
    * Read content of specific secret
    */
    [OperationContract()]
    [WebInvoke(Method = "POST",
       ResponseFormat = WebMessageFormat.Json,
       BodyStyle = WebMessageBodyStyle.Wrapped,
       UriTemplate = "ReadSecret")]
       String ReadSecret(Ticket ticket, String secretName);
    /*
    * Write content to specific secret
    */
    [OperationContract()]
    [WebInvoke(Method = "PUT",
      ResponseFormat = WebMessageFormat.Json,
      BodyStyle = WebMessageBodyStyle.Wrapped,
      UriTemplate = "WriteSecret")]
     void WriteSecret(Ticket ticket, String secretName, String secretData);
    /*
    * Delete content of specific secret
    */
    [OperationContract()]
    [WebInvoke(Method = "DELETE",
      ResponseFormat = WebMessageFormat.Json,
      BodyStyle = WebMessageBodyStyle.Wrapped,
      UriTemplate = "DeleteSecret")]
    void DeleteSecret(Ticket ticket, String secretName);
  }

[DataContract]
public enum ResourceActions
  {
    [EnumMember]
    Read = 0,
    [EnumMember]
    Write = 1,
    [EnumMember]
    Delete = 2,
   }
[DataContract]
public class PolicyElement
  {
    public PolicyElement(String id)
       {
         cred_id = id;
       }
    [DataMember]
    public String cred_id { get; set; }
  }
[DataContract]
```

```
  public class Policy
    {
       public Policy()
       {
         policy = new List<PolicyElement>();
       }
        [DataMember]
        public List<PolicyElement> policy { get; set; }
    }
    [DataContract]
    public class Ticket
    {
        [DataMember]
        public String jwt { get; set; } // JSON Web Token
    }
}
```

Methods

GetAuthPolicy method

The GetAuthPolicy() method returns a list of policies (credentials) which are required in order to access (read, write or delete) specific Secret of a named user.

Syntax

```
List<Policy> GetAuthPolicy(String userName, UInt16 nameType, String secretName,
ResourceActions action);
```

Parameter	Description
userName	Name of the user whose Secret needs to be managed.
userNameType	Format of the user name provided in the userName parameter. See **"User class" on page 65** for a detailed description of user name formats.
secretName	Name of the Secret need to be accessed, "SystemLogonInfo" for example.
action	Action requested for this Secret. Right now we will support 3 types of action: 1) Read, 2) Write and 3) Delete.

Return values

List of policies (credentials) which would be required to access such Secret;

Notes

GetAuthPolicy () should be implemented as HTTP GET using JSON as response format.

The following example of use of GetAuthPolicy():

```
http://www.mycompany.com/ DPWebSecretService.svc/ GetAuthPolicy?
       user=someone@mycompany.com&type=6&secret=SystemLogonInfo&action=Read
```

This URL would request policies which would be required to read secret with name SystemLogonInfo which belongs to user with name someone@mycompany.com.

The example response would be the following:

```
{"GetAuthPolicyResult":
  [
```

```
    {"policy":[
      {"cred_id":"AC184A13-60AB-40e5-A514-E10F777EC2F9"},
      {"cred_id":"8A6FCEC3-3C8A-40c2-8AC0-A039EC01BA05"}
    ]},
    {"policy":[
      {"cred_id":"AC184A13-60AB-40e5-A514-E10F777EC2F9"},
      {"cred_id":"E750A180-577B-47f7-ACD9-F89A7E27FA49 "}
    ]}
  ]
}
```

This response means user someone@mycompany.com needs to provide “Fingerprint AND PIN” or “Fingerprint AND Bluetooth” to be allowed to read SystemLogonInfo Secret. Note that we must use “braceless” GUID representation in our APIs in URLs and JSON representation.

DoesSecretExist method

DoesSecretExist method verifies whether or not a specific secret of a specified user exists.

Syntax

```
bool DoesSecretExist(String userName, UInt16 nameType, String secretName);
```

Parameter	Description
userName	Name of the user whose Secret needs to be verified.
userNameType	Format of the user name provided in the userName parameter. See **“User class” on page 65** for a detailed description of user name formats.
secretName	Name of the Secret need to be verified, “SystemLogonInfo” for example.

Return values

true – If secret with the specified name exists in the DigitalPersona database.

false – If secret with the specified name exists in the DigitalPersona database

Notes

DoesSecretExist () should be implemented as HTTP GET using JSON as the response format.

The following is an example of using DoesSecretExist ():

```
http://www.mycompany.com/ DPWebSecretService.svc/ DoesSecretExist?user=
someone@mycompany.com&type=6&secret=SystemLogonInfo
```

The URL above requests information as to whether or not a secret with the name “SystemLogonInfo” exists for the user someone@mycompany.com.

The following is an example of the HTTP response to the request.

```
{"DoesSecretExistResult":true}
```

This response means that a secret with the name “SystemLogonInfo” exists for the user someone@mycompany.com.

ReadSecret method

The ReadSecret method will return the content of a specific secret for the user if the authentication provided in the Ticket satisfies the policy.

Syntax

```
String ReadSecret(Ticket ticket, String secretName);
```

Parameter	Description
ticket	JSON Web Token returned by the Authentication Service.
secretName	Name of the Secret need to be read.

Returns

String which represents Base64url encoded secret content.

Notes

ReadSecret () should be implemented as HTTP POST using JSON as the response format.

The following URL is an example of using ReadSecret ().

```
http://www.mycompany.com/DPWebSecretService.svc/ReadSecret
```

Below is an example of an HTTP BODY of a ReadSecret request.

```
{"secretName":"SystemLogonInfo","ticket":{"jwt":"eyJ0eXAiOiJKV1QiLA0KICJhbGc
iOiJIUzI1NiJ9.eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leG
FtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ.dBjftJeZ4CVP-B92K27uhbUJU1p1rwW1gFWFOEjXk"}}
```

Below is example of the response.

```
{"ReadSecretResult":"eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9"}
```

WriteSecret method

The WriteSecret method will write the content of a specific secret of the user if authentication provided in a Ticket satisfies the policy. If a secret with the specified name does not exist, it will be created. There is no separate CreateSecret method and separate authorization for secret creation.

```
void WriteSecret(Ticket ticket, String secretName, String secretData);
```

Parameter	Description
ticket	JSON Web Token returned by the Authentication Service.
secretName	Name of the Secret to be written, “SystemLogonInfo” for example.
secretData	Base64url encoded data to be written to the secret.

Returns

None

Notes

WriteSecret () should be implemented as HTTP PUT using JSON as the response format.

The following URL is an example of using WriteSecret ().

```
http://www.mycompany.com/ DPWebSecretService.svc/WriteSecret
```

Below is an example of HTTP BODY of the WriteSecret request.

```
{
  "secretName":"SystemLogonInfo",
  "secretData":"eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9",
  "ticket":
    {"jwt":"eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJqb2UiLA0KICJl
    eHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ.dB
```

```
    jftJeZ4CVP-B92K27uhbUJU1p1rwW1gFWFOEjXk"}
}
```

DeleteSecret method

The DeleteSecret method will clear the secret content and delete the secret object (if any). The caller must provide a Ticket which satisfies the policy.

Syntax

```
void DeleteSecret(Ticket ticket, String secretName);
```

Parameter	Description
ticket	JSON Web Token returned by the Authentication Service.
secretName	Name of the Secret to be deleted, “SystemLogonInfo” for example.

Returns

None.

Notes

DeleteSecret () should be implemented as HTTP DELETE using JSON as the response format.

The following URL is an example of using DeleteSecret ().

```
http://www.mycompany.com/DPWebSecretService.svc/DeleteSecret
```

Below is an example of HTTP BODY of a DeleteSecret request:

```
{
  "secretName":"SystemLogonInfo",
  "ticket":{"jwt":"eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJqb2Ui
  LA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cn
  VlfQ.dBjftJeZ4CVP-B92K27uhbUJU1p1rwW1gFWFOEjXk"}
}
```

Data Contracts

ResourceActions enumerator

ResourceActions enumerates possible actions the caller may perform with a particular resource (Secret).

```
public enum ResourceActions
{
  [EnumMember]
  Read = 0,
 [EnumMember]
  Write = 1,
  [EnumMember]
  Delete = 2,
}
```

We support only these actions in the current version of the API.

- Read
- Write
- Delete.

PolicyElement class

The PolicyElement class describes one policy element or Credential.

```
public class PolicyElement
  {
  public PolicyElement(String id)
  {
    cred_id = id;
  }
  [DataMember]
  public String cred_id { get; set; }
  }
```

Parameter	Description
cred_id	Unique Id of credential.

The following credentials are supported in this version.

```
{D1A1F561-E14A-4699-9138-2EB523E132CC} - User password;
{AC184A13-60AB-40e5-A514-E10F777EC2F9} - Fingerprints;
{8A6FCEC3-3C8A-40c2-8AC0-A039EC01BA05} - PIN;
{D66CC98D-4153-4987-8EBE-FB46E848EA98} - Smart Cards;
{F674862D-AC70-48ca-B73E-64A22F3BAC44} - Contactless/Proximity cards;
{B49E99C6-6C94-42DE-ACD7-FD6B415DF503} - Live Questions;
{E750A180-577B-47f7-ACD9-F89A7E27FA49} - Bluetooth;
** MISSING **
```

Below is an example of JSON representation of a credential. This policy element describes a Fingerprint credential. You must use the “braceless” GUID representation in our API in URLs and JSON representation.

```
{"cred_id":"AC184A13-60AB-40e5-A514-E10F777EC2F9"}
```

Policy class

This class describes one authentication policy. To access a resource multiple authentication policies could be applied. In this case, the relationship between the authentication policies would be OR.

```
public class Policy
  {
  public Policy()
  {
     policy = new List<PolicyElement>();
  }
  [DataMember]
  public List<PolicyElement> policy{ get; set; }
}
```

Below is an example of the JSON representation of a policy. This policy describes an authentication policy which requires “Fingerprint AND PIN” credentials to be provided.

```
{"policy":[
{"cred_id":"AC184A13-60AB-40e5-A514-E10F777EC2F9"},
{"cred_id":"8A6FCEC3-3C8A-40c2-8AC0-A039EC01BA05"}
]},
```

Ticket class

The result of successful authentication is a Ticket. The format of the Ticket is JSON Web Token (JWT). Additional details about JWT are provided in the topic **“JSON Web Token (JWT)” on page 71**.

```
public class Ticket
  {
  [DataMember]
   public String jwt { get; set; } // JSON Web Token
  }
```

DP Web Authentication Policy API 9

THIS CHAPTER DESCRIBES THE API FOR THE DIGITALPERSONA WEB AUTHENTICATION POLICY SERVICE.

The goal of the Web Authentication Policy Service (WAPS) is to specify the authentication policy required for a specific user to manage a Resource. It is implemented as a GUI-less Web Service.

WAPS can be located inside or outside the firewall, depending on the logic of the specific WAPS implementation. Location inside the firewall is more common since WAPS may need connection to AD, DigitalPersona Server or other customer DBs.

WAPS can use any protocol the customer chooses for listening. However, it is important that the Service Provider (SP) using WAPS must always have access to it.

WAPS can be hosted in any supported environment and developed using any Web development framework as long as SP can communicate with it.

IDPWebPolicy interface

The DigitalPersona Web Authentication Policy Service (WAPS) is implemented as a WCF service. The IDPWebPolicy interface has only one method, GetPolicyList(), as described in the next topic.

```
namespace DPWcfPolicyService
{
  [ServiceContract]
  public interface IDPWebPolicy
  {
  /*
  * Return policy for specific resource, specific user and specific action
  */
  [WebInvoke(Method = "GET",
    ResponseFormat = WebMessageFormat.Json,
    BodyStyle = WebMessageBodyStyle.Wrapped,
    UriTemplate = "GetPolicyList?user={userName}&type={nameType}&uri=
      {resourceUri}&action={action}")]
  List<Policy> GetPolicyList(String userName, UInt16 nameType,
    String resourceUri, ResourceActions action);
  /*
  * Return policy for specific resource, specific user, specific action using
    contextual/behavior information
  */
  [OperationContract()]
  [WebInvoke(Method = "POST",
    ResponseFormat = WebMessageFormat.Json,
    BodyStyle = WebMessageBodyStyle.Wrapped,
    UriTemplate = "GetPolicyListEx")]
  List<Policy> GetPolicyListEx(String userName, UInt16 nameType, String
    resourceUri, ResourceActions action, ContextualInfo info);
  }
  [DataContract]
  public enum ResourceActions
  {
    [EnumMember]
    Read = 0,
    [EnumMember]
    Write = 1,
    [EnumMember]
```

```
    Delete = 2,
    }
  [DataContract]
  public class PolicyElement
  {
    public PolicyElement(String id)
    {
      cred_id = id;
    }
  [DataMember]
  public String cred_id { get; set; }
    }
  [DataContract]
  public class Policy
  {
    public Policy()
    {
      policy = new List<PolicyElement>();
    }
    [DataMember]
    public List<PolicyElement> policy{ get; set; }
    }
  [DataContract]
  public class ContextualInfo
  {
    [DataMember]
    public Boolean behavior { get; set; }
    [DataMember]
    public Boolean ip { get; set; }
    [DataMember]
    public Boolean device { get; set; }
   [DataMember]
    public Boolean altusInstalled { get; set; }
    [DataMember]
    public String computer { get; set; }
    [DataMember]
    public String domain { get; set; }
    [DataMember]
    public String user { get; set; }
    [DataMember]
    public Boolean insideFirewall { get; set; }
    [DataMember]
    public Boolean remoteSession { get; set; }
  }
}
```

GetPolicyList method

GetPolicyList() method returns a list of policies that consist of authentication options; at least one of them must be met in order to access (read, write or delete) the specific Secret of the user.

Syntax

```
List<Policy> GetPolicyList(String userName, UInt16 nameType, String resourceUri,
ResourceActions action);
```

Parameter	Description
userName	Name of the user whose Secret needs to be managed.
userNameType	Format of the user name provided in the userName parameter. See **"User class" on page 65** for a detailed description of user name formats.
resourceUri	Uri which represent resource to be accessed. For example, a Secret name like "SystemLogonInfo."
action	Action requested for this resource. Currently the following actions are supported: 1) Read, 2) Write and 3) Delete.

Returns

List of policies of which one must be met in order to access the resource.

Notes

GetPolicyList () should be implemented as HTTP GET using JSON as the response format.

The following is an example of using GetPolicyList().

```
http://www.mycompany.com/DPWebPolicy/DPWebPolicyService.svc/GetPolicyList?
user=someone@mycompany.com&type=6&uri=SystemLogonInfo&action=Read
```

This URL requests a list of policies required to read a secret with the name SystemLogonInfo, which belongs to a user with the name someone@mycompany.com.

The example response would be the following.

```
{"GetPolicyListResult":
  [
    {"policy":[
      {"cred_id":"AC184A13-60AB-40e5-A514-E10F777EC2F9"},
      {"cred_id":"8A6FCEC3-3C8A-40c2-8AC0-A039EC01BA05"}
    ]},
    {"policy":[
      {"cred_id":"AC184A13-60AB-40e5-A514-E10F777EC2F9"},
      {"cred_id":"E750A180-577B-47f7-ACD9-F89A7E27FA49"}
    ]}
  ]
}
```

This response means that the user someone@mycompany.com needs to provide "Fingerprint AND PIN" or "Fingerprint AND Bluetooth" to be allowed to read the SystemLogonInfo Secret.

GetPolicyListEx method

GetPolicyListEx() method is extension of GetPolicyList() method which adds Contextual/Behavior information and returns list of policies which are authentication options of which at least one must met in order to access (read, write or delete) the specific Secret of the specific user.

Syntax

```
List<Policy> GetPolicyListEx(User user, String resourceUri, ResourceActions
action, ContextualInfo info);
```

Parameter	Description
userName	Name of the user whose Secret needs to be managed.
resourceUri	Uri which represent resource to be accessed. For example, a Secret name like “SystemLogonInfo.”
action	Action requested for this resource. Currently the following actions are supported: 1) Read, 2) Write and 3) Delete.
info	Contextual/Behavior information provided (see detailed description below).

Returns

List of policies of which one must be met to access such resource;

GetPolicyListEx () should be implemented as HTTP POST using JSON as response format.

The following example of use of GetPolicyListEx():

```
http://www.digitalpersona.com/DPWebPolicy/DPWebPolicyService.svc/GetPolicyListEx
```

Below is an example of HTTP request body:

```
{
  "user":{"name":"someone@mycompany","type":6},
  "resourceUri":"SystemLogonInfo",
  "action":1,
  "info":
  {
    "behavior":true,
    "ip":true,
    "device":true,
    "altusInstalled":true,
    "computer":"computername.mycompany.net",
    "domain":"mycompany.net",
    "user":"someone@mycompany.com",
    "insideFirewall":true,
    "remoteSession":false
  }
}
```

This URL would request policies which would be required to read secret (in this example Secret is requested resource) with name SystemLogonInfo which belongs to user with name someone@mycompany.com.

The example response would be the following:

```
{"GetPolicyListExResult":
  [
    {"policy":[
      {"cred_id":"AC184A13-60AB-40e5-A514-E10F777EC2F9"},
      {"cred_id":"8A6FCEC3-3C8A-40c2-8AC0-A039EC01BA05"}
    ]},
    {"policy":[
      {"cred_id":"AC184A13-60AB-40e5-A514-E10F777EC2F9"},
      {"cred_id":"E750A180-577B-47f7-ACD9-F89A7E27FA49"}
    ]}
  ]
```

```
}
```

This response means that the user someone@mycompany.com needs to provide "Fingerprint AND PIN" or "Fingerprint AND Bluetooth" to be allowed to read SystemLogonInfo Secret.

Data Contracts

ResourceActions enumerator

ResourceActions enumerates the possible actions the caller may apply to a particular resource.

```
public enum ResourceActions
{
  [EnumMember]
  Read = 0,
  [EnumMember]
  Write = 1,
  [EnumMember]
  Delete = 2,
}
```

We support only three actions in this version of WAPS.

- Read
- Write
- Delete

PolicyElement class

The PolicyElement class describes one policy element.

```
public class PolicyElement
{
  public PolicyElement(String id)
  {
    cred_id = id;
  }
  [DataMember]
  public String cred_id { get; set; }
}
```

Parameter	Description
cred_id	Unique Id of credential.

The following credentials are supported in this version.

```
{D1A1F561-E14A-4699-9138-2EB523E132CC} - User password;
{AC184A13-60AB-40e5-A514-E10F777EC2F9} - Fingerprints;
{8A6FCEC3-3C8A-40c2-8AC0-A039EC01BA05} - PIN;
{D66CC98D-4153-4987-8EBE-FB46E848EA98} - Smart Cards;
{F674862D-AC70-48ca-B73E-64A22F3BAC44} - Contactless/Proximity cards;
{B49E99C6-6C94-42DE-ACD7-FD6B415DF503} - Live Questions;
{E750A180-577B-47f7-ACD9-F89A7E27FA49} - Bluetooth;
*** MISSING ***
```

Below is an example of JSON representation of a fingerprint credential. The relationship between policy elements within one policy is AND. Note that we must use “braceless” GUID representation in our API in URLs and JSON representation.

```
{"cred_id":"AC184A13-60AB-40e5-A514-E10F777EC2F9"}
```

Policy class

This class describes one authentication policy. To access a particular resource, multiple authentication policies could be applied. In this case the relationship between the authentication policies would be OR.

```
public class Policy
{
  public Policy()
  {
    policy = new List<PolicyElement>();
  }
  [DataMember]
  public List<PolicyElement> policy{ get; set; }
}
```

Below is an example of JSON representation of a policy. This policy describes an authentication policy which requires “Fingerprint AND PIN” credentials to be provided.

```
{"policy":[
  {"cred_id":"AC184A13-60AB-40e5-A514-E10F777EC2F9"},
  {"cred_id":"8A6FCEC3-3C8A-40c2-8AC0-A039EC01BA05"}
]},
```

ContextualInfo class

This class describes Contextual/Behavior information.

```
[DataContract]
public class ContextualInfo
{
  [DataMember]
  public Boolean behavior { get; set; }
  [DataMember]
  public Boolean ip { get; set; }
  [DataMember]
  public Boolean device { get; set; }
  [DataMember]
  public Boolean altusInstalled { get; set; }
  [DataMember]
  public String computer { get; set; }
  [DataMember]
  public String domain { get; set; }
  [DataMember]
  public String user { get; set; }
  [DataMember]
  public Boolean insideFirewall { get; set; }
  [DataMember]
  public Boolean remoteSession { get; set; }
    }
```

Parameter	Description
behavior	Boolean value which is true if user behavior is matched, false otherwise.
ip	Boolean value which is true if user IP is matched with the trained data, false otherwise.
device	Boolean value which is true if user device is matched with the trained data, false otherwise.
altusInstalled	Boolean value which is true if DigitalPersona client software is installed on client machine, false otherwise.
computer	DNS name of the client machine.
domain	AD domain client computer belongs to.
user	UPN name of the user currently logged on client machine.
insideFirewall	Boolean value which is true if client machine located inside corporate firewall, false otherwise.
remoteSession	Boolean value which is true user access client machine remotely, false otherwise.

Below is an example of JSON representation of ContextualInfo:

```
{
  "behavior":true,
  "ip":true,
  "device":true,
  "altusInstalled":true,
  "computer":"kloizinD7.crossmatch.net",
  "domain":"crossmatch.net",
  "user":"Kirill.Lo@crossmatch.com",
  "insideFirewall":true,
  "remoteSession":false
}
```

Policy Configuration

dpWebDefaultPolicies configuration element

The dpWebDefaultPolicies element in the Web.config file contains general configuration information about the Authentication Policies.

Put the dpWebDefailtPolicies element within the configuration element in a Web.config. The dpWebDefailtPolicies element has no attributes.

The dpWebDefaultPolicies element may include the following elements:

dpWebPolicies configuration element

The dpWebPolicies element in the Web.config file contains configuration information about standard Authentication Policies (authentication policies when step-up authentication is not triggered).

Put the dpWebPolicies element within the dpWebDefaultPolicies element in a Web.config. The dpWebPolicies element has no attributes.

The dpWebPolicies element may include the following elements:

dpWebPolicy configuration element

The dpWebPoliciy element in the Web.config file contains configuration information about one particular Authentication Policy.

The following table lists and describes the attributes of the dpWebPoliciy element.

Attribute	Data type	Description
name	String	Readable name of authentication policy. This attribute is required.
tokens	String	List of authentication tokens separated by semicolon (;) in token GUID format. This attribute is required.

Below are few examples of dpWebPolicies element:

```
<dpWebPolicy name="Password" tokens="D1A1F561-E14A-4699-9138-2EB523E132CC" />
```

The element above represents Password only policy.

```
<dpWebPolicy name="Fingerprint" tokens="AC184A13-60AB-40E5-A514-E10F777EC2F9" />
```

The element above represents Fingerprint only policy.

```
<dpWebPolicy name="Fingerprint AND Password" tokens="AC184A13-60AB-40E5-
A514-E10F777EC2F9; D1A1F561-E14A-4699-9138-2EB523E132CC" />
```

The element above represents Fingerprint AND Password policy.

Example of dpWebPolicies Configuration Element

Below is an example of dpWebPolicies Configuration Element:

```
<dpWebPolicies>
  <dpWebPolicy name="Password" tokens="D1A1F561-E14A-4699-9138-2EB523E132CC" />
  <dpWebPolicy name="Fingerprint" tokens="AC184A13-60AB-40E5-A514-E10F777EC2F9" />
  <dpWebPolicy name="Contactless" tokens="F674862D-AC70-48CA-B73E-64A22F3BAC44" />
  <dpWebPolicy name="Smart Card" tokens="D66CC98D-4153-4987-8EBE-FB46E848EA98" />
  <dpWebPolicy name="One-time Password" tokens="324C38BD-0B51-4E4D-BD75-200DA0C8177F" />
</dpWebPolicies>
```

Those policies should allow to use Password OR Fingerprint OR Contactless cards OR Smart Cards OR OTP if step up authentication is not triggered.

dpWebStepUpPolicies configuration element

The dpWebStepUpPolicies element in the Web.config file contains configuration information about step-up Authentication Policies (authentication policies when step-up authentication is triggered).

Put the dpWebStepUpPolicies element within the dpWebDefailtPolicies element in a Web.config. The dpWebStepUpPolicies element has no attributes.

The dpWebStepUpPolicies element may include one or multiple dpWebPoliciy elements.

Below is an example of dpWebStepUpPolicies Configuration Element:

```
<dpWebStepUpPolicies>
  <dpWebPolicy name="Fingerprint AND Password" tokens="AC184A13-60AB-40E5-
    A514-E10F777EC2F9; D1A1F561-E14A-4699-9138-2EB523E132CC" />
  <dpWebPolicy name="Fingerprint AND PIN" tokens="AC184A13-60AB-40E5-A514-
    E10F777EC2F9; 8A6FCEC3-3C8A-40c2-8AC0-A039EC01BA05" />
</dpWebStepUpPolicies>
```

Those policies would force user to use Fingerprint AND Password OR Fingerprint AND PIN if step-up authentication is triggered.

dpWebStepUpTriggers configuration element

The dpWebStepUpTriggers element in the Web.config file contains configuration information about triggers which can trigger Step-up Authentication.

Put the dpWebStepUpTriggers element within the dpWebDefailtPolicies element in a Web.config. The dpWebStepUpTriggers element has no attributes.

The dpWebStepUpTriggers element may include the following elements:

dpWebStepUpTrigger Configuration Element

The dpWebStepUpTrigger element in the Web.config file contains configuration information about one particular Step Up Authentication trigger.

The following table lists and describes the attributes of the dpWebStepUpTrigger element.

Attribute	Data type	Description
name	String	Trigger name. This attribute is required.

Below are few examples of dpWebStepUpTrigger element:

```
<dpWebStepUpTrigger name="behavior" />
```

The element above represents Behavior trigger (step-up authentication if user behavior does not matched).

```
<dpWebStepUpTrigger name="insideFirewall" />
```

The element above represents inside firewall trigger (trigger step up authentication if user machine located outside corporate network).

List of Triggers supported

In this release we will support triggers below:

Parameter	Description
behavior	Trigger step up authentication if user behavior does not match.
ip	Trigger step up authentication if user IP does not match with the trained data stored in the BehavioSec cloud.
device	Trigger step up authentication if user device does not match with the trained data stored in the BehavioSec cloud.
altusInstalled	Trigger step up authentication if DPCA is not installed on client machine.
computer	Trigger step up authentication if client machine is not trusted.
domain	Trigger step up authentication if client machine does not belong to trusted AD domain.
user	Trigger step up authentication user who is trying to access web server is not currently logon user on local machine.
insideFirewall	Trigger step up authentication if local machine is not located in corporate network (not inside corporate firewall).
remoteSession	Trigger step up authentication if user connected to local machine remotely.

Example of dpWebStepUpTriggers Element

Below is an example of dpWebStepUpTriggers Configuration Element:

```
<dpWebStepUpTriggers>
  <dpWebStepUpTrigger name="behavior" />
  <dpWebStepUpTrigger name="insideFirewall" />
</dpWebStepUpTriggers>
```

Those settings will trigger step-up authentication if user behavior does not match or local machine located outside firewall.

Example of dpWebDefaultPolicies Element

Below is an example of dpWebDefaultPolicies configuration element:

```
<dpWebDefaultPolicies>

  <dpWebPolicies>
    <dpWebPolicy name="Password" tokens="D1A1F561-E14A-4699-9138-2EB523E132CC" />
    <dpWebPolicy name="Fingerprint" tokens="AC184A13-60AB-40E5-A514-E10F777EC2F9" />
  </dpWebPolicies>

  <dpWebStepUpPolicies>
    <dpWebPolicy name="Fingerprint AND Password" tokens="AC184A13-60AB-40E5-
      A514-E10F777EC2F9; D1A1F561-E14A-4699-9138-2EB523E132CC" />
  </dpWebStepUpPolicies>

  <dpWebStepUpTriggers>
    <dpWebStepUpTrigger name="behavior" />
    <dpWebStepUpTrigger name="insideFirewall" />
  </dpWebStepUpTriggers>

</dpWebDefaultPolicies>
```

The policy above will allow user to use Fingerprint OR Password if step-up authentication is not triggered; force user to use Fingerprint AND Password if step-up authentication is triggered and user behavior and computer location outside firewall will trigger step-up authentication.

10 WAS Credentials Data Forma

THIS CHAPTER DESCRIBES DATA MEMBERS FOR VARIOUS SUPPORTED CREDENTIALS.

In the **IDPWebAuth** interface (see **"IDPWebAuth interface" on page 59**) we define a number of methods for user authentication and identification which have as an input parameter an object of the **Credential** class. This chapter defines and describes the format of the **data** member for each **Credential** supported by DigitalPersona.

The **Credential** class is defined as follows:

```
[DataContract]
public class Credential
{
  [DataMember]
  public String id { get; set; } // unique id (Guid) of credential
  [DataMember]
  public String data { get; set; } //  credential data
}
```

It is obvious that the **data** member of the **Credential** class should be different for different types of credentials.For example, the **data** member of the Fingerprint **Credential** is different from the same member of the Password **Credential**.

Fingerprint credential

The following ID is defined for Fingerprint credentials.

```
{AC184A13-60AB-40e5-A514-E10F777EC2F9}
```

The data for fingerprint credential is defined in WCF terminology as follows.

```
[DataContract][Flags]
public enum BioFactor
{
```

```
  [EnumMember]
  MULTIPLE =              0x0001,
  [EnumMember]
  FACIAL_FEATURES =       0x0002,
  [EnumMember]
  VOICE =                 0x0004,
  [EnumMember]
  FINGERPRINT =           0x0008,
  [EnumMember]
  IRIS =                  0x0010,
  [EnumMember]
  RETINA =                0x0020,
  [EnumMember]
  HAND_GEOMETRY =         0x0040,
  [EnumMember]
  SIGNATURE_DYNAMICS =    0x0080,
  [EnumMember]
  KEYSTROKE_DYNAMICS =     0x0100,
  [EnumMember]
  LIP_MOVEMENT =          0x0200,
  [EnumMember]
  THERMAL_FACE_IMAGE =    0x0400,
  [EnumMember]
  THERMAL_HAND_IMAGE =    0x0800,
  [EnumMember]
  GAIT =                  0x1000,
}

[DataContract]
public class BioSampleFormat
{
  public BioSampleFormat(UInt16 owner, UInt16 id)
  {
    FormatOwner = owner;
    FormatID = id;
  }
  [DataMember]
  public UInt16 FormatOwner { get; set; }
    // Biometric owner ID registered with IBIA (http://www.ibia.org/base/
        cbeff/_biometric_org.phpx).
    // 51 for DigitalPersona, 49 for Neurotechnologija
  [DataMember]
  public UInt16 FormatID;   // Vendor specific format ID
}

[DataContract][Flags]
public enum BioSampleType
{
  [EnumMember]
  RAW             = 0x01,      // Fingerprint image
  [EnumMember]
  INTERMEDIATE    = 0x02,      // Fingerprint feature set
  [EnumMember]
  PROCESSED    = 0x04,         // Fingerprint template
```

```
  [EnumMember]
  RAW_WSQ_COMPRESSED    = 0x08, // WSQ compressed image
  [EnumMember]
  ENCRYPTED    = 0x10,
  [EnumMember]
   SIGNED    = 0x20,
}

[DataContract]
public enum BioSamplePurpose
{
  [EnumMember]
  ANY = 0,         // Any purpose
  [EnumMember]
  VERIFY = 1,      // Purpose is verification
  [EnumMember]
  IDENTIFY = 2,    // Purpose is identification
  [EnumMember]
  ENROLL =   3,    // Purpose is enrollment
  [EnumMember]
  ENROLL_FOR_VERIFICATION_ONLY = 4, //  Purpose is enrollment for verification
  [EnumMember]
  ENROLL_FOR_IDENTIFICATION_ONLY =  5,  // Purpose is enrollment for identification
  [EnumMember]
  AUDIT = 6,       // Purpose is audit
}

[DataContract]
public enum BioSampleEncryption
{
  [EnumMember]
  NONE = 0,        // Data is not encrypted
  [EnumMember]
  XTEA = 1,        // XTEA encryption with well-known key
}

[DataContract]
public class BioSampleHeader
{
  [DataMember]
  public BioFactor Factor { get; set; }
    // Biometric factor. Must be set to 8 for fingerprint
  [DataMember]
  public BioSampleFormat Format { get; set; }
    // Format owner (vendor) information
  [DataMember]
  public BioSampleType Type { get; set; }
    // type of biometric sample
  [DataMember]
  public BioSamplePurpose Purpose { get; set; }
    // Purpose of biometric sample
  [DataMember]
  public SByte Quality { get; set; }
    // Quality of biometric sample.
```

```
    // If we don't support quality it should be set to -1.
  [DataMember]
  public BioSampleEncryption Encryption { get; set; }
    // Encryption of biometric sample.
}

[DataContract]
public class BioSample
{
  public BioSample()
  {
    Header = new BioSampleHeader();
  }
  [DataMember]
  public UInt16 Version { get; set; }
    // header version. Must be set to 1 in this version
  [DataMember]
  public BioSampleHeader Header { get; set; }
    // Biometric sample header
  [DataMember]
  public String Data { get; set; }
    // Base64url encoded biometric sample data
}
```

BioSample class

The BioSample class represents a general biometric sample.

```
[DataContract]
public class BioSample
{
  public BioSample()
  {
    Header = new BioSampleHeader();
  }
  [DataMember]
  public UInt16 Version { get; set; }
    // header version. Must be set to 1 in this version
  [DataMember]
  public BioSampleHeader Header { get; set; }
    // Biometric sample header
  [DataMember]
  public String Data { get; set; }
    // Base64url encoded biometric sample data
}
```

Data Member	Description
Version	Version of Biometric Sample format. Must be 1 for this API version.
Header	Base64url encoded biometric sample data. See Base64url encoded biometric sample data. See **“BioSampleHeader class” on page 100**..
Data	Base64url encoded biometric sample data. See **“Biometric Sample Data” on page 103**.

BioSampleHeader class

The BioSampleHeader provides detailed information about a Biometric sample.

```
[DataContract]
public class BioSampleHeader
{
  [DataMember]
  public BioFactor Factor { get; set; }
    // Biometric factor. Must be set to 8 for fingerprint
  [DataMember]
  public BioSampleFormat Format { get; set; }
    // Format owner (vendor) information
  [DataMember]
  public BioSampleType Type { get; set; }  // type of biometric sample
  [DataMember]
  public BioSamplePurpose Purpose { get; set; }
    //Purpose of biometric sample
  [DataMember]
  public SByte Quality { get; set; }
    // Quality of biometric sample.
    // If we don't support quality, it should be set  to -1.
  [DataMember]
  public BioSampleEncryption Encryption { get; set; }
    // Encryption of biometric sample.
}
```

Data Member	Description
Factor	Biometric factor presented in this Biometric sample. This version only supports fingerprint, so this member should be set to 8.
Format	Format of Biometric sample. See **"BioSampleFormat class" on page 100**.
Type	Type of Biometric sample. See **"BioSampleType enumeration" on page 101**.
Purpose	Purpose of Biometric sample. See **"BioSamplePurpose enumeration" on page 102**.
Quality	Quality of Biometric sample. Unsupported in this version. Set to -1.
Encryption	Define what kind of encryption algorithm was used to protect the Biometric sample. See **"BioSampleEncryption enumeration" on page 103** for details).

BioSampleFormat class

BioSampleFormat describes the vendor-specific format of the Biometric sample.

```
[DataContract]
public class BioSampleFormat
{
  public BioSampleFormat(UInt16 owner, UInt16 id)
  {
    FormatOwner = owner;
    FormatID = id;
  }
  [DataMember]
  public UInt16 FormatOwner { get; set; }
    // Biometric owner ID registered with IBIA (http://www.ibia.org/base/
    // cbeff/_biometric_org.phpx).
    // 51 for DigitalPersona, 49 for Neurotechnologija
  [DataMember]
  public UInt16 FormatID;  // Vendor specific format ID
}
```

Data Member	Description
FormatOwner	Represents the vendor which produced this Biometric sample. Values are assigned and registered by the International Biometric Industry Association (IBIA) (*http://www.ibia.org/base/cbeff/_biometric_org.phpx*). The supported values are: 51 – for DigitalPersona 49 – for Neurotechnology
FormatID	This value is assigned by the Format Owner and may optionally be registered by the IBIA. Unsupported in this version, so it should be set to 0.

BioSampleType enumeration

The `BioSampleType` enumeration defines the type of Biometric sample.

```
[DataContract][Flags]
public enum BioSampleType
{
  [EnumMember]
  RAW              = 0x01,   // Fingerprint image
  [EnumMember]
  INTERMEDIATE     = 0x02,   // Fingerprint feature set
  [EnumMember]
  PROCESSED= 0x04,           // Fingerprint template
  [EnumMember]
  ENCRYPTED= 0x10,
  [EnumMember]
  SIGNED= 0x20,
}
```

Data Member	Description
RAW	The sample is a raw (unprocessed) biometric sample. For fingerprints, this means the sample is a Fingerprint Image.
INTERMEDIATE	The sample is a partially processed biometric sample.For fingerprints, this means the sample is a Fingerprint Feature Set.
PROCESSED	The sample is a fully processed biometric sample. For fingerprints, this means the sample is a Fingerprint Template.
ENCRYPTED	Not supported in this version.
SIGNED	Not supported in this version.

BioSamplePurpose enumeration

The BioSamplePurpose details the purpose of this biometric sample.

```
[DataContract]
public enum BioSamplePurpose
{
  [EnumMember]
  ANY = 0,                              // Any purpose
  [EnumMember]
  VERIFY = 1,                           // Verification
  [EnumMember]
  IDENTIFY = 2,                         // Identification
  [EnumMember]
  ENROLL = 3,                           // Enrollment
  [EnumMember]
  ENROLL_FOR_VERIFICATION_ONLY = 4,     // Enrollment for verification
  [EnumMember]
  ENROLL_FOR_IDENTIFICATION_ONLY = 5, // Enrollment for identification
  [EnumMember]
  AUDIT = 6,                            // Audit
}
```

Data Member	Description
ANY	The purpose of this biometric sample is undefined and it can be used for any purpose.
VERIFY	This biometric sample was created for verification (authentication) purposes only.
IDENTIFY	This biometric sample was created for identification purposes only.
ENROLL	This biometric sample was created for enrollment purposes.
ENROLL_FOR_VERIFICATION_ONLY	This biometric sample was created for audit purposes.

This document is describing data for authentication SDK which means we are concerning about verification and identification so the following purposes should be supported: [Is this still relevant?]

ANY - biometric sample can be used for both verification and identification.

VERIFY - biometric sample can be used for verification only.

IDENTIFY - biometric sample can be used for identification only.

Notes

If a fingerprint image is sent as the biometric sample, the ANY purpose should be set.

If feature extraction is done by the DigitalPersona engine, with any flag except FT_PRE_REG_FTR and FT_REG_FTR, it can be used for both verification and identification so ANY purpose could be provided.

BioSampleEncryption enumeration

`BioSampleEncryption` specifies the encryption algorithm that was used to protect the biometric sample.

```
[DataContract]
public enum BioSampleEncryption
{
  [EnumMember]
  NONE = 0,
  [EnumMember]
  XTEA = 1,      // XTEA encryption with well-known key
}
```

Enum Member	Description
NONE	The biometric sample provided is not encrypted.
XTEA	The biometric sample provided is encrypted using the DigitalPersona implementation of XTEA with a hardcoded key.

Notes

We strongly recommend using XTEA encryption for biometric samples (as we do in DigitalPersona Pro and DigitalPersona products), but we recognize the complexity of XTEA implementation on different platforms (such as Android) so unencrypted samples are accepted as well.

Biometric Sample Data

The following data formats for fingerprint samples are supported.

- Fingerprint Feature Set
- Raw Fingerprint Image

Fingerprint Feature Set

To specify a Fingerprint Feature Set in the Biometric sample, the following values in **`BioSampleHeader`** must be set.

Value	**Description**
Factor	must be set to 8 (*FINGERPRINT*).
FormatOwner	If the DigitalPersona engine was used for feature extraction, ***FormatOwner*** must be set to **51**. If Neurotechnologija engine was used for feature extraction, ***FormatOwner*** must be set to **49**.
Type	must be set to 2 (*INTERMEDIATE*).
Purpose	can be set to 0 (*ANY*).

Value	Description
Quality	should be set to -1.
Encryption	If XTEA encryption was used, ***Encryption*** must be set to 1. Otherwise, set to 0.

Because we treat Fingerprint Feature Set as an opaque blob, we should provide the Base64Url encoded feature set blob as the ***Data*** member of **BioSample**.

Below is an example of JSON representation of Fingerprint Feature Set.

```
{
  "Version":1,
  "Header":
  {
    "Factor":8,           // Fingerprint
    "Format":
    {
      "FormatOwner":51,     // DigitalPersona engine
      "FormatID":0
    },
    "Type":2,            // Feature Set
    "Purpose":0,         // Any purpose
    "Quality":-1,
    "Encryption":0       // Unencrypted
  },
  "Data":"eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9"
    // Base64url encoded Feature Set
}
```

Fingerprint image

To specify a Fingerprint Image in the Biometric sample, the following values in **`BioSampleHeader`** must be set.

Value	Description
Factor	must be set to 8 (*FINGERPRINT*).
FormatOwner	is ignored and could be set to 51;
Type	must be set to 1 (RAW);
Purpose	is ignored and can be set to 0 (*ANY*);
Quality	should be set to -1.
Encryption	If XTEA encryption was used, ***Encryption*** must be set to 1. Otherwise, set to 0.

Notes

We cannot treat fingerprint image as an opaque blob because the image has many parameters which need to be passed like image size, image resolution, etc.

We use **`the FD_Image`** format which we are using in “C” code as a prototype for Web image representation to simplify the image conversion on the DigitalPersona Server.

The following classes are defined in WBF terminology for specifying an image in the biometric sample.

```
/* uImageType */
[DataContract]
public enum FpImageType
{
  [EnumMember]
  UNKNOWN =                       0,
  [EnumMember]
  BLACK_WHITE =                   1,
  [EnumMember]
  GRAY_SCALE  =                   2,
  [EnumMember]
  COLOR       =                   3,
}
/* uPadding */
[DataContract]
public enum FpImagePadding
{
  [EnumMember]
  NO_PADDING  =                     0,
  [EnumMember]
  LEFT_PADDING =                    1,
  [EnumMember]
  RIGHT_PADDING =                   2,
}
/* uPolarity */
[DataContract]
public enum FpImagePolarity
{
  [EnumMember]
  UNKNOWN_POLARITY     =            0,
  [EnumMember]
  NEGATIVE_POLARITY    =            1,
  [EnumMember]
  POSITIVE_POLARITY    =            2,
}
/* uRGBcolorRepresentation */
[DataContract]
public enum FpImageColorRepresantation
{
  [EnumMember]
  NO_COLOR_REPRESENTATION = 0,
  [EnumMember]
  PLANAR_COLOR_REPRESENTATION = 1,
  [EnumMember]
  INTERLEAVED_COLOR_REPRESENTATION = 2,
}
[DataContract]
public class FpDataHeader
{
  [DataMember]
  public Byte uDataType { get; set; }
/* must be 1 which represents 2D image*/
  [DataMember]
  public UInt64 DeviceId { get; set; }
```

```
    /* reserved for future use            */
    /* FD_DEVICE_TYPE provides the following information:          */
    /* 0000     -  05BA      - 0001       - 01             - 04     */
    /* reserved    vendor Id   product Id    major revision    minor revision*/
  [DataMember]
  public UInt64 DeviceType { get; set; }
    /* image acquisition or device testing may take some time.
    /* Progress indicator    */
   /* gives a feed back on the progress of the transaction.*/
  [DataMember]
  public Byte iDataAcquisitionProgress { get; set; } /* should be 100*/
}
[DataContract]
public class FpImageFormat
{
  [DataMember]
  public Byte uDataType { get; set; }
  /* is 1 which represents 2D image */
    [DataMember]
    public FpImageType uImageType { get; set; }
    /* B&W, gray or color images                */
    [DataMember]
    public Int32 iWidth { get; set; }
    /* image width [pixel]                      */
    [DataMember]
    public Int32 iHeight { get; set; }
    /* image height [pixel]                     */
    [DataMember]
    public Int32 iXdpi { get; set; }
    /* X resolution [DPI]                       */
    [DataMember]
    public Int32 iYdpi { get; set; }
    /* Y resolution [DPI]                       */
    [DataMember]
    public UInt32 uBPP { get; set; }   /*number of bits per pixel    */
    [DataMember]
    public FpImagePadding uPadding { get; set; }
    /* right or left padding                    */
    [DataMember]
    public UInt32 uSignificantBpp { get; set; }
    /* number of significant bits per pixel     */
   [DataMember]
   public FpImagePolarity uPolarity { get; set; }
    /* positive=black print on white background  */
   [DataMember]
   public FpImageColorRepresantation uRGBcolorRepresentation { get; set; }
   [DataMember]
   public UInt32 uPlanes { get; set; }   /* color planes.*/
}
[DataContract]
public enum FpImageCompression
{
  [EnumMember]
  NONE = 0,                    // Data is not compressed
```

```
  [EnumMember]
  JASPER_JPEG = 1,              // Jasper JPEG compression
  [EnumMember]
  WSQ = 2,                      // WSQ compression
}
[DataContract]
public class FpImage
{
  public FpImage()
  {
    Header = new FpDataHeader();
    Format = new FpImageFormat();
  }
  [DataMember]
  public Byte Version { get; set; }
  [DataMember]
  public FpDataHeader Header { get; set; }
  [DataMember]
  public FpImageFormat Format { get; set; }
  [DataMember]
  public FpImageCompression Compression { get; set; }
  [DataMember]
  public String Data { get; set; }
}
```

Value	Description
Version	Version of Fingerprint Image format. Must be 1 in this version.
Header	Header which specifies details of the fingerprint imaging device.
Format	Format of the fingerprint image. It details image size, image resolution, etc.
Compression	Compression algorithm used to compress the fingerprint image. In this version the only supported compression algorithm is Jasper JPEG.
Data	Base64url encoded fingerprint image.

Below is an example of fingerprint image JSON representation.

```
{
  "Version":1,
  "Header":
  {
    "uDataType":1,        // 2D image
    "DeviceId":0,
    "DeviceType":49264417347272704, // U.are.U 5200 device
    "iDataAcquisitionProgress":100
  },
  "Format":
  {
    "uDataType":1,        // 2D image
    "uImageType":2,       // Gray scale
    "iHeight":400,        // 400 pix width
    "iWidth":400,         // 400 pix height
    "iXdpi":500,          // 500 dpi
```

```
    "iYdpi":500,          // 500 dpi
    "uBPP":8,             // 8 bits per pixel
    "uPadding":2,         // right padding
    "uSignificantBpp":8   // 8 significant bits per pixel
    "uPolarity":2,        // positive polarity
    "uPlanes":1,          // 1 plane
    "uRGBcolorRepresentation":0   // no color representation
  },
  "Compression":0,        // uncompressed
  "Data":"eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9" // Base64url encoded image
}
```

Creating a JSON representation of a BioSample from a fingerprint image

To create a JSON representation of **BioSample** from a fingerprint image, perform the following steps.

1 Capture a fingerprint image using any supported capturing device.

2 Base64url encode the image raw bytes. These raw bytes should not include any metadata like image size or resolution; those data will be provided in the JSON Header and Format sections.

3 Knowing the fingerprint image metadata information, create a JSON representation of the **FpImage** class.

4 Base64url encode UTF-8 representation of JSON representation of **FpImage**;

5 Create JSON representation of **BioSample** as described in **"BioSample class" on page 99**, setting the string we got in step #5 as **Data** member of **BioSample**;

Below we give an example of how to create a JSON representation of **BioSample** from a fingerprint image.

Let's assume you capture a fingerprint image using a U.are.U 5200 device. The raw image bytes are the following:

```
[123,34,116,121,112,34,58,34,74,87,84,34,44,10,32,34,97,108,103,34,58,34,
32,82,83,50,53,54,34,125]
```

The Base64url representation of this image is:

```
eyJ0eXAiOiJKV1QiLAogImFsZyI6IiBSUzI1NiJ9
```

Knowing the image width and height, resolution, etc we can create a JSON representation of the **FpImage** class. (Note that the sequence of nodes in JSON representation is unimportant.)

```
{"Compression":0,"Data":"eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9","Forma
t":{"iHeight":400,"iWidth":400,"iXdpi":500,"iYdpi":500,"uBPP":8,"uDataTyp
e":1,"uImageType":2,"uPadding":2,"uPlanes":1,"uPolarity":2,"uRGBcolorRepr
esentation":0,"uSignificantBpp":8},"Header":{"DeviceId":0,"DeviceType":49
264417347272704,"iDataAcquisitionProgress":100,"uDataType":1},"Version":1
}
```

Base64url encoding UTF-8 representation of image above, we get the following string:

```
eyJDb21wcmVzc2lvbiI6MCwiRGF0YSI6ImV5SjBlWEFpT2lKS1YxUWlMQTBLSUNKaGJHY2lPa
UpJVXpJMU5pSjkiLCJGb3JtYXQiOnsiaUhlaWdodCI6NDAwLCJpV2lkdGgiOjQwMCwiaVhkcG
kiOjUwMCwiaVlkcGkiOjUwMCwidUJQUCI6OCwidURhdGFUeXBlIjoxLCJ1SW1hZ2VUeXBlIjo
yLCJ1UGFkZGluZyI6MiwidVBsYW5lcyI6MSwidVBvbGFyaXR5IjoyLCJ1UkdCY29sb3JSZXBy
ZXNlbnRhdGlvbiI6MCwidVNpZ25pZmljYW50QnBwIjo4fSwiSGVhZGVyIjp7IkRldmljZUlkI
jowLCJEZXZpY2VUeXBlIjo0OTI2NDQxNzM0NzI3MjcwNCwiaURhdGFBY3F1aXNpdGlvblByb2
dyZXNzIjoxMDAsInVEYXRhVHlwZSI6MX0sIlZlcnNpb24iOjF9
```

Finally we can create a JSON representation of **BioSample**:

```
{
  "Version":1,
  "Header":
  {
    "Factor":8,
    "Format":
    {
      "FormatOwner":51,
      "FormatID":0
    },
  "Type":1,
  "Purpose":0,
  "Quality":-1,
  "Encryption":0
  },
  "Data":"eyJDb21wcmVzc2lvbiI6MCwiRGF0YSI6ImV5SjBlWEFpT2lKS1YxUWlMQTBLSUNKaG
  JHY2lPaUpJVXpJMU5pSjkiLCJGb3JtYXQiOnsiaUhlaWdodCI6NDAwLCJpV2lkdGgiOjQwMCwi
  aVhkcGkiOjUwMCwiaVlkcGkiOjUwMCwidUJQUCI6OCwidURhdGFUeXBlIjoxLCJ1SW1hZ2VUe
  XBlIjoyLCJ1UGFkZGluZyI6MiwidVBsYW5lcyI6MSwidVBvbGFyaXR5IjoyLCJ1UkdCY29sb3JS
  ZXByZXNlbnRhdGlvbiI6MCwidVNpZ25pZmljYW50QnBwIjo4fSwiSGVhZGVyIjp7IkRldmlj
  ZUlkIjowLCJEZXZpY2VUeXBlIjo0OTI2NDQxNzM0NzI3MjcwNCwiaURhdGFBY3F1aXNpdGlv
  blByb2dyZXNzIjoxMDAsInVEYXRhVHlwZSI6MX0sIlZlcnNpb24iOjF9"
}
```


Creating Fingerprint Credentials from BioSample(s)

Follow these steps to create a JSON representation of **`Credential`** class which we can send in one of the **`IDPWebAuth`** interface methods from one or more biometric samples.

1 Base64url encode the binary representation of any fingerprint feature set(s) or encode any fingerprint image(s) as described in the previous topic.

2 Create a JSON representation **`BioSample`**(s).

3 Combine one or more **`BioSample`**s in a JSON array using square brackets []. NOTE: we support multiple biometric samples in one **`Credential`** to handle two (multi) finger identification/verification.

4 Base64url encode UTF-8 representation of JSON array of **`BioSample`**s in a string;

5 Create a JSON representation of the credentials using the Fingerprint ID as the **`id`** member and the string created in step 4 as the **`data`** member.

For example we have a fingerprint feature set which we would like to send to the DigitalPersona Server for identification. This feature set is created using the DigitalPersona fingerprint engine and the following bytes array represents this feature set.


```
[123,34,116,121,112,34,58,34,74,87,84,34,44,10,32,34,97,108,103,34,58,34,
32,82,83,50,53,54,34,125]
```


The Base64url encoded representation of this fingerprint feature set is:


```
eyJ0eXAiOiJKV1QiLAogImFsZyI6IiBSUzI1NiJ9
```


The JSON representation of **`BioSample`** class for this feature set is:


```
{
  "Version":1,
  "Header":
  {
    "Factor":8,
```

```
    "Format":
    {
      "FormatOwner":51,
      "FormatID":0
    },
  "Type":2,
  "Purpose":0,
  "Quality":-1,
  "Encryption":0
  },
  "Data":"eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9"
}
```


Combining the **`BioSample(s)`** in an array we have the following JSON representation:


```
[{
  "Version":1,
  "Header":
  {
    "Factor":8,
    "Format":
    {
      "FormatOwner":51,
      "FormatID":0
    },
    "Type":2,
    "Purpose":0,
    "Quality":-1,
    "Encryption":0
    },
  "Data":"eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9"
}]
```


The Base64url encoded UTF-8 representation of the JSON array above is:


```
W3sNCiJWZXJzaW9uIjoxLA0KIkhlYWRlciI6DQp7DQoiRmFjdG9yIjo4LA0KIkZvcm1hdCI6D
Qp7DQoiRm9ybWF0T3duZXIiOjUxLA0KIkZvcm1hdElEIjowDQp9LA0KIlR5cGUiOjIsDQoiUH
VycG9zZSI6MCwNCiJRdWFsaXR5IjotMSwNCiJFbmNyeXB0aW9uIjowIA0KfSwNCiJEYXRhIjo
iZXlKMGVYQWlPaUpLVjFRaUxBMEtJQ0poYkdjaU9pSklVekkxTmlKOSIJCQkJDQp9XQ0K
```


Finally we create a JSON representation of the **`Credential`** class which we can send for identification (verification):


```
{"id":"AC184A13-60AB-40e5-A514-E10F777EC2F9",
"data":"W3sNCiJWZXJzaW9uIjoxLA0KIkhlYWRlciI6DQp7DQoiRmFjdG9yIjo4LA0KIkZvc
m1hdCI6DQp7DQoiRm9ybWF0T3duZXIiOjUxLA0KIkZvcm1hdElEIjowDQp9LA0KIlR5cGUiOj
IsDQoiUHVycG9zZSI6MCwNCiJRdWFsaXR5IjotMSwNCiJFbmNyeXB0aW9uIjowIA0KfSwNCiJ
EYXRhIjoiZXlKMGVYQWlPaUpLVjFRaUxBMEtJQ0poYkdjaU9pSklVekkxTmlKOSIJCQkJDQp9
XQ0K"}
```


AuthenticateUser

To call the AuthenticateUser() method, a fingerprint credential must already have been created as described in the previous section.

Below is an example of HTTP Body for fingerprint authentication with the fingerprint credential created in the previous section.

```
{
  "user":
  {
    "name":"someone@mycompany.com",
    "type":6
  },
  "credential":
  {
    "id":"AC184A13-60AB-40e5-A514-E10F777EC2F9",
    "data":"W3sNCiJWZXJzaW9uIjoxLA0KIkhlYWRlciI6DQp7DQoiRmFjdG9yIjo4LA0KIkZvc
    m1hdCI6DQp7DQoiRm9ybWF0T3duZXIiOjUxLA0KIkZvcm1hdElEIjowDQp9LA0KIlR5cGUiOj
    IsDQoiUHVycG9zZSI6MCwNCiJRdWFsaXR5IjotMSwNCiJFbmNyeXB0aW9uIjowIA0KfSwNCi
    JEYXRhIjoiZXlKMGVYQWlPaUpLVjFRaUxBMEtJQ0poYkdjaU9pSklVekkxTmlKOSIJCQkJDQp
    9XQ0K"
  }
}
```

IdentifyUser

To call the IdentifyUser() method, the caller must create a fingerprint credential as described previously.

Below is an example of HTTP Body for fingerprint identification with the fingerprint credential created previously.

```
{
  "credential":
  {
    "id":"AC184A13-60AB-40e5-A514-E10F777EC2F9",
    "data":"W3sNCiJWZXJzaW9uIjoxLA0KIkhlYWRlciI6DQp7DQoiRmFjdG9yIjo4LA0KIkZvc
    m1hdCI6DQp7DQoiRm9ybWF0T3duZXIiOjUxLA0KIkZvcm1hdElEIjowDQp9LA0KIlR5cGUiO
    jIsDQoiUHVycG9zZSI6MCwNCiJRdWFsaXR5IjotMSwNCiJFbmNyeXB0aW9uIjowIA0KfSwNCi
    JEYXRhIjoiZXlKMGVYQWlPaUpLVjFRaUxBMEtJQ0poYkdjaU9pSklVekkxTmlKOSIJCQkJDQp
    9XQ0K"
  }
}
```

GetEnrollmentData

To ask information about enrolled fingerprints, the client should send an IDPWebAuth-> GetEnrollmentData request to the server. Below is an example of such a request:

```
https://www.mycompany.com/DPWebAuthService.svc/GetEnrollmentData?user=
someone@mycompany.com&type=6&cred_id=AC184A13-60AB-40e5-A514-E10F777EC2F9
```

The result of a successful GetEnrollmentData request should be a Base64url encoded UTF-8 representation of a list (array) of fingerprints (fingerprint positions) enrolled by the user. Below is the ANSI 381 list of valid fingerprint positions.

Fingerprint position	Value		Fingerprint position	Value
Unknown	0		Left thumb	6
Right thumb	1		Left index finger	7
Right index finger	2		Left middle finger	8
Right middle finger	3		Left ring finger	9

Fingerprint position	Value		Fingerprint position	Value
Right ring finger	4		Left little finger	10
Right little finger	5			

The following GetEnrollmentDataResult indicates that the user has their right thumb, right index finger and left middle fingers enrolled.

```
[{"position":1},{"position":2},{"position":8}]
```

CustomAction

CustimAction is not currently supported for the Fingerprint Credential.

Password Credential

The following ID is defined for Password Credential.

```
{D1A1F561-E14A-4699-9138-2EB523E132CC}
```

Follow these steps to create a Password Credential:

1 Base64url encode UTF-8 representation of the password. NOTE: It is not necessary to include null terminating character to the UTF-8 representation.

2 Create a JSON representation of the Password credential, setting the Password Credential ID as **`the id`** member and the string we created in step #1 as the **`data`** member.

For example, we create a JSON representation of the following password: P@ssw0rd. The Base64url encoded UTF-8 representation of this password is:

```
UEBzc3cwcmQ
```

Finally we create a JSON representation of the **`Credential`** class which we can send for password authentication to the DigitalPersona Server.

```
{"id":"D1A1F561-E14A-4699-9138-2EB523E132CC",
"data":"UEBzc3cwcmQ"}
```

AuthenticateUser

To call the AuthenticateUser() method, the caller must create a password credential as described above.

Below is an example of HTTP Body for password authentication with the password credential created above:

```
{
  "user":
  {
    "name":"someone@mycompany.com",
    "type":6
  },
  "credential":
  {
    "id":"D1A1F561-E14A-4699-9138-2EB523E132CC",
    "data":"UEBzc3cwcmQ"
  }
}
```

IdentifyUser

The Password credential does not support user identification, so an IdentifyUser() call with a Password credential will return a "Not implemented" error.

GetEnrollmentData

The password credential does not support the GetEnrollmentData() call and will return a "Not implemented"error.

CustomAction

The following CustomAction operations are supported for the Password Credential.

- Password Randomization
- Password Reset

Password Randomization

For Password Randomization, the Action ID is "4".

A valid ticket has to be provided to perform Password Randomization. The ticket owner has to have the rights to randomize the user password.

A valid user for whom the password needs to be randomized needs to be provided.

The data parameter of the Credential class should be set to "null".

Below is a valid example of HTTP Body of Password Randomization request.

```
{
  "ticket":{"jwt":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"},
  "user":
  {
    "name":"someone",
    "type":9
  },
  "credential":
  {
    "id":"D1A1F561-E14A-4699-9138-2EB523E132CC ",
    "data":null
  },
  "actionId":4
}
```

This call will send a Password Randomization request for a Non AD user with the account name of "klozin".

NOTE: We can guarantee successful Password Randomization only for Non AD Users in DigitalPersona LDS installations. We can randomize password for AD user only in DigitalPersona AD installation.

Password Reset

Password Reset Action ID is "13".

Valid ticket has to be provided to perform Password Reset. Ticket owner has to have rights to "set user password".

Valid user to whom password needs to be reset needs to be provided.

Data parameter of Credential class should be set to Base64Url encoded UTF8 representation of user new password.

Below is a valid example of HTTP Body of Password Reset request:

```
{
```

```
 "ticket":{"jwt":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"},
 "user":
{
"name":"klozin",
"type":9
},
 "credential":
{
"id":"D1A1F561-E14A-4699-9138-2EB523E132CC ",
"data":"UEBzc3cwcmQ"
},
"actionId":13
}
```

This call will send Password Reset request for Altus user with account name "klozin".

NOTE: We can guarantee successful Password Reset only for Altus Users in Altus LDS installations. We can reset password for AD user only in Altus AD installation.

PIN Credential

The following ID is defined for the PIN Credential.

```
{8A6FCEC3-3C8A-40c2-8AC0-A039EC01BA05}
```

Follow these steps to create a PIN Credential:

1 Base64url encode UTF-8 representation of PIN. NOTE: It is not necessary to include null terminating character to UTF-8 representation.

2 Create a JSON representation of the PIN credential, setting PIN Credential ID as **`id`** member and the string we created in step #1 as **`data`** member;

For example, we create a JSON representation of the following PIN: 1234. The Base64url encoded UTF-8 representation of this PIN is:

```
MTIzNA
```

Finally we create a JSON representation of **`Credential`** class which we can send for PIN authentication to the DigitalPersona Server.

```
{"id":"8A6FCEC3-3C8A-40c2-8AC0-A039EC01BA05",
"data":"MTIzNA"}
```

AuthenticateUser

To call the AuthenticateUser() method, the caller must create a PIN credential as described above.

Below is an example of HTTP Body for PIN authentication with the previously created PIN credential.

```
{
  "user":
  {
    "name":"someone@mycompany.com",
    "type":6
  },
  "credential":
  {
    "id":"8A6FCEC3-3C8A-40c2-8AC0-A039EC01BA05",
    "data":"MTIzNA"
  }
}
```

IdentifyU[ser

The PIN credential does not support user identification, so an IdentifyUser() call with a PIN credential will return a "Not implemented" error.

GetEnrollmentData

The PIN credential does not support the GetEnrollmentData() call, and will return a "Not implemented" error.

CustomAction

CustomAction is not currently supported for the PIN Credential.

Live Questions Credential

The following ID is defined for the Live Questions Credential.

```
{B49E99C6-6C94-42DE-ACD7-FD6B415DF503}
```

Before a user can answer their Live Questions, the software needs to know which questions to ask the user, the user language id, etc. To ask for this information, the software sends a **IDPWebAuth-> GetEnrollmentData** request to the server. Below is an example of such a request.

```
https://www.digitalpersona.com/DPWebAuthService.svc/GetEnrollmentData?user=
someone@mycompany.com&type=6&cred_id=B49E99C6-6C94-42DE-ACD7-FD6B415DF503
```

In the following topic, details are provided about the data returned as a result of this request.

GetEnrollmentData

The result of a successful GetEnrollmentData request should be a Base64url encoded UTF-8 representation of the list (array) of questions enrolled by the user. The LiveQuestion class represents every question in the list.

```
[DataContract]
public enum LiveQuestionType
{
  [EnumMember]
  REGULAR = 0,                                    // Regular Question
  [EnumMember]
  CUSTOM = 1,                                     // Custom Question
}
[DataContract]
public class LiveQuestion
{
  [DataMember]
  public Byte version { get; set; }
    // version of Live Question class. Should be set to 1.
  [DataMember]
  public Byte number { get; set; }                // Question number
  [DataMember]
  public LiveQuestionType type { get; set; } // Question type
  [DataMember]
  public Byte lang_id { get; set; }
    // Language ID should be used to display this question
  [DataMember]
  public Byte sublang_id { get; set; }
    // Sublanguage ID should be used to display this question
  [DataMember]
  public UInt32 keyboard_layout { get; set; }
```

```
    // Keyboard layout should be used to type answer
  [DataMember]
  public String text { get; set; }
    // Custom question text. For custom questions only.
}
```

Value	Description
version	Version of Live Question format. Must be 1 in this version.
number	Live Question number. This number has a meaning only for regular questions (see below).
type	Type of Live Question. We support two types of Live Question: 1) Regular and 2) Custom.
lang_id	This primary language id was used to display the question during enrollment. The client application should (if possible) use the same primary language id to display the question.
sublang_id	This sublanguage id was used to display the question during enrollment. The client application should (if possible) use the same sublanguage id to display the question.
keyboard_layout	This keyboard layout was used for typing an answer to the question during enrollment. The client application must use the same keyboard layout for the user to enter the answer to this question.
text	Question text. Text should be provided for both Custom and Regular questions and must be in the appropriate language (correspond *lang_id* and *sublang_id* as described above).

Below is an example of the JSON representation of a **`Regular`** Live Question:

```
{
  "version":1,                // must be set to 1
  "number":2,                 // question number in regular question list
  "type":0,                   // type - regular question
  "lang_id":9,                // language - English
  "sublang_id":1,             // Sublanguage - English US
  "keyboard_layout":1033,     // Keyboard layout - English US
  "text”:”What was the name of the first school you attended?”
                              // text of the question
}
```

Below is an example of JSON representation of a **`Custom`** Live Question:

```
{
  "version":1,                // must be set to 1
  "number":102,               // question number
  "type":1,                   // type - custom question
  "lang_id":9,                // language - English
  "sublang_id":1,             // Sublanguage - English US
  "keyboard_layout":1033,     // Keyboard layout - English US
  "text":"Date of your employment."    // custom question text
}
```

Parsing GetEnrollmentDataResult

These are the steps to process (parse) **GetEnrollmentDataResult**.

1 Get the string provided in **GetEnrollmentDataResult**.

2 Base64url decode the string to get the UTF-8 representation of the Live Questions list.

3 Decode the UTF-8 string to a format compatible with the JSON parser.

4 Using the JSON parser, parse the string to objects of the **LiveQuestion** class.

The following is an example of **GetEnrollmentDataResult** parsing.

The client application makes the following call:

```
https://www.mycompany.com/DPWebAuthService.svc/GetEnrollmentData?user=
someone@mycompany.com&type=6&cred_id=B49E99C6-6C94-42DE-ACD7-FD6B415DF503
```

and receives the following result:

```
{"GetEnrollmentDataResult":"W3sia2V5Ym9hcmRfbGF5b3V0IjoxMDMzLCJsYW5nX2lkIjo5L
CJudW1iZXIiOjIsInN1YmxhbmdfaWQiOjEsInRleHQiOm51bGwsInR5cGUiOjAsInZlcnNpb24iOj
F9LHsia2V5Ym9hcmRfbGF5b3V0IjoxMDMzLCJsYW5nX2lkIjo5LCJudW1iZXIiOjYsInN1Ymxhbmd
aWQiOjEsInRleHQiOm51bGwsInR5cGUiOjAsInZlcnNpb24iOjF9LHsia2V5Ym9hcmRfbGF5b3V0I
joxMDMzLCJsYW5nX2lkIjo5LCJudW1iZXIiOjYsInN1YmxhbmdfaWQiOjEsInRleHQiOiJEYXRlIG
9mIHlvdXIgZW1wbG95bWVudC4iLCJ0eXBlIjoxLCJ2ZXJzaW9uIjoxfV0"}
```

Parsing the result would give us the following string:

```
W3sia2V5Ym9hcmRfbGF5b3V0IjoxMDMzLCJsYW5nX2lkIjo5LCJudW1iZXIiOjIsInN1Ymxhbmdfa
WQiOjEsInRleHQiOm51bGwsInR5cGUiOjAsInZlcnNpb24iOjF9LHsia2V5Ym9hcmRfbGF5b3V0Ij
oxMDMzLCJsYW5nX2lkIjo5LCJudW1iZXIiOjYsInN1YmxhbmdfaWQiOjEsInRleHQiOm51bGwsInR
5cGUiOjAsInZlcnNpb24iOjF9LHsia2V5Ym9hcmRfbGF5b3V0IjoxMDMzLCJsYW5nX2lkIjo5LCJu
dW1iZXIiOjYsInN1YmxhbmdfaWQiOjEsInRleHQiOiJEYXRlIG9mIHlvdXIgZW1wbG95bWVudC4iL
CJ0eXBlIjoxLCJ2ZXJzaW9uIjoxfV0
```

Base64url decoding the string above will give us the following JSON representation:

```
[{"keyboard_layout":1033,"lang_id":9,"number":2,"sublang_id":1,
"text":”What was the name of the first school you attended?”,
"type":0,"version":1},{"keyboard_layout":1033,"lang_id":9,"number":6,
"sublang_id":1,"text":”Who was your first employer?”,
"type":0,"version":1},{"keyboard_layout":1033,"lang_id":9,"number":102,
"sublang_id":1,"text":"Date of your employment.","type":1,"version":1}]
```

Using the JSON parser, we have the following Live Questions:

```
{
  "version":1,
  "number":2,
  "type":0,
  "lang_id":9,
  "sublang_id":1,
  "keyboard_layout":1033,
  "text":”What was the name of the first school you attended?”
}
{
  "version":1,
  "number":6,
  "type":0,
  "lang_id":9,
  "sublang_id":1,
```

```
  "keyboard_layout":1033,
  "text":”Who was your first employer?”
}
{
  "version":1,
  "number":102,
  "type":1,
  "lang_id":9,
  "sublang_id":1,
  "keyboard_layout":1033,
  "text":"Date of your employment."
}
```

Creating the Live Questions Credential

The **`LiveAnswer`** class represents the answers to **`LiveQuestion`**.

```
[DataContract]
public class LiveAnswer
{
  [DataMember]
  public Byte version { get; set; }
    // Version of Live Question class. Should be set to 1.
  [DataMember]
  public Byte number { get; set; }     // Question number
  [DataMember]
  public String text { get; set; }     // Question answer
}
```

Value	Description
version	Version of Live Answer format. Must be 1 in this version.
number	Live Question number. This number must correspond to the question number received in GetEnrollmentDataResult.
text	Answer text.

Below is the JSON representation of **`LiveAnswer`** to the question number 102 from example in **“Parsing GetEnrollmentDataResult” on page 117**.

```
{
  "version":1,// must be set to 1
  "number":102,// question number
  "text":"04/24/2009"// custom question text
}
```

Steps to create a Live Questions Credential

To create the Live Questions Credential, perform the following steps.

1 Collect user answers.

2 Create a JSON representation of **`LiveAnswer`** for every answer.

3 Combine those answers in a JSON array.

4 Base64url encode UTF-8 representation of string created in step #3 above.

5 Create a JSON representation of the **`Credential`** class using Live Question Credential ID as **`id`** member and the string created in step #4 as the **`data`** member.

For example, assume the user gave the following answers to the Live Questions in the above example.

```
Canyon Middle
SampleCo
04/24/2009
```

Based on those answers, the client application must create the following JSON representation for the array of **`LiveAnswer`** objects:

```
[
{
  "version":1,
  "number":2,
  "text":"Canyon Middle "
},
{
  "version":1,
  "number":6,
  "text":"SampleCo"
},
{
  "version":1,
  "number":102,
  "text":"04/24/2009"
}
]
```

Base64url encoding UTF-8 representation of string above, creates the following value:

```
W3sidmVyc2lvbiI6MSwibnVtYmVyIjoyLCJ0ZXh0IjoiTmV3IFlvcmsifSx7InZlcnNpb24iO
jEsIm51bWJlciI6NiwidGV4dCI6IkNhbnlvbiBNaWRkbGUifSx7InZlcnNpb24iOjEsIm51bW
JlciI6MTAyLCJ0ZXh0IjoiMDQvMjQvMjAwOSJ9XQ
```

Finally we create the JSON representation of the Live Question Credentials:

```
{"id":"B49E99C6-6C94-42DE-ACD7-FD6B415DF503",
"data":"W3sidmVyc2lvbiI6MSwibnVtYmVyIjoyLCJ0ZXh0IjoiTmV3IFlvcmsifSx7InZlc
nNpb24iOjEsIm51bWJlciI6NiwidGV4dCI6IkNhbnlvbiBNaWRkbGUifSx7InZlcnNpb24iOj
EsIm51bWJlciI6MTAyLCJ0ZXh0IjoiMDQvMjQvMjAwOSJ9XQ"}
```

AuthenticateUser

To call the AuthenticateUser() method, the caller must create the LiveQuestions credential as described above.

Below is an example of HTTP Body for LiveQuestions authentication with the LiveQuestion credential previously created.

```
{
  "user":
  {
    "name":"someone@mycompany.com",
    "type":6
  },
  "credential":
  {
    "id":"B49E99C6-6C94-42DE-ACD7-FD6B415DF503",
    "data":"W3sidmVyc2lvbiI6MSwibnVtYmVyIjoyLCJ0ZXh0IjoiTmV3IFlvcmsifSx7InZl
```

```
    cnNpb24iOjEsIm51bWJlciI6NiwidGV4dCI6IkNhbnlvbiBNaWRkbGUifSx7InZlcnNpb24i
    OjEsIm51bWJlciI6MTAyLCJ0ZXh0IjoiMDQvMjQvMjAwOSJ9XQ"
  }
}
```

IdentifyUser

The LiveQuestions credential does not support user identification, so an IdentifyUser() call with a LiveQuestions credential will return a "Not implemented" error.

CustomAction

CustomAction is not currently supported for the Live Questions Credential.

Proximity Card Credential

The following ID is defined for the Proximity Card Credential:

```
{1F31360C-81C0-4EE0-9ACD-5A4400F66CC2}
```

We treat the Proximity Card Data (Prox Card ID) as an opaque blob. Follow these steps to create the Prox Card Credential.

1 Base64url encode Prox Card ID.

2 Create the JSON representation of the **`Credential`** class, setting the Proximity Card Credential ID as the **`id`** member and the string we created in step #1 as the **`data`** member; **.**

For example, the client gets Prox Card ID and it is represented by the following byte array:

```
[123,34,116,121,112,34,58,34,74,87,84,34,44,10,32,34,97,108,103,34,58,34,
32,82,83,50,53,54,34,125]
```

The Base64url encoded value of Prox Card ID is:

```
eyJ0eXAiOiJKV1QiLAogImFsZyI6IiBSUzI1NiJ9
```

Then we create a JSON representation of the **`Credential`** class which we can send for Prox Card identification or authentication to the DigitalPersona Server:.

```
{"id":"1F31360C-81C0-4EE0-9ACD-5A4400F66CC2",
"data":"eyJ0eXAiOiJKV1QiLAogImFsZyI6IiBSUzI1NiJ9"}
```

AuthenticateUser

To call the AuthenticateUser() method, the caller must create a Proximity Card credential as described above.

Below is an example of HTTP Body for Proximity Card authentication with a previously created Proximity card credential.

```
{
  "user":
  {
    "name":"someone@mycompany.com",
    "type":6
  },
  "credential":
  {
    "id":"1F31360C-81C0-4EE0-9ACD-5A4400F66CC2",
    "data":"eyJ0eXAiOiJKV1QiLAogImFsZyI6IiBSUzI1NiJ9"
  }
}
```

IdentifyUser

To call the IdentifyUser() method, the caller must create a Proximity Card credential as described above.

Below is an example of HTTP Body for Proximity Card identification with the previously created Proximity card credential.

```
{
  "credential":
  {
    "id":"1F31360C-81C0-4EE0-9ACD-5A4400F66CC2",
    "data":"eyJ0eXAiOiJKV1QiLAogImFsZyI6IiBSUzI1NiJ9"
  }
}
```

GetEnrollmentData

The Proximity Card credential does not support the GetEnrollmentData() call, and will return a "Not implemented" error.

CustomAction

CustomAction is not currently supported for the Proximity Card Credential.

Time-Based OTP (TOTP) Credential

The following ID is defined for the TOTP Credential:

```
{324C38BD-0B51-4E4D-BD75-200DA0C8177F}
```

We treat the OTP code as a regular string. Follow these steps to create an OTP Credential.

1 Base64url encode the UTF-8 representation of the OTP code. NOTE: It is not necessary to include a null terminating character in the UTF-8 representation.

2 Create a JSON representation of the OTP credential, setting TOTP Credential ID as the *id* member and the string we created in step #1 as the *data* member;

For example, to create a JSON representation of the following OTP: 123456:

The Base64url encoded UTF-8 representation of such OTP is:

```
MTIzNAdT
```

We then create a JSON representation of the Credential class which we can send for OTP authentication to the DigitalPersona Server:

```
{"id":"324C38BD-0B51-4E4D-BD75-200DA0C8177F",
"data":"MTIzNAdT"}
```

NOTE: To use Push Notification OTP caller must use word "push" instead of OTP code. Below is example of JSON representation of Credential class which we can send for Push Notification OTP authentication to the DigitalPersona Server:

```
{"id":"324C38BD-0B51-4E4D-BD75-200DA0C8177F",
"data":"cHVzaA"}
```

AuthenticateUser

To call the AuthenticateUser() method, the caller must create a TOTP credential as described above.

Below is an example of HTTP Body for TOTP authentication with a previously created TOTP credential.

```
{
  "user":
```

```
  {
    "name":"somebody@mycompany.com",
    "type":6
  },
  "credential":
  {
    "id":"324C38BD-0B51-4E4D-BD75-200DA0C8177F",
    "data":"MTIzNAdT"
  }
}
```

Below is an example of HTTP Body for Push Notification OTP authentication with TOTP credential created above.

```
{
  "user":
  {
    "name":"someone@mycompany.com",
    "type":6
  },
  "credential":
  {
    "id":"324C38BD-0B51-4E4D-BD75-200DA0C8177F",
    "data":"cHVzaA"
  }
}
```

IdentifyUser

The TOTP credential does not support user identification, so an IdentifyUser() call with a TOTP credential will return a "Not implemented" error.

GetEnrollmentData

As result of successful GetEnrollmentData request should be a Base64url encoded UTF-8 representation OTP based information. We introduce OTPEnrollmentData class to represent this information,

```
[DataContract]
public class OTPEnrollmentData
{
  [DataMember]
  public String pn_tenant_id { get; set; }    // Push Notification Tenant Id.
  [DataMember]
  public String pn_api_key { get; set; }      // Push Notification API Key.
  [DataMember]
  public String nexmo_api_key { get; set; }   // Nexmo API Key.
  [DataMember]
  public String nexmo_api_secret { get; set; }// Nexmo API Secret
  [DataMember]
  public String phoneNumber { get; set; } // Last 4 digits of registered Phone number
  [DataMember]
  public String serialNumber { get; set; } // Last 4 digits of HD OTP Serial number
}
```

Value	Description
pn_tenant_id	Push Notification Tenant Id. This is optional parameter and if customer does not subscribe for DigitalPersona Push Notification this parameter will be set to null.
pn_api_key	Push Notification API Key. This is optional parameter and if customer does not subscribe for DigitalPersona Push Notification this parameter will be set to null.
nexmo_api_key	API Key for Nexmo SMS Gateway. The real API key would not be returned here. If Nexmo API Key is set by the customer, world "set" would be returned here. Otherwise this parameter would be omitted.
nexmo_api_secret	API Secret for Nexmo SMS Gateway. The real API Secret would not be returned here. If Nexmo API Secret is set by the customer, world "set" would be returned here. Otherwise this parameter would be omitted.
phoneNumber	Last 4 digits of registered phone number which would be used for SMS OTP.
serialNumber	Last 4 digits of hardware OTP token serial number assigned to this user.

Below is an example of JSON representation of GetEnroomentDataResult:

```
{
  "pn_tenant_id":"crossmatch.com",
  "pn_api_key":"25317484582342934",
  "nexmo_api_key":"2345467865",
  "nexmo_api_secret":"2345467865",
  "phoneNumber":"7304",
  "serialNumber":"7895",
}
```

Parsing GetEnrollmentDataResult

One must have the following steps to process (parse) GetEnrollmentDataResult:

1 Get string provided in GetEnrollmentDataResult.

2 Base64url decode the string above to get UTF-8 representation of OTPEnrollmentData.

3 Decode UTF-8 string to format compatible with JSON parser (Unicode?).

4 Using JSON parser, parse string above to objects of OTPEnrollmentData class.

CustomAction

The following CustomAction operations are currently supported for the TOTP Credential.

- Send SMS OTP Request
- Send E-Mail OTP Request

Send SMS OTP Request

There are two possible options for an SMS OTP request.

- SMS request for Enrollment
- SMS request for Authentication

Send SMS Request for Enrollment

Send SMS Request operation Action ID is "513".

Caller does not need to provide valid ticket to perform this operation so ticket parameter may be set to "null".

Valid user to whom SMS needs to be send needs to be provided.

Data parameter of Credential class should be set to Base64Url encoded Json Representation of OTPEnrollData class:

The following class will represent enrollment sample for TOTP credentials:

```
[DataContract]
public class OTPEnrollData
{
  [DataMember]
  public String otp{ get; set; }      // String with OTP verification code
  [DataMember]
  public String key { get; set; }     // String with Base64Encoded OTP key
  [DataMember]
  public String phoneNumber { get; set; }     // User phone number
}
```

Value	Description
otp	String with OTP verification code. In case of Send SMS action this parameter should omitted or set to "null".
key	Bease64Url Encoded TOTP key.
phoneNumber	User's phone number.

Below is a valid example of HTTP Body of Send SMS request for Enrollment:

```
{
  "ticket":{"jwt":null},
  "user":
  {
    "name":"someone",
    "type":9
  },
  "credential":
  {
    "id":"324C38BD-0B51-4E4D-BD75-200DA0C8177F",
    "data":"eyJ0eXAiOiJKV1QiLAogImFsZyI6IiBSUzI1NiJ9"
  },
  "actionId":513
}
```

This call will send Enrolment Send SMS request for DigitalPersona user with account name "klozin".

Send SMS Request for Authentication

Send SMS Request operation Action ID is "513".

Caller does not need to provide valid ticket to perform this operation so ticket parameter may be set to "null".

Valid user to whom SMS needs to be send needs to be provided.

Data parameter of Credential class should be set "null".

Below is a valid example of HTTP Body of Send SMS request for Authentication.

```
{
  "ticket":{"jwt":null},
  "user":
  {
    "name":"someone",
    "type":9
  },
  "credential":
  {
    "id":"324C38BD-0B51-4E4D-BD75-200DA0C8177F",
    "data":null
  },
  "actionId":513
}
```

This call will send Authentication Send SMS request for DigitalPersona user with account name "klozin".

Send E-Mail OTP Request

Send E-Mail OTP Request operation Action ID is "514".

Caller does not need to provide valid ticket to perform this operation so ticket parameter may be set to "null".

Valid user to whom E-mail needs to be send needs to be provided. User e-mail address would be retrieved from "E-mail-Addresses" attribute (Ldap-Display-Name is "mail") of user account object in Active Directory. If "mail" attribute is not set in user account in AD, call will fail. If multiple mails set in AD, OTP code will be send to all those mail addresses.

Valid user to whom OTP code requests to be mailed needs to be provided.

Data parameter of Credential class should be set "null".

Below is a valid example of HTTP Body of Send E-Mail OTP request:

```
{
  "ticket":{"jwt":null},
  "user":
  {
    "name":"someone@mycompany.com",
    "type":6
  },
  "credential":
  {
    "id":"324C38BD-0B51-4E4D-BD75-200DA0C8177F",
    "data":null
  },
  "actionId":514
}
```

This call will send OTP code over e-mail for AD user with UPN name "someone@mycompany.com".

Smart Card Credential

The following ID is defined for Smart Card Credential:

```
{D66CC98D-4153-4987-8EBE-FB46E848EA98}
```

AuthenticateUser

The data for a smart card credential is a Base64url encoded UTF-8 representation of the JSON array of the CDPJsonSCAuthToken classes. Every public key is represented by the CDPJsonSCAuthToken class.

```
[DataContract]
public class CDPJsonSCAuthToken
{
  [DataMember]
  public Byte version { get; set; }        // version
  [DataMember]
  public UInt64 timeStamp { get; set; }    // current time
  [DataMember]
  public String keyHash { get; set; }      // public key's hash.
  [DataMember]
  public String signature { get; set; }    // signature
}
```

where:

Parameter	Description
Byte **version**	The version of the CDPJsonSCAuthToken object, which must be set to 1 for the current implementation.
UInt64 **timeStamp**	The UTC time when the object is created (64-bit value representing the number of 100-nanosecond intervals since January 1, 1601).
String **keyHash**	Public key's hash, Base64url UTF-8 encoded string. The public key from the Smart Card must be imported in the *PUBLICKEYBLOB* format (see https://msdn.microsoft.com/en-us/library/ee442238.aspx). After that, the RSA256 hash (see https://en.wikipedia.org/wiki/Secure_Hash_Algorithm) of the key must be calculated. The resulting 32 bytes must be Base64url encoded to the string - **KeyHash** = Base64urlEncode(**RSA256** Hash (**PUBLICKEYBLOB** (PuK))
String **signature**	Timestamp and Public key's hash, Base64url UTF-8 encoded string. The Timestamp (8 bytes) and Public key's *RSA256* Hash (32 bytes) must be combined into a 40 byte array, where the first 8 bytes is the Timestamp and the remainder is the hash. This 40 bytes blob must be hashed again with RSA256 and then signed with the Smart Card's Private Key. The signature algorithm used is specified when the Smart Card key pair is originally created, usually RSA. The resulting signature blob must be Base64url encoded into the string: **KeyHash** = Base64urlEncode(Sign(PrK)(**RSA256** Hash(**Timestamp** + **RSA256** Hash (**PUBLICKEYBLOB** (PuK))))

To create the Smart Card Credential for authentication, follow these steps on the client.

1 Enumerate the asymmetric key pairs on the Smart Card or select the exact key pair to use.

2 Create a JSON representation of the CDPJsonSCAuthToken class for every key pair from step #1.

3 Combine the CDPJsonSCAuthToken classes into JSON array, even if there's only one CDPJsonSCAuthToken.

4 Base64url UTF-8 encode the representation of the string created in step 3.

5 Create a JSON representation of the Credential class using the Smart Card Credential ID as the *id* member and the string created in step 4 as a *data* member.

IdentifyUser

This method is not supported. The Smart Card token does not support user identification.

GetEnrollmentData

This method returns the list of Smart Card credentials enrolled for the user. It can be used to select the Smart Card token for authentication or to delete the token using the DeleteUserCredentials method. Te method does not require that the Smart Card be inserted into the reader when it is called.

As result of this call, a string presenting the Base64url encoded UTF-8 representation of the JSON array of the CDPJsonSCEnrolledToken classes will be returned. Every enrolled Smart Card credential is represented by the CDPJsonSCEnrolledToken class:

```
[DataContract]
public class CDPJsonSCEnrolledToken
{
  [DataMember]
  public Byte version { get; set; }         // version
  [DataMember]
  public UInt64 timeStamp { get; set; }     // enrollment time
  [DataMember]
  public String keyHash { get; set; }       // public key's hash.
  [DataMember]
  public String nickname { get; set; }      // token's nickname
}
```

where:

Parameter	Description
Byte **version**	The version of the CDPJsonSCEnrolledToken object, which must be set to 1 for the current implementation.
UInt64 **timeStamp**	The UTC time when the token is enrolled (64-bit value representing the number of 100-nanosecond intervals since January 1, 1601).
String **keyHash**	Public key's hash, Base64url UTF-8 encoded string. The public key from the Smart Card must be imported in the *PUBLICKEYBLOB* format (see http://msdn.microsoft.com/en-us/library/aa387459(VS.85).aspx). After that, the RSA256 hash (see https://en.wikipedia.org/wiki/Secure_Hash_Algorithm) of the key is calculated. The resulting 32 bytes are Base64url encoded to the string - **KeyHash** = Base64urlEncode(**RSA256** Hash (**PUBLICKEYBLOB** (PuK)) This unique string can be used to unambiguously indicate the card token in the DeleteUserCredentials method.
String **nickname**	The nickname of the smart card token.This string, along with enrollment time, can be used in the UI to list enrolled tokens.

CustomAction

CustomAction is not currently supported for the Smart Card Credential.

Face Credential

One of the following third-party SDKs can be used to support the Face Credential in your application.

- Cognitec FVSDK ver. 9.1.0
- Innovatrics IFace SDK ver. 3.1.0

All the DigitalPersona Servers and Clients in the environment must use the same SDK. *Enrollment data is not compatible between the SDKs!*

The following ID is defined for Face Credentials: {85AEAA44-413B-4DC1-AF09-ADE15892730A}

The BioSample class used in description below is declared in "Altus 1 1 WAS Credentials Format.docx"

AuthenticateUser

The data for Face Authentication is a Base64url encoded UTF-8 representation of the JSON array, containing *BioSample* objects(s).

The following data members for BioSample should be provided for authentication.

Data member	Description
BioFactor	Must be set to 2 (DP_BIO_FACTOR::FACIAL_FEATURES)
BioSamplePurpose	Must be set to 0 (Any purpose) or 1 (purpose Verification)
BioSampleEncryption	Must be 0 (not encrypted) or 1 (XTEA encryption)
BioSampleType	Must be one of the following: DP_BIO_SAMPLE_TYPE::PROCESSED DP_BIO_SAMPLE_TYPE::RAW

DP_BIO_SAMPLE_TYPE::PROCESSED image

Indicates a face template in the internal SDK format.

- Cognitec FIR format;
- Innovatrics Template format.

When the input credential data is already a face template (type 4), then only one BioSample object is expected in the array.

Data member	**Description**
BioSampleType	Must be 4.
BioSampleFormat->FormatOwner	"Organization Identifier" number from the International Biometrics Identity Association. • 0x63 (99) for Cognitec • 0x35 (53) for Innovatrics
Data	Base64url encoded JSON representation of the CDPJsonFIR class described below.

```
[DataContract]
public class CDPJsonFIR
{
  [DataMember]
  public Byte Version { get; set; }    // version
  [DataMember]
  public long SDKVersion { get; set; } // SDK version
```

```
  [DataMember]
  public String Data { get; set; }     // FIR object
}
```

where:

Data member	Description
Byte *Version*	Specifies the version of the CDPJsonFIR object. It must be set to 1.
ULONG *SDKVersion*	Specifies the version of the SDK: • Cognitec FVSDK, must be not less than 0x90100 (ver. 9.1.0). • Innovatrics IFace SDK, must be not less than 0x30100 (ver. 3.1.0).
String *Data*	Contains a Base64url encoded BYTE array. • Cognitec SDK: this BYTE array is a serialized Cognitec FIR object, created using FIR::writeTo() method. • Innovatrics SDK: this BYTE array is a Template object, created using IFACE_CreateTemplate function.

If *BioSampleEncryption* is set to 1 (XTEA encryption), then this data is encrypted.

Below is an example of JSON representation of the CDPJsonFIR object containing the Cognitec FIR object:

```
{
"Version":1,
"SDKVersion":?590080??, //? ?0x90100?, ver. 9.1.0
"Data":"2lvbiI6MSwibnVtYmVyIjoyLCJ0ZXh0IjoiTmV3IFlvcmsifSx7InZlcnNpb24iOjEsI
m51bWJlciI6NiwidGV4dCI6IkNhbnlvbiBNaWRkbGUifSx7InZlcnNpb24iOjEsIm51bWJlciI6MT
AyLCJ0ZXh0IjoiMDQvMjQvMjAwOSJ9XQ" // Base64url encoded serialized FIR object
}
```

Example of JSON representation of the authentication array containing the face FIR template:

```
{
  [{
  "Version":1,
  "Header":
  {
    "Factor":2,           // Facial features
    "Format":
    {
      "FormatOwner":99,   // Cognitec
      "FormatID":0
    },
    "Type":4,             // Face template
    "Purpose":1,          // Verification
    "Quality":-1,
    "Encryption":0        // Unencrypted
  },
  "Data":"eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9"
    // Base64url encoded CDPJsonFIR object
  }]
}
```

DP_BIO_SAMPLE_TYPE::RAW image

Indicates a raw face image. It's recommended to have a minimum of ten *BioSample* objects containing raw images in the authentication array for successful verification.

The only type of raw image supported by the current version is a jpeg file.

BioSampleType - must be 1.

BioSampleFormat->FormatOwner - not used, must be 0.

Data - Base64url encoded JSON representation of the CDPJsonFaceImage class described below:

```
[DataContract]
public class CDPJsonFaceImage
{
  [DataMember]
  public Byte Version   { get; set; }  // version
  [DataMember]
  public DP_FACE_IMAGE_TYPE ImageType  { get; set; } // type of image
  [DataMember]
  public String ImageData { get; set; } // face image
}
```

where:

Data member	Description
Byte *Version*	Specifies the version of the CDPJsonFaceImage object. It must be set to 1.
DP_FACE_IMAGE_TYPE *ImageType*	type of image. Must be set to 1, JPEG_FILE:
String *ImageData*	contains Base64Url encoded raw image data, according to the *ImageType*. In the current version, it's a jpeg file. If BioSampleEncryption is set to 1 (XTEA encryption), this data is encrypted.

```
typedef enum DP_FACE_IMAGE_TYPE
{
  JPEG_FILE = 1, // Base64Url encoded JPEG file
}DP_FACE_IMAGE_TYPE;
```

Below is an example of JSON representation of the CDPJsonFaceImage object containing the raw face image (jpeg file):

```
{
  "Version":1,
  "ImageType":1,// JPEG_FILE
  "ImageData":"W3sidmVyc2lvbiI6MSwibnVtYmVyIjoyLCJ0ZXh0IjoiTmV3IFlvcmsifSx7
  InZlcnNpb24iOjEsIm51bWJlciI6NiwidGV4dCI6IkNhbnlvbiBNaWRkbGUifSx7InZlcnNp
  b24iOjEsIm51bWJlciI6MTAyLCJ0ZXh0IjoiMDQvMjQvMjAwOSJ9XQ"
    // Base64url encoded jpeg file
}
```

Example of JSON representation of the authentication array containing the raw face images (jpeg files):

```
{
  [{
  "Version":1,
```

```
  "Header":
  {
    "Factor":2,         // Facial features
    "Format":
    {
      "FormatOwner":0, // Not used
      "FormatID":0
    },
    "Type":1,           // Raw image
    "Purpose":1,        // Verification
    "Quality":-1,
    "Encryption":0      // Unencrypted
    },
  "Data":"WF0T3duZXIiOjUxLA0KIkZvcm1hdElEIjowDQp9LA0KIlR5cGUiOjIsDQoiUHVycG9z
  …
  …
  ZSI6MCwNCiJRdWFsaX"   // Base64url encoded CDPJsonFaceImage object
  },
  …
  …
  {
  "Version":1,
  "Header":
  {
    "Factor":2,         // Facial features
    "Format":
    {
      "FormatOwner":0, // Not used
      "FormatID":0
    },
    "Type":1,           // Raw image
    "Purpose":1,        // Verification
    "Quality":-1,
    "Encryption":0      // Unencrypted
    },
  "Data":"SGVhZGVyIjp7IkRldmljZUlkIjowLCJEZXZpY2VUeXBlIjo0OTI2NDQxNzM0NzI3Mjc
  …
  …
  wNCwiaURhdGFBY3F1a"   // Base64url encoded CDPJsonFaceImage object
  }]
}
```

The following steps will be needed to create the Face authentication packet.

1 Create JSON representation of BioSample(s).

2 Combine those JSON representations in a JSON array ([]).

3 Base64Url encode the string created in step #2.

4 Create a JSON representation of the Credential class using Face Credential ID as id member and a string created in step #3 as a data member.

IdentifyUser

This method is not supported. The Face Credential does not support user identification.

GetEnrollmentData

This method is not supported.

CustomAction

CustomAction is not supported for the Face Credential.

Contactless Card Credentials

The following ID is defined for Contactless Card Credential:

{F674862D-AC70-48ca-B73E-64A22F3BAC44}

AuthenticateUser

The data for the contactless card credential is a Base64url encoded UTF-8 representation of the JSON CDPJsonCLCAuthToken class:

```
[DataContract]
public class CDPJsonCLCAuthToken
{
  [DataMember]
  public Byte version { get; set; } // version
  [DataMember]
  public String UID { get; set; }   // card UID
  [DataMember]
  public String OTP { get; set; }   // TOTP
}
```

where:

Data member	Description
Byte *version*	Specifies the version of CDPJsonCLCAuthToken object, must be set to 1.
String *UID*	Card's unique ID, array of 64 bytes, presented as Base64url UTF-8 encoded string.
`String OTP`	`6 symbols time based one-time password, generated by the client, presented as string.`

To create the Contactless Card Credential for authentication, following steps must be performed on the client:

1 Read the card UID and the symmetric key stored on the Contactless card, create the SHA 256 hash of this key.

2 Use the key hash as a TOTP seed, generate the OTP.

3 Create a JSON representation of the CDPJsonCLCAuthToken class.

- Use the OTP string created in step #2;
- Base64url UTF-8 encode the card UID;

4 Base64Url encode the JSON representation of the CDPJsonCLCAuthToken class;

5 Create a JSON representation of the Credential class using Contactless Card Credential ID as id member and string created in step #4 as a data member.

IdentifyUser

To call IdentifyUser() method, you must first create the Contactless Card credential (CDPJsonCLCAuthToken) as described above. Below is an example of HTTP Body for Contactless Card identification with Contactless card credential created above.

```
{
  "credential":
  {
    "id":" F674862D-AC70-48ca-B73E-64A22F3BAC44",
    "data":"LAogImFs1QiLAogImFeyJ0eXAiOiJKVBSUzI18"
  }
}
```

GetEnrollmentData

This method is not supported.

CustomAction

CustomAction is not currently supported by Contactless Card Credential.

Windows Integrated Authentication (WIA) Credential

The following ID is defined for the WIA Credential.

{AE922666-9667-49BC-97DA-1EB0E1EF73D2}

The following functions are *not* supported by the WIA Credential:

- GetEnrollmentData
- IdentifyUser
- AuthenticateUser
- AuthenticateUserTicket
- CustomAction

CreateUserAuthentication

The *User* parameter of CreateuserAuthentication is ignored, and the credentialId parameter should be set to AE922666-9667-49BC-97DA-1EB0E1EF73D2.

Below is an example of HTTP Body required to create Extended Authentication for WIA:

```
{
  "user":null,
  "credentialId":"AE922666-9667-49BC-97DA-1EB0E1EF73D2"
}
```

CreateTicketAuthentication

The *ticket* parameter of CreateTicketAuthentication must be a valid Ticket, and the credentialId parameter should be set to AE922666-9667-49BC-97DA-1EB0E1EF73D2.

Below is an example of HTTP Body required to create Extended Authentication for WIA:

```
{
  "ticket":{"jwt":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"},
  "credentialId":"AE922666-9667-49BC-97DA-1EB0E1EF73D2"
}
```

ContinueAuthentication

The *authId* parameter of ContinueAuthentication must be a valid authentication handle returned by CreateUserAuthentication or CreateTicketAuthentication. The *authData* parameter is Based64Url encoded data returned by the WIA client.

Below is an example of HTTP Body for WIA authentication handshake:

```
{
  "authId":657854,
  "authData":"`eypZ3NhZGhhc2Rma0FTREZLYWZyZGtB"
}
```

DestroyAuthentication

The *authId* parameter of DestroyAuthentication must be a valid authentication handle returned by CreateUserAuthentication or CreateTicketAuthentication.

Below is an example of HTTP Body to destroy WIA authentication:

```
{
  "authId":657854
}
```

Email Credential

The following ID is defined for the Email Credential:

```
{7845D71D-AB67-4EA7-913C-F81E75C3A087}
```

The Email credential is an auxiliary credential and cannot be used alone to log on to STS. It has to be combined with other Primary credential(s).

The email credential is some string (details of the string is TBD) which will be sent to the user over e-mail and user can present the very same string during E-mail authentication.

Upon initiating email verification (see the CustomAction defined on page 136), the user should receive an email similar to the following.

To verify your email address, please click the following link:

```
https://sts.yourdomain.com/verifymail&user=John.Doe@yourdomain.com&type=6
&data=COqe3faVtaeWdBD4ncmH4r3C9EQ
```

The URL in the example above has several components.

1 The URL of the service which will process the email verification. In the example it's https://sts.yourdomain.com.

2 The name of the function that will be used for e-mail verification. In the example it's *verifymail*.

3 The user name. In the example it's *user=John.Doe@yourdomain.com*.

4 The user name type. In example it's *type=6* (which signifies a UPN name).

5 Email verification data. In the example it's *data=COqe3faVtaeWdBD4ncmH4r3C9EQ*. This data (COqe3faVtaeWdBD4ncmH4r3C9EQ) has to be provided in the *credential* object's *data* field when calling the AuthenticateUser request (see below). The data will be valid for 30 minutes afterthe email is sent.

AuthenticateUser

To call the AuthenticateUser() method, the caller must constrict the authentication request using the "data" field of the verification URL received through email as the *data* field of the *credential* object.

Below is an example of HTTP Body for email authentication with the Email credential described in the example above.

```
{
  "user":
  {
    "name":"Kirill.Lo@crossmatch.com",
    "type":6
  },
  "credential":
  {
    "id":"7845D71D-AB67-4EA7-913C-F81E75C3A087",
    "data":"COqe3faVtaeWdBD4ncmH4r3C9EQ"
  }
}
```

IdentifyUser

Not supported.

GetEnrollmentData

As result of successful GetEnrollmentData request should be a Base64url encoded UTF-8 representation E-mail based information. We introduce MailEnrollmentData class to represent this information:

```
[DataContract]
public class MailEnrollmentData
{
  [DataMember]
  public Boolean has_mail { get; set; }
  // true if user has mail data enrolled
}
```

Data member	Description
has_mail	is True if the user has email data enrolled and False if not. Usually a user email address is stored as the "mail" attribute in AD or LDS, but in the future we may use other mechanisms as well to find a user's email address such as querying a custom SQL database.

Below is an example of JSON representation of GetEnroomentDataResult.

```
  {
    "has_mail":true
  }
```

Parsing GetEnrollmentDataResult

Use the following steps to process (parse) GetEnrollmentDataResult.

1 Get string provided in GetEnrollmentDataResult.

2 Base64url decode the string above to get UTF-8 representation of MailEnrollmentData.

3 Decode UTF-8 string to a format compatible with the JSON parser (Unicode?).

4 Using the JSON parser, parse the string to objects of the MailEnrollmentData class.

CustomAction

The following CustomAction operations are currently supported by the Email credential.

Send Email Verification Request;

The Send Email Verification Request operation Action ID is "16".

The caller does not need to provide a valid ticket to perform this operation, so the ticket parameter may be set to "null".

A valid user to whom the email should be sent needs needs to be provided.

The *data* parameter of the Credential class does not need to be provided and can be set to “null”.

Below is a valid example of HTTP Body of Send E-mail Verification request:

```
{
  "ticket":{"jwt":null},
  "user":
  {
    "name":"someone@mycompany.com",
    "type":9
  },
  "credential":
  {
    "id":"7845D71D-AB67-4EA7-913C-F81E75C3A087",
    "data":null
  },
  "actionId":16
}
```

This call will send an email verification request to the user John.Doe@yourdomain.com. An example of the email request was provided above.

NOTE: The user must have valid e-mail address stored in our database (AD or LDS) to be able to send the email. If the email address cannot be found in the user record, the following error will be returned.

```
E_ADS_PROPERTY_NOT_FOUND (0x8000500D)
```

The property was not found in the cache. The property may not have an attribute, or is invalid.

U2F Device Credential

The following ID is defined for the U2F Credential.

```
{5D5F73AF-BCE5-4161-9584-42A61AED0E48}
```

AuthenticateUser

This method is not supported.

IdentifyUser

This method is not supported.

GetEnrollmentData

This method is not supported.

CustomAction

The following CustomAction operations are currently supported by the FIDO U2F Credential.

Request Application Id from Server

Application Id request operation Action ID is "17".

The caller does not need to provide a valid ticket to perform this operation, so the ticket parameter may be set to "null".

The caller does not need to provide a valid user name or type to perform this operation, so the *name* parameter may be set to "null" and the *type* parameter can be set to 0.

The *data* parameter of Credential class does not need to be provided and can be set to “null”.

Below is a valid example of HTTP Body of Application Id request:

```
{
  "ticket":{"jwt":null},
  "user":
  {
    "name":null,
    "type":0
  },
  "credential":
  {
    "id":"5D5F73AF-BCE5-4161-9584-42A61AED0E48",
    "data":null
  },
  "actionId":17
}
```

Below is a valid example of HTTP Body of Application Id response:

```
{
  "AppId":"https://sts.crossmatch.com/AppId.json"
}
```

If the AppId is not set on the Server, an appropriate error will be returned.

CreateUserAuthentication

The *credentialId* parameter should be set to 5D5F73AF-BCE5-4161-9584-42A61AED0E48.

Below is an example of HTTP Body required to create Extended Authentication for FIDO U2F:

```
{
  "user":{"name":"John.Doe@yourdomain.com","type":6},
  "credentialId":" 5D5F73AF-BCE5-4161-9584-42A61AED0E48"
}
```

CreateTicketAuthentication

The *ticket* parameter of CreateTicketAuthentication must be a valid Ticket, and credentialId parameter should be set to 5D5F73AF-BCE5-4161-9584-42A61AED0E48.

Below is an example of HTTP Body required to create Extended Authentication for WIA:

```
{
  "ticket":{"jwt":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"},
  "credentialId":"5D5F73AF-BCE5-4161-9584-42A61AED0E48"
}
```

ContinueAuthentication

The *authId* parameter of ContinueAuthentication must be a valid authentication handle returned by CreateUserAuthentication or CreateTicketAuthentication. The *authData* parameter is Based64Url encoded data of FIDO U2F handshake.

Below is an example of HTTP Body for FIDO U2F authentication handshake:

```
{
  "authId":657854,
  "authData":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"
}
```

Handshake data is defined as:

```
[DataContract]
public class U2FHandshake
{
  [DataMember]
  public HandshakeType handshakeType { get; set; }
    // Type of U2F specific handshake.
  [DataMember]
  public String handshakeData { get; set; }
    // Base64Url encoded handshake data.
}
```

Where

```
/// <summary>
/// Handshake types supported by U2F.
/// </summary>
[DataContract]
public enum HandshateType
{
  /// <summary>
  /// Client request challenge from U2F server.
  /// </summary>
  [EnumMember]
  ChallengeRequest = 0,

  /// <summary>
  /// Server respond with U2F challenge.
  /// </summary>
  [EnumMember]
  ChallengeResponse = 1,

  /// <summary>
  /// Client request data verification.
  /// </summary>
  [EnumMember]
  AuthenticationRequest = 2,

  /// <summary>
  /// Server respond with result of data verification.
  /// </summary>
```

```
  [EnumMember]
  AuthenticationResponse = 3,
}
```

The value of *handshakeData* depends on the handshake type. There are four types of handshake.

- Challenge Request
- Challenge Respond
- Authentication Request
- Authentication Respond

Challenge Request

Client send Challenge Request as first step of U2F authentication handshakes. handshakeData of Challenge Request is ignored and should be set to null.

Below is an example of Json representation of Challenge Request:

```
{
  "handshakeType":0,
  "handshakeData":null
 }
```

Challenge Respond

Challenge Respond will be returned by the U2F Server as authData member of ExtendedAUTHResult returned by Challenge Request call.

Below is an example of Challenge Respond:

```
{
  "handshakeType":1,
  "handshakeData":"Z3NhZGhhc2Rma0FTREZLYWZyZGtB"
 }
```

handshakeData of Challenge Respond should be provided in the following format:

```
[DataContract]
public class U2FChallengeRespond
{
  [DataMember]
  public String version { get; set; }        // version.
  [DataMember]
  public String appId { get; set; }          // application ID.
  [DataMember]
  public String challenge { get; set; }      // Base64Url encoded challenge.
  [DataMember]
  public String keyHandle { get; set; }      // Base64Url encoded key handle.
}
```

Data member	Description
version	Version of supported FIDO standard, must be set to "U2F_V2" for current implementation.
appId	Application ID.
challenge	Base64Url encoded challenge.
keyHandle	Base64Url encoded key handle.

Authentication Request

The client sends an Authentication Request as the second step of U2F authentication handshaking. *handshakeData* of the Authentication Request has the following format.

```
[DataContract]
public class U2FAuthenticationRequest
{
  [DataMember]
  public String version { get; set; }          // version.
  [DataMember]
  public String appId { get; set; }            // application ID.
  [DataMember]
  public String serialNum { get; set; }        // serial number of U2F device.
  [DataMember]
  public String signatureData { get; set; }
    //  Base64Url encoded signature of challenge.
  [DataMember]
  public String clientData { get; set; }
    //  U2F token authentication counter.
}
```

Data member	Description
version	Version of supported FIDO standard, must be set to "U2F_V2" for current implementation.
appId	Application ID.
serialNum	U2F device serial number.
signatureData	Base64Url encoded signature of client data.
clientData	Base64Url encoded client data.

See the following webpage for details:

https://fidoalliance.org/specs/fido-u2f-v1.0-nfc-bt-amendment-20150514/fido-u2f-raw-message-formats.html

Client data contains challenge which was generated previously.

Authentication Respond

Authentication Respond is not supported at this time and reserved for future use. The result of Authentication Request will be returned in ExtendedAUTHResult. Json Web Token will be returned if authentication succeeded or error if authentication failed (see topic 4.9 of Web Authentication Service design document for details).

DestroyAuthentication

The *authId* parameter of DestroyAuthentication must be a valid authentication handle returned by CreateUserAuthentication or CreateTicketAuthentication.

Below is an example of HTTP Body to destroy FIDO U2F authentication:

```
{
  "authId":657854
}
```

Section Three - Windows Authentication

This section includes the following chapters:

Chapter Number and Title	Purpose	Page
11 - Introduction		142
3 - Installation		94
12 - Using the SDK		92
14 - Custom Authentication Policies		150

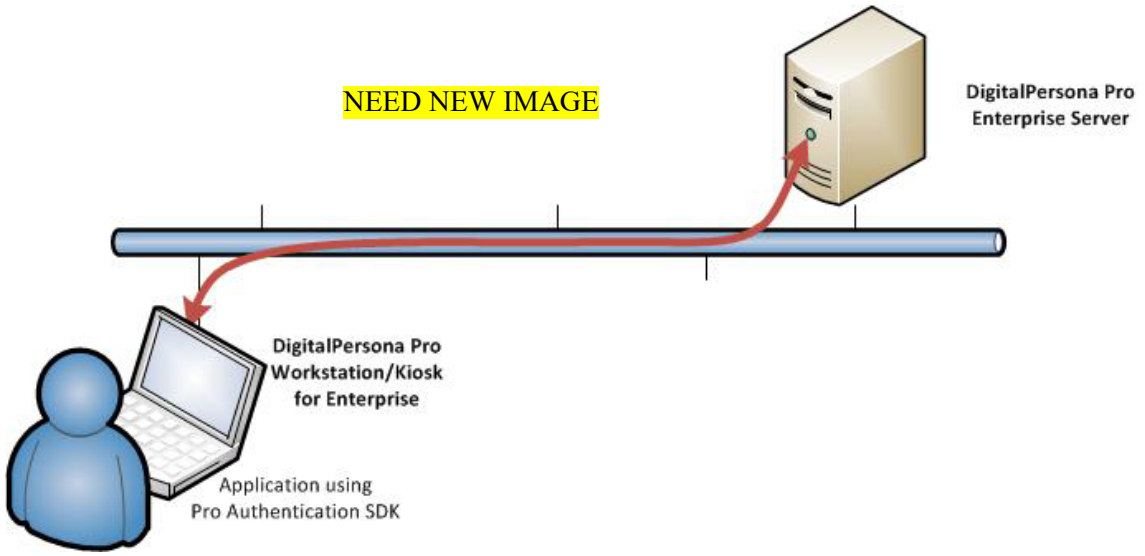
The DigitalPersona AUTH API is a subset of the DigitalPersona Access Management API that provides authentication and identification on the Windows Platform. User enrollment must be handled through a DigitalPersona client.

The AUTH API is automatically installed as part of these DigitalPersona clients:

- DigitalPersona AD Workstation or LDS Workstation
- DigitalPersona AD Kiosk or LDS Kiosk

This documentation, sample code (for C++ and .NET), and any necessary includes are available in a separate DigitalPersona Access Management API package available from Crossmatch and your channel partner or reseller.

When you install a DigitalPersona Workstation or Kiosk client, the DigitalPersona AUTH API runtime is installed as well. As shown in the diagram below, your application runs on workstations that are also running one of the DigitalPersona clients.



The API can be used for the following:

- Authenticating users with the authentication policy and user interface used by DigitalPersona Workstation/Kiosk and optionally reading a user secret.

  The DPAlAuthenticate function displays the multi-factor authentication dialog and matches the supplied credentials against the user's enrolled credentials. The customizable dialog box accepts the credentials required by the authentication policy set by the DigitalPersona administrator. On successful authentication, DPAlAuthenticate can optionally return user secrets to the application.
- Identifying users by searching in the DigitalPersona database to find the user and authenticate them.
- The DPAlIdentAuthenticate function displays the multi-factor identification dialog and identifies the user based on the credentials supplied. The customizable dialog box allows the user to provide the credentials required by the current authentication policy. If the identification succeeds, DPAlIdentAuthenticate can optionally return the user name and secret to the application.
- Retrieving and saving user secrets. Secrets are cryptographically protected and are released to an application only after successful authentication of the user. Secrets are stored in the DigitalPersona database and roam with the rest of the user data.

- Implementing custom authentication policies which extend the DigitalPersona administrator's policies or create new policies.

The DigitalPersona AUTH API observes all of the settings in the DigitalPersona software regarding its communications with the server, supported credentials, policies, etc.

For advanced users, your application can require additional credentials (i.e., you can create a custom authentication policy), but if secret release is required, your application's must meet the requirements of the policy set by the DigitalPersona administrator.

Target Audience

This guide is for developers who have a working knowledge of the C++ programming language. In addition, readers must have an understanding of the DigitalPersona product and its authentication terminology and concepts.

Chapter Overview

Chapter 11, Introduction (this chapter), describes the audience for which this guide is written; cites a number of resources that may assist you in using the Access Management API; identifies the minimum system requirements needed to run the Access Management API; and lists the DigitalPersona products supported by the Access Management API.

Chapter 12, Windows API, describes the typical workflow for authentication on the desktop and describes the functions in the API.

Chapter 13, Windows API Sample Application, describes the features of the sample application.

Chapter 14, Custom Authentication Policies, describes how authentication policies are stored, how to extend an authentication policy and how to define a new authentication policy.

Windows API 12

THIS CHAPTER PROVIDES INSTRUCTIONS ON USING THE DIGITALPERSONA AUTH API.

This chapter describes the standard workflow for using the Altus AUTH SDK API and lists the functions provided.

Two sample applications are provided, one for C++ and one for .NET.

The information in this chapter is extracted from `DPAltusAuthSdkApi.h`.

For terminology and concepts, see the *DigitalPersona Administrator Guides* and the *DigitalPersona CLient Guide.*

Workflow

The normal process is:

call DPAlInit to initialize

. . .

call functions in the API
  if a API function returned an array, string or BLOG, call DPAlFreeBuffer to release the memory

. . .

call `DPAlTerm` to release resources

Note that if you are calling only one function in the API, then you don't need to call `DPAlInit` and `DPAlTerm` directly because the function will initialize and terminate for itself automatically. However when you are making multiple calls, it is faster and more efficient to initialize and terminate directly.

The API provides:

1. **Authentication**: Verifying that a user is who they claim to be by checking that the provided credentials (password, fingerprint, etc.) match their username's credentials in the DigitalPersona database.

   The `DPAlAuthenticate` function displays the multi-factor authentication dialog and matches the supplied credentials against the user's enrolled credentials. The customizable dialog box accepts the credentials required by the authentication policy set by the DigitalPersona administrator.

   Optional: On successful authentication, `DPAlAuthenticate` can return user secrets and the type of credential(s) that the user provided for authentication.
2. **Identification**: Searching in Active Directory to find the user and authenticate them. **For Kiosk environments only.**

   The `DPAlIdentAuthenticate` function displays the multi-factor identification dialog and identifies the user based on the credentials supplied. The customizable dialog box allows the user to provide the credentials required by the current authentication policy.

   Optional: If the identification succeeds, `DPAlIdentAuthenticate` can return the user name, the type of credential(s) used to authenticate and user secrets.

Authentication Policies

The simplest option provided by DigitalPersona AUTH API is to authenticate a user using the session authentication policy defined by the DigitalPersona administrator. In this case, you do not need to know how policies work and you may simply pass NULLs to the API for all parameters that take an authentication policy.

For more information on authentication policies, see Chapter 14, *Custom Authentication Policies,* on *page 150*.

Functions

This section lists the Pro Authentication API functions. For a detailed description of each function's parameters, consult the header file `DPAlAuthSdkApi.h`.

Function	Description
`DPAlInit`	Initialize the authentication functions. Calling this function is optional -- if you do not call it, the system will initialize itself. However if you are going to call multiple authentication functions, it is more efficient and provides better performance if you initialize and terminate explicitly.
`DPAlTerm`	Terminate the authentication process, release resources. You must call `DPAlTerm` once for each time that you called `DPAlInit`.
`DPAlAuthentication`	Display multi-factor authentication dialog and authenticate a user. Optionally return the type of credentials used to authenticate and/or a secret upon successful authentication. Note that this function performs a 1-to-1 comparison -- matching a user's credentials against their enrolled credentials in the DigitalPersona database.
`DPAlIdentAuthenticate`	**For Kiosk environments only.** Display multi-factor identification dialog and identify a user. Optionally return the username, the type of credential(s) used to authenticate and/or user secret(s) upon successful identification. Note that this function performs a 1-to-many comparison -- searching Active Directory to find the user -- and then authenticates the user.
`DPAlReadAuthPolicy`	Read an authentication policy.
`DPAlWriteSecret`	Save the requested secret (authenticated users only).
`DPPTDoesSecretExist`	Check to see if a secret exists.
`DPAlBufferFree`	Release memory buffer allocated by the other functions in the API.
`DPAlFormatMessage`	Returns a message string corresponding to an error code generated by the API. The string is returned in the language of the current user.

Windows API Sample Application 13

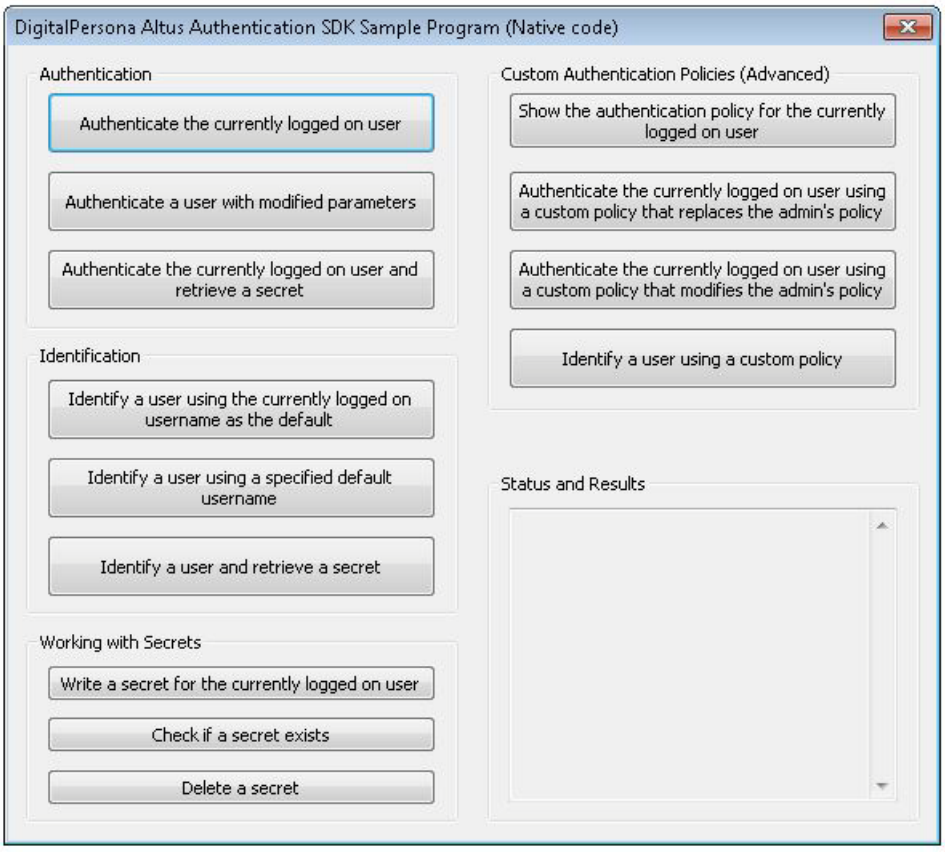
THIS CHAPTER PROVIDES INSTRUCTIONS ON USING THE DIGITALPERSONA AUTH API.

The sample application source code is provided in the folder *Windows API/C++ and .Net AUTH API* (including the Visual Studio 2008 project) from the DigitalPersona API package.

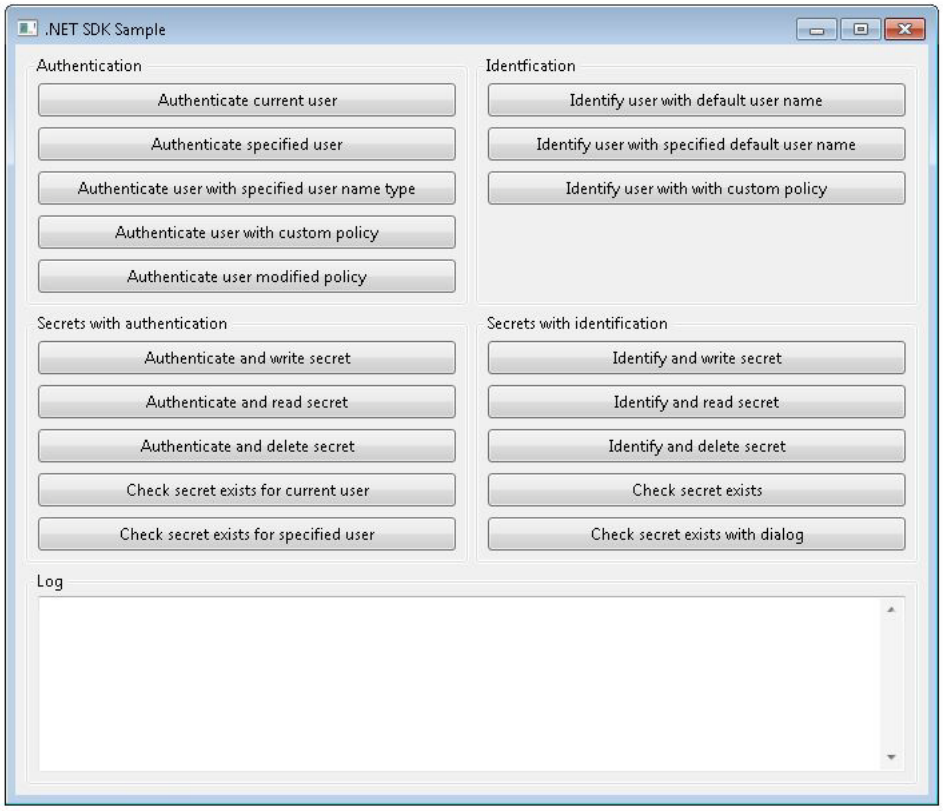
There are two sets of sample code. One for C++ and one for .NET.

The sample application displays a set of buttons that demonstrate a variety of tasks that you might perform with the DigitalPersona API, such as *Authentication, Identification, Working with Secrets* and Custom Authentication Policies. When you run the sample application, the main screen looks like one of the images below.

The source code header file, **DP**`AltusAuthSdk`**Api.h,** *includes detailed comments describing what each button does.*



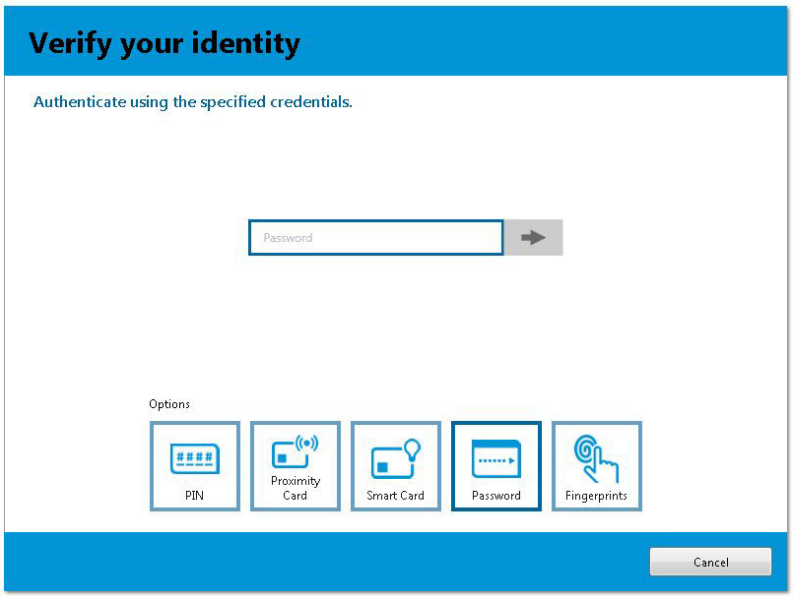
C++ Sample UI



.NET Sample UI

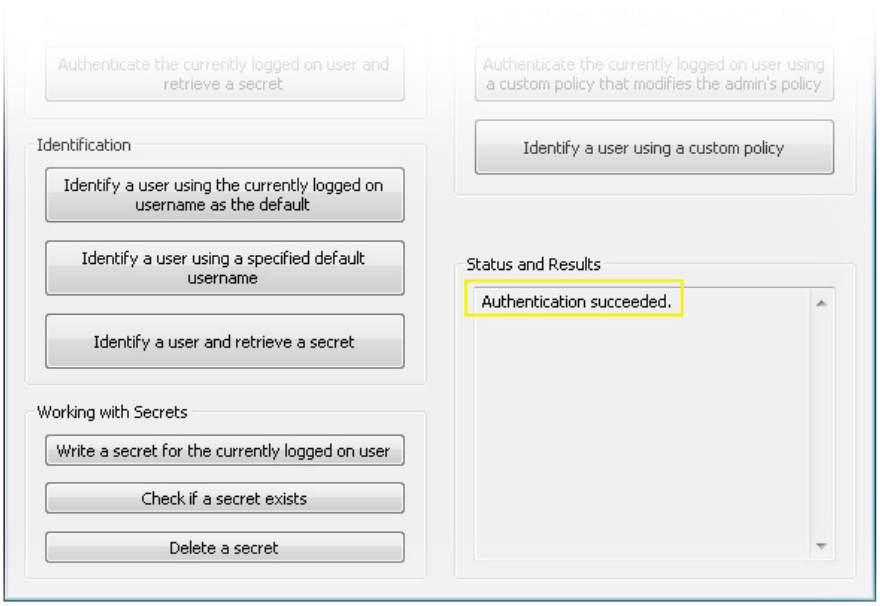
Note that to run the sample application you must have a DigitalPersona client installed.

For example, to authenticate a user, click on the **Authenticate the currently logged on user** button. A standard DigitalPersona dialog box will appear, similar to the one shown below. The actual dialog box will vary depending on your environment and the DigitalPersona client that you have installed.



You can authenticate with any method that you have set up. After you authenticate, the result of the operation will appear in the main screen of the sample program as shown below, in this case the message **Authentication**

succeeded as shown below.



Section Four - Appendix

This section includes the following chapters:

Custom Authentication Policies 14

THIS CHAPTER DESCRIBES HOW TO CREATE AND WORK WITH CUSTOM AUTHENTICATION POLICIES.

This material is for advanced developers only. [IT APPLIES TO WINDOWS, WEB AUTH AND AUTHENTICATION DURING WEB ENROLLMENT?]

Technical support queries regarding custom authentication policies should be sent to `ProSDK`*CAPsupport@digitalpersona.com* rather than the usual DigitalPersona technical support.

If you need DigitalPersona to authenticate users ***and*** return user secrets, then you will need to satisfy the authentication policy defined by the DigitalPersona administrator. You can also choose to define your own custom authentication policy but if you do, your custom policy may not be sufficient for secret release. *DigitalPersona* ***will not release secrets unless you satisfy the authentication policy defined by the DigitalPersona administrator.***

The DigitalPersona administrator must define the authentication policy or policies. Some examples of authentication policies might be:

- Users can authenticate with either a fingerprint or a password but we don't need both fingerprint AND password.
- Users can authenticate with password OR with a smartcard; if they use their smartcard, then they must also enter their PIN.

Consult the *DigitalPersona DigitalPersona or DigitalPersona AD Administrator Guide* (available at *http://www.digitalpersona.com/support/reference-material/pro-reference-material/)* for more information on policies.

Note that an authentication policy is defined for a user on a specified workstation. Users may have different policies, and a policy for one user may not work if you try to use it to retrieve a secret for another user.

How an Authentication Policy is Represented

An authentication policy is represented by an array of ***credential masks***.

To create a data representation of the DigitalPersona administrator's authentication policy, we make a list of all credentials or credential combinations that are permitted. Then we create a credential mask for each valid credential or combination. Each mask has a bit set for every credential that is required in this combination. As long as the user supplies one valid combination of credentials (i.e., satisfies at least one credential mask) they will be authenticated.

Each credential mask is a 64-bit value with each bit representing one valid credential. The bits are defined as:

Table 14.1: Definition of bits in each credential mask

Hex Bit Mask	Credential
0x01	Password
0x02	Fingerprint
0x04	Smart card
0x10	Face
0x20	Contactless card
0x80	PIN

Table 14.1: Definition of bits in each credential mask

Hex Bit Mask	Credential
0x100	Proximity card
0x200	Bluetooth

The simplest authentication policy consists of a single (binary) credential mask:

```
00000000000000000000000000000000000000000000000000000000000000001
```

which represents a policy that requires users to provide a password (the password bit is set).

As another example, if the authentication policy requires BOTH a password and fingerprint, the authentication policy would consist of this credential mask (password and fingerprint bits both set):

```
00000000000000000000000000000000000000000000000000000000000000011
```

If the user can authenticate either with a password OR a fingerprint, the authentication policy would consist of an array containing the following two credential masks. The first credential mask has the bit set for password access and the second mask has the bit set for fingerprint access. As long as a user satisfies ONE of these credentials masks, the user will be authenticated.

```
00000000000000000000000000000000000000000000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000010
```

As another example, the policy below shows the two credential masks for the authentication policy that requires users to authenticate by providing both their fingerprint and their password OR by providing their smartcard.

```
00000000000000000000000000000000000000000000000000000000000000011
00000000000000000000000000000000000000000000000000000000000000100
```

A fairly typical default authentication policy is shown below. This policy allows the user to use ANY one of: password, fingerprint, or smart card.

```
00000000000000000000000000000000000000000000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000010
00000000000000000000000000000000000000000000000000000000000000100
```

The order of credential masks within the Policy array is unimportant. The bits that correspond to each credential type are consistent for all credential masks (i.e., bit 0 always represents that a password is required).

Extending an Authentication Policy

If you want to extend the DigitalPersona administrator's authentication policy, you can read the current policy by calling `DPAlReadAuthPolicy`. You can then modify the credentials masks or add new credentials masks to the authentication policy array. You can then pass your authentication policy to calls to `DPAlAuthenticate` or `DPAlIdentAuthenticate` and your authentication policy will be used instead of the policy defined by the DigitalPersona administrator.

As a best practice, we recommend that this feature only be used to make the DigitalPersona administrator's authentication policy more strict. *DigitalPersona will not release secrets if your policy is less stringent than the existing DigitalPersona authentication policy.*

For example, consider the case where the existing DigitalPersona authentication policy is to allow fingerprints or passwords. In that case, the authentication policy would consist of these two credential masks:

```
00000000000000000000000000000000000000000000000000000000000000001
00000000000000000000000000000000000000000000000000000000000000010
```

If you require that users also use face recognition in addition to either a fingerprint or password, you would update the authentication policy's credential masks to this:

```
00000000000000000000000000000000000000000000000000000000000010001
00000000000000000000000000000000000000000000000000000000000010010
```

In this case, DigitalPersona will release secrets because the policy is stricter than the original.

However if you update the authentication policy to allow a different kind of credential entirely (for example, by adding a new credential mask that allows smart cards), then DigitalPersona will *not* release secret data.

Creating a New Authentication Policy

You can also create your own custom policy. To do this, simply create the appropriate credentials masks and pass your authentication policy to `DPAlAuthenticate` or `DPAlIdentAuthenticate`.

DigitalPersona will not release secrets if your policy is less strict than the existing authentication policy set by the DigitalPersona administrator.

Web Smart Card support 15

THIS CHAPTER DESCRIBES DESCRIBES THE USE OF CONTACTED PKI SMART CARDS IN ALTUS V1.3 FOR WEB AUTHENTICATION, INCLUDING THE AUTHENTICATION END ENROLLMENT ALGORITHMS, WORKFLOWS, IMPLEMENTATION AND DATA FORMATS.

Overview

Contacted PKI Smart Card is a powerful authentication token. The combination of two authentication factors (the card belonging to the user and the PIN known only to the user) alone with the using of the card's RSA private key (which never leaves the card) makes it a perfect and strongest solution for the WEB authentication.

The format of database user record of enrolled Smart Card credentials is defined in the "Smart Card Enrollment Data Format" document[1]. This format was not changed since the early Altus versions and still is used for Altus Client (not WEB-based) Smart Card authentication and enrollment.

The PKI SC WEB authentication token for Altus v1.3 is built according to the requirements described in the "Altus WEB AUTH SDK" documentation[2].

Authentication algorithm

Unlike the commonly used PKI SSL protocol which needs multiple transactions between the client and server to establish the secure WEB connection, the Altus SC authentication algorithm is built to use only one operation, the single call to the server. This remarkably simplifies the operation, especially in the multi- server environment, without the impact on security.

The idea of Altus WEB SC (WASSC) authentication is following:

1 On the client, where the SC is inserted to the reader, the AWEBSC client enumerates the asymmetric key pairs on the card and uses all the available private keys from the card to sign the nonce data.

- We have to use all the private keys from the card because in general the client has no information which key from the card was used for user enrollment:

  $PrK[] = \{ PrK_1 , \ldots PrK_n \};$

  If the client has the information, which key pair must be used for authentication (for ex., the user has selected one of the card certificates in the Card Authentication UI page or one of the enrolled smart card tokens in the list of enrolled tokens), then only this selected key can be used in the following algorithm. But anyway the data structure sent to the server must be presented as an array, even if it contains only one key data:

  $PrK[] = \{ PrK_1 \};$
- As a nonce data, the current time is used in the UTC (8 bytes array) format:

  Nonce = GetCurrentTime(UTC);
- For every private key, the hash of the public key paired to this private key is added to the nonce, and this combination is signed with this private key:

  $Hash = Hash(PuK_i);$

  $Sig_i = Signature(PrK_i)(Nonce + Hash);$
- For every private key, the data sent to the server is a combination of the Nonce, the hash of the public key and the Signature:

  $AuthData_i = Nonce + Hash + Sig_i;$

1. https://tfs.digitalpersona.com/sites/DPCollection1/Enterprise6/Shared Documents/Altus 1.3/Smart Card Enrollment Format.docx
2. https://tfs.digitalpersona.com/sites/DPCollection1/Enterprise6/Shared Documents/Altus 1.2/Altus 1 1 Web AUTH SDK.docx

- The array of AuthData is sent to the server:

  $AuthData[] = \{AuthData_1, \ldots, AuthData_b\}$

2 On the server:

- First, the timestamp of the received data is checked. The server checks if the Nonce provided by the client complies with the limitations in time difference between the server and the client, i.e.

  If (($Nonce_1$ - GetCurrentTime(UTC)) < AheadTimeThreshold) AND

  (GetCurrentTime(UTC) - $Nonce_1$) < BehindTimeThreshold))

  Unlike the time-based OTP solutions, we consider here that both cases: client time is behind the server and client time is ahead of the server are possible, so both thresholds are the same: AheadTimeThreshold = BehindTimeThreshold = **3 min**. This is a default value, but it may be changed by the server's settings.

  All the Nonces in the AuthData array are the same, so only first Nonce is used for calculations.

  This operation is done first, as it's less resource consuming comparing to the cryptographic operations described below.

  If the timestamp of the client data does not match the current server time, the authentication fails with "Out of time" error message.

- The user's SC enrollment record is opened and the proper public key found on this record is used to verify the signatures sent by the client. We have to try all the public keys from the SC enrollment record against all the hashes sent by the client because multiple smart cards (i.e. multiple keys) are allowed to be enrolled for the user:

  $PuKs[] = \{ PuKs_1 , \ldots PuKs_m \};$

  For every $Hash_i$ in AuthData[]

  ```
  {
      For every PuKs_j in PuKs[]
      {
          If (Hash_i = = Hash(PuKs_j) )
          {
              Proceed authentication (i, j);
              Exit;
          }
      }
  }
  ```

  If the hash of one of the public keys stored in the user's SC enrollment record matches one of the hashes sent by the client, then the authentication process is continued with this public key and this nonce/signature data. If no matches were found, the authentication fails with "Not enough information to authenticate" error.

- The server verifies the signature of the data sent by the client, using the found public key from the user SC enrollment record and the found Nonce + Hash record from the authentication data sent by the client:

  If (Sig_i = = VeryfySignature(PuK_j)($Nonce_i$ + $Hash_i$))

  ```
  {
      Authentication succeeded!
  }
  ```

If the signature verification succeeds, the Jason authentication Web Token (JWT) is returned to the client, as described in "Altus Web AUTH Service.docx"[1]. . If the signature verification fails, the "Access denied" error is returned.

Authentication Implementation

The Altus WEB SC (WASSC) authentication token is implemented according to the "Altus WAS Credentials Format.docx"[2] to comply with the demands of **`IDPWebAuth`** interface and **`Credential`** class. The **`Credential`** class is defined as follows:

```
[DataContract]
public class Credential
{
    [DataMember]
    public String id { get; set; }    // unique id (Guid) of credential
    [DataMember]
    public String data { get; set; } //  credential data
}
```

The following ID is defined for Smart Card Credential:

{D66CC98D-4153-4987-8EBE-FB46E848EA98}

Following methods of the **`IDPWebAuth`** interface are supported by WASSC authentication token:

WebAthenticate

To call this method, the client should send **IDPProWebAUTHMgr -> AuthenticateUser** request to the server.

The data for smart card credential is a Base64url encoded UTF-8 representation of the JSON array of the DPJsonSCAuthToken classes. Every public key is represented by the CDPJsonSCAuthToken class:

```
[DataContract]
    public class CDPJsonSCAuthToken
    {
        [DataMember]
        public Byte version { get; set; }     // version
        [DataMember]
        public UInt64 timeStamp { get; set; }     // current time
        [DataMember]
        public String keyHash { get; set; }   // public key's hash.
        [DataMember]
        public String signature { get; set; } // signature
    }
```

where:

Byte **version** - version of CDPJsonSCAuthToken object, must be set to 1 for current implementation.

UInt64 **timeStamp** – Nonce, the UTC time of the moment when the object is created (64-bit value representing the number of 100-nanosecond intervals since January 1, 1601).

String **keyHash** – Public key's hash, Base64url UTF-8 encoded string.

1. https://tfs.digitalpersona.com/sites/DPCollection1/Enterprise6/Shared Documents/Altus 1.1/Altus 1 1 Web AUTH Service.docx
2. https://tfs.digitalpersona.com/sites/DPCollection1/Enterprise6/Shared Documents/Altus 1.1/Altus 1 1 WAS Credentials Format.docx

The public key from the Smart Card must be imported in the **PUBLICKEYBLOB**[1] format. After that, the **RSA256** has[2]h of the key must be calculated. The resulting 32 bytes must be Base64url encoded to the string:

KeyHash = Base64urlEncode(**RSA256** Hash (**PUBLICKEYBLOB** (PuK))

String **signature** – Timestamp and Public key's hash, Base64url UTF-8 encoded string.

The Timestamp (8 bytes) and Public key's **RSA256** Hash (32 bytes) must be combined into 40 bytes array, where the first 8 bytes is Timestamp and the rest is the hash. This 40 bytes blob must be hashed again with **RSA256** and then signed with the Smart Card's Private Key. The signature algorithm used is specified when the Smart Card key pair is originally created, usially it's **RSA**. The resulting signature blob must be Base64url encoded into the string:

KeyHash = Base64urlEncode(Sign(PrK)(**RSA256** Hash(**Timestamp** + **RSA256** Hash (**PUBLICKEYBLOB** (PuK))))

To create the Smart Card Credential for authentication, following steps must be performed on the client:

1 Enumerate the asymmetric key pairs on the Smart Card **or** select the exact key pair to use;

2 Create a JSON representation of the CDPJsonSCAuthToken class for every key pair from step #1;

3 Combine the CDPJsonSCAuthToken classes into JSON array, even if there's only one DPJsonSCAuthToken;

4 Base64url UTF-8 encode the representation of string created in step #3;

5 Finally, create a JSON representation of the **`Credential`** class using Smart Card Credential ID as **`id`** member and string created in step #4 as a **`data`** member.

WebIdentify

This method is not supported. The Smart Card token does not support user identification.

WebGetData

This method returns the list of Smart Card credentials enrolled for the user. It can be used to select the Smart Card token for authentication or to delete with the *WebDelete* method. There's no need to insert the Smart Card to the reader to call this method.

To call this method, the client should send **IDPProWebAUTHMgr -> GetEnrollmentData** request to the server.

As result of this call, a string presenting the Base64url encoded UTF-8 representation of the JSON array of the CDPJsonSCEnrolledToken classes will be returned. Every enrolled Smart Card credential is represented by the CDPJsonSCEnrolledToken class:

```
[DataContract]
    public class CDPJsonSCEnrolledToken
    {
        [DataMember]
        public Byte version { get; set; }      // version
        [DataMember]
        public UInt64 timeStamp { get; set; }      // enrollment time
        [DataMember]
        public String keyHash { get; set; }    // public key's hash.
        [DataMember]
        public String nickname { get; set; }    // token's nickname
    }
```

1. http://msdn.microsoft.com/en-us/library/aa387459(VS.85).aspx
2. https://en.wikipedia.org/wiki/Secure_Hash_Algorithm

where:

Byte **version** - version of CDPJsonSCEnrolledToken object, must be set to 1 for current implementation.

UInt64 **timeStamp** – The UTC time of the moment when the token was enrolled (64-bit value representing the number of 100-nanosecond intervals since January 1, 1601).

String **keyHash –** Public key's hash, Base64url UTF-8 encoded string.

The public key from the Smart Card is imported in the **PUBLICKEYBLOB** format. After that, the **RSA256** hash of the key is calculated. The resulting 32 bytes are Base64url encoded to the string:

KeyHash = Base64urlEncode(**RSA256** Hash (**PUBLICKEYBLOB** (PuK))

This unique string can be used to unambiguously indicate the card token in the *WebDelete* method.

String **nickname –** the nickname of the smart card token.

This string along with enrollment time can be used in the UI to list the enrolled tokens.

Enrollment algorithm

On WEB Smart Card enrollment, one of the public keys stored on the card must be selected and sent to the server.

1 On the client, where the SC is inserted to the reader:

- The enrollment UI must enumerate the appropriate key pairs (or X.509 certificates) on the card, and the user must select one of them. Please note that this operation does not need the card PIN.
- The user must provide a text string - the nickname for the enrolling smart card token to distinguish it from other smart card tokens enrolled for this particular user. By default, the nickname usually contains the card name, like "*SmartCafe Expert 72K DI v3.2*".

  Please note that nicknames are not unique in the WASSC, so it's up to the user to make a good practical nickname for the enrolled token.

2 On the server:

- The server enrollment procedure uses the card public key to encrypt the user sensitive data and create a database user record of enrolled Smart Card, as defined in the "Smart Card Enrollment Data Format" document. Public key encryption does not need the Smart Card presence at that time.

  The time of the enrollment is added to the record, alone with the token's nickname.

Enrollment Implementation

The WEB enrollment of the WASSC authentication token is implemented according to the "Altus 1 2 Web Enroll Service.docx"[1] to comply with the demands of **`IDPWebEnroll`** interface and **`Credential`** class. The **`Credential`** class is defined as following:

```
[DataContract]
public class Credential
{
    [DataMember]
    public String id { get; set; }   // unique id (Guid) of credential
    [DataMember]
    public String data { get; set; } //  credential data
}
```

The following ID is defined for Smart Card Credential:

1. https://tfs.digitalpersona.com/sites/DPCollection1/Enterprise6/Shared Documents/Altus 1.2/Altus 1 2 Web Enroll Service.docx

{D66CC98D-4153-4987-8EBE-FB46E848EA98}

Following methods of the **`IDPWebEnroll`** interface are supported by WASSC authentication token:

WebEnroll

To call this method, the client should send **IDPProWebEnrollMgr -> EnrollUserCredentials** request to the server.

The data for smart card credential is a Base64url encoded UTF-8 representation of the JSON representation of the CDPJsonSCEnrollData class. CDPJsonSCEnrollData class is defined as following:

```
[DataContract]
    public class CDPJsonSCEnrollData
    {
        [DataMember]
        public Byte version { get; set; }     // version
        [DataMember]
        public String key { get; set; }       // public key
        [DataMember]
        public String nickname { get; set; } // token's nickname
    }
```

where:

Byte **version** - version of CDPJsonSCEnrollData object, must be set to 1 for current implementation;

String **key** – public key, Base64url UTF-8 encoded string.

The public key from the Smart Card must be imported in the **PUBLICKEYBLOB** format. After that, it must be Base64url encoded to the string:

Key = Base64urlEncode((**PUBLICKEYBLOB** (PuK))

String **nickname** – the nickname of the smart card token. It's suggested to use the card name, like "*SmartCafe Expert 72K DI v3.2*", as a part of the nickname. The server limits the length of the nickname to

255 symbols, the exceeding symbols will be cut off.

To create the Smart Card Credential for enrollment, following steps must be performed on the client:

1 Enumerate the asymmetric key pairs on the Smart Card **and** select the exact key pair to use. WASSC supports RSA keys of any length but at least 1024 bit length key is suggested for security reason.

2 Create a JSON representation of the CDPJsonSCEnrollData class for the public key selected on step #1;

3 Base64Url encode string created in step #2;

4 Finally, create a JSON representation of the **`Credential`** class using Smart Card Credential ID as **`id`** member and string created in step #3 as a **`data`** member.

WebDelete

To call this method, the client should send **IDPProWebEnrollMgr -> DeleteUserCredentials** request to the server.

To delete the particular Smart Card credential, the input data for this call must be a string presenting the Smart Card Public key's hash, calculated as described in 4.1 or received from the server as described in 4.3.

To delete all the Smart Card credentials enrolled for this particular user, the input data for this call must be an empty string (not a NULL pointer).

Smart Card data format 16

THIS CHAPTER DESCRIBESTHE DATA FORMAT USED WITH SMART CARD ENROLLMENT.

Described below is a data blob used with SC Authentication Provider. The data has variable size, so the main idea is to create it as a fixed-size header followed by a set of tagged records. Data in the blob is stored in a little-endian format.

bRecordNumber is a number of tagged records, 5 records in ver. 2.0.

Version 2.0 defines five valid record tags:

RP – Payload record RC – Certificate record RK – Public key record

RT – Transport key record RD – Card nickname record.

Common constant-size header	WORD wAllSize		Size of all data object, including this field, in bytes
	WORD wVersion	BYTE bMajorVersion	Version = 2.0
		BYTE bMinorVersion	
	BYTE bRecordNumber		Number of records = 5
	FILETIME ftRegistration Time	DWORD dwLowDateTime	Data and time of SC enrollment – 64-bit value representing the number of 100-nanosecond intervals since January 1, 1601
		DWORD dwHighDateTime	
User Key Payload record – contains UKP encrypted by Transport key (EUPK).	BYTE bTag[2] = 'RP'		User record tag
	WORD wRecordSize		Record size, including this field, in bytes
	BYTE bPayload[wRecordSize - 2]		Encrypted UKP
Certificate record – contains information about the certificate connected to the SC public key, according to the X.509 ver.3 specification[1]. This record is optional.	BYTE bTag[2] = 'RC'		Certificate record tag.
	WORD wRecordSize		Record size, including this field, in bytes
	WORD wSNSize		Size of the certificate's Serial Number, in bytes, as described in the CRYPT_INTEGER_BLOB format.
	BYTE bCertSerialNumber[wSNSize]		Serial number of the certificate, as described in the CRYPT_INTEGER_BLOB format.
	WORD wINSize		Size of certificate's Issuer Name, in bytes, as described in the CERT_NAME_BLOB format.
	BYTE bCertIssuerName [wINSize]		Issuer Name of the certificate, as described in the CERT_NAME_BLOB format.
Public key record – contains SC public key.	BYTE bTag[2] = 'RK'		SC public key record tag
	WORD wRecordSize		Record size, including this field, in bytes
	BYTE bKeyData[wRecordSize - 2]		Smart Card public key, in PUBLICKEYBLOB[2] format.
Transport key record – contains transport key encrypted by the SC public key (ETRK).	BYTE bTag[2] = 'RT'		Transport key record tag
	WORD wRecordSize		Record size, including this field, in bytes.
	BYTE bKeyData[wRecordSize - 2]		Encrypted transport key
Nickname record - user-friendly card name used in DP's UI.	BYTE bTag[2] = 'RD'		Nickname record tag.
	WORD wRecordSize		Record size, including this field, in bytes (256 * 2 + 2).
	BYTE bData[wRecordSize - 2]		Data - zero terminated Unicode string, max size 255 symbols.

Index