Using the SignOn Library

Revision History

Version	Date	By	Comments
1	06/01/10	Brian Graversen	Initial release of document
2	06/05/10	Brian Graversen	Removed UserDomain from the documentation
3	13/07/10	Brian Graversen	Updated documentation with regards to return values and the overloaded method GetValue

Introduction

This document contains a guide to using the SignOn Library. Its intended audience is developers of software that needs to use the library. The library is developed as a C# component, which can be accessed directly by other .NET application, as well as through COM from COM-aware applications. A wrapper for Java is also available, where pure Java classes exposes the same functionality as its C# counterparts, and the implementation simple passes the calls to the C# library through native calls.

Common for all languages is that a client builds a request and then passes it to the SignOn method (the only available method on the SignOn Library) which either authenticates the user, or returns an error. Depending on which plugins are used, a number of parameters must be supplied in the request. This guide assumes the use of the CAPI PKI plugin for authentication, and the SOSI GW Plugin for logging in to the STS. For details of using other plugins, see the documentation of those plugins.

We will start by looking at code samples in C#, Visual C (COM) and Java. The full code samples are available in the Sample Code folder bundled with the SignOn Library. Below we will look into the imports parts only, and leave out the “cruft”.

To actually run the code, a working SOSI Gateway is required. Otherwise the GW plugin must be commented out in the code – this still allows the PKI plugin to be tested. Assuming that test certificates are used, the gateway should be configured to test mode.

Test certificates that are accepted by the test STS can be downloaded from http://www.sosi.dk/twiki/bin/view/Support/TestSTSadgang

Using C#

All code samples in this section are taken from the “C# Client” project found in the “Sample Code” folder.

Step 1 – building a SignOn request

```
ISignOnRequest request = new SignOnRequest();

request.LogFile = @"c:\logs\demoAppLog.log";
request.LogLevel = "DEBUG";
request.VerifyPlugins = true;
request.ReturnLog = false;
```

An instance of the SignOnRequest class is the only parameter to the SignOn method, and the above properties are the meta-data settings for the request. None of the properties are mandatory, and the possbibly values (and defaults) are explained in the table below

Property	Default	Description
LogFile	(null)	If set, this property should point to a file which the user has permissions to write to, since all log-information will be written to this file. If not set, nothing will be logged to file.
LogLevel	INFO	Possible values are DEBUG, INFO and ERROR, and they describe the amount of log-information calling the SignOn method will produce (DEBUG will result in the most log-information)
VerifyPlugins	TRUE	Should always be set to TRUE, since it will case the

		library to verify the integrity of all plugins at runtime – It should be set to FALSE only when testing new plugins (that has not yet been signed by SDSD)
ReturnLog	FALSE	If set to TRUE, all log-information will be returned as part of the SignOnResponse, and can be parsed and written to logfile by the calling application.

Once the SignOnRequests meta-data has been set, the plugins can be loaded and configured. The following small code sample shows how to use the SignOnRequest instance to configure and load the GW Plugin – please read the GW Plugin documentation for the description of all the settings available to this plugin.

```
request.AddPlugin(@"libs\GWPlugin.dll", -1, false);

request.AddInitialContext(GwCtxKey.CareProviderId, "19343634");
request.AddInitialContext(GwCtxKey.CareProviderName, "TestProvider");
request.AddInitialContext(GwCtxKey.ITSystemName, "SOSITEST");
request.AddInitialContext(GwCtxKey.GWAddress,
                  "http://localhost:8080/sosigw/service/sosigw");
request.AddInitialContext(GwCtxKey.Issuer, "SOSITEST");
request.AddInitialContext(GwCtxKey.MedcomRole, "læge");
```

The AddPlugin method call takes three arguments, the first is either the name of a plugin installed in the Global Assembly Cache (GAC) or a relative path to a plugin DLL. Using the relative path is the recommended approach, but if the plugins are installed in the GAC on the users machine, accessing them through their name is also possible.

The second parameter is the timeout setting (-1 to disable timeout) in miliseconds, so setting the value to 3000 would require the plugin to execute in maximum 3 seconds – using more time than allowed will result in that plugin failing to execute.

The last argument is a boolean, and when set to TRUE, it will allow the SignOn Library to continue even though the plugin failed (or timed out). This is usefull if more than one plugin is configured, and the caller don't care if they all succeed.

All the subsequent calls to AddInitialContext in the above sample code are merely settings for the GW Plugin. Each plugin has its own set of settings, and the documentation for that plugin should be consulted for the full description of each setting.

The various authentication plugins are configured in much the same way, except for a few additional settings, which are shown below, where the CAPI PKI Plugin is used as an example

```
request.AddPlugin(@"libs\CAPIPKIPlugin.dll", -1, false);
request.Password = "Test1234";
request.UserName = "<Not Used By CAPI Plugin>";
```

An authentication plugin might take additional settings, which are set using AddInitialContext like in the GW Plugin example above, but the CAPI PKI Plugin does not require anything besides the Password. For completeness sake, the special authentication properties on the SignOnRequest are all shown above (Password and UserName). Check the documentation for the authentication plugin that you need to use, to see which of these are needed (and what values to give).

The "more-than-one-certificate" case

Common for all PKI Plugins (a special breed of authentication plugins) is that they access

certificates, and the user may have more then one of these. If such a situation occours, the SignOn method will return an errorcode indicating this, as well as a list of possible choices. These choices should be presented to the user, and after he picks one of them, the call to SignOn is made again, but this time an additional property is set on the SignOnRequest

```
request.SetCertificateChoice(<Users Certificate Choice>);
```

The SignOnRequest instance from the previous call can be reused, or reconstructed, whatever makes sense for the vendor application you are building. When a certificate choice is set in the request, the PKI plugin will use this certificate if more than one is available.

Step 2 – Calling SignOn
In the SignOn method call, we will use the request instance created in step 1.

```
SignOnFacade signOnFacade = new SignOnFacade();

ISignOnResponse response = signOnFacade.SignOn(request);
```

The response object contains several properties about the call, which should be checked by the calling application. They are described in the table below, and sample code shown afterwards

Property	**Description**
Status	This is the property that should be checked first, since it indicates the result of the SignOn method call. The possible values are described in the Error Handling section of this document.
FailedPlugin	This is the name of the plugin that failed – it is only set if a plugin actually failed, and caused the SignOn method call to stop.
FailedPluginMessage	This is the message/reason given by the failed plugin (if any) – it only indicates the actual error, and the log should be consulted for further details.
LogEntries	If configured (ReturnLog = TRUE in the SignOnRequest), or if no LogFile was set, then all logentries created by the library and the plugins, are returned as an array in this field, to be parsed by the vendor application.
CertificateChoices	If (and only if) a PKI plugin encountered the special case, where a user has more than one certificate, then this field contains an array of possible choices, from which the user should pick one – see the “more-than-one-certificate” section above.

Besides these static fields, generic output from the plugins is available in a name/value set on the SignOnResponse – the actual “keys” for the name/values can be found in the plugin documentation. The sample code below will show how to extract some of the more common values.

```
if (response.Status == StatusCodes.Ok)
{
    string loa = response.GetValue(SignOnCtxKey.LevelOfAssurance);
    string idCardHandle = response.GetValue(SignOnCtxKey.IdCardHandle);
}
```

The LevelOfAssurance value is set by the authentication plugins, and reflect the level of authentication that the user has gone through (from 1 to 4), the concept of *Level of Assurance* is well documented in “SDSD Authentication” [1]. The IdCardHandle value is set by the GW Plugin, and is a reference to the ID Card that has been issued during the SignOn method. The client

application should store this value securely, since it will need it for later calls through the SOSI GW.

How to handle the various error cases is not exactly language specific, and is postponsed to the section about error handling found later in this document.

Using COM

The SignOnLibrary.dll file is a .NET assembly, but it also contains COM interfaces, which can be accessed by any COM aware application. All the sample code below is written in Visual C, but it can be accessed from other COM aware languages as well – the syntax should be the same as when accessing other COM component. The samples below are all taken from the "Visual C COM Client" sample project found in "Sample Code" folder bundled with the SignOn Library.

Generating a TLB file from the assembly
Before programming against the application, we will need to create an interface file (TLB), which can be included in the client application. The .NET framework has a tool for this purpose called regasm.exe (Register Assembly). If the file is not in the classpath, and can be found at this location (the exact location depends on your version of Windows)

```
c:\windows\Microsoft.NET\Framework\v2.0.50727\RegAsm.exe
```

The following command is issued on the SignOnLibrary.dll to create the TLB file

```
regasm.exe SignOnLibrary.dll /codebase /tlb:SignOnLibrary.tlb
```

The regasm command also registers the SignOnLibrary.dll as a COM component on this machine, so it is ready to be used from COM aware applications. When deploying to the users machine, the notes regarding deployment[2] should be read.

Using the TLB in the source code

```
#import "SignOnLibrary.tlb" raw_interfaces_only rename("GetObject", "GetObjectA");
```

The above import statement is used to gain access to the interfaces on the SignOn Library, and should be added in the top of the sourcefile that uses the SignOn Library. This is Visual C specific, and other import statements may be required for other COM aware languages. It is important to note that the GetObject method is renamed to GetObjectA to avoid name-clashing with existing methods, but this is the only modification needed to the import statement.

We will now walk through the same steps as described in the C# section above, but we will leave out some of the details regarding the various properties, as these are the same, and not language specific. Please read the C# section before reading the following steps.

Step 1 – Building a SignOn request
An instance of the SignOnRequest class can be created using the interface class through the following bit of code. The meta data properties are then set using the methods on this object

```
SignOnLibrary::ISignOnRequestPtr pRequest(__uuidof(SignOnLibrary::SignOnRequest));

BSTR logFile = SysAllocString(L"c:\\logs\\demoAppLog.log");
BSTR logLevel = SysAllocString(L"DEBUG");

pRequest->put_LogFile(logFile);
pRequest->put_LogLevel(logLevel);
pRequest->put_VerifyPlugins(true);
pRequest->put_ReturnLog(false);
```

```
SysFreeString(logFile);
SysFreeString(logLevel);
```

Except that we have to allocate/free the strings to be passed through the COM calls, the syntax and flow is very similar to the C# sample code, and the description of the various properties can be found in that section of this document.

Loading and configuring plugins is done using the same syntax, and brief samples are included below, just to show that it reflects the C# sample code perfectly.

```
BSTR pluginDll = SysAllocString(L"libs\\GWPlugin.dll");
pRequest->AddPlugin(pluginDll, -1, false);
SysFreeString(pluginDll);

BSTR careProviderId = SysAllocString(L"19343634");
pRequest->AddInitialContext_3(SignOnLibrary::GwCtxKey_CareProviderId, careProviderId);
SysFreeString(careProviderId);
```

The important thing to note is that overloading on the AddInitialContext() method is done by having numbered versions of the method (not all COM aware languages supports overloading), so the version that takes a GwCtx enum is AddInitialContext, while the version that takes Signer keys is suffixed by _2. The list below can be consulted for which to use, if your IDE does not give you this information for free

AddInitialContext	SignOnLibrary::GwCtxKey
AddInitialContext_2	SignOnLibrary::SignerCtxKey
AddInitialContext_3	BSTR

Note that the 3rd choice allows for passing any string value – some plugins may require configuration settings that are not yet part of the official context bundles.

The same goes for the GetValue method on the SignOnResponse class – it is also overloaded, like this

GetValue	SignOnLibrary::SignOnCtxKey
GetValue_2	BSTR

Step 2 – Calling SignOn

Similar to the C# sample code, a SignOnFacade instance is created, the SignOn method called, and a SignOnResponse object returned. The syntax through COM calls is the following.

```
SignOnLibrary::ISignOnResponsePtr pResponse;
SignOnLibrary::ISignOnFacadePtr pFacade(__uuidof(SignOnLibrary::SignOnFacade));
pFacade->SignOn(pRequest, &pResponse);
```

The response object can be queried for the same properties as the C# counterpart, and the following brief bit of code shows how to extract status and the loa/idcard properties

```
SignOnLibrary::StatusCodes status;
pResponse->get_Status(&status);

if  (status == SignOnLibrary::StatusCodes_Ok)
{
        BSTR loa;
        BSTR idCardHandle;

        pResponse->GetValue(SignOnLibrary::SignOnCtxKey_IdCardHandle, &idCardHandle);
        pResponse->GetValue(SignOnLibrary::SignOnCtxKey_LevelOfAssurance, &loa);
```

```
}
```

Handling errors is common for all languages, and the section on Error Handling should be consulted for the various errors that can occur, and how they should be handled.

Using Java

The Java wrapper is designed to look almost identically to the C# version. All code samples in this section are taken from the Java example client, which can be found in the subversion repository.

Creating a request

```
SignOnRequest request = new SignOnRequest();

request.setPassword("Test1234");
request.setLogFile("c:\\logs\\javaKlient.log");
request.setReturnLog(true);
request.setVerifyPlugins(false);
request.addPlugin(new PluginEntry("CAPIPKIPlugin.dll", 30000, false));
```

The parameters are the same as for the C# case, see above for the explanation.

Invoking the SignOn library

Once a request has been constructed, the library is invoked in the following way:

```
SignOnFacade facade = new SignOnFacade();
SignOnResponse response = facade.signOn(request);
```

Error Handling

The SignOnLibrary does not throw any exceptions. Instead errors are reported by setting the status property on the response objects. The possible status codes are defined in the StatusCodes enum, and listed below with a brief description. Sample code follows on how to handle these situations. The sample code is in C#, but other languages should handle the situations similarly.

StatusCode	Numeric	Description
Ok	0	No errors occurred during the SignOn method call.
LoadPluginError	1	One of the configured plugins could not be loaded. Consult the logs for details. Most common errors are: - Plugin file does not exist - Signature on plugin file is invalid - Plugin file is corrupt *Note that if the plugin was configured to continue-on-error, the SignOn method will only log this error, but continue with the next configured plugin, and this StatusCode will not be returned.*
InvokePluginError	2	One of the configured plugins failed during execution. Consult the logs for details. *Note that if the plugin was configured to continue-on-error, the SignOn method will only log this error, but continue with the next configured plugin, and this StatusCode will*

		not be returned.
InternalError	3	An error occurred inside the SignOn Library framework, not related to any of the plugins. Consult the logs for details.
InvalidLogLevel	4	The configured LogLevel on the SignOnRequest was invalid. Only “DEBUG”, “INFO” and “ERROR” is accepted. Correct your code.
NoCertificatesError	5	This is a special return code flagged by an authentication/PKI plugin. It states that the user currently attempting do perform a sign operation has no valid certificates. The user must install a valid certificate on the machine he is using before he can continue!
MultipleCertificatesError	7	This is a special return code flagged by an authentication/PKI plugin. It states that the user has more than one certificate, and that no certificate was chosen using the CertificateChoice on the SignOnRequest. The CertificateChoices property on the SignOnResponse contains an array of possible choices – one of these must be chosen by the user.
InvalidCertificateError	6	This is a special return code flagged by an authentication/PKI plugin. It states that the user has more than one certificate, but that the certificate set in the CertificateChoice on the SignOnRequst is not one of the valid choices – likely this is due to an error in the code calling the SignOn Library – check your code, or contact the PKI plugin developers for support.
InvalidCredentialsError	8	This is a special return code flagged by an authentication/PKI plugin, indicating that the password (or one of the full set of credentials: UserName and Password) entered by the user is incorrect.
TimeoutError	9	One of the configured plugins failed to perform its operation in the time allowed. Check the FailedPlugin property of the SignOnResponse to see which plugin failed – also check the logs for additional details. *Note that if the plugin was configured to continue-on-error, the SignOn method will only log this error, but continue with the next configured plugin, and this StatusCode will not be returned.*

Some of these errors cannot be handled by the code, and should just result in the SignOn failing, but others can be handled either programmatically or by user-interaction. We will look at some C# sample code for handling each error

```
switch (response.Status)
{
	case StatusCodes.Ok:
		// Extract the plugin information, and store the IDCardHandle.
		string loa = response.GetValue(SignOnCtxKey.LevelOfAssurance);
		string idCardHandle = response.GetValue(SignOnCtxKey.IdCardHandle);
		break;
	case StatusCodes.InternalError:
		// Abort the SignOn and check the logs for details
```

```
                break;
            case StatusCodes.NoCertificatesError:
                // Inform the user that he has no valid certificates
                break;
            case StatusCodes.MultipleCertificatesError:
                // Extract the possible choices of certificates, prompt the
                // user, and call SignOn again with this choice.
                CertificateChoice[] choices = response.CertificateChoices;
                // TODO: prompt the user... this code just picks the first certificate
                request.SetCertificateChoice(choices[0]);
                response = facade.SignOn(request);
                // TODO: parse response.Status again
                break;
            case StatusCodes.LoadPluginError:
            case StatusCodes.InvokePluginError:
            case StatusCodes.TimeoutError:
                // Check response.FailedPlugin to see which plugin failed,
                // and consult the logs for additional information. Also the field
                // response.FailedPluginMessage can contain additional information from
                // the plugin; this is also stored in the log – but some plugins may use
                // this field to pass additional statuscode information to be used
                // programatically by the client.
                break;
            case StatusCodes.WrongPasswordError:
                // Inform the user that the username/password was incorrect
                // and prompt him for these credentials again (retry SignOn)
                break;
            case StatusCodes.InvalidLogLevel:
            case StatusCodes.InvalidCertificateError:
                // These errors should not occur in production – check your code
                break;
    }
```

The above switch contains only the bare minimum required to handle the different status codes that can be returned by the SignOn method, but it should give a good idea on how the various status codes are grouped, and what should be done when they occur.

References

[1]	SDSD Authentication	http://www.sdsd.dk/Det_goer_vi/Programmer/~/media/Files/arkitektur/autentfikationsstrategier.ashx
[2]	Deployment Notes	Bundled with the SignOn Library