DIGITAL SUNDHED

SAMMENHÆNGENDE DIGITAL SUNDHED I DANMARK

Plugin Development for 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

Introduction

Writing plugins for the SignOn Library is done in .NET, all the sample code below is written in C#, but any .NET language can be used, as long as it allows you to implement the plugin interface.

To get started, you need the SignOnLibrary.dll file, which is the SignOn Library assembly containing, among other things, the interface(s) that you need to implement. For a quick overview, the interfaces are shown below (they are quite small).

The GW Plugin and the CAPI PKI Plugins are part of the SignOn Library, and the code for these two plugins can be read for details on how a full plugin could be implemented. This document will attempt to describe the structure and workings of basic plugins.

Interfaces for implementing plugins

```
public interface IPlugin
{
	string Name { get; }
}

public interface IActionPlugin : IPlugin
{
	void Invoke(ISignOnContext signOnContext, IApplicationContext applicationContext);
	void Release();
}

public interface IAuthPlugin : IPlugin
{
	void Invoke(ISignOnContext signOnContext, IApplicationContext applicationContext,
			string username, string password);
	void Release();
}

public enum DigestType { MD5, SHA1, SHA256 };
public interface IPKIPlugin
{
	byte[] Sign(byte[] digest, DigestType digestType);
}
```

The only two interfaces that are interesting for actual implemention is the IActionPlugin and IAuthPlugin interfaces. The IPlugin interface is merely a technicality, and the IPKIPlugin interface is a special interface that certain authentication interfaces exposes to other plugins (see the section on authentication plugins for details).

This document will talk about action plugins and authentication plugins separately, and cover the PKI plugin as a special case in the authentication plugin section.

General Setup

Assuming that Visual Studio is used as the IDE, the following quick guide can be used to initiate a new project; other IDEs likely requires something very similar.

Start a new project with "Class Library" as the output type (to create a DLL instead of an EXE). Add the SignOnLibrary.dll as a reference, and ensure that the .NET Framework target is set to 2.0 (all plugins and the library must work with .NET 2.0 and higher).

The output needs to be strongname signed, so right-click on the project, and go to the "Signing" tab. Check "Sign the assembly" and just generate a new strongname key – this will generate a file which should be stored as part of the project – later releases of the same plugin should reuse this

strongname key. The SignOn Library cannot load assemblies that are not strongname signed.

Implementing an Action Plugin

Create a new class that implements the IActionPlugin interface. Something like the following should work as a starting point

```
using SignOnLibrary.Interfaces;

namespace MyActionPluginNS
{
	public class MyActionPlugin : IActionPlugin
	{
		public string Name { get { return "MyActionPluginName"; } }

		public void Invoke(ISignOnContext signOnContext, IApplicationContext appContext)
		{
		}

		public void Release()
		{
		}
	}
}
```

The above skeleton code contains the bare minimum for a working action plugin. The Invoke() method should perform the plugins workload, and the Release() method should be used to cleanup any data or resources that needs to be free'd.

The section on working with the context variables (signOnContext and appContext) should be read to understand how the plugin interacts with the framework and other plugins.

Implementing an Authentication Plugin

Much like the action plugin is created by implementing the IActionPlugin interface, an authentication plugin is created by implementing the IAuthPlugin interface. The following bit of code should work as a good starting point

```
using SignOnLibrary.Interfaces;

namespace MyAuthPluginNS
{
	public class MyAuthPlugin : IAuthPlugin
	{
		public string Name { get { return "MyAuthPluginName"; } }

		public void Invoke(ISignOnContext signOnContext, IApplicationContext appContext,
				string username, string password)
		{
			// example on how to set the level-of-assurance
			signOnContext.SetValue(SignOnCtxKey.LevelOfAssurance, "3");
		}

		public void Release()
		{
		}
	}
}
```

Note that the only real difference is in the Invoke() method, which takes additional credentials as input parameters. An authentication is also required to set the *Level-of-Assurance* upon succesful verification of the credentials – please read the document "SDSD Authentication" [1] regarding valid values of *Level-of-Assurance*.

Special note regarding authentication plugins and Level-of-Assurance

The value may only be increased, never decreased. So if the current plugin would set the value to "3" upon verification of the given credentials, and another plugin has set the value to "4" previously, then this plugin should not modify the value. A more correct approach (than the one used in the sample code above) would be something like the following:

```
string loa = signOnContext.GetValue(SignOnCtxKey.LevelOfAssurance, "3");

if (loa != null && Convert.ToInt32(loa) >= 3)
{
        ; // do nothing
}
else
{
        signOnContext.SetValue(SignOnCtxKey.LevelOfAssurance, "3");
}
```

Notes regarding PKI plugins

PKI plugins are special authentication plugins, which besides doing authentication, also exposes functionality to other plugins, so they can use the users certificate to cryptographically sign data. An authentication plugin that wishes to expose such functionality would have an internal class that implements the IPKIPlugin interface, which it would store in the ApplicationContext that is passed to it as an input on the Invoke() method. The following very simple code shows how this could be done (the CAPI PKI Plugin does this, so looking at that production code might be more rewarding).

```
internal class MySigner : IPKIPlugin
{
        internal MySigner(<relevant information>)
        {
        }

        public byte[] Sign(byte[] digest, DigestType digestType)
        {
                // perform a signature operation on the digest
        }
}
```

The above class is then instantiated by the authentication plugin (passing whatever is needed for the class to perform the signing operation if called later by other plugins) and stored in the ApplicationContext like this

```
        public void Invoke(ISignOnContext signOnContext, IApplicationContext appContext,
                                string username, string password)
        {
                // [... snip all other code in the Invoke method ...]


                MySigner signer = new MySigner(<relevant information>);
                applicationContext.SetObject(PkiCtxKey.PkiPlugin, signer);

                // [... snip all other code in the Invoke method ...]
        }
```

Other plugins can now extract the MySigner instance from the ApplicationContext and call the Sign() method. When Release() is called on the authentication plugin, it should also ensure that no further Sign() calls can be made to the MySigner instance.

Notes regarding interoperability with the GW Plugin

When writing PKI-enabled authentication plugins that should work with the GW Plugin, it is important to note that the GW Plugin needs to know which certificate (X509Certificate2) that is used when the Sign() method is called – the following bit of code should be added to the Invoke() call, after the clients certificate has been chosen (and the credentials verified)

```
X509Certificate2 clientCertificate = ...
applicationContext.SetObject(GwCtxKey.ClientCertificate, clientCertificate);
```

This allows the GW Plugin to extract the certificate for its own use.

Working With Context Variables

When Invoke() is called on a plugin it is given references to two context variables, the SignOnContext and the ApplicationContext. The SignOnContext is used to store information that should be returned to the calling application – currently this only covers a "handle" for the issued IDCard (for the GW Plugin) and the *Level-of-Assurance* (for all authentication plugins).

The plugins do not need to worry about cleaning up after themselves in case of errors, should a plugin fail for one reason or another, the SignOn Library framework will ensure that none of the modifications made by the plugin to the context variables are stored (a rollback mechanism is in effect).

Logging

The client can configure the SignOn Library in various ways regarding logging, but no matter what, all plugins are always given an instance of the ILogger interface, which they can use to log relevant information. It can be extracted from the ApplicationContext when Invoke() is called (and stored locally if needed), and used throughout the plugin.

```
public void Invoke(ISignOnContext signOnContext, IApplicationContext appContext,
                        string username, string password)
{
        // [... snip all other code in the Invoke method ...]


        ILogger logger = null;
        object tmp = applicationContext.GetObject(AppCtxKey.LoggerPlugin);
        if (tmp != null && tmp is ILogger)
        {
                logger = (ILogger) tmp;
        }

        // [... snip all other code in the Invoke method ...]
}
```

Error Signaling

The SignOn Library will call Invoke() on each of the configured plugins, in the order that the plugins where added, and the only way that a plugin has of indicating that an error occured during the call to Invoke(), is to throw an exception. The SignOn Library will catch all exceptions, and handle them according to the following rules.

1. If the plugin was configured to continue-on-error, the exception is merely logged, and the SignOn Library will continue with the next configured plugin and no further steps are performed – otherwise continue with step 2.
2. If the exception is of the type SignOnException (found in the SignOnLibrary namespace), it will check the type of error indicated by the exception to detect if this information needs to be passed on to the client (e.g. The wrong password was entered by the user).
3. The name of the plugin (IPlugin.Name) is stored in the output as SignOnResponse.FailedPlugin, and the exception message as SignOnResponse.FailedPluginMessage – this allows the plugin to pass some information back to the client if needed (it is not recommended, but possible).

4. The exception is logged to the internal logging mechanism in the SignOn Library framework.
5. The execution of the SignOn() method is stopped and a SignOnResponse.Status is set to indicate what went wrong (possible influenced by the exception if it is of the type SignOnException).

SignOnException – what is it for?

For most error situations, simply throwing an exception is fine. This will result in the SignOn Library returning an InvokePluginError to the calling application, and all the relevant information will be available both in SignOnResponse.FailedPlugin/FailedPluginMessage as well as the logs.

In the special cases, where the client program can actually do something sensible (like ask the user for his password again in case he entered it incorrectly), the SignOnException enters the picture.

By throwing a SignOnException instead of any other exception type, the SignOn Library will return a different status code than InvokePluginError – the sample below shows how to use the exception, and the table shows the possible options, and when to use them.

```
if (<user entered wrong password>)
{
        throw new SignOnException(PluginError.WrongPassword, "Incorrect Credentials");
}
```

PluginError Type	Status Code Mapped	Description
TimeOut	TimeOutError	Should not be used by the plugins, it is used by the SignOn Library in case any plugin fails to perform within the given time.
InvalidCredentials	InvalidCredentialsError	Can be used by authentication plugins when the given credentials are incorrect.
MultipleCertificates	MultipleCertificatesError	Can be used by PKI plugins when the user has more than one certificate. When throwing an exception with this flag, it should also set the possible choices on the SignOnContext (see the multiple certificates section below)
NoCertificates	NoCertificatesError	Can be used by PKI plugins when the user does not have any valid certificates.
InvalidCertificateChoice	InvalidCertificateError	Can be used by PKI plugins when the calling application has attempted to select a certificate that does not exist (likely an implementation error?).

Handling the multiple-certificates situation

Authentication plugins that accesses a users certificates can experience the situation that the user has more than one certificate – rather than let the plugin decide which to use, the decision is left up to the user/client. The plugin should throw a SignOnException with type MultipleCertificates, and

set the possible CertificateChoices on the SignOnContext. The following bit of code shows how this could be done

```
if (<user has more than one certificate>)
{
        List<X509Certificate2> clientCertificates = ... <from somewhere> ...
        List<CertificateChoice> certificateChoices = new List<CertificateChoice>();

        foreach (X509Certificate2 certificate in clientCertificates)
        {
                string prettyName = ... <displayed to the user> ...
                string identifier = ... <used to identify the certificate internally> ...

                CertificateChoice choice = new CertificateChoice(prettyName, identifier);
                certificateChoices.Add(choice);
        }

        signOnContext.PKIContext.CertificateChoices = certificateChoices.ToArray();
        throw new SignOnException(PluginError.MultipleCertificates, ">1 Certificate");
}
```

Basically each possible certificate is mapped to a CertificateChoice, which consists of two entries; a PrettyName which is displayed to the user (something readable), and an internal identifier. The plugin cannot keep state, so the identifier has to be something that can be used to locate that exact certificate when called at a later time (e.g. the certificates sha1-digest).

If the Invoke() method is called, and the user has more than one certificate, the ApplicationContext should be checked for a CertificateChoice before returning the above error – the caller could be calling for the second time, and thus a choice about which certificate to use could be included in the request.

```
string identifier = (string) applicationContext.GetObject(PkiCtxKey.CertificateChoice);
```

If the identifier above is not-null, it should be checked against the possible certificates available to that user, and if no matches are found, a SignOnException with the type InvalidCertificateChoice should be thrown. Note that the identifier-string returned in the above line is created by the block of code that creates all the possible CertificateChoices.

Additional Notes

A single .NET assembly (DLL) should not contain more than one plugin implementation. The SignOn Library will only load a single plugin from a given DLL, and if the DLL contains more than one plugin implementation, the first plugin encountered will be used.

References

[1]	SDSD Authentication	http://www.sdsd.dk/Det_goer_vi/Programmer/~/media/Files/arkitektur/autentfikationsstrategier.ashx